INSTITUT FÜR MATHEMATIK

TUHH Technische Universität Hamburg

Development of a Conversational Interface Based on Institution-Specific Documentation through LLM Finetuning

Forschungsprojektarbeit

von
Philip Suskin
aus Hamburg
Matrikelnummer: 53890
Studiengang: Computer Science

January 4, 2024

Erstprüfer: Dr. Jens-Peter M. Zemke
Betreuer: Dr. Jens-Peter M. Zemke

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

> "Development of a Conversational Interface Based on Institution-Specific Documentation through LLM Finetuning"

selbständig und ohne unzulässige fremde Hilfe verfasst 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. Ich versichere, dass die eingereichte schriftliche Fassung der auf dem beigefügten Medium gespeicherten Fassung entspricht.

Ort, Datum	Unterschrift

Contents

List of Figures

Abstract

Modern advancements in the field of Natural Language Processing, driven primarily by Large Language Models, open the door not only to a plethora of AI-driven applications, but to the ability to produce such applications at or near the industry standard without the need for industrial-grade computational resources. In particular, requirements regarding GPU memory have sunk drastically as a result of recent advancements. However, it is not always clear what implications these improvements have for smaller organizations planning to work with Large Language Models. We review the extent to which consumer-grade hardware is capable of training language models of varying parameter count and analyze the performance of resulting models. Furthermore, we share our experience regarding training methodology and evaluative measures through an intuitive practical use case.

1 Introduction

Language models have rapidly made their way into a variety of applications, such as conversational interfaces, document summarization, sentiment analysis, and many more. In particular, conversational interfaces have begun to reshape the way humans gather and manage information. From an institutional perspective, these interfaces could be implemented to offer clients automated assistance with a plethora of common tasks and queries specific to the institution's domain. Implementing an industrial-strength interface for automated dialog currently involves using **Large Language Models**, language models apt to handle the complexity of language-based learning tasks through the substantial number of parameters in their architecture.

1.1 Large Language Models

Large Language Models (LLMs), in the context of Natural Language Processing (NLP), are models which process textual data, most commonly implemented using **Transformers**. Transformers are a neural network architecture developed by Vaswani et al. 2017 aimed at computing contextualized **embeddings** of **tokenized** sequences of text, thereby producing a comprehensive semantic representation of the input. This works using a mechanism called **self-attention**.

Early instances of large language models are found in models such as GPT-1 (Radford and Narasimhan 2018), the first iteration of the Generative Pre-trained Transformers (GPT) models, developed by OpenAI and released in June 2018. This model began paving the way for future generative language models. In October of the same year, the Bidirectional Encoder Representations (BERT) (Devlin et al. 2018) model, developed by Google, was released. It capitalized on the ability to learn contextual representations based on context both to the left and to the right of words, thereby achieving high performance on multiple NLP tasks, such as sentiment analysis and text classification. In the following years, further GPT iterations were released, yielding GPT-2 (Radford, Wu, et al. 2019) in February 2019 and GPT-3 (Brown et al. 2020) in June 2020. More recently, the LLaMA (Touvron, Lavril, et al. 2023) and LLaMA-2 (Touvron, Martin, et al. 2023) models, developed by Meta AI, were released in February and July 2023, respectively. These two open-source models exhibit state-of-the-art benchmark performance and are provided at various *parameter counts*. As implied by the name, the parameter count details the number of parameters in the respective model architecture. The LLaMA models were developed at parameter counts of 7b (7 billion), 13b, 33b, and 65b. The LLaMA-2 models were developed at parameter counts of 7b, 13b, 34b, and 70b, though the 34b version has not been publicly released to date. For reference, a 7b model occupies approx. 13.3 GB of storage, and this parameter-count-to-storage ratio holds for models of higher parameter count.

1.1.1 Tokens

Tokens are basic units of text that form the sequences processed by LLMs. They follow a fixed vocabulary, in which each token is assigned a unique integer ID, managed by a *tokenizer*. With the help of a tokenizer, raw text can be tokenized by encoding sub-sequences of text into their respective ids as they are encountered. The simplest tokenizers are *whitespace tokenizers*, which split text by spaces, creating a vocabulary that consists of all words encountered in the provided text corpora. However, this method of tokenization can lead to many problems once the vocabulary is established. For example, the risk of encountering out-of-vocabulary words is high and requires a colossal vocabulary to minimize. For this reason, the current standard approach to tokenization involves *Byte-Pair Encoding (BPE)*, a method that generates a vocabulary consisting of the subwords that occur most frequently in the reference corpora (Sennrich, Haddow, and Birch 2015). In order to set up BPE tokenization, the frequency with which each word occurs in the training data is first established. Following this, a base vocabulary containing all characters found in these words is generated, whereby the vocabulary entries are called *symbols*. Symbols are then iteratively merged according to the symbol pair that occurs most frequently in the original word frequency measurement until the target vocabulary size is attained. An application of BPE tokenization is provided using the LLaMA tokenizer on the example sentence "This evaluation serves the analysis of recorded BDE time tickets." in Listing 1. This example demonstrates that BPE symbols can be (and, given sufficient vocabulary size, often are) entire words.

```
['This', 'evaluation', 'serves', 'the', 'analysis', 'of', 'recorded', 'B', 'DE', 'time', 'tick
    ↪ ', 'ets', '.']
```

Listing 1: BPE tokenization example

Though it depends on the tokenizer, estimates on the token-to-word ratio that can be expected for general text under BPE amount to around 4 tokens for every 3 words (Petrov et al. 2023). The ratio achieved by the LLaMA tokenizer on our dataset does not stray too far from this estimate, sitting at approx. 1.72 tokens per word (instead of $1.\overline{3}$). The increased ratio on our dataset is arguably explainable by the high frequency of technical terminology and acronyms occurring in the training samples.

Based on their assigned ID, tokens are one-hot encoded in order to create a vector representation suitable for the LLM architecture, with a typical one-hot encoding dimension, i.e., vocabulary size, on the order of 10^4 (32000 for LLaMA/LLaMA-2). Subsequently, embeddings are formed from these vectors, which aim to establish semantic representations of tokens at a reduced dimension compared to the one-hot encoding dimension.

1.1.2 Embeddings

In general, embeddings form a lower-dimensional continuous representation out of one-hot encoded tokens within a designated vocabulary. In contrast to one-hot token encodings, typical embeddings have a dimension on the order of 10^3 (4096 for LLaMA-2-7b). In

an abstract sense, the continuous representation formed by an embedding, which can be expressed as a vector $\boldsymbol{v} = (v_1, \ldots, v_n)$, infuses, or better, *embeds*, a semantic description into its elements v_i, though this description is context-free for elementary encoders such as *word2vec* (Mikolov et al. 2013). Still, non-contextual embeddings possess valuable properties and relationships to other embeddings, such as reduced distance[1] to semantically related embeddings in the present vector space. A further property of these embeddings is their ability to produce *linear word analogies* (Allen and Hospedales 2019). A linear word analogy f is an invertible transformation over a set of ordered token pairs S for which $f(\boldsymbol{v}) = \boldsymbol{w} \wedge f^{-1}(\boldsymbol{w}) = \boldsymbol{v} \quad \forall(\boldsymbol{v}, \boldsymbol{w}) \in S$ holds and f has the form $\boldsymbol{v} \mapsto \boldsymbol{v} + \boldsymbol{r}$. An example commonly referenced in order to exemplify this property is the analogy $\boldsymbol{v}_{\text{king}} + (\boldsymbol{v}_{\text{woman}} - \boldsymbol{v}_{\text{man}}) \approx \boldsymbol{v}_{\text{queen}}$, where $\boldsymbol{v}_x$ is the embedding vector of some token x. Analogies like these arise in the embedding vector space due to co-occurrence probabilities aligning with the structure of the analogy (Ethayarajh, Duvenaud, and Hirst 2018). More formally, an analogy "$\boldsymbol{a}$ is to $\boldsymbol{b}$ as $\boldsymbol{c}$ is to $\boldsymbol{d}$" holds in the respective embedding vector space if and only if $\frac{p(\boldsymbol{w}|\boldsymbol{a})}{p(\boldsymbol{w}|\boldsymbol{b})} \approx \frac{p(\boldsymbol{w}|\boldsymbol{c})}{p(\boldsymbol{w}|\boldsymbol{d})} \quad \forall \boldsymbol{w} \in T$, where T is the token vocabulary and $p(\boldsymbol{v}_x|\boldsymbol{v}_y)$ is the co-occurrence probability for the tokens x and y, i.e., the probability that x is found in the current context window given that y is found in it (Pennington, Socher, and Manning 2014). Another key insight regarding embeddings is the fact that the discussed properties do not result from a pre-defined embedding space, but instead arise inherently as embeddings are learned during training. Embeddings are learned through a designated embedding layer positioned at the front of the Transformer Architecture, detailed in Section 1.2.

Despite these important properties, lack of consideration for context in embeddings can inhibit LLMs from adequately processing linguistic features required to model natural language. Attention is a mechanism aimed at providing context in addition to embedding the semantic description of tokens, in order to alleviate this linguistic constraint.

1.1.3 Attention

Attention can be expressed as a functional mapping of a query and a set of key-value pairs to an output, where these elements are expressible as the vectors $q \in \mathbb{R}^{d_k}, k \in \mathbb{R}^{d_k}, v \in \mathbb{R}^{d_v}$, respectively (Vaswani et al. 2017, Section 3.2). Specifically, the output is computed as a weighted sum of the values, where the weight assigned to each value derives from a compatibility function of the query with the corresponding key. In practice, attention is computed on sets of queries, keys, and values simultaneously, where the sets are expressed as the matrices $Q \in \mathbb{R}^{n \times d_k}, K \in \mathbb{R}^{n \times d_k}, V \in \mathbb{R}^{n \times d_v}$, respectively, with $n \in \mathbb{N}$ as the sequence length. An important component of attention is *softmax*, an activation function defined on $\mathbb{R}^n \to \mathbb{R}^n$ that transforms output neuron activations into a probability

[1] Distances between embeddings are generally computed using Euclidean or cosine distance, both of which exhibit the phenomenon of reduced magnitude between semantically related embeddings.

distribution over the classification space[2], defined as:

$$\text{softmax}(\boldsymbol{x})_i := \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \in \mathbb{R}^n. \tag{1.1}$$

The prevailing implementation of attention is *Scaled Dot-Product Attention*, see (Vaswani et al. 2017, Equation 1), defined as follows:

$$\text{Attention}(Q, K, V) := \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \in \mathbb{R}^{n \times d_v}.$$

Scaled Dot-Product Attention first computes the scaled product of the queries and keys, before applying softmax column-wise, after which the result is multiplied by the attention values. The product of the queries and keys is scaled according to the reciprocal of the square root of their shared dimension $\frac{1}{\sqrt{d_k}}$ in order to counteract the potential for softmax to face *vanishing gradients*, which can occur when the input to softmax is of large magnitude. The vanishing gradient problem refers to a phenomenon in back-propagation in which gradients become extremely small across many consecutive backward passes, hindering the training process by causing negligible updates to weights, particularly in early layers. The structure of Scaled Dot-Product Attention is visualized in Figure 1 (Vaswani et al. 2017, Figure 2).

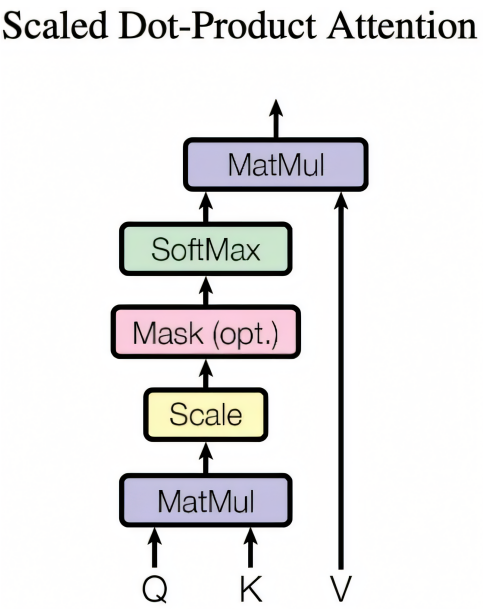


Figure 1: Visualization of Scaled Dot-Product Attention (from Vaswani et al. 2017)

In order to improve efficiency for the repeated computation of the attention function, the queries, keys, and values are linearly projected h times with learned projections to the

[2]In the case of LLMs, the softmax function transforms the output activations into a probability distribution over the corresponding tokenizer vocabulary.

dimensions d_k, d_k, and d_v, respectively, instead of computing the attention function once with d-dimensional vectors, where d is the pre-trained model dimension. This allows for the parallel execution of the attention function on the projected queries, keys, and values, yielding d_v-dimensional intermediate vectors. These are then concatenated and projected again, resulting in the final output vectors. This process is called *Multi-Head Attention*, see (Vaswani et al. 2017, Section 3.2.2), formally defined as follows:

$$\begin{aligned}\text{MultiHead}_{W^Q,W^K,W^V,W^O}(X,Y) &:= \text{Concat}(\text{head}_1,\ldots,\text{head}_h)W^O \in \mathbb{R}^{n\times d},\\ \text{head}_i &:= \text{Attention}(\underbrace{YW_i^Q}_{Q_i},\underbrace{XW_i^K}_{K_i},\underbrace{XW_i^V}_{V_i}),\end{aligned}$$

where $X \in \mathbb{R}^{n\times d}$ and $Y \in \mathbb{R}^{n\times d}$ are the inputs to the current attention unit, the learned projections are parameter matrices $W_i^Q \in \mathbb{R}^{d\times d_k}$, $W_i^K \in \mathbb{R}^{d\times d_k}$, $W_i^V \in \mathbb{R}^{d\times d_v}$, and $W^O \in \mathbb{R}^{hd_v\times d}$, and Concat$(\cdot)$ denotes column-wise concatenation of matrices. The inputs X, Y are based on block outputs within the encoder-decoder structure of the Transformer architecture, presented in more detail in Section 1.2. For self-attention, it holds that equivalent inputs $X = Y$ representing the previous encoder/decoder block output are passed, whereas for cross-attention, which connects the encoder and decoder in the Transformer architecture, X forms the output of the last encoder block and Y forms the output of the previous decoder block (Soltius 2023). The structure of Multi-Head Attention is visualized in Figure 2 (Soltius 2023).

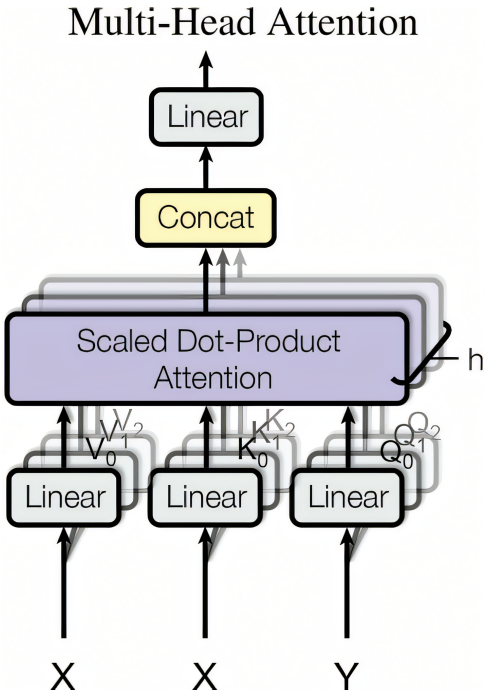


Figure 2: Visualization of Multi-Head Attention (from Soltius 2023)

The discussed components of the attention mechanism form the basis for the implementation of the Transformer model architecture.

1.2 Transformers

The Transformer Architecture is unique in the sense that it relies entirely on self-attention for the computation of its input and output representations (Vaswani et al. 2017, Section 3). As a sequence transduction model, the Transformer comprises an encoder-decoder structure, where the encoder maps a tokenized input sequence $\boldsymbol{x} = (x_1, \ldots, x_n)$ to a sequence of continuous embeddings $\boldsymbol{z} = (z_1, \ldots, z_n)$, and the decoder maps $\boldsymbol{z}$ to an output sequence $\boldsymbol{y} = (y_1, \ldots, y_m)$, sequentially. The Transformer implements the encoder as well as the decoder using stacked self-attention and *position-wise feed-forward networks (FFN)* (Vaswani et al. 2017, Section 3.3). The FFN is a layer architecture comprising two learned linear transformations separated by a ReLU activation, see (Vaswani et al. 2017, Equation 2), defined as follows:

$$\mathrm{FFN}_{W_1,W_2,b_1,b_2}(x) := \max(0, xW_1 + b_1)W_2 + b_2 \in \mathbb{R}^{n\times d},$$

where $x \in \mathbb{R}^{n\times d}$ is an input sequence, $W_1 \in \mathbb{R}^{d\times d_{\mathrm{FFN}}}$ and $b_1 \in \mathbb{R}^{n\times d_{\mathrm{FFN}}}$ are the weight and bias, respectively, for the first linear transformation, and $W_2 \in \mathbb{R}^{d_{\mathrm{FFN}}\times d}$ and $b_2 \in \mathbb{R}^{n\times d}$ are the weight and bias, respectively, for the second linear transformation.

Additionally, d-dimensional *positional encodings* are added to the otherwise position-agnostic embeddings in order to inject information regarding the order of tokens in the sequence. Both learned and fixed positional encodings are possible implementations and have been shown to yield similar results (Vaswani et al. 2017, Section 3.5). The positional encoding method used by the original authors of the Transformer architecture comprises alternating sine and cosine functions, see (Vaswani et al. 2017, Section 3.5), according to the following definitions:

$$\mathrm{PE}(\mathrm{pos}, 2i) = \sin\left(\mathrm{pos}/10000^{\frac{2i}{d}}\right) \in [-1, 1],$$
$$\mathrm{PE}(\mathrm{pos}, 2i+1) = \cos\left(\mathrm{pos}/10000^{\frac{2i}{d}}\right) \in [-1, 1],$$

where pos represents the positional index of a token and $2i$ or $2i+1$ represent the index along the embedding dimension. Both LLaMA and LLaMA-2 implement *rotary position embedding (RoPE)*, a method of positional encoding based on rotation matrices (Su et al. 2021). Instead of adding a term to an embedding, RoPE prepends a matrix factor $\boldsymbol{R}^d_{\Theta,m} \in \mathbb{R}^{d\times d}$ to the learned attention projections in order to inject information regarding the relative position of a token within a given sequence. Here,

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}$$

is in the form of a rotation matrix where $\Theta = \{\theta_i = 10000^{-2(i-1)/d}, \quad i \in [1, 2, \ldots, d/2]\}$ and m is the positional index of a token. This method has been shown to exhibit numerous valuable properties, such as flexibility regarding sequence length and long-term inter-token decay, i.e., a decreasing inner product of query and key vectors as the absolute difference in their positional indices increases (Su et al. 2021, Section 3.4.3).

The Transformer Architecture is displayed in Figure 3 (Vaswani et al. 2017, Figure 1). Within this architecture, the dimension of the feed-forward layers d_{FFN} is typically $4d$. For reference, the pre-trained model employed by the original authors of the Transformer Architecture possesses dimensions of $d = 512$, $d_k = 64$, $d_v = 64$, $d_{\text{FFN}} = 2048$, as well as $h = 8$ attention heads and encoder/decoder stacks of size $N = 6$.

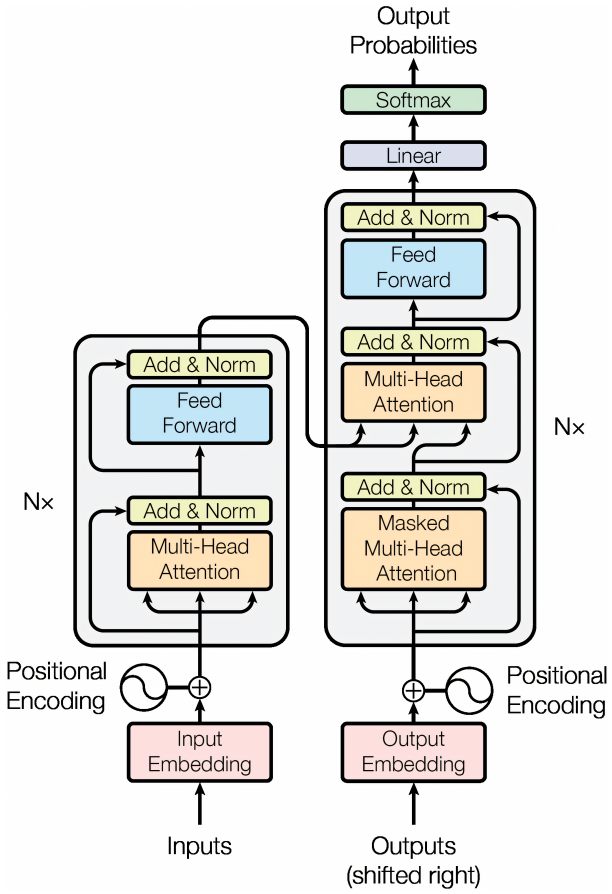


Figure 3: Visualization of the Transformer Architecture (from Vaswani et al. 2017)

In practice, encoder-decoder, encoder-only, and decoder-only models are all put to use in different situations. Encoder-decoder models, such as those within the Marian framework (Junczys-Dowmunt et al. 2018), aim to generate text that inherently contrasts the input text, like machine translation. Encoder-only models, such as BERT, focus on providing contextual word embeddings, which commonly play a large role for downstream classification tasks. Decoder-only models, such as the various GPT models, excel at generative language tasks. We focus on decoder-only models for our work.

1.3 Training

Training LLMs from scratch is generally performed *auto-regressively* and demands extraordinary compute and memory capabilities. The auto-regressive training of LLMs is an unsupervised learning technique which processes tokenized sequences of text extracted from large corpora by iteratively predicting the token that follows the sequence. In addition, many LLMs are trained with a subsequent *reinforcement learning* step in order to align models with human preferences and/or reduce unwanted bias and implicit toxicity, which can come in the form of prejudices such as racism and sexism. Reinforcement learning is an online machine learning technique which trains an agent to form an optimal decision-making policy by interacting with a defined environment, receiving feedback according to a reward function, and adjusting its behavior in a way that maximizes cumulative rewards. In the context of NLP, *reinforcement learning with human feedback (RLHF)* is generally implemented, a type of reinforcement learning in which human evaluation of model responses constitutes the reward. To this end, a reward model is trained on a dataset containing human-annotated rankings of model-generated responses to various prompts, which is then used to provide feedback to the LLM acting as an agent in order to train a reinforcement learning policy aligned with human preference.

Due to the substantial number of parameters in an LLM (commonly in the billions) and number of tokens processed during training (commonly in the trillions), training from scratch historically leads to CO_2 emissions in the 10s or even 100s of tons (Luccioni, Viguier, and Ligozat 2022) and costs estimated to start at $10 million (Miller 2023). As a result, this is generally not a feasible option for organizations. Luckily, a variety of open-source pre-trained LLMs, also called *foundation models*, are available on platforms such as Hugging Face (Hugging Face 2016). This provides users with the opportunity to use these models for inference, i.e., generation of model responses based on passed prompts, or tailor them to more specific needs through a process called **Finetuning**.

1.4 Finetuning

Finetuning is a technique within the field of *transfer learning* which involves adapting a pre-trained neural network for a particular learning task by training it on a smaller, often domain-specific dataset. In terms of model parameters, finetuning equates to altering a pre-trained model weight matrix W_0 by some delta matrix ΔW in order to minimize loss on the defined learning task:

$$W_0 + \Delta W.$$

Full Finetuning involves updating all elements of the original weight matrix through conventional back-propagation. In this case, the original weight matrix in its entirety is iteratively adjusted by gradient updates:

$$W_{i+1} = W_i + \Delta W_i.$$

Despite the reduced dataset size compared to that when training from scratch, due to the substantial number of parameters in the large language model architecture, full finetuning remains very costly and may be over budget for most organizations. This is due, in particular, to the extreme VRAM consumption (over 100GB even for smaller models) required for back-propagation (Hu et al. 2021, Section 4.2).

Parameter-Efficient Finetuning (PEFT) aims to attain comparable performance to full finetuning while using fewer computational resources. In particular, the memory overhead which must be managed by the GPU as well as the storage of weight matrices is dramatically reduced. General PEFT approaches involve finetuning only an (external) subset of weights, while freezing most of the original weights. In this case, the original weight matrix is iteratively adjusted by gradient updates that affect only a subset of its elements:

$$W_{i+1} = W_0 + \Delta W(S),$$

where S denotes the set of parameters that constitute the contents of the structured matrix for the external weight update $\Delta W(S)$.

As a result, saving the finetuned model or checkpoints thereof only requires saving the smaller subset of weights, since this subset can be added to the pre-trained weight matrix during inference. In addition, the GPU memory requirement is reduced, since gradients and gradient updates only need to be computed for non-frozen weights. These improvements make LLM finetuning a more feasible endeavor for all organizations. The resources we consumed while systematically finetuning models of various parameter count are listed in Table 1. The average duration is calculated from a set of systematically finetuned models, detailed in Section 4. The effective costs result from the hourly renting rates on runpod.io at the time of training. Carbon emission estimates were conducted using the Machine Learning Impact calculator presented in (Lacoste et al. 2019).

Parameter Count	GPU	Avg. Duration	Effective Costs	CO_2 Emissions
7b	RTX A4500	7:51:06.36	\$2.83	1.12 kg
13b	RTX A4500	13:33:40.80	\$4.88	1.89 kg
70b	A100	1 day, 6:55:46.06	\$61.55	4.73 kg

Table 1: Resource consumption for various finetuned models

Initial implementations of PEFT approaches involve freezing all pre-trained model weights and training only a task-specific finetuning head which is appended to the model (Houlsby et al. 2019). Since the finetuning head consists purely of additional layer(s) positioned at the back of the adapted model, this approach fails to learn through adjustments to internal model representations, instead providing access only to the output embedding. An extension to these implementations involves training multiple intermediate layers, called *adapter layers*, injected throughout the pre-trained model architecture. This approach yields promising results regarding model performance, though it leads to increased

latency during inference and lowered computational efficiency in general due to the significant increase in total model parameters (Houlsby et al. 2019, Section 3.2). Yet another advancement to PEFT approaches, though specific to language models, called *Prefix-Tuning*, developed by researchers at Stanford, focuses on the optimization of input vectors fed into a pre-trained model instead of adjusting the model architecture through conventional parameter-tuning. More specifically, prefix-tuning prepends supplementary task-specific context to each input vector, which is optimized through a training procedure in which all pre-trained model weights remain frozen, in order to steer the model to desirable outputs (Li and Liang 2021). Despite demonstrating comparable performance to adapter-tuning methods, prefix-tuning is limited in its capabilities by its sole focus on input embeddings. We focus on more recent implementations of PEFT approaches that aim to alleviate the discussed constraints and offer additional benefits for finetuning, namely **LoRA** and **QLoRA**, two state-of-the-art techniques which take advantage of various properties to reduce compute.

1.4.1 LoRA

Low-Rank Adaptation (LoRA) of LLMs, developed by Hu et al. 2021, proposes a way to decompose the gradient update matrix into lower-rank matrix factors named *adapters*, thereby reducing the memory reserved for back-propagation. This concept hinges on the discovery that the parameters of LLMs reside in a low intrinsic dimension (Aghajanyan, Zettlemoyer, and Gupta 2020), thus their updates ΔW can be approximated with low-rank matrix decompositions (Hu et al. 2021, Section 1). Adapters are a concept originally proposed in the form of an additional set of lower-dimensional trainable layers positioned between the original layers of the pre-trained model (Houlsby et al. 2019). LoRA avoids the inference latency associated with adding these additional layers by merging the learned weights with the pre-trained weights during inference (Hu et al. 2021, Section 6).

At its core, LoRA capitalizes on the properties of the *Singular Value Decomposition (SVD)*. The SVD factorizes a matrix $M \in \mathbb{C}^{m\times n}$ into matrix factors $U \in \mathbb{C}^{m\times m}$, $\Sigma \in \mathbb{R}^{m\times n}_{\geq 0}$, and $V^* \in \mathbb{C}^{n\times n}$, containing left-singular vectors, singular values (along the diagonal), and right-singular vectors of M, respectively, where U, V are orthonormal matrices (Golub and Van Loan 2013, Chapter 2.4). This decomposition allows for a method called *truncation*, in which the matrix factors are condensed into the forms $U_t \in \mathbb{C}^{m\times t}$, $\Sigma_t \in \mathbb{R}^{t\times t}_{>0}$, and $V_t^* \in \mathbb{C}^{t\times n}$, where the t largest (non-zero) singular values and corresponding singular vectors are used. The truncated SVD provides an optimal low-rank approximation of the original matrix M and thereby provides a conceptual foundation for the low-rank approximations of adapters respective to pre-trained weight matrices. In the case of LoRA, a pre-trained weight matrix $W \in \mathbb{R}^{d\times d}$ designated to be adapted is not decomposed and truncated, but is instead a term to which the product BA of the low-rank matrix factors $A \in \mathbb{R}^{r\times d}$ and $B \in \mathbb{R}^{d\times r}$ is added, where $r \ll d$ is the reduced rank, A is randomly initialized, and B is zero initialized (Hu et al. 2021, Section 4.1). B is chosen to be zero-initialized in order to ensure that the weight update BA is $\mathbf{0}$

initially, i.e., finetuning commences with total parameters equivalent to the pre-trained model weights. The discussed LoRA principles are depicted in Figure 4 (Hu et al. 2021, Figure 1). In this figure, A is represented by a normal distribution $\mathcal{N}(0, \sigma^2)$ with mean 0 and standard deviation σ. However, we employ a different method for the random initialization of A based on a zero-centered uniform distribution $\mathcal{U}(-a, a)$ on the interval $[-a, a]$, detailed in Section 2.3.

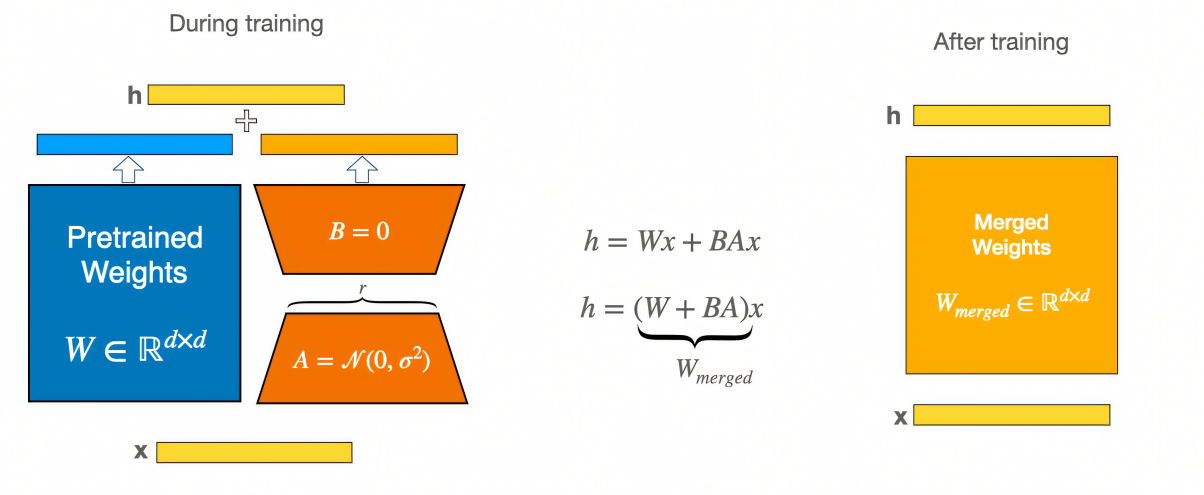


Figure 4: Visualization of LoRA Adapters (from Hu et al. 2021)

Due to the reduced rank of the matrix factors A and B, the number of trainable parameters per weight matrix W is lowered from d^2 to $2rd$, which is significantly lower for $r \ll d$. LoRA aims to leverage the reduced dimensionality of the adapter matrices in a parameter-efficient approach to finetuning. Though this approach is agnostic to training objective, its implementation can be exemplified for the language modeling problem (Hu et al. 2021, Section 2). Given a pre-trained auto-regressive language model $P_\Phi(y|x)$ parameterized by parameters Φ and a downstream task represented by a training dataset of context-target pairs $\mathcal{Z} = \{(x_i, y_i)\}_{i=1,\ldots,N}$, where x_i and y_i are tokenized sequences, full finetuning initializes the model to pre-trained weights Φ_0 and subsequently updates the weights to $\Phi_0 + \Delta\Phi$ by repeatedly following the gradient to maximize the conditional language modeling objective, defined as follows:

$$\text{Obj}_{\text{Full}} = \max_{\Phi} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y_t|} \log P_\Phi(y_t|x, y_{<t}), \tag{1.2}$$

where $|y_t|$ is the length, i.e., number of tokens, of the t-th tokenized output sequence y_t. The equation for the maximization of the conditional language modeling objective stems from the application of *cross-entropy loss*. Cross-entropy loss is a loss function designed for learning tasks in which the output comes in the form of a probability distribution which denotes the likelihood of each label within a set of possible labels, defined as:

$$C(p) := -y_{i,j} \log x_{i,j} = -\log P(y_{i,j}) \in [0, \infty), \tag{1.3}$$

where $p = (x_{i,j}, y_{i,j})$ is an output pair in which $x_{i,j}$ is the model output and $y_{i,j}$ is the true value. Since the true value $y_{i,j}$ is a one-hot encoded vector corresponding to the index of the correct label, the cross-entropy error can alternatively be described in terms of the probability assigned to the correct label by the model output $x_{i,j}$, also shown in (1.3). Finally, the conditional language modeling objective, effectively cross-entropy loss on next-token prediction, involves minimizing the cross-entropy loss, which can be equivalently described as maximizing its additive inverse, explaining the formulation of (1.2).

LoRA proposes an approach that encodes the parameter increment $\Delta\Phi = \Delta\Phi(\Theta)$ by a small subset of parameters Θ with $|\Theta| \ll |\Phi_0|$, where $|\Theta|$ is the cardinality of the set of trainable parameters and $|\Phi|$ is the cardinality of the set of pre-trained parameters. This modifies the original optimization problem in (1.2) to one over Θ:

$$\text{Obj}_{\text{LoRA}} = \max_{\Theta} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y_t|} \log P_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x, y_{<t}).$$

Given a pre-defined set of weight matrices to adapt, referred to as *modules*, one can calculate the percentage reduction in trainable parameters induced by LoRA finetuning by applying the following calculation:

$$\begin{aligned} f_{\text{trainable}} &= \frac{\text{no. trainable params}}{\text{no. pre-trained params}} = \frac{|\Theta|}{|\Phi|} = 2\frac{\hat{L}_{\text{LoRA}}}{\hat{L}_{\text{total}}}\frac{r}{d} \in [0,1], \\ |\Theta| &= 2 \times \hat{L}_{\text{LoRA}} \times d \times r, \\ |\Phi| &= \hat{L}_{\text{total}} \times d^2, \end{aligned}$$

where $\hat{L}_{\text{LoRA}}$, $\hat{L}_{\text{total}}$ are the number of LoRA/total modules, respectively, d is the pre-trained model dimension, and r is the reduced rank. As an example, LoRA finetuning on LLaMA-2-7b, for which $\hat{L}_{\text{total}} = 7$ and $d = 4096$ hold, using $r = 4$ and $\hat{L}_{\text{LoRA}} = 2$, the fraction of trainable parameters is as low as $f_{\text{trainable}} = \frac{1}{1792} \approx 0.056\%$.

LoRA has been shown to exhibit multiple advantages over full finetuning (Hu et al. 2021, Section 5.1). Aside from the benefits regarding memory allocation during back-propagation, LoRA saves finetuned adapters with much lower storage requirements than fully-finetuned weights. Thus, deploying multiple instances of finetuned models only requires a single instance of the pre-trained model weights alongside the respective fine-tuned adapters. LoRA also produces no additional latency during inference. Regular adapter layers have to be processed sequentially, which bottlenecks the parallelism neural networks rely on for low latency. LoRA instead only adds a term to the weights in the forward pass, which can then be calculated like normal. Furthermore, LoRA doesn't lead to a reduction in input sequence length. Since LoRA categorically affects weight update matrices, it has no bearing on the input, so the sequence length can remain the same. The benefits of LoRA were further observed practically (Hu et al. 2021, Section 4.2):

finetuning GPT-3 175b with Adam lead to a drop in VRAM consumption from 1.2TB to 350GB, reduced checkpoint size from 350GB to 35MB, and increased training speed by 25%. Finally, empirical trials showed that adapting only the attention weights, particularly W_q (query) and W_v (value), with a reduced rank $r = 4$ yields near-optimal performance while significantly reducing compute.

1.4.2 QLoRA

Quantized Low-Rank Adaptation (QLoRA), developed by Dettmers, Pagnoni, et al. 2023, is an extension of LoRA, which further reduces the memory overhead for finetuning. This is achieved by three core innovations: 4-bit NormalFloat (NF4), Double Quantization, and Paged Optimizers (Dettmers, Pagnoni, et al. 2023, Section 1).

In order to understand the motivation for QLoRA, as well as the computational benefit it provides, an understanding of the relevant numeric datatypes, including those proposed by the IEEE 754 standard ("IEEE Standard for Binary Floating-Point Arithmetic" 1985), must be established. To this end, an overview of these datatypes is presented in Table 2. The table provides a comparison of the number of bits reserved for each of portion of the floating point format. The green cells highlight the fact that the 16-bit Brain Float (BF16) shares the same amount of exponent bits as a regular[3] 32-bit float (FP32), instead of the 5 exponent bits in a regular 16-bit float (FP16). By extension, the number of precision bits in a 16-bit Brain Float amounts only to 7 instead of 10. As such, the Brain Float datatype is equipped to handle a large range of values, reducing susceptibility to overflow, and allows for more efficient computation, coming at the cost of reduced precision.

Format	No. of bits			Range of values	
	sign	exp	mantissa	min.	max.
FP32	1	8	23	$2^{-126} \approx 1.18 \times 10^{-38}$	$(1 - 2^{-24})2^{128} \approx 3.4 \times 10^{38}$
BF16	1	8	7	$2^{-126} \approx 1.18 \times 10^{-38}$	$(1 - 2^{-8})2^{128} \approx 3.39 \times 10^{38}$
FP16	1	5	10	$2^{-14} \approx 6.1 \times 10^{-5}$	$(1 - 2^{-11})2^{16} = 65504$

Table 2: Overview of relevant computer number formats

The next concept we discuss is *quantization*. In general, quantization refers to the concept of re-assigning certain values of a larger, i.e., more memory-consuming, datatype to values of a smaller datatype. Values are re-assigned according to the *quantization bin* they fall into when normalized. Given the values $w_1, \ldots, w_n$, normalization is performed by dividing each value by the *quantization constant*, which is the *absmax* of the values $m = \max_i |w_i|$. Subsequently, re-assignment according to respective quantization bin is assessed by mapping each normalized value w_i/m to the nearest value of the quantization datatype q_j and storing the index $c_i = \text{argmin}_j |q_j - w_i/m|$ of the nearest value within the quantization datatype. The information that is lost due to quantized values deviating

[3]Here, a "regular" k-bit float refers to the k-bit IEEE 754 floating point format.

from their respective original values is referred to as *quantization error* and is generally sought to be minimized. An example of quantization and resulting quantization error is presented for an example 2-bit datatype with arbitrary input values in Listing 2. From this example, the importance of matching the quantization datatype distribution to the input distribution for minimizing quantization error becomes clear, as the quantization datatype in the example is an asymmetric datatype skewed toward high values, making it unsuitable for typical distributions, such as uniform and normal distributions. The inverse operation to quantization, *dequantization*, is applied using the quantization constant calculated during quantization, as a result of which the quantization constant must be stored.

Quantization datatype values: $\{-1, 0.5, 0.7, 1\}$
Input values: $\{10, -1, 3, 8, -9\}$

1. Normalize input with absmax = 10: $\{10, -1, 3, 8, -9\} \rightarrow \{1, -0.1, 0.3, 0.8, -0.9\}$
2. Assign each input to nearest value in quantization datatype:
   ↪ $\{1, -0.1, 0.3, 0.8, -0.9\} \rightarrow \{1, 0.5, 0.5, 0.7, -1\}$

Quantization error calculated by the absolute difference between normalized input and
   ↪ quantization result: $\{|1-1|, |-0.1-0.5|, |0.3-0.5|, |0.8-0.7|, |-0.9-(-1)|\} =$
   ↪ $\{0, 0.6, 0.2, 0.1, 0.1\}$
Dequantization is conducted by denormalizing with absmax of original input values:
   ↪ $\{1, 0.5, 0.5, 0.7, -1\} \rightarrow \{10, 5, 5, 7, -10\}$

Listing 2: Quantization on example 2-bit datatype

A conceptual extension of quantization lies in *block-wise quantization*, which divides the input into smaller, independently quantized blocks (Dettmers, Lewis, et al. 2021). This makes quantization more resistant to outliers, since the increased quantization error associated with outliers is restricted to the block containing the outlier, instead of affecting all values.

The 4-bit NormalFloat datatype is an information-theoretically optimal datatype for normally distributed data, a status achieved by ensuring an even distribution of values across all quantization bins after quantization (Dettmers, Pagnoni, et al. 2023, Section 3, 4-bit NormalFloat Quantization). This makes it suitable for the quantization of LLMs, since pre-trained model weights are generally distributed along a zero-centered normal distribution (Dettmers, Pagnoni, et al. 2023, Appendix F). This is due to pre-trained model weights being initialized according to such a distribution, with training yielding no significant influence on the statistical characteristics of the initialized state. Using this insight, the NF4 values q_j are calculated based on the *quantiles* of the normal distribution. Quantiles are values which partition a distribution or finite set of values into subsets of maximally equal likelihood or occurrence, respectively. The construction of NF4 values, see (Yoshida 2023), follows the process detailed in Enumeration 1. The choice of 4-bit precision for the NF4 datatype is motivated by systematic experiments across multiple

model parameter counts loaded in varying levels of precision, which demonstrate that 4-bit precision provides an almost universally optimal trade-off between total model bits and zero-shot accuracy (Dettmers and Zettlemoyer 2022). These findings ignore the additional improvement to the number of *Floating Point Operations Per Second (FLOPS)* able to be executed by the GPU hardware during training, which shares an inversely proportional relationship to the bits in the quantization datatype $n_{FLOPS} \propto \frac{1}{k}$, where n_{FLOPS} is the number of FLOPS executed during training and k is the number of bits in the qunatization datatype. Thus, a reduction in precision to the quantization datatype leads directly to reduced training duration proportional to the relative reduction.

1. Set $\delta = \frac{1}{2}\left(\frac{1}{32} + \frac{1}{30}\right)$
2. Compute 8 evenly spaced probability values $p_1, \ldots, p_8$ such that $p_1 = \delta$ and $p_8 = \frac{1}{2}$
3. Find their pre-images under the Gaussian CDF, $\Phi : \tilde{q}_i = \Phi^{-1}(p_i)$ for $i = 1, \ldots, 8$
4. Compute 9 evenly spaced probability values $r_8, \ldots, r_{16}$ such that $r_8 = \frac{1}{2}$ and $r_{16} = 1 - \delta$
5. Set $\tilde{q}_i = \Phi^{-1}(r_i)$ for $i = 9, \ldots, 16$ (r_8 is ignored since $\tilde{q}_8$ is already set to 0)
6. Normalize the $\tilde{q}$s to the range [-1, 1] to get the final code: $q_i = \frac{\tilde{q}_i}{\max_i |\tilde{q}_i|}$

Enumeration 1: NF4 Construction

The constructed NF4 code defines a set of values in the range $[-1, 1]$ with $q_1 = -1$, $q_8 = 0$, and $q_{16} = 1$. The largest unnormalized quantization value $\tilde{q}_i$ is equivalent to $\Phi^{-1}(1 - \delta) \approx 1.848$, such that the final quantization values q are quantiles of the normal distribution $\mathcal{N}\left(0, \frac{1}{\Phi^{-1}(1-\delta)}\right)$. The final quantization values are visualized over the corresponding normal distribution in Figure 5.

Double Quantization, as the name suggests, involves quantizing the quantization constants computed by an initial quantization in order to achieve a further reduction in memory footprint per parameter (Dettmers, Pagnoni, et al. 2023, Section 3, Double Quantization). This is a valuable innovation, because while quantization allows for more efficient storage and computation, there is still a non-negligible memory overhead associated with storing absmax quantization constants. An example from the authors of QLoRA quantifies this overhead, illustrating that 4-bit quantization with 32-bit constants and a blocksize of 64 adds $\frac{32}{64} = 0.5$ bits per parameter (one 32-bit quantization constant for every 64 parameters). The essence of double quantization lies in reducing this overhead by quantizing the computed quantization constants. Discussion of the example is continued by the QLoRA authors in order to quantify the reduction of the memory overhead, demonstrating that a second quantization with 8-bit constants and a blocksize of 256 yields a memory reduction from 0.5 bits per parameter to $\frac{8}{64} + \frac{32}{(64 \cdot 256)} \approx 0.127$ bits per parameter. Thus, in total, the memory overhead of 4.5 bits per parameter is reduced to approx. 4.127 per parameter, leading to an overall memory reduction of approx. 8.3%.

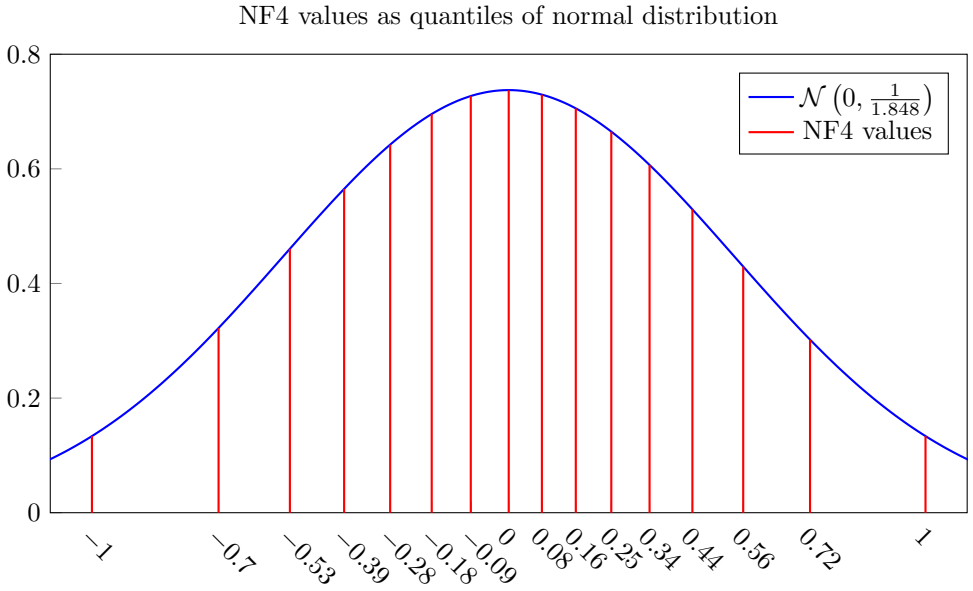


Figure 5: Visualization of NF4 values over corresponding normal distribution

Finally, Paged Optimizers, which make use of NVIDIA unified memory, allow for page-to-page transfers between the CPU and GPU in cases where the GPU runs out of memory (Dettmers, Pagnoni, et al. 2023, Section 3, Paged Optimizers). They are implemented using a lazy offloading algorithm which automatically transfers the optimizer state memory in cases of memory spikes that exceed GPU memory. These memory spikes generally occur in cases where a large mini-batch, i.e., a batch containing higher than usual sequence lengths, requires more GPU memory than is available to maintain the corresponding optimizer state. In this case, a designated paging engine evicts the allocated memory for the optimizer state to the CPU RAM and pages it back into GPU memory when it is required for optimizer updates.

In conjunction with each other as well as LoRA, these innovations characterize the implementation of QLoRA, see (Dettmers, Pagnoni, et al. 2023, Equation 5, QLoRA), defined for an arbitrary linear layer with one LoRA adapter as follows:

$$Y^{\text{BF16}} = X^{\text{BF16}}\text{doubleDq}(c_1^{\text{FP32}}, c_2^{\text{FP8}}, W^{\text{NF4}}) + X^{\text{BF16}}B^{\text{BF16}}A^{\text{BF16}} \in \mathbb{R}^{n\times d}, \tag{1.4a}$$

$$\text{doubleDq}(c_1^{\text{FP32}}, c_2^{\text{FP8}}, W^{\text{NF4}}) := \text{dq}(\text{dq}(c_1^{\text{FP32}}, c_2^{k\text{-bit}}), W^{\text{NF4}}) = W^{\text{BF16}} \in \mathbb{R}^{d\times d}, \tag{1.4b}$$

where $Y \in \mathbb{R}^{n\times d}$ is the output, $X \in \mathbb{R}^{n\times d}$ is the input, $c_1 \in \mathbb{R}$, $c_2 \in \mathbb{R}$ are the quantization constants, $W \in \mathbb{R}^{d\times d}$ is the quantized weight matrix, $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times d}$ are the LoRA adapters, and $\text{dq}(\cdot)$, $\text{doubleDq}(\cdot)$ represent single and double quantization, respectively. The variable superscripts declare the datatype of the variable. From (1.4a), it follows that QLoRA implements a storage datatype (NF4) for the model architecture and a computation datatype (BF16) for forward and backward passes as well as computing adapter gradients. The conceptual composition of QLoRA in comparison to previously discussed finetuning methods is visualized in Figure 6 (Dettmers, Pagnoni, et al. 2023,

Figure 1).

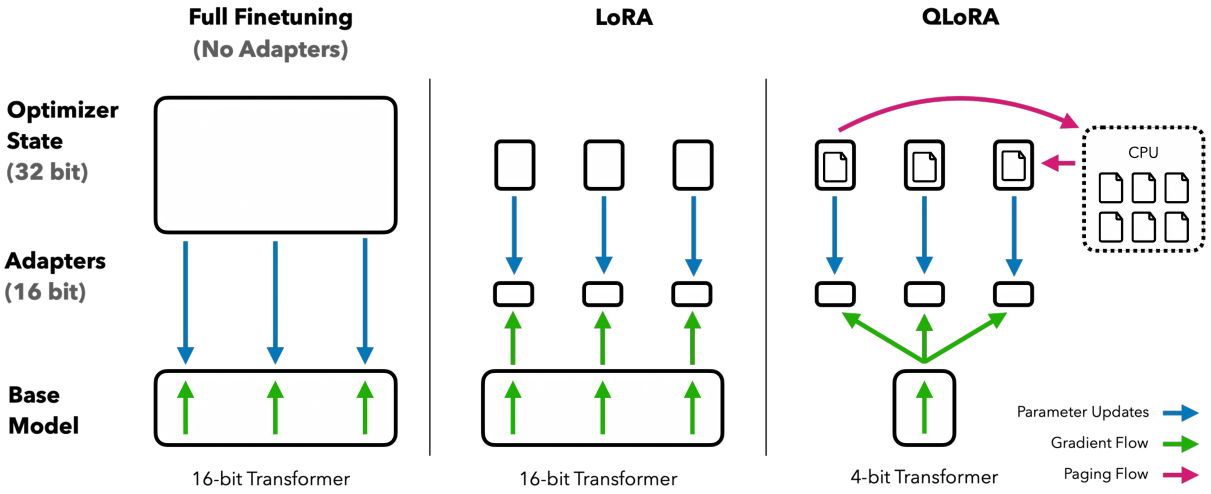


Figure 6: QLoRA in comparison to LoRA and Full Finetuning (from Dettmers, Pagnoni, et al. 2023)

QLoRA benefits not only from the advantages of LoRA, but also from reduced memory consumption without loss in performance, while additionally loosening the strict threshold on GPU memory (Dettmers, Pagnoni, et al. 2023, Section 2). Benefits regarding memory requirements were also observed for QLoRA practically (Dettmers, Pagnoni, et al. 2023, Section 4): finetuning a 65B parameter model resulted in a reduction in average memory consumption from over 780GB to under 48GB without degradation in performance (regarding inference latency and response accuracy) when compared to 16-bit full finetuning. This significant memory reduction enables finetuning to be performed on 48GB graphics cards such as the RTX A6000 or A40, or alternatively on two consumer-grade 24GB graphics cards, such as the RTX 3090 or RTX 4090. Finally, empirical trials showed that a blocksize of 64 for 4-bit NormalFloat quantization on W, a blocksize of 256 for 8-bit float subsequent quantization (on quantization constants), and 16-bit Brain Float as the computation datatype (for the weight gradients of LoRA parameters) yields performance which matches 16-bit full finetuning and 16-bit LoRA finetuning while significantly reducing memory consumption.

1.5 Evaluation

The goal of evaluation is to assess the quality of model inferences respective to the model's domain and learning task. To this end, established metrics are used define distance/similarity between output and reference tokenized string sequences (Yan 2023).

For example, Bilingual Evaluation Understudy (BLEU), a precision-based metric, measures the relative occurrence of *n-grams* in a generated output $\hat{S}$ also appearing in the corresponding reference sequence S. For a given tokenized sequence $y = y_1 y_2 \cdots y_K$ and

token length $n \in \mathbb{N}$, the set of n-grams G_n is defined as:

$$G_n(y) := \{y_1 \cdots y_n, y_2 \cdots y_{n+1}, \cdots, y_{K-n+1} \cdots y_K\}.$$

Based on this definition, the measure of precision implemented by BLEU, see (Papineni et al. 2002, Section 2.1.1), for a given token length n is defined as follows:

$$\text{precision}_n(\hat{S}, S) := \frac{\sum_{g \in G_n(\hat{S})} \min\left(C\left(g, \hat{S}\right), C\left(g, S\right)\right)}{\sum_{g \in G_n(\hat{S})} C\left(g, \hat{S}\right)} \in [0, 1],$$

where $C(g, S)$ represents the number of times an n-gram g occurs in some sequence S. Another component integrated into BLEU is a brevity penalty, which penalizes sentences of length $r \in \mathbb{N}$ below the reference sequence length $c \in \mathbb{N}$, defined as:

$$\text{BP}(\hat{S}, S) := \begin{cases} 1 & \text{if } r \leq c, \\ e^{-\left(\frac{r}{c-1}\right)} & \text{otherwise.} \end{cases}$$

The implementation of a brevity penalty is motivated by the goal to avoid bias toward short sentences that would otherwise arise as a result of the high relative occurrence of select words in short sentences. The score produced by BLEU, see (Papineni et al. 2002, Section 2.3), is calculated by taking the weighted geometric mean of the precision_n values calculated for all positive integer token lengths and applying the brevity penalty:

$$\text{BLEU}_w(\hat{S}, S) := \text{BP}(\hat{S}, S) \cdot \exp\left(\sum_{n=1}^{\infty} w_n \ln \text{precision}_n(\hat{S}, S)\right) \in [0, 1],$$

where $w_n \in w$ are the weights for each token length n. Weights are commonly set to $\frac{1}{n}$.

Other methods, such as BERTScore (T. Zhang et al. 2019), involve measuring the semantic similarity between sequences by comparing their token embeddings. We focus on and make use BLEU for performance evaluation in our work.

2 Experimental Setup

Our experimental analysis is performed in cooperation with classix, a software development company newly founded in 1994 through a change of name from Brenner Datensysteme GmbH, founded in 1983. Since its inception, classix has continually developed an object-oriented framework for building and maintaining business applications, with which multiple large-scale enterprise resource planning (ERP) systems have been produced. The development of this framework reflects a comprehensive approach to data modelling based on object-oriented components, which facilitates the production of individual software solutions through abstraction. The abstract enterprise data model at the heart of this concept is embodied by the CyberEnterprise architecture, which remains consistent regarding its original structure in the modern era, despite constant improvement regarding the functionality of software applications based on this model. Using the CyberEnterprise architecture as a foundation, a domain-specific programming language called InstantView and corresponding framework comprising various business applications called AppsWarehouse was developed. These systems facilitate the production of highly individualized business solutions in short amounts of time.

We detail the preparation of training data and method for evaluation of model performance for our finetuning implementations, as well as the resources that are currently available for custom implementations of finetuning in this section. More specifically, resources for foundation models, finetuning parameters, and the applied training algorithm are presented, with an added focus on software implementation.

2.1 Data

The raw data used as a foundation for finetuning training data stems from the documentation of the classix software infrastructure, which exists both in the form of a technical documentation on `https://instantview.org`, as well as a general, more practical documentation on `https://appswarehouse.de`. These documentations provide descriptions for classix classes and modules, respectively, in both English and German, in order to facilitate effective navigation and utilization of tools and systems within the classix infrastructure. For the learning tasks we propose, solely the AppsWarehouse documentation serves as a source of training data, since its contents provide more general and digestible insight into the classix infrastructure.

Since the documentation is a web documentation written in HTML, a semantic parsing of all pages was performed, which systematically retrieves relevant blocks of text into a JSON document. The format of the resulting documents for each documentation is presented in Section A of the appendix.

The blocks of text extracted from the documentation were pre-processed according to the respective requirements for auto-regressive and *Alpaca* training data formats. Pre-processing for auto-regressive finetuning involves ensuring that the text sequences of the training data are used for gradient optimization in their entirety. Pre-processing

for "Alpaca-style" finetuning involves injecting specific surrounding text into training sequences in order to produce a consistent scheme designed for instruction following in conversational dialog. This scheme, adapted by researchers at Stanford, comprises specifically-worded *instruction*, *input*, and *output* sequences, where the instruction sequence defines the prompted task, the optional input sequence defines supplementary context for the instruction, and the output sequence defines the desired model response (Taori et al. 2023). The conceptual roots for structuring sequences according to this scheme lie in *instruction-tuning*, a finetuning scheme in which training data is prepared in a way that improves the ability of language models to follow natural language instructions (S. Zhang et al. 2023). The Alpaca dataset contains 52000 sequences generated using the *self-instruct* framework, a framework for generating instruction-tuning training sequences which largely circumvents the need for human annotation (Y. Wang et al. 2022). An example training sequence from the Alpaca dataset is presented in Listing 3. Sequences that follow the instruction-tuning scheme are generally finetuned such that only on the output sequences are used for gradient optimization.

```
Below is an instruction that describes a task, paired with an input that provides
    ↪ further context. Write a response that appropriately completes the request.

### Instruction: Explain why the following fraction is equivalent to 1/4

### Input: 4/16

### Response: The fraction 4/16 is equivalent to 1/4 because both numerators and
    ↪ denominators are divisible by 4. Dividing both the top and bottom numbers by
    ↪ 4 yields the fraction 1/4.
```

Listing 3: Alpaca training sequence

2.2 Model and Parameters

Foundation models were accessed through Hugging Face, a platform which offers access to a host of models, datasets, and algorithms for various machine learning tasks. Of the provided models, versions of the LLaMA and LLaMA-2 models were used as foundation models for finetuning.

The parameters we used for finetuning are based on those that were used for finetuning the *guanaco* model family, which constitutes the basis for analysis of QLoRA by its original authors. These parameters were largely adopted and adjusted for individual learning tasks. Although LoRA makes up a direct part of the implementation of QLoRA, some of the parameters employed by the authors of QLoRA differ from those proposed by the authors of LoRA. These differences are detailed in Table 3.

Furthermore, we make our own adjustments to parameters, primarily in order to tailor them to the training datasets we use. These adjustments are detailed in Table 4.

Parameter	LoRA	QLoRA
LoRA rank	4	64
LoRA modules	q, v **or** q, k, v, o	all
LoRA dropout	0.0	0.1
LoRA alpha	128	16

Table 3: Comparison of differing finetuning parameters proposed by LoRA vs. QLoRA authors

Parameter	QLoRA	Us
eval_dataset_size	1024	128
max_eval_samples	1000	100
source_max_len	16	1024
target_max_len	512	1024

Table 4: Comparison of differing finetuning parameters QLoRA proposes vs. ones we implement

A full list of the parameters we employ (for finetuning LLaMA-2-7b, as an example) is located in Section B of the appendix.

2.3 Training algorithm

The weights of the LoRA adapters are initialized such that the first matrix factor B is zero-initialized and the second matrix factor A is initialized using a *uniform Kaiming distribution* (He et al. 2015). In general, Kaiming initialization is an initialization method that accounts for non-linear activation functions, particularly rectifiers, such as the *SwiGLU* (Shazeer 2020) activation function we make use of, by ensuring uniform variance throughout all layers (He et al. 2015, Section 2.2). Its derivation is accordingly based on analysis of the layer-wise progression of variance in both forward and back-propagation. In the case of forward propagation, the derivation involves the l-th layer's response function, defined as:

$$\boldsymbol{y}_l = \boldsymbol{W}_l \boldsymbol{x}_l + \boldsymbol{b}_l,$$

where $\boldsymbol{y}_l \in \mathbb{R}^d$ is the response, $\boldsymbol{W}_l \in \mathbb{R}^{d \times d}$ is the weight matrix, $\boldsymbol{x}_l \in \mathbb{R}^d$ is the input, and $\boldsymbol{b}_l \in \mathbb{R}^d$ is the bias. The goal is to find some $\boldsymbol{W}_l$ such that $\mathrm{Var}[\boldsymbol{y}_l] = \mathrm{Var}[\boldsymbol{y}_{l-1}]$.

Since $\boldsymbol{b}_l$ is assumed to be zero-initialized, the elements of $\boldsymbol{W}_l$ are mutually independent while sharing some distribution, the elements of $\boldsymbol{x}_l$ are assumed to also be mutually independent while sharing some distribution, and $\boldsymbol{W}_l$ and $\boldsymbol{x}_l$ are independent of each other, the response function can be represented in terms of variances on random variables:

$$\mathrm{Var}[y_l] = d\mathrm{Var}[w_l x_l], \tag{2.1}$$

where y_l, w_l, and x_l are random variables of $\boldsymbol{y}_l$, $\boldsymbol{W}_l$, and $\boldsymbol{x}_l$, respectively. The presence of the factor d in this representation is explained by the additional dimension of $\boldsymbol{W}_l = [\boldsymbol{w}_{l,1}, \ldots, \boldsymbol{w}_{l,d}]$, such that this matrix, when multiplied by $\boldsymbol{x}_l = [x_{l,1}, \ldots, x_{l,d}]^T$, effectively produces a sum of d vectors $x_{l,1}\boldsymbol{w}_{l,1} + x_{l,2}\boldsymbol{w}_{l,2} + \cdots + x_{l,d}\boldsymbol{w}_{l,d}$, where each vector is individually represented by the product of random variables $w_l x_l$. Assuming w_l has zero mean, the representation in (2.1) can be rewritten according to the variance of the product of independent variables:

$$\mathrm{Var}[y_l] = d\mathrm{Var}[w_l]\mathrm{E}[x_l^2].$$

Assuming w_{l-1} has a symmetric distribution around 0 and $b_{l-1} = 0$, y_{l-1} also has a symmetric distribution around 0 (and thereby zero mean), which, for the $ReLU$ activation function[4], leads to $\mathrm{E}[x_l^2] = \frac{1}{2}\mathrm{Var}[y_{l-1}]$. This can be explained intuitively, since $\mathrm{Var}[y_{l-1}] = \mathrm{E}[y_{l-1}^2]$ holds due to y_{l-1} having zero mean, and the necessary conclusion $\mathrm{E}[y_{l-1}^2] = 2\mathrm{E}[x_l^2]$ is explained by y_{l-1} having a symmetric distribution around 0, as well the definition for ReLU $x_l = \max(0, y_{l-1})$, as a symmetric distribution around 0 provides exactly one additional source for each value present in its square over the same distribution with no values in the negative domain. The resulting equation and its cumulative counterpart for all layers up to L are presented in the following:

$$\mathrm{Var}[y_l] = \frac{1}{2}d\mathrm{Var}[w_l]\mathrm{Var}[y_{l-1}], \tag{2.2a}$$

$$\mathrm{Var}[y_L] = \mathrm{Var}[y_1]\left(\prod_{l=2}^{L}\frac{1}{2}d\mathrm{Var}[w_l]\right). \tag{2.2b}$$

The relationship between $\mathrm{Var}[y_L]$ and $\mathrm{Var}[y_1]$ in (2.2b) demonstrates that the relevant product must take on the value of a scaler, as optimal initialization avoids exponential reduction or amplification of the magnitude of input signals. Thus, a sufficient condition for such initialization can be proposed:

$$\frac{1}{2}d\mathrm{Var}[w_l] = 1 \quad \forall l.$$

The back-propagation case is handled similarly, using layer gradients as a basis for analysis, and yields a compatible condition.

For the SwiGLU activation function used in LLaMA and LLaMA-2 with a given pre-trained model dimension d, the uniform Kaiming distribution generates a uniform distribution on the interval $[-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}}]$, resulting in a mean of 0 and a variance of $\frac{1}{3d}$. The derivation of these definitions is located in Section C of the appendix.

[4]The authors of Kaiming Initialization focus specifically on ReLU. Other activation functions can produce coefficients that differ from the $\frac{1}{2}$ found in relevant calculations. This will be based on the relationship between the expectation of a squared random variable with a symmetric distribution around 0 and a squared random variable with the respective activation function applied to the same distribution.

SwiGLU, see (Shazeer 2020, Equation 5), is based on the *Swish* (Ramachandran, Zoph, and Le 2017) and *Gated Linear Units (GLU)* (Shazeer 2020) activation functions, defined as follows:

$$\begin{aligned}\text{SwiGLU}(x, W, V, b, c, \beta) &:= \text{Swish}_\beta(xW + b) \otimes (xV + c) \in \mathbb{R}^{d_{\text{swish}}},\\ \text{Swish}_\beta(x) &:= x\sigma(\beta x) \in \mathbb{R}^{d_{\text{swish}}},\end{aligned}$$

where $x \in \mathbb{R}^{\text{sequence length}\times \text{d}}$ is an input sequence, $W \in \mathbb{R}^{d\times d_{\text{swish}}}$ and $b \in \mathbb{R}^{d_{\text{swish}}}$ are the weight and bias for the linear transformation in the Swish component, $V \in \mathbb{R}^{d\times d_{\text{swish}}}$ and $c \in \mathbb{R}^{d_{\text{swish}}}$ are the weight and bias for the linear transformation in the gate component, $\beta \in \mathbb{R}$ is a learned parameter controlling the smoothness of the function, $\sigma : \mathbb{R} \to \mathbb{R}$ is the *sigmoid* activation function (applied to each matrix element), and $\otimes$ represents the component-wise product. It benefits from properties such as smoothness, non-monotonicity, and a gating mechanism, yielding high performance relative to other established GLU variants, particularly for tasks related to language modeling (Shazeer 2020, Section 3).

The optimization algorithm used during training is *AdamW*[5]. AdamW is an algorithm similar to *Adaptive Moment Estimation (Adam)*, differing only in its implementation of weight decay. Adam is an efficient stochastic optimization algorithm requiring only first-order gradients that, using L_2 regularization, implements weight decay as an extension of the gradient calculation of the loss function (Kingma and Ba 2014):

$$\boldsymbol{g}_t \leftarrow \nabla f_t\left(\boldsymbol{\theta}_{t-1}\right) + \lambda\boldsymbol{\theta}_{t-1},$$

where $\nabla f_t\left(\boldsymbol{\theta}_{t-1}\right) \in \mathbb{R}^n$ is the gradient of the loss function with respect to the parameters $\boldsymbol{\theta}_{t-1} \in \mathbb{R}^n$ at the time step $t-1$, $\quad t \in \mathbb{N}$, and $\lambda \in \mathbb{R}$ is the regularization parameter.

AdamW proposes a correction to the way weight decay is implemented in Adam by decoupling the weight decay from the optimization steps taken with respect to the loss function (Loshchilov and Hutter 2017). In algorithmic terms, this equates to implementing weight decay as an extension of the parameter update itself, instead of the loss gradient:

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t\left(\alpha\hat{\boldsymbol{m}}_t/(\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda\boldsymbol{\theta}_{t-1}\right),$$

where $\alpha = 0.001$ and $\epsilon = 10^{-8}$ are pre-defined constants, $\eta_t \in \mathbb{R}$ is the schedule multiplier, and $\hat{\boldsymbol{m}}_t \in \mathbb{R}^n$, $\hat{\boldsymbol{v}}_t \in \mathbb{R}^n$ are the first/second moment vectors, respectively.

Finetuning was performed for 1875 iterations, also called *steps*, equivalent to approx. 30 epochs in our case[6], and, for LLaMA-2-7b with our finalized dataset (detailed in Section 3), yielded the loss curves displayed in Figure 7. The shape of these curves is characteristic for the finetuning of all models for this dataset regardless of parameter

[5] 32-bit Paged AdamW

[6] The dependency between steps and epochs arises from the number of training batches used, since a step denotes the forward and backward pass of a single batch, whereas an epoch denotes the forward and backward pass of all training samples.

count, and thus provides a generalized representation of the progression of loss during finetuning for the implemented training procedure. In general, loss is computed using cross-entropy on next-token model predictions. More specifically, given the tokenized input sequence $\boldsymbol{x} = (x_1, \ldots, x_n)$ and output sequence $\boldsymbol{y} = (y_1, \ldots, y_m)$ of an arbitrary training sample, cross entropy loss (see (1.3)) is computed sequentially over all model output predictions:

$$C_{\text{total}} = -\sum_{i=1}^{m} \log P_{\Phi}(y_i | x, y_{<i}) \in [0, \infty).$$

The evaluation loss values stem from periodic evaluation every 187 iterations (approx. 3 epochs) on a 100-sample evaluation dataset randomly extracted from the original training data. The MMLU loss values result from the default evaluation on the MMLU dataset. The MMLU dataset consists of multiple-choice questions pertaining to various fields, such as astronomy, elementary mathematics, philosophy, and many more. These questions are designed to measure a model's multitask accuracy as well as its academic and professional understanding (Hendrycks et al. 2020). For this reason, it is expected, if not hoped for, that a model's multitask accuracy decreases as it becomes more adapted to a specific domain through finetuning.

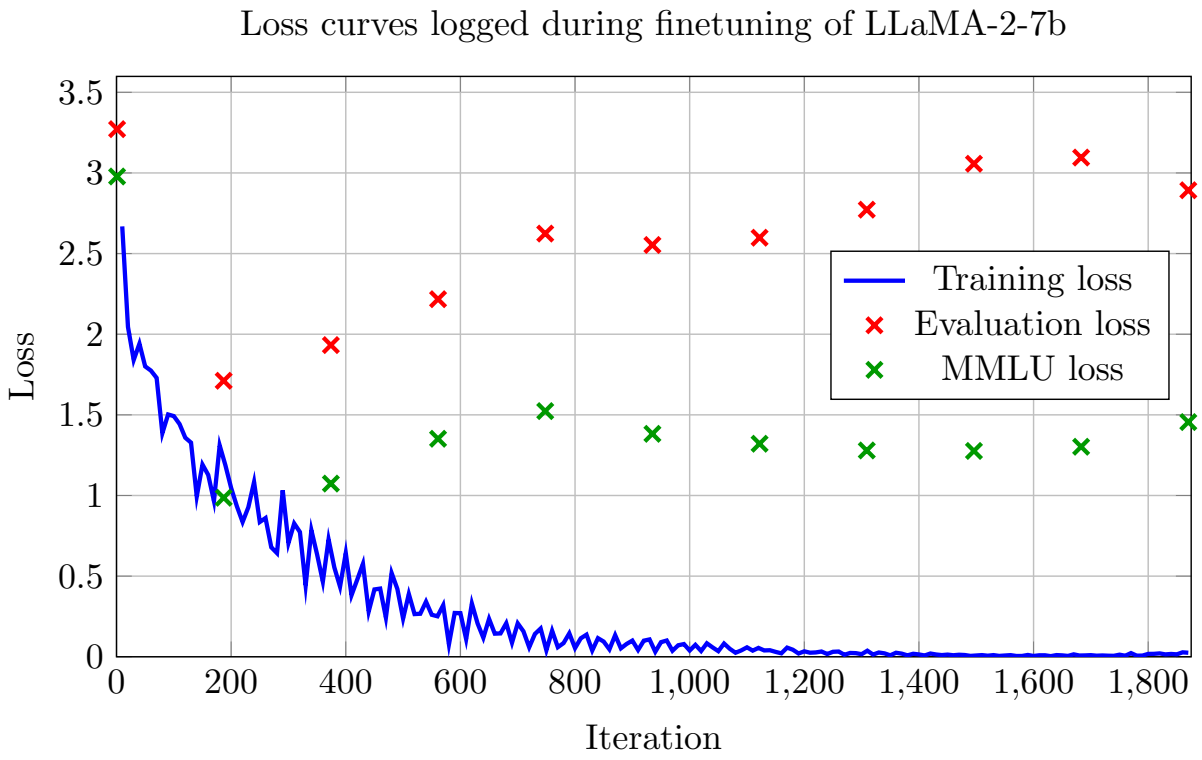


Figure 7: Loss curves for LLaMA-2-7b trained with r=64 on finalized dataset

While the shape of the training loss curve conforms with expectations for the development of training accuracy on deep learning tasks, the evaluation and MMLU loss curves behave

in an unusual fashion. One can observe a significant drop in both evaluation and MMLU loss after the first 187 iterations, followed by inconsistent development that generally trends upward, more so for evaluation loss than MMLU loss. This may be explained by initial iterations providing the model with relevant general information regarding the classix infrastructure and overall business concepts, while later iterations overfit the model to information regarding the classix modules present in the training data. However, for the proposed learning task, detailed in Section 3, an overfitted model is not necessarily an undesirable result.

2.4 Evaluation

Model performance was evaluated by measuring the accuracy of generated module descriptions. Module descriptions were generated by prompting models to describe a given module contained in the training data, after which the produced output is compared to the genuine module description. Due to their complexity, it is generally important for modules to be described nearly verbatim with respect to the original module description. Furthermore, an embedding-based semantic comparison between generated and true module descriptions is susceptible to hallucination, since hallucinated responses are often thematically accurate despite being inappropriate with respect to their exact content. Thus, module description accuracy is evaluated using BLEU. As a necessary word of warning regarding the use of the popular term "hallucination" to describe undesirable model responses, it should be considered that LLMs, as auto-regressive models designed solely for next-token prediction, do not and cannot share the human perspective on reasonable and desirable behavior regarding natural language. In this sense, "hallucination" is a rather anthropomorphic term for, contrary to the way the term is usual used, determinable and reproducible behavior.

In order to ensure reproducible results, the *temperature* inference parameter was minimized, causing the model to respond deterministically. Temperature, along with the *top-k* and *top-p* parameters, defines the degree of randomness in model inferences. More specifically, temperature operates on softmax (see (1.1)), augmenting the definition to either accentuate or diminish the impact of large activations, which can be observed in the following definition for softmax parameterized with temperature $T \in \mathbb{R}_+$:

$$\operatorname{softmax}_T(\boldsymbol{x})_i := \frac{e^{\frac{x_i}{T}}}{\sum_{j=1}^{n} e^{\frac{x_j}{T}}} \in \mathbb{R}^n.$$

The modified definition illustrates that for $0 < T < 1$, the relative likelihood assigned to high activations is increased, whereas it is decreased for $1 < T < \inf$. This, by extension, indicates that as T approaches 0, the maximal activation $x_{\max}$ approaches 1, making token selection deterministic:

$$\lim_{T \to 0} \operatorname{softmax}_T(\boldsymbol{x})_i = \begin{cases} 1 & \text{if } x_i > x_j \quad \forall j \neq i, \\ 0 & \text{otherwise.} \end{cases}$$

In this extreme case, the modified softmax function acts simply as a traditional max function.

2.5 Software implementation

Currently, most of the support for QLoRA finetuning is fundamentally written in Python. The Python modules responsible for providing the core functionality of QLoRA finetuning are the torch, peft, transformers, and bitsandbytes modules. The torch module, made available as part of the PyTorch framework, is responsible for the efficient execution of deep learning training algorithms by providing an implementation for GPU-accelerated tensor computation. We use this module for the execution of our training algorithm and for later analysis of model weights. The peft module is part of a Hugging Face framework responsible for implementations of various PEFT methods (Mangrulkar et al. 2022). The PEFT methods currently supported are LoRA, Prefix-Tuning, P-Tuning, Prompt Tuning, AdaLoRA, LLaMA-Adapter, and IA3, though we focus on and make use of the LoRA implementation. The transformers module is part of another Hugging Face framework responsible for providing access to state-of-the-art pre-trained models, as well as support for various finetuning prerequisites, such as tokenization, as well quantization and inference configurations (Wolf et al. 2020). In addition, this module supports the configuration of various callbacks to be executed during training, such as periodic model evaluation and logging of model loss. The bitsandbytes module, developed by Tim Dettmers, is responsible for efficient implementations of quantization and low-bit inference, as well as providing implementations of low-bit optimizers.

Our implementations of QLoRA finetuning and model inference are based on Python scripts which can be found in the QLoRA Github. Based on this, we detail the basic setup and procedure required to reproduce our implementation.

2.5.1 Installation

Our implementation is based on Linux, specifically Ubuntu 22.04, since a fully functional version of bitsandbytes has not been implemented for Windows to date. The first required installation is that of the CUDA Toolkit, which provides the GPU-accelerated libraries necessary for efficient finetuning. We used CUDA version 11.7. Additionally, a compatible Python version (3.6+) must be installed. For a minimal finetuning implementation, only the following Python modules must be installed: accelerate, bitsandbytes, datasets, peft, transformers. However, other modules are also defined as installation requirements in the QLoRA Github, which offer additional functionality such as evaluation metrics during training, listed in Listing 4.

```
bitsandbytes==0.40.0, transformers==4.31.0, peft==0.4.0, accelerate==0.21.0, einops
    ↪ ==0.6.1, evaluate==0.4.0, scikit-learn==1.2.2, sentencepiece==0.1.99, wandb
    ↪ ==0.15.3
```

Listing 4: Python module requirements

A final requirement pertaining to the use of LLaMA-2 models, specifically, is authentication with an access token of a Hugging Face account which has been approved by Meta. As of the current date, users wishing to access a LLaMA-2 model through Hugging Face must apply for approval by filling out the required form. After approval, which generally takes 1-2 days, a user access token must be generated on the approved Hugging Face account and added to the git credential manager through the commands listed in Listing 5.

```
git config --global credential.helper store
huggingface-cli login
```

Listing 5: LLaMA-2 authentication through git credential manager

2.5.2 Finetuning

We detail the steps necessary for finetuning on an arbitrary custom dataset using LLaMA-2-7b in Python, though both the dataset and model are completely interchangeable. The first step involves loading the pre-trained model and corresponding tokenizer with calls to respective *from_pretrained* functions, as well as adding special tokens to the LLaMA tokenizer using *add_special_tokens*, detailed in Listing 6.

Afterwards, a preprocessing step prepares the model for QLoRA finetuning using the *prepare_model_for_kbit_training* and *get_peft_model* functions, detailed in Listing 7.

Following this, a step for loading the training dataset is required. Datasets on Hugging Face can be loaded directly using the *load_dataset* function, or from local files using the *Dataset* class. In our case, we load a local dataset in the JSON format using the *from_json* member function, detailed in Listing 8.

Finally, QLoRA finetuning can be performed using the *train* member function of a defined *Seq2SeqTrainer* object, detailed in Listing 9.

2.5.3 Inference

After finetuning, adapter checkpoints can be loaded in conjunction with the corresponding pre-trained model in order to use the finetuned model for inference. The finetuned model can be loaded with calls to specific *from_pretrained* functions, similar to the loading process during finetuning, detailed in Listing 10.

Subsequent calls to the *generate* member function of the *model* object can be used to generate inferences on a given prompt, detailed in Listing 11.

```python
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig

model_id = "meta-llama/Llama-2-7b-hf"

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True, # Use double quantization
    bnb_4bit_quant_type="nf4", # NF4 as storage datatype
    bnb_4bit_compute_dtype=torch.bfloat16 # BF16 as copmutation datatype
)

tokenizer = AutoTokenizer.from_pretrained(model_id, tokenizer_type="llama", padding_side="right",
    ↪ use_auth_token=True)

model = AutoModelForCausalLM.from_pretrained(model_id, quantization_config=bnb_config,
    ↪ device_map="auto", use_auth_token=True)

tokenizer.add_special_tokens({
    "eos_token": tokenizer.convert_ids_to_tokens(model.config.eos_token_id),
    "bos_token": tokenizer.convert_ids_to_tokens(model.config.bos_token_id),
    "unk_token": tokenizer.convert_ids_to_tokens(
        model.config.pad_token_id if model.config.pad_token_id != -1 else tokenizer.pad_token_id
    ),
})
```

Listing 6: Loading pre-trained model

```python
from peft import prepare_model_for_kbit_training, LoraConfig, get_peft_model

model.gradient_checkpointing_enable()
model = prepare_model_for_kbit_training(model)

config = LoraConfig(
    r=64, # LoRA rank
    lora_alpha=16, # LoRA Alpha (weight update scaling factor)
    lora_dropout=0.1, # LoRA dropout (10% dropout)
    target_modules=["q_proj", "v_proj"], # LoRA modules (in this case, the query and key attention
    ↪ projections)
    bias="none", # No bias neurons
    task_type="CAUSAL_LM" # Learning task as causal language modeling (next-token prediction)
)

model = get_peft_model(model, config)
```

Listing 7: Pre-processing pre-trained model

```
from datasets import load_dataset, Dataset

# load_dataset("tatsu-lab/alpaca")

dataset_path = "path/to/dataset.json"

dataset = Dataset.from_json(path_or_paths=dataset_path)
```

Listing 8: Loading training dataset

```
import transformers

trainer = transformers.Seq2SeqTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=dataset,
    args=transformers.TrainingArguments(
        per_device_train_batch_size=1, # Training batch size per GPU
        do_eval=True, # Perform evaluation on sampled evaluation data
        do_mmlu_eval=True, # Perform evaluation on MMLU data
        per_device_eval_batch_size=1, # Evaluation batch size per GPU
        max_steps=1875, # Maximum total steps
        eval_steps=187, # Evaluation interval
        save_steps=500, # Model checkpoint saving interval
        logging_steps=10, # Training metrics logging interval
        optim="paged_adamw_32bit", # Optimization algorithm
        learning_rate=2e-4, # Learning rate
        gradient_accumulation_steps=16, # Gradient update every 16 batch gradient accumulations
        max_grad_norm=0.3, # Normalize gradients to prevent them from getting too large
        warmup_ratio=0.03, # Number of iterations before learning rate attains its designated value
    ↪ through linear progression starting from 0
        output_dir="path/to/output" # Directory in which to save model checkpoints
    ),
    data_collator=transformers.DataCollatorForCausalLM(tokenizer, source_max_len=1024,
    ↪ target_max_len=1024)
)

model.config.use_cache = False

trainer.train()
```

Listing 9: Executing QLoRA finetuning

```python
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
from peft import PeftModel

model_id = "meta-llama/Llama-2-7b-hf"
adapter_path = "path/to/adapter/checkpoint"

tokenizer = AutoTokenizer.from_pretrained(model_id)
tokenizer.bos_token_id = 1 # Fixing some of the early LLaMA HF conversion issues.

model = AutoModelForCausalLM.from_pretrained(
    model_id,
    torch_dtype=torch.bfloat16,
    device_map="auto",
    load_in_4bit=True,
    quantization_config=BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_compute_dtype=torch.bfloat16,
        bnb_4bit_use_double_quant=True,
        bnb_4bit_quant_type="nf4",
    )
)

model = PeftModel.from_pretrained(model, adapter_path)
model.eval()
```

Listing 10: Loading finetuned model

```python
from transformers import GenerationConfig

prompt = "What is the meaning of life?"
promptAlpaca = f"Below is an instruction that describes a task. Write a response that appropriately
    ↪ completes the request.\n\n### Instruction:\n{instruction}\n\nResponse:"

inputs = tokenizer(promptAlpaca, return_tensors="pt").to('cuda')

outputs = model.generate(
    **inputs,
    generation_config=GenerationConfig(
        do_sample=True,
        max_new_tokens=64, # Maximum number of tokens in the model response
        top_p=0.9,
        temperature=0.7,
    )
)

text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(text) # "42!"
```

Listing 11: Inference using finetuned model

3 Methodology

We detail a sequence of finetuning implementations involving two learning tasks that pertain to the classix infrastructure in this section. Specifically, the training data, foundation model, and finetuning parameters utilized, as well as a qualitative assessment of finetuned model performance is presented.

3.1 Learning Task 1: Rudimentary understanding of the classix infrastructure

As an introduction to LLM finetuning, the task of developing a rudimentary understanding of concepts relevant to the classix infrastructure was proposed. For simplicity, dialog capability was not yet deemed a prerequisite. Thus, finetuning was performed auto-regressively, thereby adapting the model for text completion.

3.1.1 Training Data

To compose text sequences for training, text data was extracted from each article in the parsed JSON document by concatenating the module name and description (if present), followed by block name and description for all blocks in the article. For example, the sequence for the module *utilpurc* is displayed in Listing 12.

```
utilpurc: This module is used for the inspection of purchasing data. Results window:
    ↪ This window lists the data to be checked as a result of an inspection run.
    ↪ Module name: utilpurc.mod.
```

Listing 12: utilpurc module sequence

The resulting tokenized sequences exhibited statistical characteristics presented in Table 5.

Statistical characteristic	Value
no. of sequences	914
mean sequence length	427.84
standard deviation of sequence length	611.51
minimum sequence length	5
maximum sequence length	7276
25th percentile sequence length	88
50th percentile sequence length	231
75th percentile sequence length	515
no. of sequences maximally 2000 tokens long	892 (97.6%)

Table 5: Statistical characteristics of tokenized sequences (auto-regressive)

3.1.2 Model and Parameters

As an initial experimental finetuning attempt, setup and execution was conducted on a local computer housing a 16GB RTX 4080 graphics card. This allowed for the finetuning of LLaMA-7b in approx. 3.5 hours. Aside from the *target_max_len*, which was increased from 512 to 2000, training parameters were left identical to those used for training guanaco-7b. This parameter was increased to leverage LLaMA-7b's maximal 2048 token sequence length in order to fit most sequences into a forward pass without the need for sequence partitioning.

3.1.3 Results

In order to acquire a general first impression of the capabilities of finetuned models, an approach guided by heuristics was employed for assessing model performance, rather than systematic evaluation. To this end, the following points of interest were investigated:

1. Does the model "speak in classix terms"?
2. How does the model respond to vague prompts, which are relevant to multiple different modules?
3. Can and does the model produce responses verbatim from a module description?
4. Can the model respond to questions appropriately?

Each point is answered based on conclusive observations in the following:

1. The model speaks in terms that could be attributed to classix by an unfamiliar party, but is riddled with hallucinations and false claims behind a confident facade. The model does not consistently provide reliable information on modules within the classix infrastructure.
2. The model is observably inclined to hallucation if the prompt does not provide the necessary context regarding the module being referenced. In cases where a prompt lacks the necessary context, the model will respond with the description of an arbitrary module of its choice, which can easily vary for a single prompt given a sufficient temperature value.
3. The model can produce responses verbatim from a module description, though this wasn't observed to occur frequently, even if the prompt contained a verbatim part of a module description.
4. The model can respond to questions appropriately, though it is unreliable. Occasionally, the model interprets a question as a rhetorical device, to which it responds with further rhetorical devices in an almost philosophical manner.

Exemplary model inferences to support these observations are presented in Section D of the appendix.

3.2 Learning Task 2: Denomination of classix modules in a dialog format

After establishing auto-regressive capabilities, the feasibility of instruction-tuning for chat dialog was investigated. A relevant learning task comprising the denomination of classix modules was proposed to structure dialog data.

3.2.1 Training Data

Based on the same training data as in the previous learning task, data was prepared in bi-directional pairs for each module, such that for each module, one training sample for description given a module name and one for denomination given a module description was generated. The training data further took inspiration from the Alpaca dataset, in order to align the foundation model with conversation dialog. Initial training data took on the form depicted in Listing 13.

```
1. [Below is an instruction that describes a task, paired with an input that provides
    ↪ further context. Write a response that appropriately completes the request.

### Instruction:
What is the name of this module?

### Context:
This is the description of the module <Modulename>: <Module Description>

### Response:], [The name of the module is <Modulename>.]

2. [Below is an instruction that describes a task. Write a response that appropriately
    ↪ completes the request.

### Instruction:
What is the purpose of the module <Modulename>?

### Response:], [The purpose of the module <Modulename> is <Module Description
    ↪ >.]
```

Listing 13: Alpaca-style training data format

In this format, the variables in angle brackets are replaced with the corresponding module data and the square brackets denote the input and output sequences, respectively. The input and output sequences are distinguished in order to control which tokens are used for loss and gradient updates. Tokens in the input sequence are masked with "IGNORE_INDEX" labels to avoid their use for loss calculation, which is thereby only applied to tokens in the output sequence. The numbers 1 and 2 enumerate the two training samples which make up a bi-directional pair for a given module.

The use of this dataset is motivated by the objective to train an LLM to understand rudimentary concepts specific to the classix infrastructure for a relevant learning task, namely the denomination of modules. The first sample in the format serves precisely the purpose of the learning task by providing the description of a module as context and querying the name of the module being described. The second sample serves the purpose of training the foundation model on concepts and associations relevant to the classix infrastructure (an added benefit to this sample is that the models are trained to reliably provide verbatim descriptions of modules they are queried on).

Typical phrases were defined and randomly pooled from when formatting instructions and responses in order to add diversity to the prompts which the model can manage as well as the responses the model generates. A Python dictionary containing these phrases is presented in Section E of the appendix.

The formatted sequences were subsequently evaluated on their tokenized lengths and removed[7] from the dataset if the length exceeded the limits defined by *source_max_len* and *target_max_len*, respectively, in order to avoid sequence partitioning.

Statistical characteristics are presented for the input and output sequences used as training data in Table 6.

Statistical characteristic	Input	Output
no. of sequences	1260	1260
mean sequence length	125.61	84.79
standard deviation of sequence length	133.65	130.64
minimum sequence length	44	7
maximum sequence length	994	948
25th percentile sequence length	47	11
50th percentile sequence length	66	24
75th percentile sequence length	150.25	103.25

Table 6: Statistical characteristics of tokenized sequences (Alpaca-style)

3.2.2 Model and parameters

Since the functional capability of finetuning was established locally, further finetuning was performed on a GPU cloud service. Here, finetuning was performed using an 80GB A100 graphics card. This allowed for the finetuning of LLaMA-7b in approx. 4 hours. Due to the new input/output format in the training sequences, the parameters for *source_max_length* and *target_max_len* were modified to equal 1024 each, making full use of LLaMA-7b's maximal 2048 token sequence length.

[7] In total, 7 sequence pairs, i.e., 14 total sequences, were removed due to excess length.

3.2.3 Results

It should be noted that the two "learning directions" introduced during training, i.e., the denomination of modules based on a description and the description of modules based on a name, do not exhibit identical performance despite each direction's comprising an equal number of training samples. This could be observed for model inferences generated on prompts representing each of these directions. Thus, each direction is analyzed separately.

Firstly, the description of modules, though not intended as the primary learning objective, produced notable accuracy, with model inferences often yielding the verbatim module description when prompted to describe a given module. However, when prompted with an invalid module description, the model hallucinates heavily, often merging terms and concepts from multiple unrelated modules. An example for this is presented for an imaginary "feedmamm" module in Listing 14. In this example, terms such as "supplier declaration" and "tax authority" both exist within the training data, but appear in completely unrelated modules. Tax authorities are referenced in the *ecvatsta* module, wherein the URL appearing in the example is also present. However, the "Federal Trade Office (Bundesanzeigerstelle)" is not referenced in the genuine module description (or anywhere in the training data). Instead, the genuine module description makes reference to the "Federal Central Tax Office (BZSt)". This is a subtle distinction, but it highlights the risk for proper names to be recalled incorrectly, since these are broken down into individual pieces during tokenization, making it harder for the model to identify them as coherent concepts.

```
Prompt: Describe the feedmamm module.

Response: The purpose of the feedmamm module is the following: This module is used
    ↪ to record supplier declarations. It is used if a supplier declares itself and all its
    ↪ affiliated companies to be independent. This independent supplier declaration
    ↪ must be reported to the tax authorities. The Federal Trade Office (
    ↪ Bundesanzeigerstelle) can be reached on the Internet under https://www.bzst.
    ↪ bund.de/contact/.
```

Listing 14: Hallucinated description given invalid module name

In contrast to the description of modules, the denomination of modules, which represents the primary learning objective, produced low accuracy, with model inferences rarely yielding the correct module name, even in cases where a module is described using large parts or the entirety of the verbatim module description. The finetuned model demonstrates an observable propensity to hallucation, generally responding not with the name of the wrong module, but with a fabricated module name altogether. An example of this is found in Listing 15, where the fabricated "financial" module is returned, despite the correct module, *balanfac*, being described verbatim in the prompt. Also presented in Listing 15 is an example demonstrating the fixation of the finetuned module on certain parts of the prompt, as the presence of the name *billofma* directly invokes a corresponding

model response, despite the prompt's containing many other possible indicators of which module is being described.

```
Prompt: With this module, the annual and period balances of a general ledger or
    ↪ personal account posted in financial accounting are displayed. Which module is
    ↪ being described?

Response: The module being described is financial.

Prompt: This is the description of the module billofma: Feeding guinea pigs and
    ↪ groundhogs. Which module describes Donald Trump's presidency?

Response: The module being described is billofma.
```

Listing 15: Hallucinated module names

The difference in accuracy in these results is presumably explained by the asymmetry of the two training directions. While the training samples for the description of modules give the finetuned model a clear sequence of tokens to optimize on, the training samples for the denomination of modules force the finetuned model to attempt to learn to extract the relevant information from a given prompt in order to produce an accurate module name. Since only the module name and surrounding response formulation provide tokens for gradient optimization in these samples, module denomination mainly involves training the attention heads on recognizing relevant indicators from the corresponding description. Furthermore, the samples for the description of modules contain significantly longer outputs comparatively, which provide more instances for gradient optimization.

3.3 Performance Improvement on Learning Task 2

During evaluation of the previous model, specifically regarding propensity to hallucation, a jarring error was identified. Since the module name itself is included in the context for samples querying the module name, any module name passed into a prompt would be identified by the model as the queried module name, regardless of the contents of the module description. To avoid this, finetuning for the same learning task was performed using our finalized dataset containing corrected training data, in which the module name does not appear in the context for the module description. Instead, the module description is placed directly after the context tag, as shown in Listing 16.

```
### Context:
<Module Description>
```

Listing 16: Context without module name

The foundation model and training parameters remained identical to the previous finetuning implementation.

3.3.1 Results

Since samples for the learning direction responsible for the description of modules remained unchanged, no changes to finetuned model performance were observed for model inferences generated on prompts representing this direction. The module denomination learning direction, however, exhibited notable improvements, particularly regarding the behavior being targeted by the changes to the training data, namely fixation on module names provided in prompts. To this end, the prompt which previously demonstrated the model's fixation on passed module names now also takes additional portions of text into account, as demonstrated in Listing 17.

```
Prompt: This is the description of the module billofma: Feeding guinea pigs and
    ↪ groundhogs. Which module describes Donald Trump's presidency?

Response: The module being described is trumppres.
```

Listing 17: Improvement to hallucinated module names

Accuracy, even for prompts containing extensive descriptions, remained low, however. An example for a successful case exists for the *balanfac* module, where the entire verbatim module description is contained in the prompt, presented in Listing 18.

```
Prompt: With this module, the annual and period balances of a general ledger or
    ↪ personal account posted in financial accounting are displayed. Which module is
    ↪ being described?

Response: This module is balanfac.
```

Listing 18: Successful module denomination

Through evaluation of model performance on module denomination, an interesting quality of the finetuned model was observed. The model, rather than *classifying* from a fixed set of names, is observably trained to *generate* a plausible module name given some description. This distinction is demonstrated using the query of a fabricated module in Listing 19. In this example, a query for the name of a module that describes "feeding mammals" is met with a response which contains the plausible and fitting module name "feedmamm". This name is not only semantically accurate, but also shares the syntactic structure of many module names within AppsWH, comprising an 8-letter concatenation of descriptive subwords according to the common AppsWH module naming scheme.

```
Prompt: This is the context of a module: feeding guinea pigs and groundhogs. Which
    ↪ module describes feeding mammals?

Response: The module being described is feedmamm.
```

Listing 19: Generative rather than classifying model behavior

4 Analysis

Though our experimental results do not fulfill expectations for the proposed learning task of module denomination, they offer insight to finetuned model behavior and provide a basis for more rigorous analysis regarding performance on the secondary learning objective, i.e., the description of modules. To this end, analysis using our finalized dataset was extended into a more mathematical domain. More specifically, our work comprises three types of analysis, each pertaining to the state of the adapter matrices after finetuning. For the first type, finetuned model performance on the generation of module descriptions is evaluated. To this end, BLEU scores are computed in order to measure the accuracy of model-generated module descriptions with respect to the genuine module description. The next form of analysis investigates the differences of finetuned adapter matrices to their respective initialized states. Here, the absolute change in the elements of these matrices is measured, which quantifies the change effected by finetuning numerically and ensures, given sufficient magnitude, that "lazy training" has not taken place, a phenomenon wherein model weights are hardly altered during finetuning. In addition, the distribution of the singular values of the adapter differences is analyzed, as well as the the impact on performance when adding truncated finetuned adapter matrix differences to respective initialized states, particularly compared to the performance of adapters finetuned conventionally with the truncated dimensions. The final type of analysis involves examining the similarity/distance between subspaces spanned by different finetuned adapter matrices. Specifically, subspaces spanned by the products of adapter matrices of modules in the layers of various models (differing in parameter count and LoRA rank[8]) were compared among each other.

In order to facilitate systematic analyses, the finetuning results of three differently-sized LLaMA-2 models at various LoRA ranks were prepared. The specific finetuning implementations performed for the preparation of these results involved finetuning the 7b, 13b, and 70b parameter count versions of LLaMA-2, where the 7b version was trained with LoRA ranks from $r = 1$ to $r = 64$, the 13b version was trained with ranks from $r = 8$ to $r = 64$, and the 70b version was trained with $r = 32$ and $r = 64$, each in increments of powers of 2. To further promote comparable results between models trained at a given parameter count with differing LoRA rank, adapter weight matrices are initialized such that their rows are pooled from the rows of the largest initialized weight matrix ($r = 64$). This ensures that the image of a smaller initialized adapter weight matrix is always a subset of the image of the largest matrix.

4.1 Module description performance analysis

The performance of finetuned models can be measured on the basis of model-generated module description accuracy, since this task is directly linked to the amount of information

[8]Only models sharing a common parameter count can be used as a basis for subspace analysis, since differences in parameter count translate to differences in context length and thereby lead to differently dimensioned adapter matrices.

stored in adapters of various dimensions. We thus generate module descriptions for finetuned models of all proposed parameter counts and LoRA ranks using all modules contained within the training data, and subsequently measure the BLEU scores of these generated descriptions with respect to the genuine module description. Resulting scores are presented visually and in the form of relevant statistics for evaluations that both include and exclude evaluation samples, as these are samples on which the models are not finetuned. For the visual presentation, we choose to measure the cumulative BLEU score occurrence over the BLEU value range $[0, 1]$ for a given sample count $n \in \mathbb{N}$ according to the following definition:

$$F(x) = \sum_{i=1}^{n} \chi_i(x), \quad x \in [0, 1],$$
$$\chi_i(x) := \begin{cases} 1 & \text{if BLEU}(\hat{S}_i, S_i) \leq x, \\ 0 & \text{otherwise,} \end{cases}$$

where $\hat{S}_i$ and S_i are the i-th model-generated and genuine module descriptions, respectively. For the statistical evaluation, we provide an overview of metrics including mean BLEU score, standard deviation on BLEU scores, number of perfect scores, i.e., generated module descriptions which precisely match the genuine module description, as well as final training loss.

After evaluating BLEU scores for each model-generated module description, we analyze the worst performing module descriptions for their statistical characteristics, with particular focus on tokenized sequence length, as well as for their common traits.

4.2 Adapter difference analysis

The extent of the change in adapter matrices as a result of finetuning can be quantified in a variety of ways. We opt to visualize the difference between finetuned adapter matrices and their respective initialized states graphically (we refer to these differences as *finetuned adapter deltas* in our work for simplicity), in addition to evaluating statistics pertaining to the absolute change per adapter matrix element induced by finetuning. We then move on to visualize the distribution of singular values of finetuned adapter deltas, which provides insight regarding the density of information contained in lower-dimensional approximations of learned adapter adjustments. Finally, we evaluate the performance of a model with adapters defined as the sum of truncated finetuned adapter deltas and respective adapter initializations, while also drawing comparisons to a model with adapters originally finetuned with the truncated dimensionality. Truncation is performed using the truncated SVD. We opt to truncate finetuned adapter deltas and add the result to the respective adapter initialization rather than truncating finetuned adapters themselves, because, for the A LoRA adapter specifically, while the singular value distribution of finetuned adapter deltas is heavily skewed toward a small subset of large singular values (see Figure 11), the singular value distribution of finetuned adapter matrices has a considerably larger minimal baseline, making later singular values less negligible (see Section F

of the appendix). It should be noted that for the B LoRA adapter, both methods of truncation are equivalent, since these adapters are zero-initialized. However, for the A factor, which is randomly initialized, the distinction regarding truncation method is of significance. This can be verified by considering finetuned adapters as the sum of matrices representing finetuned adapter deltas and respective initialized states $\Delta A + A_0$, where the initialized states can be viewed as some noise matrix $A_0 \sim \mathcal{U}(-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}})$, where d is the pre-trained model dimension and $\mathcal{U}(-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}})$ is a uniform distribution on the interval $[-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}}]$ according to the Kaiming initialization method described in Section 2.3. As such, the variance of this "noise" equals $\frac{1}{3d}$, a magnitude comparable to that of the values found in finetuned adapter deltas themselves (see Figure 9). Thus, the singular value decomposition of finetuned adapters is heavily influenced by the adapter's respective initialization, leading to less accurate truncated SVD approximations. By only truncating the finetuned adapter deltas and adding the result to the respective adapter initializations, we bypass the influence of this noise and produce more accurate finetuned adapter approximations.

4.3 Subspace analysis

Subspaces can be compared in a multitude of ways. The authors of LoRA elect to use a measure based on the Grassmann distance to assess subspace similarity (Hu et al. 2021, Section 7.2). Other options include the cosine-sine (CS) decomposition, which computes Q-Blocks with highly related singular value decompositions (Golub and Van Loan 2013, Chapter 2.5.4). We follow the procedure of subspace analysis employed by the LoRA authors.

In the case of the Grassmann distance $\delta(\cdot)$, the "distance" (or, more precisely, *metric* on the Grassmannian) between two linear subspaces $A \in Gr(i, N)$ and $B \in Gr(j, N)$, where $Gr(k, N)$ is defined to be the set of all k-dimension linear subspaces of $\mathbb{R}^N$, is defined using the *principal angles* θ_k for these subspaces:

$$\delta(A, B) := \sqrt{\sum_{k=1}^{\min(i,j)} \theta_k^2} \in [0, \frac{\pi}{2}).$$

Principle angles can be calculated using the *principle vectors* $\hat{a}_i$, $\hat{b}_i$ of the compared subspaces, where principal vectors are defined to be unit vectors within the spans of their respective bases that have minimal angle between them and are orthogonal to each other:

$$\hat{a}_k^T \hat{b}_k \text{ maximal with } \|\hat{a}_k\| = \|\hat{b}_k\| = 1 \text{ and } \hat{a}_k^T \hat{a}_l = \hat{b}_k^T \hat{b}_l = 0 \quad \forall k \neq l.$$

The principle angles are then defined as the inverse cosine of the inner products of the principle vectors $\theta_k := \arccos \hat{a}_k^T \hat{b}_k$. In practice, principal angles can be extracted from the singular values computed in a singular value decomposition $M_A^T M_B = U \Sigma V^T$ on the product of the matrices M_A, M_B acting as bases for each subspace. The principle angles

are then given by the inverse cosine of the singular values $\theta_k := \arccos \sigma_k$ (defined along the diagonal of Σ).

The subspace similarity measure employed by the authors of LoRA uses a slightly modified version of the Grassmann distance, equivalent to the following:

$$\phi(A, B, i, j) = \psi(U_A^i, U_B^j) = \frac{1}{p} \sum_{k=1}^{p} \sigma_k^2,$$

where $p = \min(i, j)$, and the practical computation is performed using the left singular vectors instead of the singular values:

$$\phi(A, B, i, j) := \frac{\|U_A^{i\mathrm{T}} U_B^j\|_F^2}{p} \in [0, 1], \tag{4.1}$$

where U^k represents the matrix containing columns of U corresponding to the top k singular vectors. Here, a similarity score $\phi(\cdot)$ of 1 indicates complete subspace overlap and a score of 0 indicates complete separation.

Since the subspace analysis is performed on the products W of adapter matrices A, B with $W = BA$, and these products share the substantial dimensionality $d \times d$ of the pretrained weight matrices, a more efficient method of computing the required left-singular matrices U_w that involves the SVDs of the factors A and B is implemented. Given the decomposed matrices U_a, Σ_a, V_a^T and U_b, Σ_b, V_b^T resulting from an SVD on the matrix factors A and B, respectively, the left-singular matrix U_w of their product W can be computed as $U_w = U_b U_c$, where U_c is the left-singular matrix resulting from an SVD on the matrix $C = \Sigma_b V_b^T U_a \Sigma_a$. Due to the sign ambiguity of the SVD, the instance of the left-singular matrix U_w computed by this method may contain differently signed columns from the instance computed by a regular SVD on the product W, but this is not relevant to the result of the modified Grassmann distance because of the Frobenius norm.

We aim to reproduce the results obtained by the authors of LoRA, who first analyze adapter subspaces in order to determine to what degree a subspace spanned by the top $i \in \mathbb{N}$ singular vectors in the left-singular matrix of a lower-rank weight update is contained by the subspace spanned by the top $j \in \mathbb{N}$ singular vectors in the left-singular matrix of a higher-rank weight update (Hu et al. 2021, Section 7.2). We follow part of the procedure employed by the LoRA authors and implement a comparison of subspaces stemming from the finetuned module $W = BA$, i.e., the product of the two finetuned adapters of a LoRA module, for the query projection of LLaMA-2-7b finetuned with rank $r = 8$ and $r = 64$, respectively. We further take advantage of the insight provided by Hu et al. 2021, Section H.1, which demonstrates that conclusions hold regardless of which model layer is analyzed, and examine the first layer (out of 32) of the finetuned models. We then follow the procedure employed by the authors of LoRA regarding analysis of subspace similarity between different randomly seeded finetuning implementations of LLaMA-2-7b with $r = 64$, by again examining subspaces stemming from the finetuned module for the query projection in the first layer.

5 Results

We detail the results obtained from each analysis in this section.

5.1 Module description performance analysis

The measure of cumulative BLEU score occurrence based on model-generated module descriptions is presented in Figure 8 for all proposed parameter counts and LoRA ranks used for finetuning LLaMA-2-7b, as well as for score assessment both including and excluding evaluation samples. Models are labelled in the form "p-r", where p and r are the parameter count and LoRA rank of a finetuned model, respectively. Since the cumulative BLEU score measure quantifies the number of module descriptions associated with a BLEU score up to a certain value, curve progressions that start off low and rise when nearing the upper bound of the domain are indicative of higher relative performance. A theoretically optimal progression takes the form of the shifted scaled Heaviside step function $n \cdot \Theta(x-1)$, where n is the number of evaluated samples, as this involves all samples producing perfect BLEU scores. From the figure, it can be observed that the cumulative measure of scores excluding evaluation samples, which comprises 120 fewer samples than the full training data, essentially shifts the measure of cumulative BLEU score occurrence including evaluation samples downward, with no impact to the performance of models relative to each other. Though this is not particularly surprising, it is indicative of uniform "extrapolative capabilities", i.e., the ability to produce accurate descriptions for modules not contained in the training data, across all finetuned models. The scores produced by foundation models, labelled by "-0" suffixes, yield a measure of baseline module description performance and provide a point of reference for the evaluation of finetuned model performance.

The statistical evaluation of measured BLEU scores is presented in Table 7. From this, it can be observed that mean score and number of perfect scores are strongly correlated ($R \approx 0.99$ for excluded evaluation samples), while also both being moderately inversely correlated to final training loss ($R \approx -0.7$ to mean score for excluded evaluation samples, and $R \approx -0.6$ to number of perfect scores). However, final training loss also appears to be tied to parameter count directly to some degree ($R \approx -0.85$), supported by the presence of disproportionately low final training loss values relative to mean score and perfect score count for 70b finetuned models. This observation is indicative of an increase in performance for models finetuned at higher parameter counts on the primary learning task, i.e., module denomination, which is not included as part of the BLEU score analysis, but is reflected in magnitude of the final training loss. The BLEU score standard deviation is generally consistent, taking on lower values when excluding rather than including evaluation samples, and taking on consistently low values for foundation models.

Regarding relative model performance, a non-monotonic progression in performance for increases in LoRA rank and parameter count can be observed in both Figure 8 and Table 7. Contrary to the baseline expectation, increased rank and parameter count were not observed to consistently lead to improved performance in our case. Instead, model

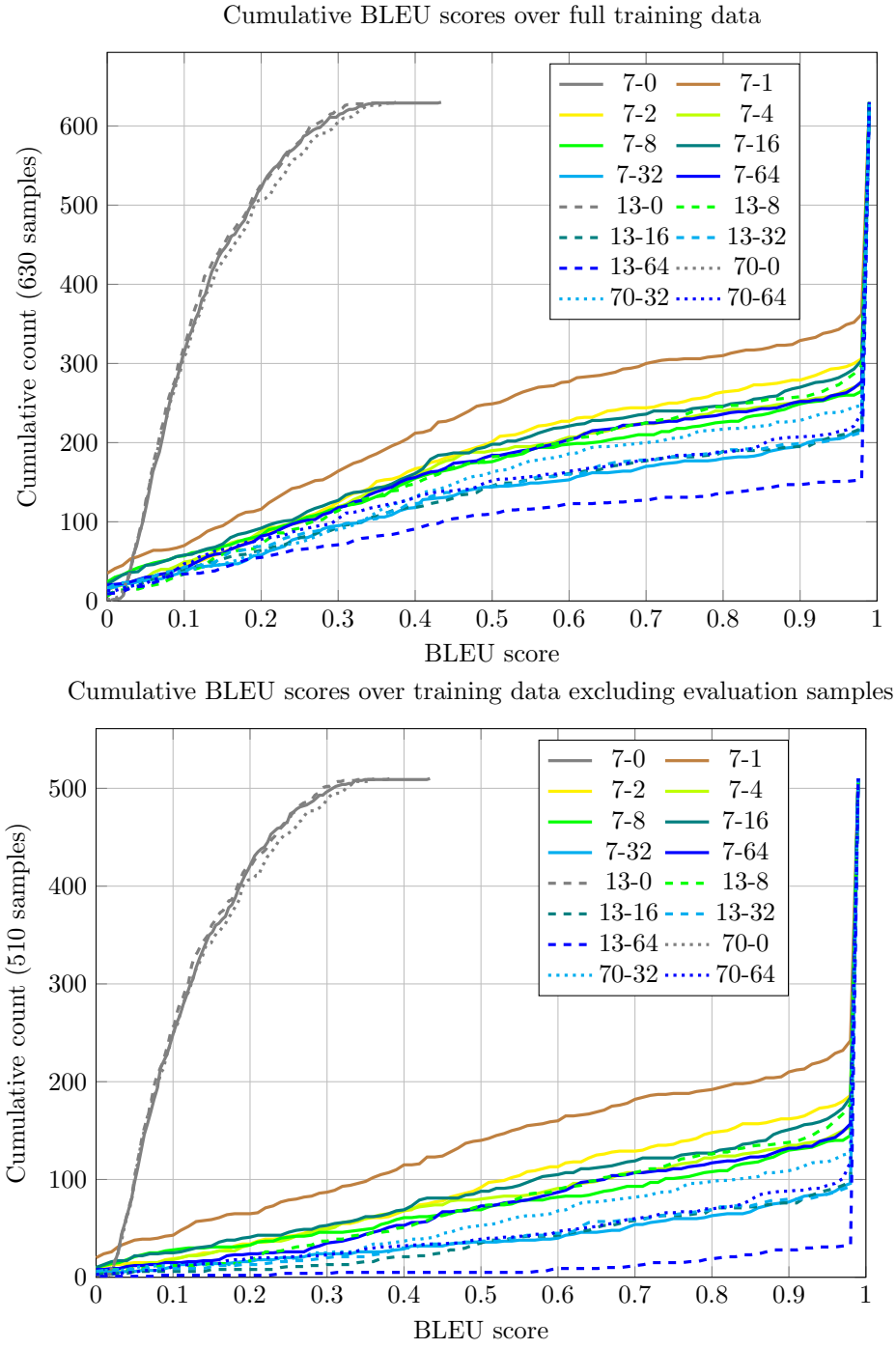


Figure 8: Cumulative BLEU score occurrence

performance, while trending upward, fluctuates as the LoRA rank is increased. For example, the 7-16 and 7-64 models are both dominated by multiple models finetuned at lower ranks. Similarly, model performance trends upward as the parameter count is increased, but shows notable fluctuation, particularly in the case of models finetuned at the

Finetuned model	Mean		Std. dev.		Perfect scores	Final training loss
	Full	No eval	Full	No eval		
7-0	0.12	0.13	0.08	0.08	0	NA
7-1	0.65	0.74	0.38	0.35	265	0.37
7-2	0.72	0.82	0.35	0.29	314	0.35
7-4	0.74	0.85	0.35	0.28	352	0.32
7-8	0.75	0.86	0.36	0.28	363	0.32
7-16	0.72	0.83	0.36	0.3	317	0.32
7-32	0.8	0.92	0.33	0.21	404	0.27
7-64	0.74	0.86	0.35	0.26	345	0.31
13-0	0.12	0.12	0.08	0.08	0	NA
13-8	0.74	0.86	0.34	0.26	328	0.28
13-16	0.8	0.93	0.33	0.19	403	0.27
13-32	0.79	0.92	0.33	0.21	410	0.27
13-64	0.85	0.98	0.3	0.1	477	0.27
70-0	0.13	0.13	0.08	0.08	0	NA
70-32	0.77	0.89	0.33	0.24	379	0.19
70-64	0.79	0.92	0.34	0.22	383	0.19

Table 7: BLEU score statistics

70b parameter count. For example, the 13-8 model is dominated by the 7-8 model, and both the 70-32 and 70-64 models are dominated by multiple models finetuned at lower parameter counts. An additional notable aspect of the data is the extreme statistical similarity between the 7-32 and 13-16 models, which in fact share the exact same final training loss. Explanations for these results could be provided by the *Chinchilla scaling law* (Hoffmann et al. 2022), dataset-specific optima, or variation in training algorithms. The Chinchilla scaling law suggests, that given a compute budget for Transformer language model training, there exists an ideal relationship between model size and number of training tokens for optimizing model performance, which scales proportionally as the compute budget is adjusted (Hoffmann et al. 2022, Section 3.4). Since we vary model size but keep our training arguments and dataset the same, sub-optimal performance for larger models could be explained by this law. With LoRA rank as an additional finetuning variable in our analysis, the existence of data-specific optima could also provide an explanation. While the Chinchilla scaling law applies to LLM training, QLoRA finetuning introduces additional variables that impact model performance. As such, there may be data-specific optimal values or ratios for finetuning variables including, but not limited to, parameter count and LoRA rank, which do not simply involve increasing these parameters in order to improve performance. Finally, the natural statistical variation associated with the probabilistic nature of training algorithms executed during finetuning could provide an explanation for the discussed results. This is supported by the observation that, for a single parameter count, performance can fluctuate in both directions as the rank is increased, which, if not attributed to natural variation, necessitates a highly

sensitive and unstable dependency between LoRA rank and model performance. In order to investigate this possibility, LLaMA-2-7b was finetuned with $r = 64$ four additional times using unique random seeds for training. Resulting model performance is presented in Table 8. The performance measures display considerable variance despite utilizing the same training arguments and dataset. For example, the number of perfect BLEU scores ranges from 345, the amount yielded by the initial finetuning of LLaMA-2-7b with $r = 64$, to 423, which constitutes the highest perfect score count out of all finetuned 7b models. This variance in performance is not accompanied by an according variance in final training loss, which is significantly more consistent, yielding a difference of approx. 6×10^{-3} between the highest and lowest final losses. In general, the results support the hypothesis that the primary source of fluctuation regarding model performance stems from the statistical variance associated with probabilistic training algorithms executed during finetuning, though this does not entirely rule out the effects of the Chinchilla scaling law or data-specific optima.

Finetuned model	Mean		Std. dev.		Perfect scores	Final training loss
	Full	No eval	Full	No eval		
7-64-0	0.74	0.86	0.35	0.26	345	0.31
7-64-1	0.79	0.91	0.34	0.24	406	0.31
7-64-2	0.81	0.94	0.33	0.19	423	0.31
7-64-3	0.76	0.87	0.34	0.27	364	0.32
7-64-4	0.77	0.89	0.34	0.24	379	0.31

Table 8: BLEU score statistics for randomly seeded trials

Based on the BLEU scores evaluated for performance analysis, we investigate properties of the worst performing module descriptions in order to determine why some descriptions perform significantly worse than others. We look at the 20 worst performing module descriptions generated by the overall best performing finetuned model, which is LLaMA-2-13b finetuned with LoRA rank $r = 64$, though we postulate, based on observation, that the conclusions drawn from our investigation hold regardless of which finetuned model is used, and that certain properties of the module descriptions used as training samples are the main reason for poor performance on those descriptions. We first investigate the statistical characteristics of the worst performing module descriptions, presented in Table 9. A key feature of the data is the mean sequence length, which is over 7 times as large as the mean sequence length over the entire output training data (see Table 6), indicating that long module descriptions are more difficult to learn effectively than shorter ones. This is indirectly supported by findings from Liu et al. 2023, which suggest that performance steadily degrades on longer contexts (Liu et al. 2023, Section 1). From a purely mathematical standpoint, an increased likelihood of poor performance for long sequences can be explained by the large number of tokens needed to be correctly predicted in sequential fashion in order to produce an accurate response, as one incorrect token in the response generally leads to incorrect tokens from that position onward. However, it can also be observed from the table that there are sequences among the worst performing module

descriptions which are not especially long, such as the minimum length sequence consisting of 118 tokens. Since sequence length is thereby not a decisive property on its own, we further investigate semantic properties of the descriptions yielding poor performance, in order to ascertain the root cause of the discrepancy in performance.

Statistical characteristic	Value
no. of sequences	20
mean sequence length	610.85
standard deviation of sequence length	211.49
minimum sequence length	118
maximum sequence length	946
25th percentile sequence length	609.75
50th percentile sequence length	657.5
75th percentile sequence length	699.5

Table 9: Statistical characteristics of the tokenized 20 worst performing module descriptions

For the investigation of semantic properties, we output the 20 examined module descriptions and compare them to their corresponding sample in the training dataset. With this, we are able to make the initial observation that each model-generated description begins accurately, and that poor performance can generally be attributed to either a premature ending of the description or a series of inaccurate tokens following an arbitrary number of accurate tokens which continues until the description is concluded. We present examples for both of these cases, demonstrating how both short and long responses can yield poor performance in the process. The example with the shorter response stems from the *localeEdit* module, and technically ends in a reasonable fashion in both semantic and syntactic terms, but omits a tremendous portion of the module description by ending prematurely. The model-generated module description is compared to the first part of the corresponding genuine description in Listing 20. The omitted portion of the genuine module description is represented by an ellipsis to save space.

```
The module localeEdit can be described as follows: Important! Please also read the
    ↪ description on the subject of location−specific data in general (e.g. different
    ↪ languages or calendars of different countries).

The module localeEdit can be described as follows: Important! Please also read the
    ↪ description on the subject of location−specific data in the installation
    ↪ documentation This module is used to create, maintain and manage location−
    ↪ specific data. These can . . .
```

Listing 20: Model-generated (top) vs. genuine (bottom) short module description

The remaining example stems from the *statturn* module, where the finetuned model begins hallucinating mid-way through its description in an unusual fashion. The model-generated module description is compared to the corresponding genuine description in

Listing 21. Since there is no mention of "Mr. X" in the training dataset[9] used for finetuning, the name must stem from prior knowledge of the LLaMA-2 foundation model. For reference, Mr. X is a fictitious character from the "Resident Evil" video game series, whom LLaMA-2 as well as any of our finetuned models can accurately describe when prompted accordingly.

```
The purpose of the module statturn is as follows: This app ... movements are referred
    ↪ to as part consumption:Goods receipts ...

The purpose of the module statturn is as follows: This app ... movements are
    ↪ monitored via the part consumption statistics (list view):Consumption/non−
    ↪ consumption of a part .... Mr. X has carried out a parts consumption statistics
    ↪ evaluation for you. You can now see which parts he thinks you have consumed
    ↪ or have not consumed in which period.... Mr. X is convinced that you will now
    ↪  be able to make use of the statistics in the appropriate context.
```

Listing 21: Model-generated (top) vs. genuine (bottom) long module description

No further consistent indicators of poor performance could be identified among the worst performing module descriptions.

5.2 Adapter difference analysis

The difference between finetuned adapter matrices and their respective initialized states, i.e., a finetuned adapter delta, is visualized for the A LoRa adapter of the query projection in the first layer of LLaMA-2-7b in Figure 9. In addition to quantifying the general magnitude of changes in adapter matrix elements, the visualization facilitates the observation of notable structural arrangements and patterns within the difference matrix. Though not entirely clear from the figure, computation of finetuned adapter deltas yielded multiple insights regarding the extent of changes in adapters incurred through finetuning. One such insight pertains to the susceptibility of the A adapter initialization to sign changes. Here, the probability of sign change lies at approx. 25% per matrix element, varying slightly by parameter count and LoRA module. Further insights are based on the results of subsequent analysis on the absolute change per matrix element in finetuned adapter deltas.

In order to further quantify the learned adapter changes embodied by finetuned adapter deltas, a statistical overview of the absolute change per adapter matrix element is presented in Table 10. Results are grouped by parameter count and LoRA module, as the absolute difference per matrix element was most consistent across varying LoRA rank and layer index. The low relative standard deviation values observable in the presented statistics, which represent the standard deviation of the mean absolute differences per

[9]The only term related to "Mr. X" found in the training data is "Mr. Proper", who is mentioned in a single module.

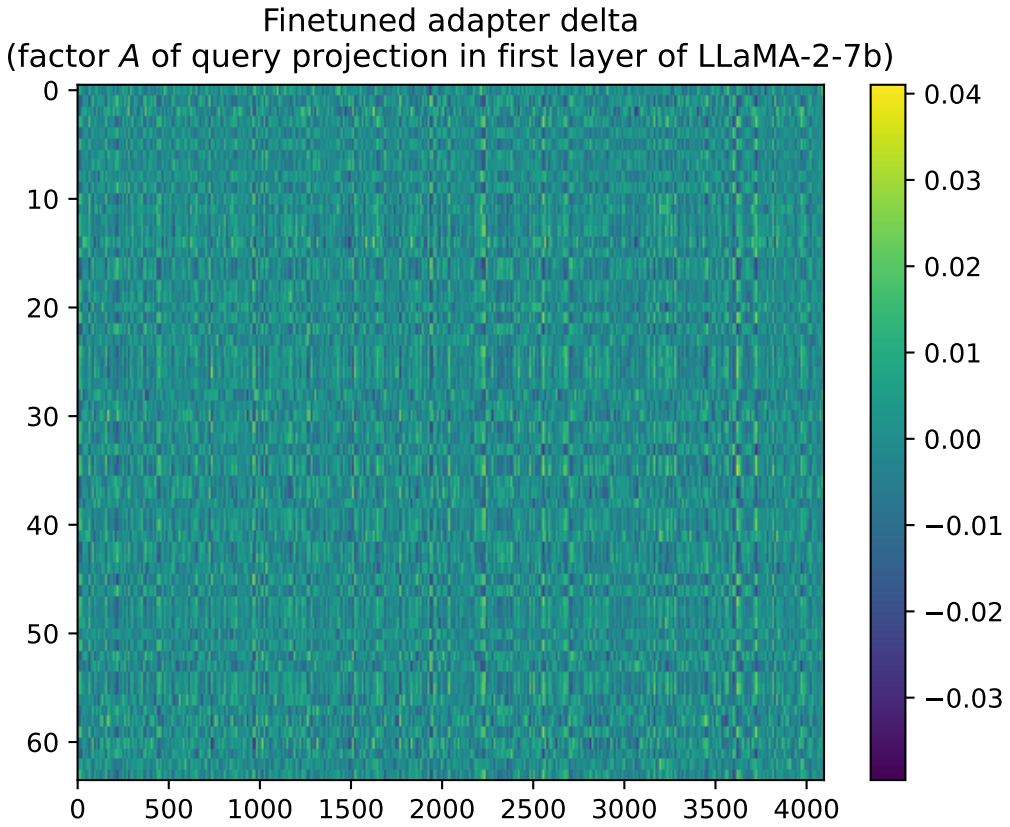


Figure 9: Difference between finetuned adapter matrix and initialized state

matrix element over all LoRA ranks and layers, are indicative of an adequate subdivision of variables for analysis. Modules containing the "mlp" prefix represent each of the learned projections in the SwiGLU activation function found in the LLaMA and LLaMA-2 foundation models (Ma, Fang, and X. Wang 2023, Figure 2). In general, it can be observed that absolute differences per matrix element share relatively consistent relationships between modules within each parameter count. For example, for the A adapter, the mean absolute differences for the q_proj, k_proj, and down_proj modules are generally approximately equal within each parameter count, with higher parameter counts producing lower mean values. Relationships between modules in the B adapter appear less consistent by comparison, though a reduction in mean value for the higher parameter counts can still be observed, as well as reduced mean values in general compared to the A adapter. Standard deviations appear fairly arbitrary, with the only arguably notable values being the particularly low standard deviations for the gate_proj and up_proj modules of the B adapter in the 70b parameter count model. As a loose guideline, the mean absolute difference per matrix element makes up approx. 1/4 of the length of the interval on which the uniform distribution used for the random initialization of A is defined. It is no coincidence that this value conforms with the aforementioned approx. 25%

probability of sign change per matrix element in A. This correspondence can be understood with the help of a few shortcuts when considering that 50% of all values in a given zero-centered uniform distribution $\mathcal{U}(-a, a)$ fall into the interval $[-\frac{a}{2}, \frac{a}{2}]$, and that each of these values possesses an approx. 50% chance of undergoing a sign change given a mean absolute change $\frac{a}{2}$ equivalent to 1/4 the length of the distribution interval, resulting in an overall 25% probability of a sign change. The true underlying explanation is more complex and is based on the exact distributions of both adapter initialization and adapter differences/changes, which we leave to be understood in the Section G of the appendix. As part of this explanation, based on our observation that adapter changes are roughly normally distributed, we approximate the distribution of adapter changes using a zero-centered normal distribution. The reason for normally distributed adapter changes could be tied to the *central limit theorem*, which states that, under appropriate conditions, the distribution of pooled sample means converges to a normal distribution as the number of pooled samples approaches infinity. As a result, finetuned adapter elements are not distributed uniformly like their initialization, and are instead approximately normally distributed as well[10]. The elements of the finetuned adapter delta and corresponding finetuned adapter of the factor A of the query projection in the first layer of LLaMA-2-7b are visualized using 100-bin histograms over normal distributions scaled according to the height of each histogram in Figure 10.

As a method for investigating the structural complexity of the evaluated matrix differences, the distribution of singular values of finetuned adapter deltas is evaluated and is visualized using linear and semi-log plots for the singular values of the adapters of all modules within LLaMA-2-7b finetuned with LoRA rank $r = 64$ in Figure 11. In order to produce a comprehensive overview of the singular value distributions within the entire model architecture, the per-index geometric mean of the singular value distributions across all layers is computed for each adapter and visualized accordingly. For comparison between ranks, an inset plot containing the respective singular values for LLaMA-2-7b finetuned with $r = 8$, as well as a further inset plot providing a magnified view of the first 8 singular values of the primary distribution (LLaMA-2-7b finetuned with $r = 64$), is also provided in Figure 11. This comparison illustrates the significantly higher uniformity in the distribution of singular values for the lower $r = 8$ rank, indicating that the learned adapter changes at this rank are closer to the information-theoretical optimum for the associated dimensionality. Concerning the relative progression of singular values among the adapters of different modules, the curves of the singular value distributions of all adapters exhibit high similarity, though variance in the magnitude of the first, i.e., largest, singular value of each adapter can be observed. For example, the first singular values of the gate_proj module exhibit high relative magnitudes, particularly in the case of the associated B adapter, which possesses the largest singular value in general. It

[10]In the case of the B adapter, finetuned adapter elements are as normally distributed as their corresponding adapter changes, since B is zero-initialized. In the case of the A factor, the distribution of finetuned adapter elements is augmented by the original uniform distribution used to initialize A, as well as the mathematical constraints imposed on gradient updates, whose influence becomes clear from the identical domain for both the adapter change and finetuned adapter element distributions in Figure 10.

Parameter count	Module	A		B	
		Mean	Std. dev.	Mean	Std. dev.
7b	self_attn.q_proj	1/114.84	1/1183.5	1/129.24	1/971.74
	self_attn.k_proj	1/114.84	1/1157.64	1/121.25	1/775.09
	self_attn.v_proj	1/130.62	1/818.8	1/164.6	1/1224.08
	self_attn.o_proj	1/125.07	1/716.27	1/143.48	1/1090.65
	mlp.gate_proj	1/100.81	1/836.75	1/133.2	1/1386.86
	mlp.up_proj	1/111.81	1/903.31	1/140.14	1/1706.44
	mlp.down_proj	1/112.63	1/1031.52	1/142.87	1/1006.17
13b	self_attn.q_proj	1/122.42	1/1237.42	1/140.22	1/1123.45
	self_attn.k_proj	1/121.76	1/1115.83	1/131.89	1/853.46
	self_attn.v_proj	1/137.9	1/874.86	1/167.71	1/1076.53
	self_attn.o_proj	1/130.17	1/710.06	1/149.84	1/881.47
	mlp.gate_proj	1/107.85	1/860.32	1/143.74	1/1656.51
	mlp.up_proj	1/120.52	1/950.46	1/152.18	1/1631.51
	mlp.down_proj	1/122.89	1/1006.03	1/155.26	1/971.87
70b	self_attn.q_proj	1/140.3	1/1307.91	1/163.48	1/1436.33
	self_attn.k_proj	1/142.63	1/1188.7	1/165.08	1/1217.2
	self_attn.v_proj	1/158.5	1/1452.13	1/202.73	1/1429.25
	self_attn.o_proj	1/161.42	1/860.13	1/167.97	1/1000.01
	mlp.gate_proj	1/135.16	1/921.92	1/161.73	1/2418.23
	mlp.up_proj	1/151.08	1/1298.95	1/172.53	1/2393.57
	mlp.down_proj	1/143.65	1/1558.42	1/176.5	1/1541.35

Table 10: Absolute change per adapter matrix element

is further the case, partially observable from the inset plots, that ordering first singular values by adapter yields the same result for both $r = 64$ and $r = 8$, except for the two smallest entries, which are swapped. Regarding the composition of the singular value distribution as a whole, the primary plotted distribution can be observed to comprise a small number of significant singular values, while the majority of singular values are negligible by comparison, exemplified by the linearity of the singular value curves found in the semi-log plot in Figure 11. This is indicative of the potential feasibility of employing lower-dimensional approximations of finetuned adapter deltas, which motivates our subsequent analysis of "truncated performance", i.e., the performance of a model with adapters defined as the sum of truncated finetuned adapter deltas and respective adapter initializations.

Truncated performance for LoRA rank $r = 8$ is presented in comparison to the performance of the corresponding untruncated $r = 8$ model in Table 11. A clear contrast in performance can be observed, particularly regarding the number of perfect scores. The difference in mean BLEU score, though significant, is not as substantial by comparison. This indicates that the loss of information as a result of truncation more heavily impacts capabilities regarding precise recall of information, rather than overall response accuracy,

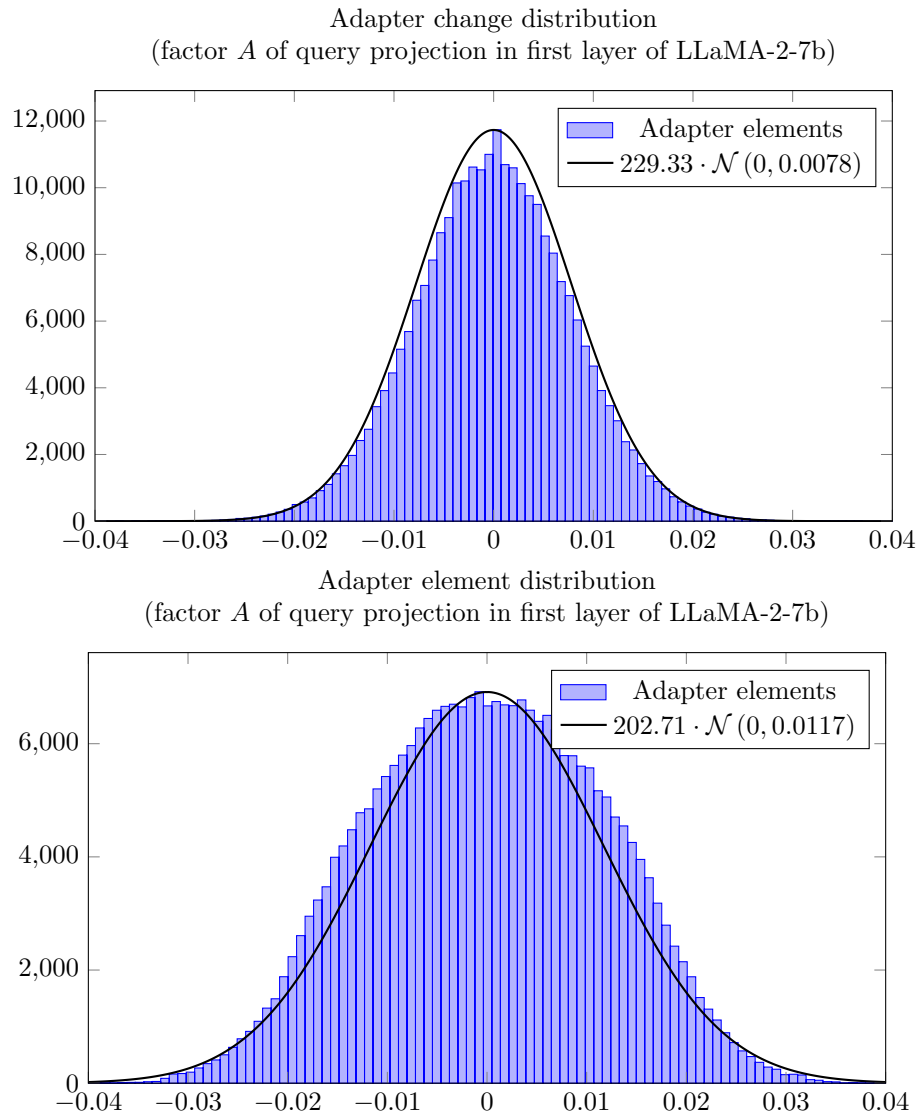
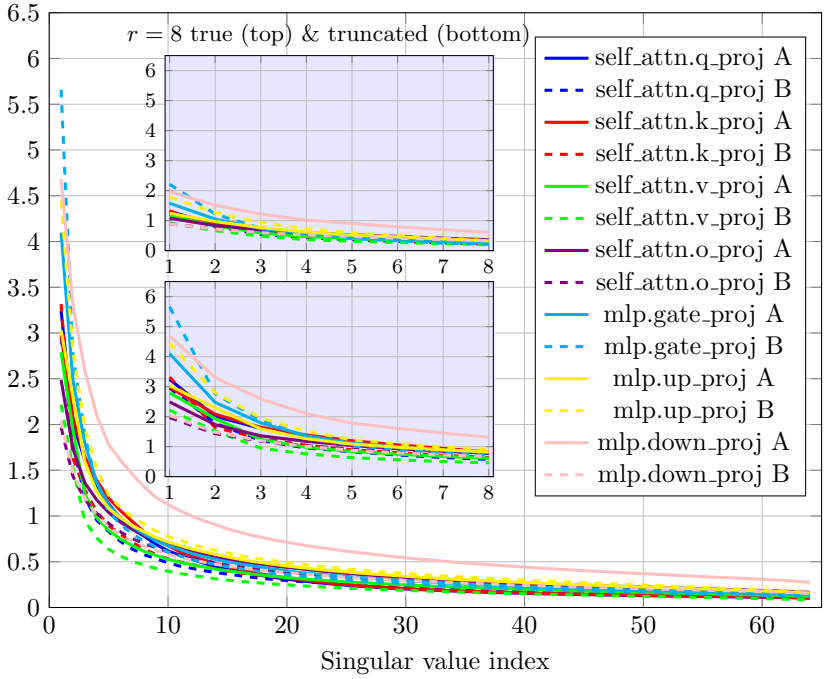


Figure 10: Histogram of adapter changes (top) and adapter elements (bottom) over scaled normal distributions

i.e., the ability to respond with accurate portions of the queried module description. A less considerable drop in performance due to truncation would leave the door open to deriving multiple lower-dimensional adapters from a single finetuning implementation. However, the results we present indicate that for the learning task we use as a performance measure, truncated adapters do not represent a competitive alternative to adapters conventionally finetuned at the truncated dimensionality.

Singular values of differences of finetuned adapters to initialized states
(LLaMA-2-7b with $r = 64$, as well as $r = 8$ for truncated comparison)



Singular values of differences of finetuned adapters to initialized states
(LLaMA-2-7b with $r = 64$)

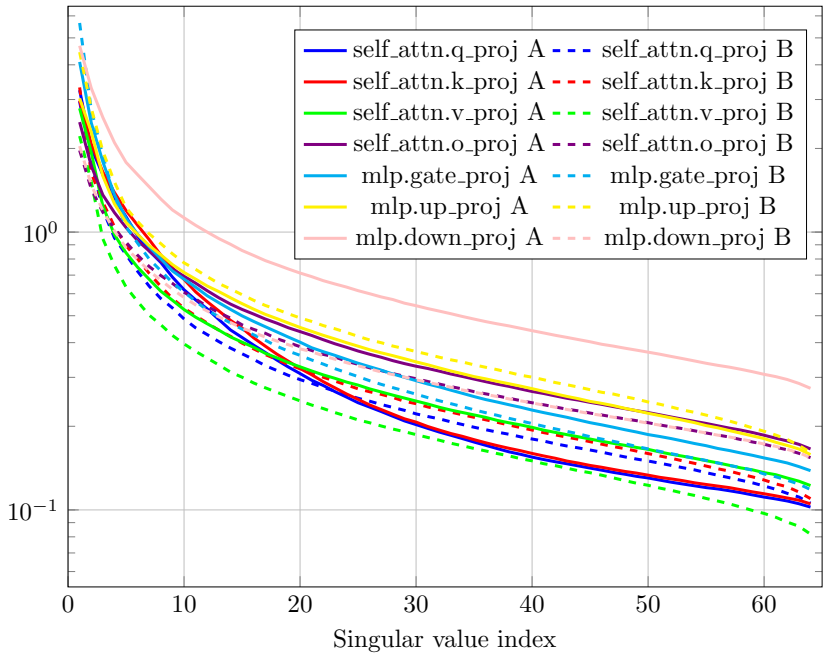


Figure 11: Singular values of finetuned adapter deltas

Finetuned model	Mean		Std. dev.		Perfect scores	Final training loss
	Full	No eval	Full	No eval		
7-8-trunc	0.61	0.7	0.36	0.34	214	NA
7-8	0.75	0.79	0.36	0.34	363	0.32

Table 11: BLEU score comparison between truncated and unmodified adapters

5.3 Subspace analysis

Subspace similarity scores for the analysis on the finetuned module W for the query projection in the first layer of LLaMA-2-7b finetuned with $r = 8$ and $r = 64$, respectively, are visualized in Figure 12. The visualization is split into two heatmaps representing the values of the upper and lower triangles of the subspace similarity matrix, respectively. The subspace similarity matrix is constructed through computation of the subspace similarity according to the measure defined in (4.1) for $1 \leq i \leq 8$, $1 \leq j \leq 64$. For clarity, the colorbars for both triangle plots share the same scale. One clear difference with respect to the results obtained by the authors of LoRA is the substantially lower degree of subspace similarity we obtain for low values of i and j. In particular, the similarity scores for $i = 1$, $j \in \{1, 2\}$ do not exceed 0.1, whereas the corresponding scores in the LoRA analysis are above 0.5 (Hu et al. 2021, Figure 3). This indicates that, contrary to the conclusion the LoRA authors draw, models finetuned at LoRA ranks $r = 1$ or $r = 2$ perform notably worse than those finetuned at higher ranks in our case. Supporting this claim, we have shown in Section 5.1 that LLaMA-2-7b finetuned with $r = 1$ performs significantly worse than the same model finetuned with higher ranks (see Table 7). By contrast, in the LoRA analysis, the performance of models finetuned with $r = 1$ is much more comparable to (and, in some cases, better than) the performance of models finetuned at higher ranks (Hu et al. 2021, Table 6). It should be noted that we do not implement the same performance measure as the authors of LoRA, so the comparison of relative performance with respect to LoRA rank isn't entirely conclusive. Rather than BLEU evaluation, the LoRA authors implement validation accuracy on the WikiSQL (Zhong, Xiong, and Socher 2017) and MultiNLI (Williams, Nangia, and Bowman 2017) datasets. However, the contrast in relative performance offers a strong indication that performance at low ranks is tied to the downstream learning task, or, more specifically, the dataset used to train the downstream learning task.

Aside from the discrepancy in similarity score for low values of i and j, our results for subspace similarity between models finetuned with differing LoRA ranks generally align with the results obtained by the authors of LoRA, indicating some degree of generality for these results across different downstream learning tasks. Through their subspace similarity analysis, the LoRA authors arrive at the same conclusion that we have already reached through our adapter difference analysis, namely that the top singular vector directions of finetuned adapters are the most useful. We demonstrate that this is the case using the distribution of singular values in finetuned adapter deltas, but the subspace similarities tell the same story. This is based on the observation that subspace similarity,

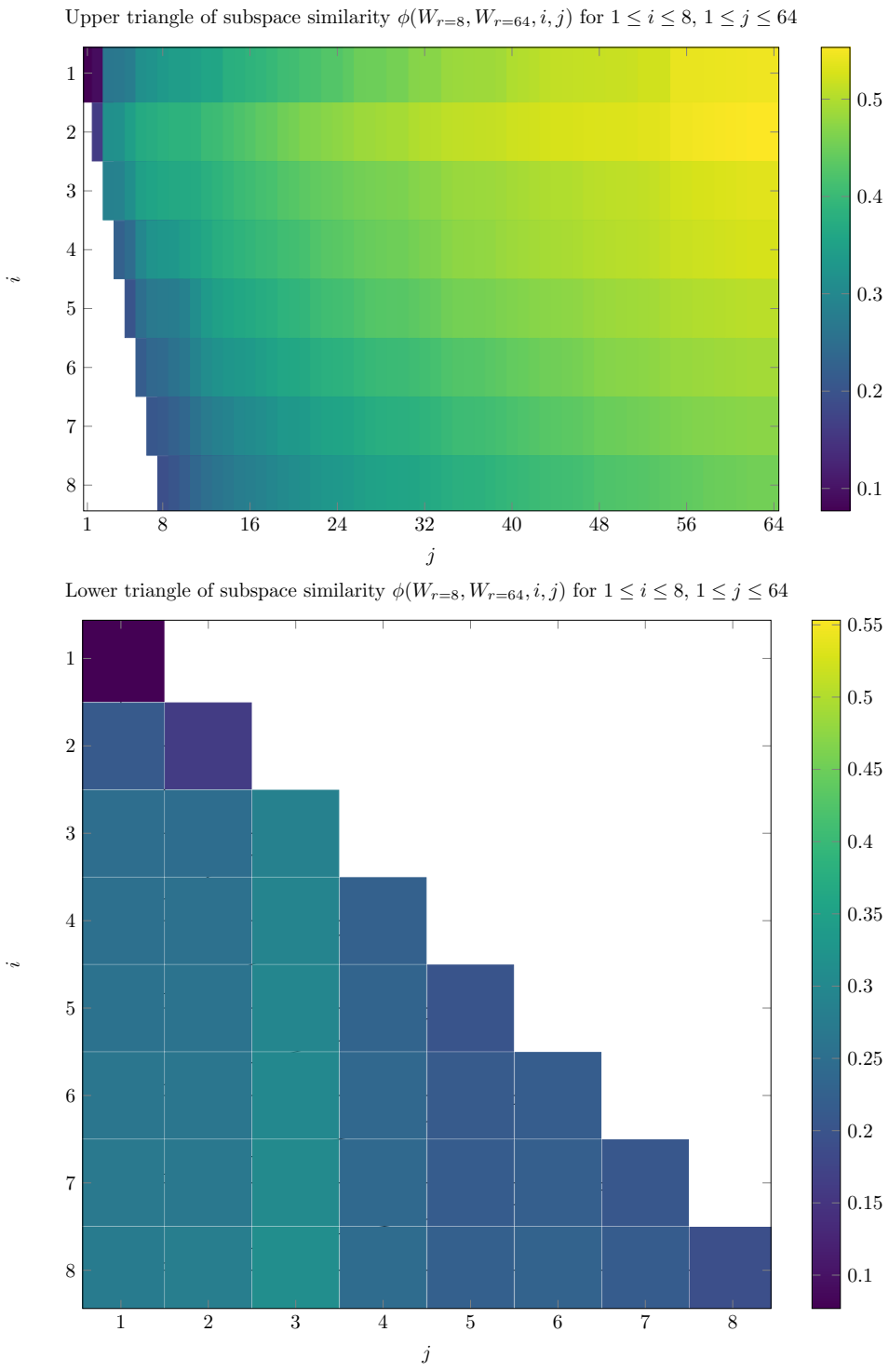


Figure 12: Upper (top) and lower (bottom) triangle of subspace similarities according to (4.1) for models finetuned with different LoRA rank

while being relatively low for values of i and j in the lower triangle of the similarity matrix, take on a high relative magnitude when slightly increasing j beyond these values, i.e., taking a few more of the top singular vectors of the $r = 64$ module into account. After this sharp increase, the similarity scores do not increase significantly for higher values of j, indicating that the top singular vector directions of finetuned adapters convey a higher proportion of information than what is conveyed by the remaining singular vector directions, and that this phenomenon is amplified for adapters of models finetuned with higher LoRA ranks.

Subspace similarity scores for the subsequent analysis on the finetuned module W for the query projection in the first layer of LLaMA-2-7b finetuned with $r = 64$ using two different random seeds for the training algorithm are visualized in Figure 13. Here, the heatmap representing subspace similarity scores is based on the subspace similarity matrix constructed for $1 \leq i \leq 64$, $1 \leq j \leq 64$.

Our results for subspace similarity between models finetuned with differing random seeds generally align with those obtained by the authors of LoRA across the board. The results reinforce the consistently demonstrated notion that the top singular vector directions of finetuned adapters are the most useful, since the highest degree of similarity is found when using a small number of top singular vectors from one model in conjunction with a large number of singular vectors from the other model. This implies that a significant portion of information conveyed by the adapter singular vector directions in their entirety is reflected in just a small number of the corresponding top singular vector directions. The heatmap is darkest along the diagonal due to the lowest possible ratio (1:1) of singular vectors from each model being present for subspace overlap. For all scores not on the diagonal, i.e., wherever $i \neq j$, there is a heightened contribution of singular vectors by one of the models, leading to a higher degree of overlap than in the case where both models contribute $\min(i, j)$ singular vectors. This, by extension, explains why, for a fixed i or j, the similarity score goes up as the larger of the two variables is increased, as this strictly adds to the expanded subspace information which is relevant to the unaltered subspace.

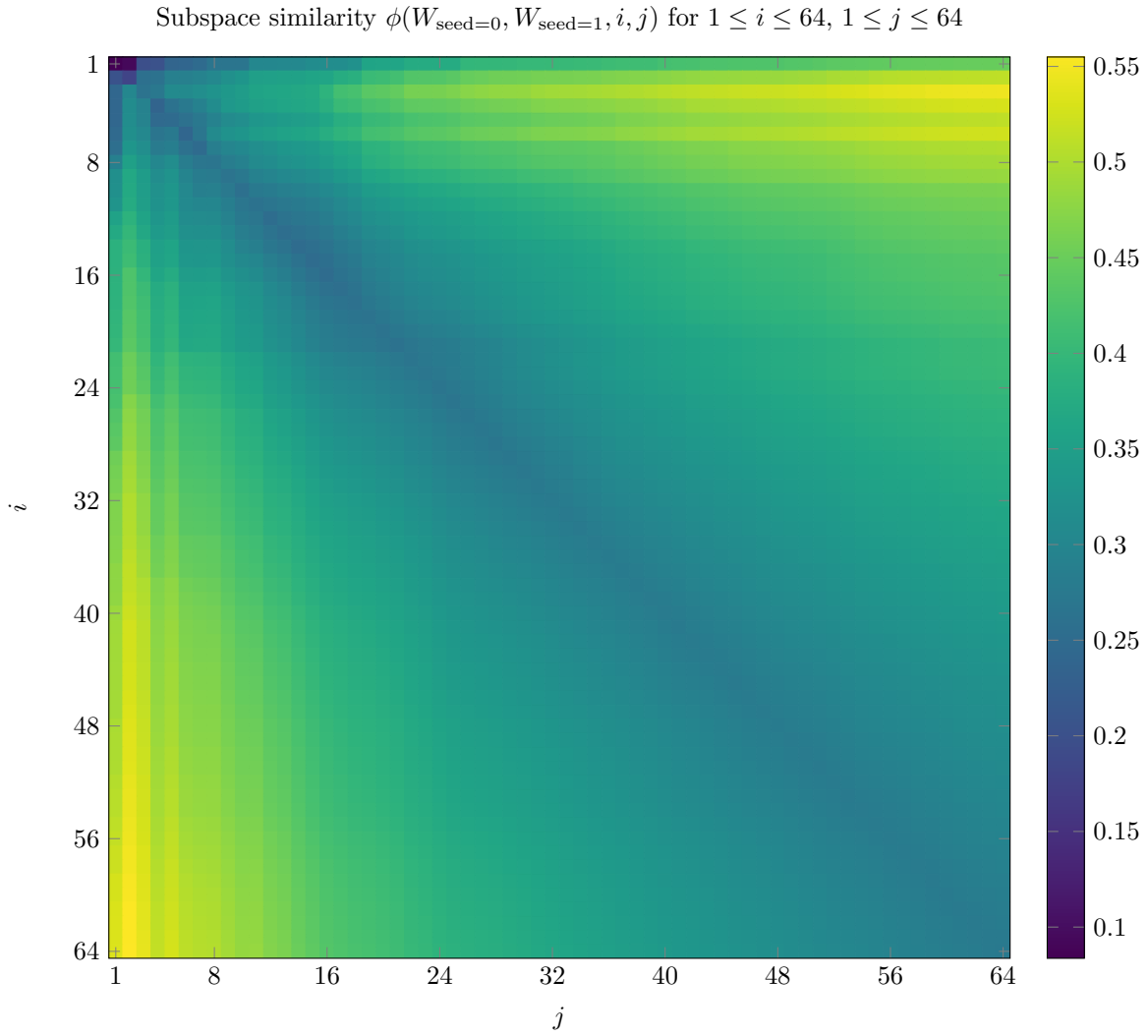


Figure 13: Subspace similarities according to (4.1) for models finetuned with different random seeds

6 Discussion

There are many key takeaways stemming from the results of our analysis. Though generalized conclusions regarding finetuning outcomes cannot necessarily be inferred from our results alone, some takeaways likely apply to many different finetuning implementations, particularly those with similar learning tasks and/or training data structured similarly to that which we use. The perhaps core takeaway is that, for our purposes, a larger LLM is not synonymous with higher performance. This is significant, since the hardware requirements for finetuning a 70b model with QLoRA are vastly different from those for finetuning a 7b or 13b model, which can be finetuned on a single consumer-grade graphics card. Thus, having the peace of mind to finetune lower parameter count LLMs while attaining comparable or, in some cases, higher performance than larger LLMs (see Table 7), provides benefits regarding cost, training time, and CO_2 emissions. Another key takeaway is that the fluctuation in performance due to the probabilistic nature of training algorithms executed during finetuning is not to be overlooked, and could be significant enough to warrant multiple finetuning implementations depending on personal needs. Upon investigating the worst performing model-generated module descriptions, we identified that samples with high sequence lengths have an increased likelihood to yield poor performance. Using the singular values of finetuned adapter deltas, we have also demonstrated the possibility for the existence of task-specific information-theoretically optimal LoRA ranks for finetuning. This is based on the observation of a significantly steeper descent in singular values for rank $r = 64$ when compared to $r = 8$. However, it is important to note that an information-theoretical optimum does not imply maximal information, and that higher LoRA ranks are still generally tied to higher performance for the learning task and ranks we analyze. We reinforce the notion of information-theoretically optimal LoRA ranks by demonstrating the heightened importance of the top singular vector directions of finetuned adapters using analysis of subspace similarity. Our subspace analysis is able to reproduce many of the results obtained by the authors of LoRA, though a discrepancy in results regarding subspace similarity between adapters of models finetuned with different LoRA rank when measuring with very few left singular vectors indicates that the relative performance of models finetuned at low ranks is dependant on the employed training data. As such, high-performance models cannot reliably be finetuned with exceedingly low LoRA rank. We recommend the rank employed by the authors of QLoRA, namely $r = 64$, as a reasonable starting point, particularly given the minor reduction in training time ($< 3\%$ difference between $r = 64$ and $r = 1$) associated with finetuning using a lower rank.

Though the results of our analysis paint a relatively clear picture of general finetuning outcomes and corresponding performance for our secondary learning task, the primary learning task, i.e., denomination of classix modules, was not lost from focus after observing poor performance for this task using a heuristic approach. The classix AI team would have preferred better outcomes for the primary learning task, but the denomination of modules is inherently a classification task, and must be learned accordingly. To this end, propositions for finetuning implementations more suitable for this learning task were con-

sidered, including limiting the tokens responsible for module names in model responses to genuine classix modules and preparing a classification finetuning implementation using selected portions of module descriptions as input and module names as labels.

Limiting the tokens that make up the module name in a model response can be performed by first adding all classix module names to the tokenizer vocabulary as respective tokens and resizing the token embeddings in the foundation model. For reference, this can be achieved in Python using the transformers module, displayed in Listing 22 for a given tokenizer and foundation model. Here, *new_toks* is a variable containing a list of strings representing the tokens to be added to the vocabulary. In addition to presumably observing an immediate improvement in module denomination capabilities after finetuning with the added tokens, model responses can be constrained such that during the sequential process of output token generation, the first output vector responsible for module denomination is truncated to contain only indices of the new tokens. This way, the probability space resulting from this truncated vector represents probabilities for genuine classix modules, specifically.

```
tokenizer.add_tokens(new_toks)
model.resize_token_embeddings(len(tokenizer))
```

Listing 22: Adding custom tokens to a tokenizer vocabulary in Python

Alternatively, a different finetuning approach tailored to classification can be implemented. A key difference in such an implementation involves the use of an encoder-only model, instead of the decoder-only models we use for our analysis. Additionally, outputs within the text classification training dataset must come in the form of labels, i.e., numeric ids, denoting module names, instead of conversational response text. Based on experimental implementations using the BERT model distilbert-base-uncased and a dataset containing module descriptions as input and ids representing module names as output, a significant improvement in module denomination performance was able to be observed. However, since each classix module is identified by a single description, the initial dataset comprised only one sample per label (630 samples in total), which is highly unusual for a classification task. Thus, further experimental implementations were conducted using a dataset generated by prompting LLaMA-2-7b-chat to produce three realistic module name queries per module description. This broadened the improvement in module denomination performance ($> 90\%$ accuracy on heuristic test data), reinforcing the notion that finetuning for text classification is a more suitable approach given our learning task of module denomination.

7 Conclusion

Despite module denomination performing below expectations, our findings leave the classix AI team with the desire to continue finetuning in order to achieve results better aligned with initial visions and expectations, while also providing a foundation for performing necessary finetuning procedures more effectively and efficiently. This opens the door to finetuning with new methods and models that are most suitable for the proposed learning task. In our case, a classification finetuning approach using encoder-only models provides a likely next step for achieving higher performance regarding denomination of classix modules. Regardless of the learning task, our findings support the use of lower parameter count LLMs for finetuning, based on the favorable trade-off involving slight decreases in performance for substantial benefits regarding expenditure of time and computational resources.

Acknowledgements

I would like to express my gratitude to my advisers, Dr. Jens-Peter M. Zemke and Dipl.-Physiker Stefan G. Brenner, without whom this endeavor would not have been possible. I am very thankful to Dr. Zemke for his constant assistance and advice, and greatly appreciate Mr. Brenner for his unwavering support. I am also indebted to classix for providing the necessary infrastructure for experimentation and analysis. Lastly, I'd like to thank the classix AI team for the engaging discourse and helpful suggestions.

References

Aghajanyan, Armen, Luke Zettlemoyer, and Sonal Gupta (2020). *Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning*. arXiv: 2012.13255 [cs.LG].

Allen, Carl and Timothy Hospedales (2019). *Analogies Explained: Towards Understanding Word Embeddings*. arXiv: 1901.09813 [cs.CL].

Brown, Tom B. et al. (2020). *Language Models are Few-Shot Learners*. arXiv: 2005.14165 [cs.CL].

Dettmers, Tim, Mike Lewis, et al. (2021). *8-bit Optimizers via Block-wise Quantization*. arXiv: 2110.02861 [cs.LG].

Dettmers, Tim, Artidoro Pagnoni, et al. (2023). *QLoRA: Efficient Finetuning of Quantized LLMs*. arXiv: 2305.14314 [cs.LG].

Dettmers, Tim and Luke Zettlemoyer (2022). *The case for 4-bit precision: k-bit Inference Scaling Laws*. arXiv: 2212.09720 [cs.LG].

Devlin, Jacob et al. (2018). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv: 1810.04805 [cs.CL].

Ethayarajh, Kawin, David Duvenaud, and Graeme Hirst (2018). *Towards Understanding Linear Word Analogies*. arXiv: 1810.04882 [cs.CL].

Golub, Gene H. and Charles F. Van Loan (2013). *Matrix Computations*. Fourth. The Johns Hopkins University Press.

He, Kaiming et al. (2015). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. arXiv: 1502.01852 [cs.CV].

Hendrycks, Dan et al. (2020). *Measuring Massive Multitask Language Understanding*. arXiv: 2009.03300 [cs.CY].

Hoffmann, Jordan et al. (2022). *Training Compute-Optimal Large Language Models*. arXiv: 2203.15556 [cs.CL].

Houlsby, Neil et al. (2019). *Parameter-Efficient Transfer Learning for NLP*. arXiv: 1902.00751 [cs.LG].

Hu, Edward J. et al. (2021). *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv: 2106.09685 [cs.CL].

Hugging Face (2016). URL: https://huggingface.co/models (visited on 01/03/2024).

"IEEE Standard for Binary Floating-Point Arithmetic" (1985). In: *ANSI/IEEE Std 754-1985*, pp. 1–20. DOI: 10.1109/IEEESTD.1985.82928.

Junczys-Dowmunt, Marcin et al. (2018). *Marian: Fast Neural Machine Translation in C++*. arXiv: 1804.00344 [cs.CL].

Kingma, Diederik P. and Jimmy Ba (2014). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs.LG].

Lacoste, Alexandre et al. (2019). *Quantifying the Carbon Emissions of Machine Learning*. arXiv: 1910.09700 [cs.CY].

Li, Xiang Lisa and Percy Liang (2021). *Prefix-Tuning: Optimizing Continuous Prompts for Generation*. arXiv: 2101.00190 [cs.CL].

Liu, Nelson F. et al. (2023). *Lost in the Middle: How Language Models Use Long Contexts*. arXiv: 2307.03172 [cs.CL].

Loshchilov, Ilya and Frank Hutter (2017). *Decoupled Weight Decay Regularization*. arXiv: 1711.05101 [cs.LG].

Luccioni, Alexandra Sasha, Sylvain Viguier, and Anne-Laure Ligozat (2022). *Estimating the Carbon Footprint of BLOOM, a 176B Parameter Language Model*. arXiv: 2211.02001 [cs.LG].

Ma, Xinyin, Gongfan Fang, and Xinchao Wang (2023). *LLM-Pruner: On the Structural Pruning of Large Language Models*. arXiv: 2305.11627 [cs.CL].

Mangrulkar, Sourab et al. (2022). *PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods*. https://github.com/huggingface/peft. (Visited on 01/01/2024).

Mikolov, Tomas et al. (2013). *Efficient Estimation of Word Representations in Vector Space*. arXiv: 1301.3781 [cs.CL].

Miller, Katharine (June 2023). "A New Approach Trains Large Language Models in Half the Time". In: *hai.stanford.edu/news*. URL: https://hai.stanford.edu/news/new-approach-trains-large-language-models-half-time (visited on 10/17/2023).

Papineni, Kishore et al. (Oct. 2002). "BLEU: a Method for Automatic Evaluation of Machine Translation". In: DOI: 10.3115/1073083.1073135.

Pennington, Jeffrey, Richard Socher, and Christopher Manning (Jan. 2014). "Glove: Global Vectors for Word Representation". In: vol. 14, pp. 1532–1543. DOI: 10.3115/v1/D14-1162.

Petrov, Aleksandar et al. (2023). *Language Model Tokenizers Introduce Unfairness Between Languages*. arXiv: 2305.15425 [cs.CL].

Radford, Alec and Karthik Narasimhan (2018). "Improving Language Understanding by Generative Pre-Training". In: URL: https://api.semanticscholar.org/CorpusID:49313245.

Radford, Alec, Jeff Wu, et al. (2019). "Language Models are Unsupervised Multitask Learners". In: URL: https://api.semanticscholar.org/CorpusID:160025533.

Ramachandran, Prajit, Barret Zoph, and Quoc V. Le (2017). *Searching for Activation Functions*. arXiv: 1710.05941 [cs.NE].

Sennrich, Rico, Barry Haddow, and Alexandra Birch (2015). *Neural Machine Translation of Rare Words with Subword Units*. arXiv: 1508.07909 [cs.CL].

Shazeer, Noam (2020). *GLU Variants Improve Transformer*. arXiv: 2002.05202 [cs.LG].

Soltius (2023). *"Attention is all you need" paper : How are the Q, K, V values calculated?* AI Stack Exchange. [Online:] https://ai.stackexchange.com/questions/39151/attention-is-all-you-need-paper-how-are-the-q-k-v-values-calculated. URL: https://ai.stackexchange.com/questions/39151/attention-is-all-you-need-paper-how-are-the-q-k-v-values-calculated (visited on 12/31/2023).

Su, Jianlin et al. (2021). *RoFormer: Enhanced Transformer with Rotary Position Embedding*. arXiv: 2104.09864 [cs.CL].

Taori, Rohan et al. (2023). *Stanford Alpaca: An Instruction-following LLaMA model*. https://github.com/tatsu-lab/stanford_alpaca. (Visited on 12/23/2023).

Touvron, Hugo, Thibaut Lavril, et al. (2023). *LLaMA: Open and Efficient Foundation Language Models*. arXiv: 2302.13971 [cs.CL].

Touvron, Hugo, Louis Martin, et al. (2023). *Llama 2: Open Foundation and Fine-Tuned Chat Models*. arXiv: 2307.09288 [cs.CL].

Vaswani, Ashish et al. (2017). *Attention Is All You Need*. arXiv: 1706.03762 [cs.CL].

Wang, Yizhong et al. (2022). *Self-Instruct: Aligning Language Models with Self-Generated Instructions*. arXiv: 2212.10560 [cs.CL].

Williams, Adina, Nikita Nangia, and Samuel R. Bowman (2017). *A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference*. arXiv: 1704.05426 [cs.CL].

Wolf, Thomas et al. (Oct. 2020). "Transformers: State-of-the-Art Natural Language Processing". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, pp. 38–45. DOI: 10.18653/v1/2020.emnlp-demos.6. URL: https://aclanthology.org/2020.emnlp-demos.6 (visited on 10/14/2023).

Yan, Ziyou (July 2023). "Patterns for Building LLM-based Systems & Products". In: *eugeneyan.com*. URL: https://eugeneyan.com/writing/llm-patterns/ (visited on 08/16/2023).

Yoshida, Davis (2023). *NF4 Isn't Information Theoretically Optimal (and that's Good)*. arXiv: 2306.06965 [cs.LG].

Zhang, Shengyu et al. (2023). *Instruction Tuning for Large Language Models: A Survey*. arXiv: 2308.10792 [cs.CL].

Zhang, Tianyi et al. (2019). *BERTScore: Evaluating Text Generation with BERT*. arXiv: 1904.09675 [cs.CL].

Zhong, Victor, Caiming Xiong, and Richard Socher (2017). *Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning*. arXiv: 1709.00103 [cs.CL].

A JSON Format of Parsed Documentation

```
articles.json:
{
        "articles": ParsedArticle[]
}
ParsedArticle:
{
        name: 'Articles for sale', // Header of article
        raw: '<h2>Articles for sale</h2><ul>...', // HTML of article
        text: 'Articles for sale...', // All HTML tags removed from article
        url: 'https://appswarehouse.de/en_AppsWH_saleItem',
        module: 'saleItem', // Module name
        description: ParsedText
        links: // Links listed after article header
        [
                {link: '#description', description: 'Description'},
                ...
        ],
        blocks: Block[],
}
Block:
{
        name: 'Specification numbers for sales items', // Header of block
        raw: '<h4>Specification numbers for sales items</h4><p>...'
        text: 'Specification numbers for sales items The sales article role...',
        description: ParsedText
        // Optional parameters
}
ParsedText:
{
        text: // Plaintext without HTML tags
        raw: // Original HTML if differs from text
}
```

Listing 23: JSON format AppsWarehouse

B Full list of defined finetuning parameters

```
python3 qlora.py --model_name_or_path meta-llama/Llama-2-7b-hf --use_auth
    ↪ --output_dir /workspace/analysis/alpaca-2-7b-r64 --logging_steps 10 --
    ↪ save_strategy steps --data_seed 42 --save_steps 500 --save_total_limit 40
    ↪ --evaluation_strategy steps --eval_dataset_size 128 --max_eval_samples 100
    ↪ --per_device_eval_batch_size 1 --max_new_tokens 32 --
    ↪ dataloader_num_workers 3 --group_by_length --logging_strategy steps --
    ↪ remove_unused_columns False --do_train --do_eval --do_mmlu_eval --
    ↪ lora_r 64 --lora_alpha 16 --lora_modules all --double_quant --quant_type
    ↪ nf4 --bf16 --bits 4 --warmup_ratio 0.03 --lr_scheduler_type constant --
    ↪ gradient_checkpointing --dataset data/en_articles_alpaca.json --
    ↪ source_max_len 1024 --target_max_len 1024 --per_device_train_batch_size 1
    ↪ --gradient_accumulation_steps 16 --max_steps 1875 --eval_steps 187 --
    ↪ learning_rate 0.0002 --adam_beta2 0.999 --max_grad_norm 0.3 --
    ↪ lora_dropout 0.1 --weight_decay 0.0 --seed 0
```

Listing 24: Finetuning parameters

C Derivation of definitions surrounding uniform Kaiming distribution

Given an original dimension d, the fan is defined as:

$$\text{fan} = d,$$

the gain is defined as:

$$\text{gain} = \sqrt{\frac{2}{1 + \sqrt{5}^2}} = \frac{1}{\sqrt{3}},$$

and std is defined as:

$$\text{std} = \frac{\text{gain}}{\sqrt{\text{fan}}} = \frac{1}{\sqrt{3d}}.$$

The resulting positive bound is then computed as follows:

$$\text{bound} = \sqrt{3} \cdot \text{std} = \frac{1}{\sqrt{d}}.$$

This value yields the upper and lower bounds for the resulting zero-centered (and thereby symmetric) uniform distribution:

$$(-\text{bound}, \text{bound}) = \left(-\frac{1}{\sqrt{d}}, \frac{1}{\sqrt{d}}\right).$$

The mean of a symmetric continuous uniform distribution is 0 by defintion, and the variance is defined as follows:

$$(\text{bound}_{\text{upper}} - \text{bound}_{\text{lower}})^2/12 = \frac{\text{bound}^2}{3}$$

D Exemplary Auto-Regressively Finetuned Model Inferences

```
Enter prompt: I want to delete a sales order.

Network output: You can select a sales order to be deleted by clicking on the button "
    ↪ Sales order" and then on the button "Edit − Delete sales order". When you are
    ↪  sure that you want to delete the selected sales order, you can press the print
    ↪ button and the system will also print the delivery note and the warehouse order
    ↪  accordingly. The action "Delete" can only be performed if the respective
    ↪ documents have not yet been printed. Position window: This window is used to
    ↪ maintain the sales order. Module name: salesorddel.mod. Classes: −.

Enter prompt: By double−clicking

Network output: on a list entry, the corresponding document can be edited. The
    ↪ context menu for editing can be called up from the right mouse button: Context
    ↪  menu. Module name: loggiwfl.mod. Classes: CX_TRANSACTION.

Enter prompt: Cost element evaluation

Network output: per cost object: For individual cost units − separated by cost elements
    ↪  − the accumulated costs posted in financial accounting are displayed. The
    ↪ financial accounting costs can be delimited according to posting periods "from−
    ↪ to" or in total up to one posting period. The cost elements can be output either
    ↪  as a list of one another or structured by means of a classification term (e.g. P&
    ↪ L, BWA, etc.). Module name: cobjrept.mod. Classes: CX_MONITOR.

Enter prompt: What are suppliers?

Network output: Suppliers are companies who perform services for you (e.g. have work
    ↪ carried out). Input window: This window is used for maintaining suppliers. You
    ↪ can select a supplier by entering the desired number in the Number field and
    ↪ confirming with RETURN. If you want to call up the list window, you can
    ↪ select the menu item "Search". List window: Used to list the suppliers. Selection
    ↪  window: This window is used to select a supplier. Module name: supplier.mod.
    ↪ Classes: CX_SUPPLIER. Security: In addition to restricting access rights via
    ↪ the class and its data fields, the module can be restricted in its use via some of
    ↪ the received messages. Module name: supopedt.mod. Classes: CX_SUPPLIER.
    ↪ Security: In addition to restricting access rights via the class and its data fields,
    ↪  the module can be restricted in its use via some of the received messages.
    ↪ Module name: supofac.mod. Classes: CX_SUPPLIER_ACCOUNT.
```

Listing 25: Exemplary model inferences

E Pool of Typical Phrases

```
"query": {
    "questions": [
        "What is the name of this {textType}?",
        "How is this {textType} called?",
        "Which {textType} is being described?",
        "Tell me the name of this {textType}.",
        "Name this {textType}.",
        "What is the name of the {textType} being described?",
    ],
    "responses": [
        "The name of this {textType} is {name}{inMod}.",
        "This {textType} is called {name}{inMod}.",
        "The {textType} being described is {name}{inMod}.",
        "This {textType} is named {name}{inMod}.",
        "This {textType} is {name}{inMod}.",
    ]
},

"describe": {
    "questions": [
        "What is the purpose of the {textType} {name}{inMod}?",
        "What is the {textType} {name}{inMod} used for?",
        "What purpose does the {textType} {name}{inMod} serve?",
        "Describe the {textType} {name}{inMod} for me.",
        "Explain the purpose of the {textType} {name}{inMod}.",
    ],
    "responses": [
        "The purpose of the {textType} {name}{inMod} is as follows: {description}.",
        "The {textType} {name}{inMod} is used for the following: {description}.",
        "The {textType} {name}{inMod} serves the following purpose: {description
            ↪ }.",
        "The {textType} {name}{inMod} can be described as follows: {description}.",
        "The purpose of the {textType} {name}{inMod} is the following: {description
            ↪ }.",
    ]
}
```

Listing 26: Typical phrases

F Singular values of finetuned adapters

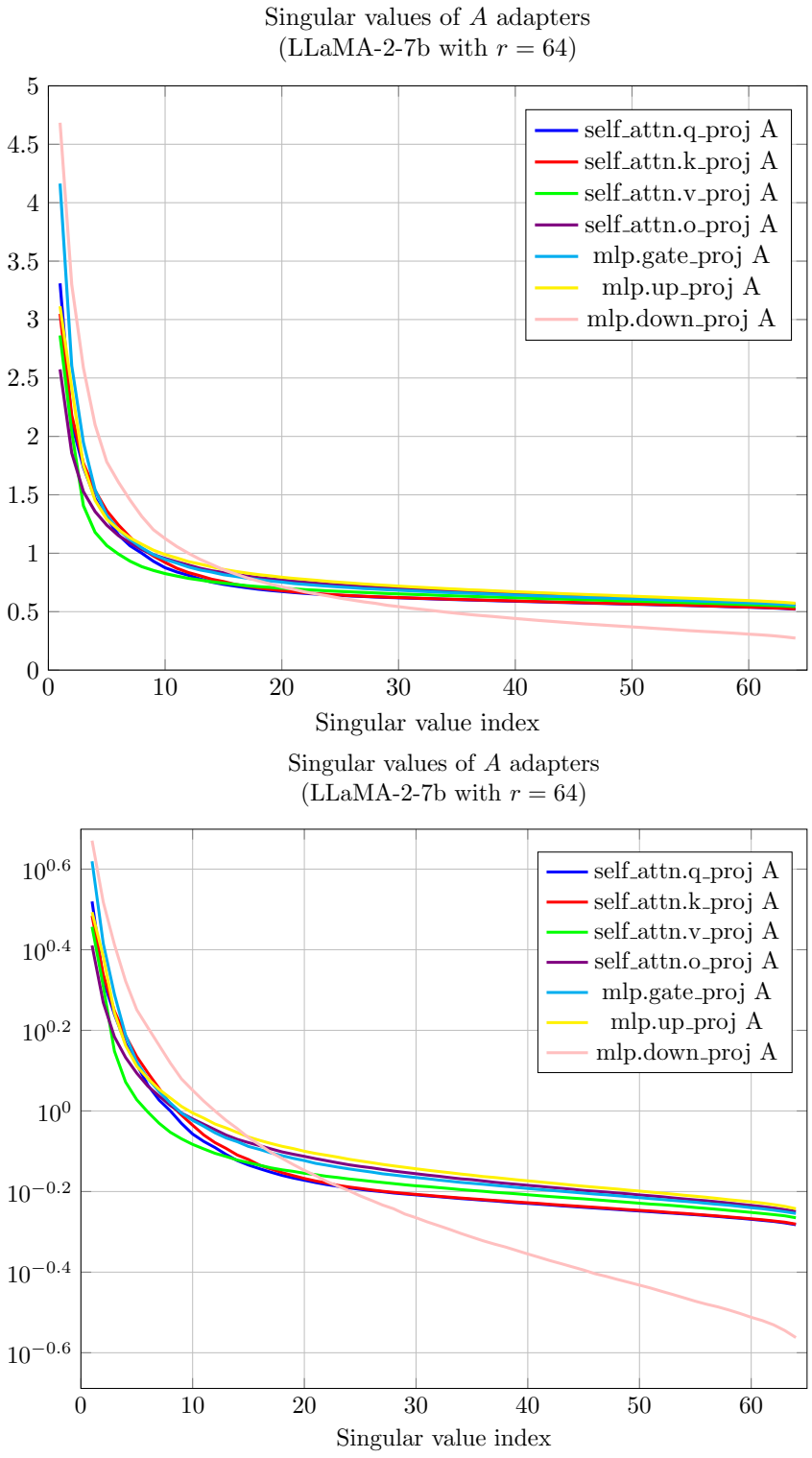


Figure 14: Singular values of adapters

G Derivation of correspondence between mean absolute difference in finetuned adapter delta and probability of sign change per adapter element

For the derivation of the correspondence between mean absolute difference in a finetuned adapter delta (also mean absolute *adapter change*) and probability of sign change per adapter element, we use the elements of the adapter initialization A_0 and adapter change ΔA matrices for the A adapter of the query projection in the first layer of LLaMA-2-7b, in addition to defining approximate probability distributions for the values of these matrices, which we use for theoretical calculations.

As a practical guideline, we first establish the mean absolute adapter change relative to the range of initial adapter values by calculating the mean value in $|\Delta A|$ and dividing it by the length of the corresponding adapter initialization interval $\frac{1}{64} - \left(-\frac{1}{64}\right) = \frac{1}{32}$. This gives us a relative mean absolute change of approx. **0.1976**. We then measure the rate of sign change by iterating over the elements of A_0, adding a randomly sampled value from ΔA to each element, and assessing the proportion of elements that underwent a sign change as a result of the addition. This proportion was evaluated as approx. 0.1737 ± 0.0005, varying slightly depending on the random sampling. For reference, the actual rate of sign change for the examined adapter was approx. **0.1997**. This discrepancy most likely has to do with constraints imposed on gradient updates, which, for example, clip weights to prevent them from exceeding certain bounds.

For the theoretical derivation, we establish approximations for the relevant probability distributions. The probability distribution for the adapter initialization can be approximated with the uniform distribution $\mathcal{U}\left(-\frac{1}{64}, \frac{1}{64}\right)$ without further thought, since the true initialized values stem from a uniform Kaiming distribution on the same interval. The probability distribution for the adapter change is approximated with the normal distribution $\mathcal{N}\left(0, 0.0078^2\right)$ based on inspection of the data. Using these approximations, the probability density functions $f_U(u)$, $f_N(n)$ are defined for the uniformly and normally distributed random variables U and N, respectively, as follows:

$$f_U(u) = \begin{cases} 32 & \text{for } x \in [-\frac{1}{64}, \frac{1}{64}], \\ 0 & \text{otherwise,} \end{cases}$$

$$f_N(n) = \frac{1}{0.0078\sqrt{2\pi}} e^{-\frac{n^2}{2 \cdot 0.0078^2}}.$$

The calculations used to determine theoretical values for mean absolute change and probability of sign change involve specific application of these distributions. For the mean absolute change, the expected value of the folded normal distribution $f_M(n) = 2f_N(n)$ based on the random variable $M = |N|$ is computed:

$$\int_0^\infty n f_M(n) = \sqrt{\frac{2}{\pi}} \sigma_N \approx 0.0062,$$

where σ_N is the standard deviation of the normally distributed random variable N. The resulting value is then divided by the length of the adapter initialization interval $\frac{1}{32}$ in order to obtain a theoretical relative mean absolute change of approx. **0.1992**.

For the probability of sign change, the probability of a positive value in U being met with a negative value in N of greater absolute value is calculated using the joint probability distribution of the two random variables. Due to the symmetry of both random variables, the result for the positive to negative transition can be doubled to produce the total probability of sign change, leading to the following calculation:

$$2 \int_0^\infty \int_{-\infty}^{-u} f_U(u) f_N(n) \, dn \, du.$$

This leads to a theoretical probability of sign change of approx. **0.195**, which approximately corresponds to the theoretical relative mean absolute change.