Institut für Maschinelle Sprachverarbeitung
Universität Stuttgart
Pfaffenwaldring 5B
D-70569 Stuttgart

Master thesis

Just-in-time Pruning of Large Prompted Language Models

Patrick Bareiß

Studiengang:	M.Sc. Computational Linguistics
Prüfer*innen:	Prof. Dr. Roman Klinger
	Prof. Dr. Sebastian Padó
Betreuer:	Prof. Dr. Roman Klinger
Beginn der Arbeit:	03.11.2023
Ende der Arbeit:	03.05.2024

Erklärung (Statement of Authorship)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und dabei keine andere als die angegebene Literatur verwendet habe. Alle Zitate und sinngemäßen Entlehnungen sind als solche unter genauer Angabe der Quelle gekennzeichnet. Die eingereichte Arbeit ist weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen. Sie ist weder vollständig noch in Teilen bereits veröffentlicht. Die beigefügte elektronische Version stimmt mit dem Druckexemplar überein.[1]

(Patrick Bareiß)

[1] Non-binding translation for convenience: This thesis is the result of my own independent work, and any material from work of others which is used either verbatim or indirectly in the text is credited to the author including details about the exact source in the text. This work has not been part of any other previous examination, neither completely nor in parts. It has neither completely nor partially been published before. The submitted electronic version is identical to this print version.

Abstract

Prompting of transformer-based autoregressive large language models (LLMs) is a powerful and ergonomic approach to solve language processing tasks (sentiment analysis, question answering, ...) using few data. However for each individual task/prompt it is also wasteful: Only a fraction of the generality in the underlying model is ever used. This raises the question: Is this also reflected in the model in the form of irrelevant model parts that do not affect the task-specific accuracy when pruned away (for instance layers)? Previous work does not address this question at a level that preserves the natural affordances of prompting: (1) Low data requirements and (2) ability to reuse the same deployed model for multiple tasks. We propose a new approach that can identify and remove irrelevant model parts if they exist, which does not require additional data (we generate it instead) and removes irrelevant parts only just before a prompt gets passed as input to the model, or just-in-time. After the prompt has been processed we re-add the removed parts to the model. In this way, we can then reuse the model and remove (potentially different) irrelevant model parts for another prompt. We identify a class of model parts for which pruning/re-adding is efficient and therefore allows for efficient just-in-time pruning. During our experiments we find that irrelevant just-in-time prunable model parts do exist for many prompts (for Mistral-7B and GPT-2-XL) and we can remove them to a substantial degree, reducing FLOPs by up to 46% while preserving the accuracy of the original model.

Acknowledgements

I would like to thank my advisor Professor Roman Klinger for helping me along this journey. Without his continued advice and support this thesis would not have been possible. Gratitude also goes out to all my friends and family who picked me back up when I felt stuck and cheered me on when things finally worked. No matter the place, no matter the time, you were always there for me. From the bottom of my heart: Thank you.

Contents

1 Introduction **9**

2 Background **15**

2.1 Neural Networks 15

2.1.1 History 15

2.1.2 Artificial Neurons 17

2.1.3 Single Layer Neural Networks 19

2.1.4 Limitations of Single Layer Neural Networks 19

2.1.5 Deep Neural Networks 20

2.1.6 Training 21

2.2 Transformer-based Language Models 27

2.2.1 Overview 27

2.2.2 High-level Architecture and Usage 28

2.2.3 Low-level Architecture 29

2.2.4 Training 31

2.2.5 Role as Foundation Models 32

2.3 Prompting using Autoregressive Language Models 32

2.3.1 Overview 32

2.3.2 Basic Approach 33

2.3.3 Advanced Prompting 35
2.4 Model Pruning 37
2.4.1 Overview 37
2.4.2 Common Components of Pruning Approaches 38
2.4.3 Taxonomy of Pruning Approaches 41

3 Related Work **45**
3.1 Overview 45
3.2 Pruning Transformers 45
3.3 Data-free Knowledge Distillation 46
3.4 Scaling-based Model Repair 47

4 Methods **49**
4.1 Overview 49
4.2 Choice of Model Part Types to Prune 51
4.3 Pruning Process 53
4.3.1 Provide Calibration Data 53
4.3.2 Assess Importance 54
4.3.3 Remove the Least Important Part 55
4.3.4 Repair 55

5 Implementation Details **57**
5.1 Overview 57
5.1.1 Model Implementation and Pruning 57
5.1.2 Evaluation 59
5.1.3 Measuring FLOPs 59

6 Experiments **61**

6.1 Overview . . . 61

6.2 Task, Prompts and Models . . . 62

6.2.1 Tasks . . . 62

6.2.2 Prompts . . . 64

6.2.3 Models . . . 66

6.3 RQ1: Existence of Sparsity . . . 67

6.3.1 Overview . . . 67

6.3.2 Results . . . 67

6.3.3 Analysis . . . 69

6.4 RQ 2: Identification of Sparsity . . . 76

6.4.1 Overview . . . 76

6.4.2 Results . . . 77

6.4.3 Analysis . . . 78

6.5 RQ 3: Repair of Pruned Models . . . 80

6.5.1 Overview . . . 80

6.5.2 Results . . . 81

6.5.3 Analysis . . . 82

6.6 RQ4: Speedup Achieved . . . 83

6.6.1 Overview . . . 83

6.6.2 Results . . . 83

6.6.3 Analysis . . . 84

7 Discussion and Conclusion **87**

A Results of Base Models **99**

Chapter 1

Introduction

Autoregressive large language models (LLMs) (Brown et al., 2020; Touvron et al., 2023; Jiang et al., 2023) are a type of machine learning model that acts as a universal text prediction machine. Given the beginning of a text, they guess what comes next. To do so, LLMs are trained on large, diverse corpora commonly scraped from the internet (Radford et al., 2019). These texts can include tweets, books, forum entries and many more types of data. In doing so LLMs learn useful general skills that help it make accurate predictions on how the text will continue. These include the assessment of sentiment (for instance to help with the prediction of a reply to an angry tweet), translation between languages (to help with prediction of multilingual text) or memorization of world knowledge (to help with prediction of news articles or historical texts) among many others (Brown et al., 2020). This makes this kind of model an attractive base to build upon when trying to solve one of these particular tasks. By further training an LLM on small amounts of additional labelled task-specific data, we can elicit this latent task knowledge already present in the model and bring it "to the surface" (known as *fine-tuning*) (Radford et al., 2019). However, more recently another approach has emerged that requires even fewer or no data at all to do so (a setting known as few- or zero-shot learning) and has therefore become a major paradigm of LLM usage: *prompting* (Brown et al., 2020).

Instead of modifying the LLM directly, prompting works by giving the model a carefully crafted input called the *prompt* instead. The prompt is designed in such a way that performing the pretraining task, e.g., predicting the next token after the prompt addresses (inadvertently) a specific task such as sentiment analysis. For instance, given the prompt

"Sentence: This movie was great! Sentiment:", a model might predict "positive" as the next word and thus also determine the text's sentiment. To address the sentiment of many different texts, a *prompt template*[1] is used, i.e., a prompt where the text is spliced in. In this example, the prompt template is: "Sentence: <text goes here> Sentiment:". A prompt template can also contain some similarly formatted input-output examples for the task, which empirically improve performance (Brown et al., 2020), however far fewer are necessary than for fine-tuning (rather: on the order of 0 to 10 examples).

But while convenient, prompting is also wasteful: For most tasks the full generality of an LLM is not required. For instance, a model need not be aware that Paris is the capital of France to determine the sentiment of "This movie is great". In some sense, while we as the task-giver know that certain text completions are a priori irrelevant for our task and do not have to be considered, the model does not and therefore has to take these possibilities into account. Assuming a form of locality exists with respect to where the model derives certain features relevant for predicting some type of text (Zhang et al., 2023b), this means that there could be model parts (such as neurons or entire layers) that are irrelevant to a task/prompt template and we may ignore them for this specific setting without affecting the task-specific accuracy. Put another way, it seems plausible that LLMs are sparse[2] in some parts of the model with respect to a specific task given as a prompt template.

The field that aims to answer the general question of finding and removing irrelevant model parts, i.e., sparsity, under different sets of constraints in neural networks is known as model pruning. In many settings, it is possible to identify and remove irrelevant parts of a model for a specific task. Task-specific model pruning (Michel et al., 2019; Peer et al., 2022) often works by assigning an importance score for each model part and then removing parts with a low score and (optionally) adjusting the remaining parts to restore performance.

Applying the techniques from model pruning to the prompting setting using LLMs still faces three open challenges:

1. **Existing work focuses on a prompt-agnostic setting** Bansal et al. (2023) already explore task-specific pruning in a prompt-based setting and find that around 70% sparsity with respect to attention heads and 20% sparsity with respect to feed-forward blocks in the model OPT-66B Zhang et al. (2022) is possible. However, they

[1] Often the definitions for prompt template and prompt are conflated (like in our Abstract). For the remainder of this work we will primarily be talking about prompt templates.

[2] We say that x is sparse in y if we can remove y from x without affecting x.

address this issue from a prompt template-agnostic lens, i.e., they prune under the assumption that a different prompt template (with different examples) is used for each instance solved for a given task. This does not match few-shot learning in a practical setting where the prompt template does not change when addressing different instances of a task (only what is inserted in the template). It could be, for instance, that some prompt templates allow much higher sparsity than others for the same task.

2. **Data is scarce** One of the main advantages of a prompt-based setting is that the data requirements to address some task are low as only a few to none examples are required for reasonable performance (Brown et al., 2020). However, most pruning approaches require access to large amounts of data to accurately prune the model. This means that while demonstrating the *existence* of prompt-specific sparsity might be possible using additional data, it is unclear if *identification* in a realistic setting is feasible, i.e., only using the model and the prompt template itself.

3. **Parts of the model should be removable just-in-time** Another major upside of prompt-based learning is that the same deployed model can solve multiple different tasks without requiring the instantiating of multiple versions (as would be necessary for fine-tuning). This *allows* the same model to be used by many different users with different tasks and prompts at the same time, i.e., only one physical model needs to be deployed on actual hardware. In fact this is often also *necessary* as modern LLMs are large and expensive to run, making the deployment of many different versions of the same base model difficult (Huang et al., 2022). A pruning approach in this setting should therefore preserve this property. As such, we ideally want to prune a model just before the instance of a specific task is passed as input in form of a prompt. And after the inference has finished, we want to return the model to its original unpruned state so that we can prune potentially different parts for the next request (as the model might be sparse in different model parts for another prompt template). In this manner, we can reuse the same deployed model and yet ignore irrelevant model parts for each individual inference. We call the application of pruning in this manner *just-in-time*. While in principle this can be done for many types of model parts, for some this is easier than for others. For instance, removing individual neurons from a layer in a neural network typically implies having to create an entirely new matrix with a row removed, which causes significant overhead to store the original. We call

a type of model part *just-in-time prunable* if it allows for a mechanism to remove it (and re-add it) without substantial overhead in this setting.[3]

In this work we aim to answer whether prompt template specific sparsity (addressing (1)) at the level of just-in-time prunable model parts (addressing (3)) exists in large language models, as well as understand if it can be *identified* based on the information contained in the prompt template (addressing (2)).

We aim to answer whether prompt template specific sparsity *exists* by employing an importance-based pruning approach that uses real task data in conjunction with the prompt template to identify irrelevant model parts. Then we examine if this sparsity can be *identified* using just the model and a prompt template. We hypothesize that this is possible by employing the same approach used to show existence but replacing the real task data with data generated using the model prompted on the prompt template. Underlying this is the assumption that the language model learns to represent the distribution of task data well enough, not only to classify it, e.g, determine the sentiment but also to produce new data points on its own, e.g., write an angry tweet. Further, we investigate if *repair* of a model that is already pruned is possible whose accuracy on a task has deteriorated. We hypothesize that even if performance is bad for the pruned model, maybe this is not because some parts of the model essential for solving a task have been removed, but instead that it is an issue of interaction between the remaining model parts. As an analogy, imagine removing parts of the walls in a house. By itself the house will collapse but it might be possible to stabilize what is still there using a bit of mortar. Similarly, we test if simply scaling the output features of the remaining model parts suffices to fix the interaction. Finally, we compare the sparsity found for the different types of model parts we examine to identify which one leads to the biggest realistic *speedup* while still preserving the performance of the unpruned model. This is because for us acceleration is the main motivation for doing just-in-time pruning to begin with.

In summary, we study the four aspects: Existence, Identifiability, Repair and Speedup. Note that for the first three we always examine the respective aspect under additional assumptions. Namely, for existence it is in general computationally hard to find maximal

[3]Of course just-in-time prunable model parts can still be pruned permanently from a model as well (for instance to save memory). In that sense just-in-time pruning is strictly more powerful than regular pruning.

sparsity if it exists (Molchanov et al., 2017). Therefore, we use an approximate approach that is well motivated by prior work and shown to be close to optimal. For Identifiability, we specifically test if a natural approach of incorporating information in the model and the prompt template in the form of generated data is sufficient. And for Repair, we focus on a weak technique (few tunable parameters). In case repair of the model is possible like this, we can assume it is also likely possible using more powerful methods. In case repair is not possible, we can infer that adapting the interaction between the model parts is not enough to repair and instead the parts themselves have to be modified.

In the following Chapters we will first outline the necessary background for our setting. Then we describe the most closely related work to our own. Afterwards we present our approach and use it to answer our research questions in a series of experiments. Finally, we conclude and contextualize our results.

Chapter 2

Background

2.1 Neural Networks

This Section explores artificial neural networks. We explain how they work and are created.

An artificial neural network, broadly, is a type of machine learning model capable of and used for representing arbitrary functions. In recent years they have shown significant success in multiple domains such as computer vision Krizhevsky et al. (2012) and natural language processing Vaswani et al. (2017) and form the basis for the type of model we study for pruning: large autoregressive language models based on the transformer architecture.

2.1.1 History

We present a brief overview of the history of neural networks. Large parts of this overview are motivated by Roberts (2024). For description of biological neurons we follow Kandel et al. (1991).[1]

Artificial neural networks were originally inspired by real biological neural networks as found in the brains of humans and most other animals. In brains, neural networks are collections of cells known as neurons. A biological neuron is roughly composed of the *cell body*, the *axon*, a long structure that emerges from the cell body and serves as an outbound

[1]Note that we simplify many aspects (such as the omission of pre-synaptic terminals) and combine information from disparate chapters of the book.

connection to other neurons as well as a number of *dendrites* that play the role of incoming connections from other neurons. The axon of a neuron is connected to the dendrites of other neurons through a structure called the *synapses*.

The main purpose of these connecting structures is the transmission of electrical signals between the neurons. This works by means of an electrical potential that is built up in the main cell body of a neuron as it receives electrical potential from other neurons through its dendrites. Once a threshold of potential is reached, a neuron is said to fire and propagates potential of its own through the axon and towards the dendrites of its outgoing neuron neighbors. After that a neuron experiences a brief period of non-excitability or refactory period before it can fire again. In this manner, biological neurons are capable of rudimentary computation by aggregating the signals of its input neurons into one new output signal. Over time and with many neurons, complex signals can be constructed that are then sufficient to act as the main executive driver of the underlying organism.

A formal model of the workings of these neural networks was first proposed by Mcculloch and Pitts (1943) in the form of temporal propositional logic, representing the workings of neural networks as electrical circuits. A few years later Hebb (1949) proposed a mechanism by which the brain learns the connections and resulting circuits. He argued that when two connected neurons fire at almost the same time, their connectivity strengthens, i.e., the amount of potential transferred through their connecting synapse increases. We can see how this mechanism allows for a neural network to build a model of the world over time by considering the following example: Assume that a neuron A fires every time a light bulb turns on and a neuron B every time its light switch is flicked. When we then flick the light switch the connection between A and B strengthens (as both fire). Over time this allows the brain to make the causal connection between the light bulb and the switch.

Once modern computers became more viable, work began on representing these theoretical models on actual non-biological hardware to learn and use them as a mechanism to represent arbitrary functions. While initial attempts to do so at IBM failed (Rochester et al., 1956), Widrow and Hoff (1960) were able to develop the models ADALINE (short for Adaptive Linear Neuron) and MADALINE (short for Many ADALINE), representing a single neuron and multiple interconnected ones respectively, and implement them on real hardware. They used their approach to predict future bits on phone linees to remove echoes and reduce noise. However, learning was still difficult, especially once work began on deeper neural networks, i.e., those where multiple neurons are connected serially and

not just in parallel. A breakthrough for efficient training of these networks was proposed by Rumelhart et al. (1986), which adjusts weights using gradient information efficiently and propagates necessary information backwards through the network. Hence the name: backpropagation.

Interest was sparked again in neural networks when research by Krizhevsky et al. (2012) showed strong performance of neural networks on ImageNet, a visual object classification challenge and dataset (Deng et al., 2009). In their work they used GPUs (graphical processing units) for training of the network and showed that much bigger and deeper networks were possible to implement with this new hardware. Following the success on many visual tasks, Vaswani et al. (2017) proposed a new type of neural network called the transformer that was also able to extend this success to a large extent to the domain of language processing. In 2020, the organization OpenAI revealed GPT-3 (Brown et al., 2020), a family of large neural networks with over one billion parameters based on the transformer architecture trained to predict the next token in a text. They showed that this kind of procedure can induce general capabilities in the model that allow it to solve more specific tasks with comparatively little data. Networks like GPT-3 form the basis for our research as we aim to reduce their prohibitive size without compromising on their capabilities.

2.1.2 Artificial Neurons

Coming back to the basics: The most fundamental component of an artificial neural network is the neuron. We model a neuron N mathematically[2] as implementing a function

$$f_N(x_1, \ldots, x_n) = \sigma(w_1 x_1 + \ldots + w_n x_n + b)$$

with $x, w \in \mathbb{R}^n$, $b \in \mathbb{R}$ (called the weights) and $n \in \mathbb{N}$. Here $\sigma(\cdot)$ refers to a nonlinear activation function that takes the output of the linear combination of x and w and further projects the outputs of the linear sum to some other point in $\mathbb{R}$. In the simplest case, $\sigma(\cdot)$ can be a thresholding function of the form

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{if } z < \theta \end{cases}$$

[2]See Bishop (1995) for one example of a similar formalism for neural networks to the one introduced here.

with θ as the thresholding value. In this specific case, a neuron is also referred to as a perceptron (when disregarding the mechanism by which the weights are arrived at) originally proposed by Rosenblatt (1958). Other choices include the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}},$$

and the rectified linear unit (ReLU) (Nair and Hinton, 2010)

$$\sigma(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}.$$

A neuron can represent different functions by choosing different values for the weights.

This formulation is a crude approximation of the behaviour of biological neurons in terms of dendrites, the cell body and the outgoing axon. Initially inputs or the outputs of other neurons x_i arrive at f_N as inputs (the role of the dendrites) and their potential is weighted in accordance to the weights w (collected in the cell body). Then, if a total amount of potential is exceeded, as modeled by $\sigma(\cdot)$ and the bias b, a signal is sent to the next neuron in the form of a higher output value (the neuron fires and sends the potential along the axon).

In order to understand how we can use artificial neurons to represent a function we are interested in let us consider the following example: We want to predict the amount of points a student got in an exam. The main indicators for this are the hours spent preparing for the exam as well as the amount of sleep (also in hours) the day before the exam. We refer to these quantities as x_1 and x_2 respectively. Assuming that x_1 and x_2 are sufficient to predict the points in the exam, this means that some true function $g : \mathbb{R}^2 \to \mathbb{R}^+$ exists that accurately maps hours spent studying and sleeping to points on the exam (we assume negative points cannot exist). In practice we do not know g but we can approximate it using a neuron N, for instance as $f_N(x_1, x_2) = \sigma(3x_1 + 2x_2 - 5)$. Here, we choose the ReLU function for σ as it accurately captures that one cannot have negative points in an exam. Then if we have a diligent student that studied for 5 hours and slept for 8, we can predict that they will receive

$$f_N(5, 8) = \sigma(3 \cdot 5 + 8 \cdot 2 - 5) = \sigma(26) = 26$$

points. On the other hand, a student that did not study at all and also partied all night is predicted to receive

$$f_N(5, 8) = \sigma(3 \cdot 0 + 8 \cdot 0 - 5) = \sigma(-5) = 0$$

points. This highlights the importance of the activation function as it can repesent non-linear aspects of g.

2.1.3 Single Layer Neural Networks

We can extend the formalism of a single neuron N to the case of m parallel (not serially connected) neurons, called a layer of neurons L, taking the same input x by replacing $w \in \mathbb{R}^n$ with $W \in \mathbb{R}^{m \times n}$, as well as $b \in \mathbb{R}$ with $b \in \mathbb{R}^m$ and defining

$$f_L(x) = \sigma(Wx + b).$$

The output is no longer a single value but a vector $o \in \mathbb{R}^m$. This formulation is known as a single-layer neural network. It can be used to represent multi-valued functions.

2.1.4 Limitations of Single Layer Neural Networks

While neurons and, by extension, single layer neural networks can already represent interesting functions, they can also fail for surprisingly simple ones: Take for instance the XOR function defined as

$$XOR(x_1, x_2) = \begin{cases} 1, x_1 \neq x_2 \\ 0, x_1 = x_2 \end{cases},$$

which takes in two inputs, compares them and returns 1 in case they are different and 0 if they are identical. Now let

$$f_L(x_1, x_2) = \sigma(w_1 x_1 + w_2 x_2 + b)$$

be a neuron (or equivalently single layer neural network with a single output) with a monotonically increasing activation function σ. Then it holds that:

$$\nexists w_1, w_2, b \in \mathbb{R} : f_L(x_1, x_2) = XOR(x_1, x_2)$$

or in other words the XOR function is not representable by one layer neural networks with monotonically increasing activation functions.

Proof One can see that this is true by considering the conditions that have to hold in order for the model to represent XOR. Consider the inputs 0 and 1. Then it has to hold that:

$$
\begin{aligned}
0 = f_L(0,0) &= \sigma(w_1 \cdot 0 + w_2 \cdot 0 + b) = \sigma(b) \\
1 = f_L(0,1) &= \sigma(w_1 \cdot 0 + w_2 \cdot 1 + b) = \sigma(w_2 + b) \\
1 = f_L(1,0) &= \sigma(w_1 \cdot 1 + w_2 \cdot 0 + b) = \sigma(w_1 + b) \\
0 = f_L(1,1) &= \sigma(w_1 \cdot 1 + w_2 \cdot 1 + b) = \sigma(w_1 + w_2 + b)
\end{aligned}
$$

Assume there is some $w_1, w_2 \in \mathbb{R}$ such that $f_L = XOR$. Then $w_1, w_2 \in \mathbb{R}^+$ because otherwise $1 = \sigma(w_1 + b) = \sigma(w_2 + b) < \sigma(b) = 0$, violating monotonicity. Then it holds that $1 = \sigma(w_1 + b) < \sigma(w_1 + w_2 + b) = 0$. This is a contradiction, therefore there are no w_1, w_2 such that $f_L = XOR$. ■

As XOR represents a simple function, this places a strong bound on the expressivity of single layer neural networks.[3]

2.1.5 Deep Neural Networks

A natural extension to the formalism of single layer neural networks are deep neural networks (sometimes also referred to as multilayer perceptron). In contrast to the simpler single layer neural network, deep neural networks with at least one hidden layer can learn and represent arbitrary (continuous) functions to an arbitrary degree of precision (Cybenko, 1989). The key change, as the name suggests, is to not map the input directly to the output using a single layer of neurons but instead passing the outputs of the first layer to one or more hidden layers of additional neurons that further process the input and only then produce the true output. This way, a network can build up more and more complex signals/activations/features over the layers, leading to higher expressivity. In fact, a two layer neural network (read: one new hidden layer) already suffices to represent XOR.

Mathematically a multilayer neural network DNN consisting of layers $L_1, \ldots, L_n$, which we denote as $DNN = \{L_i, \ldots, L_n\}$ represents the following function:

$$f_{DNN}(x) = f_{L_n}(\ldots f_{L_2}(f_{L_1}(x))),$$

i.e., the serial application of the individual layers one after the other.

[3] Originally, a similar bound was shown specifically for perceptrons by Minsky and Papert (1969)

For each layer L_i we have the free parameters from the original one-layer networks, which we will from now on refer to as $W^{(i)}$ and $b^{(i)}$ for L_i.

2.1.6 Training

Overview. To obtain a neural network approximating some desired function g a procedure is necessary to determine the optimal weights $W^{(i)}, b^{(i)}$ for this purpose. We can achieve this by modelling the finding of the parameters as an optimization problem, with the goal of minimizing some loss function $\mathcal{L}$ that is small when f_{DNN} is a good approximation of the target function g. For such a procedure to work, four components need to be considered:

- The loss function $\mathcal{L}$ with desirable properties to be easy to optimize.
- An optimization algorithm with desirable properties, such as quick convergence and robustness, capable of minimizing $\mathcal{L}$.
- A regularization mechanism that ensures f_{DNN} does not overfit the data it observes during training.
- An initial parameterization for all $W^{(i)}, b^{(i)}$.

For the rest of this Chapter we will refer to the entirety of the parameters of DNN, i.e., $W^{(i)}, b^{(i)}$ for all i as θ. We will then refer to the function represented by DNN with a specific parameterization θ as f_θ.

Loss. Most fundamentally, a loss function models the degree to which f_θ as parameterized by the weights θ approximates g on some input x. A simple example could be

$$\mathcal{L}_x(\theta) = (f_\theta(x) - g(x))^2,$$

which is 0 if $f_\theta(x) = g(x)$ (therefore minimal) but penalizes wrong predictions with a quadratic term. Further we typically want f_θ to approximate g on all inputs, therefore we often use

$$\mathcal{L}_{\mathbb{D}}(\theta) = E_{x\sim\mathbb{D}}[(f_\theta(x) - g(x))^2]$$

with $\mathbb{D}$ being the true data input generating distribution. We can then approximate $\mathcal{L}$ with the sample mean

$$\mathcal{L}_D(\theta) = \frac{1}{|D|} \sum_{(x,g(x))\in D} (f_\theta(x) - g(x))^2$$

on some *training dataset* D of input and output pairs.

There are multiple properties that make the choice of a loss function $\mathcal{L}$ preferable over alternatives:

- **Continuous** A loss function should be continuous so that the local neighborhood of a loss $\mathcal{L}(\theta + \epsilon)$ during optimization is not arbitrarily dissimilar from the loss at $\mathcal{L}(\theta)$ itself. This allows a degree of predictability of the loss.
- **Differentiable** If $\mathcal{L}$ is differentiable with respect to θ it allows us to obtain even more information about $\mathcal{L}$ useful for optimization, i.e., the gradient $\nabla_\theta \mathcal{L}$.
- **Efficient** A loss function should be fast to compute in order to speed up optimization.

Two of the most common loss functions that fulfill these criteria are the mean squared error (MSE) and the categorical cross-entropy loss (CE). The MSE loss is exactly $\mathcal{L}_D(\theta)$ as defined above, i.e., it holds that

$$\mathcal{L}_{\text{MSE}}(\theta) = \mathcal{L}_D(\theta).$$

The MSE is often used for regression problems. In contrast, the CE loss is defined only for classification. Here, we model f_θ such that it is a multivariate function with $f_{\theta,i}$ being the probability that some class c_i among multiple others C is the correct choice for the input to f_θ. Similarly g_i represents the true probability that the correct answer is c_i. Then the CE loss is defined as

$$\mathcal{L}_{CE}(\theta) = -\frac{1}{|D|} \sum_{(x,g(x))\in D} \sum_{i=1}^{|C|} g_i(x) \log(f_{\theta,i}(x)).$$

If $g_i(x) = 1$ for the correct class c_i and $g_j(x) = 0$ for all other classes $c_j \neq c_i$, the CE loss simplifies to

$$\mathcal{L}_{CE}(\theta) = -\frac{1}{|D|} \sum_{(x,g(x))\in D} \log(f_{\theta,i}(x)).$$

This simplified form of the CE loss is known as the negative log-likelihood loss.

Optimization algorithm. Aside from a loss $\mathcal{L}$ that, when minimized, leads to a good approximation of g using f_θ, the main other component is a process to find values of θ that do so.

A naïve approach we can apply for any choice of $\mathcal{L}$ is *random search*. As the name suggests in this approach we randomly sample values $\theta \sim \Theta$ from some distribution over our parameters Θ. We do so until we reach a parameterization whose loss is below some threshold or until we run out of computational resources. While effective, this approach is highly inefficient to the point that it is not practically used to train neural networks. It does, however, still find application for finding parameters of optimization algorithms (called hyperparameters) used to train neural networks.

If we assume instead that we do possess some additional knowledge about our loss function, namely that it is differentiable and thus continuous[4], we can use a much more efficient manner of optimization: *gradient descent*. The idea of this algorithm is to start out with some initial parameterization θ_0 and then iteratively make small step adjustments to it based on gradient information such that it is likely that $\mathcal{L}(\theta_{t+1}) < \mathcal{L}(\theta_t)$ holds. Over time, this decreases the loss and finds a better approximation of g. We adjust θ_t based on the equation

$$\theta_{t+1} = \theta_t - \alpha \nabla_{theta} \mathcal{L}(\theta_t),$$

where α is a hyperparameter known as the learning rate. In effect, what this equation does is exploit the fact that the direction of the gradient $\nabla_\theta \mathcal{L}(\theta_t)$ is the direction of the steepest increase in $\mathcal{L}$. By moving in the opposite direction, we therefore move in the direction of the steepest decrease and make the loss smaller.

To apply gradient descent, a general and efficient way to obtain the gradient information is necessary with respect to all the individual parameters we optimize. In general, this is achieved using the *backpropagation* algorithm (Rumelhart et al., 1986). Backpropagation is an application of the chain rule to compute the partial derivates for the individual weights $w \in \theta$ efficiently. This works by calculating the derivatives back to front, from output to input. Derivatives for intermediate values can then be reused to calculate derivatives with respect to earlier weights. To understand how this works, let us consider a simple neuron $f_N(x) = \sigma(wx + b) = z$ as an example. Further define $a = wx + b$. To calculate $\frac{\partial z}{\partial w}$,

[4] As well as analytic.

backpropagation first decomposes this into

$$\frac{\partial\, z}{\partial\, w} = \frac{\partial\, z}{\partial\, a}\frac{\partial\, a}{\partial\, w}.$$

Then $\frac{\partial\, z}{\partial\, a}$ is computed, "propagating the gradient backwards", followed by $\frac{\partial\, z}{\partial\, a}$, which can then be multiplied together to obtain the final derivative. When one has multiple layers like in a DNN, this process is continued, calculating the gradients for a and z at earlier layers based on the ones for the layers already processed and then using them for the weights at their respective layer. While it is possible to implement this procedure for a specific network architecture by hand (defining the rules on how gradients propagate where), nowadays it is more common to use a framework like PyTorch (Paszke et al., 2019) that automatically derives the gradients in this manner by tracking computations and building up an internal computation graph on which backpropagation can occur to compute its gradients.

While gradient descent serves as a good way to learn effective parameterizations of θ, it does have a set of drawbacks:

1. Gradient descent assumes we compute the gradient of the full loss $\mathcal{L}$. In case we have a loss that averages over the result of many datapoints in a training dataset (as the ones defined in this chapter are) calculating the gradient over all datapoints for each update step becomes inefficient, especially as the network gets large.
2. Gradient descent tends to get stuck in shallow local minima of $\mathcal{L}$ that do not correspond to the true global minimum of $\mathcal{L}$ (Ruder, 2016).
3. Convergence can be slow (Ruder, 2016).

To address these issues a number of modifications of the original algorithm have been proposed (and summarized by Ruder (2016)):

1. **Stochastic Gradient Descent** Instead of calculating the gradient based on the whole training dataset D, we can approximate it stochastically using a few samples from the whole dataset, referred to as the batch or minibatch. That is we replace

   $$\mathcal{L}(\theta) = \sum_{(x,g(x))\in D} \mathcal{L}_{inner}(x, g(x), \theta),$$

   and $\mathcal{L}_{inner}$ being a loss for a specific datapoint, e.g., the mean squared error, with

   $$\mathcal{L}(\theta) = \sum_{(x,g(x))\in B\subset D} \mathcal{L}_{inner}(x, g(x), \theta).$$

This way the gradient becomes faster to compute and it still approximates the same behaviour. The batch B is randomly sampled anew in each optimization step. This approximation has another benefit that manifests in the noise it introduces: By sometimes making "wrong" decisions the optimization algorithm avoids getting stuck in local but shallow minima and empirically converges more consistently to deeper ones. It is common to choose B by splitting up D into batches of a specific size. Then optimization steps continue until all these batches have been used. This is referred to as an epoch. It is common to train for a set number of epochs.

2. **Momentum** In order to improve convergence speed we can use momentum (Qian, 1999). The key idea behind this approach is that past optimization steps can help us improve the current one. To understand why imagine that we are stepping in one direction of the loss landscape during optimization and the loss drops significantly. It is likely that a nearby minima has not necessarily been reached and we would profit from going further into this direction. Intuitively, this is akin to following a ball down a hill. Instead of releasing it and catching it again after the ball has rolled for a few meters, we let it roll further and gain momentum to get down the hill quicker.

   Mathematically this is represented using a velocity vector v_t and our current position in the loss landscape θ_t is the position of the ball. The update equation then changes to

   $$v_{t+1} = \gamma v_t + \alpha \nabla_\theta \mathcal{L}(\theta_t)$$

   $$\theta_{t+1} = \theta_t - v_t$$

   with γ typically set to 0.9. This equation directly models how a ball would roll down a hill (instead of stopping it every few meters before it can build momentum) (Ruder, 2016).

3. **Adaptive Learning Rates** Finally, we can also dynamically adapt the step size α we take based on the magnitude of current gradients. If they are small we expect to be already close to a minimum and therefore would like to make α smaller in order to be able to more carefully explore the space and thereby improve convergence speed. This adaption can also be applied separately for different parameters.

   By combining parameter-specific adaptive learning rates with a form of momentum we arrive at the Adam algorithm (Kingma and Ba, 2015), which is one of the most common ways to train neural networks today.

Regularization. A major failure case that can occur when training neural networks is *overfitting*. This happens when loss is low on the training dataset D but the approximation to g breaks down on other real data sampled from $\mathbb{D}$. Colloquially speaking, instead of generalizing features and patterns, the model learns to memorize the training dataset.

We can address this issue by adding a regularization term to the loss $\mathcal{L}$ being optimized that penalizes memorization. A common choice is so-called L2 regularization, where terms of the form $||W||_2$ are added to the loss, with $||\cdot||_2$ being the euclidean norm. This suppresses large values for the weights, which makes it more unlikely that small changes in the input lead to large changes in prediction, making memorization harder.

Initialization. For optimization procedures such as gradient descent, an initial parameterization θ_0 is necessary. A common approach is to initialize the weights $W \in \theta_0$ randomly by sampling each individual one from a normal distribution with mean 0 and carefully chosen variance based on the size of the weight matrix (Narkhede et al., 2022).

Fine-tuning. In some cases, full training from scratch, i.e., a randomly initialized set of weights, is not desirable. This is especially the case as models continue to increase in size and training becomes more and more expensive. Therefore, in recent years, a new paradigm has emerged. First, a large model is trained from scratch on a huge collection of general data such as text or images, known as the *pretraining phase*. Then, to solve a particular task, a new model is initialized with the parameterization learned during pretraining, which is then further optimized on task data. That is to say, the pretraining parameterization is *fine-tuned* to be able to solve the task (Radford et al., 2019). In this manner, the model can reuse features learned during pretraining and does not have to relearn them. Therefore, fewer training steps are needed to obtain good performance on most tasks. Sometimes large pretrained models are referred to as foundation models, since they serve as the foundation to build other models upon (Bommasani et al., 2021).

However, while fine-tuning in itself addresses concerns with regards to the number of training steps needed, concerns with regards to the number of parameters to train remain unaddressed. Since models used for fine-tuning are on average large and all the parameters are updated at each step, each individual one can still take a long time and require large amounts of memory to store gradient information. As a solution to this, many *parameter-efficient fine-tuning* approaches have been proposed. Instead of updating all parameters

of the models, the idea is to update only a subset or introduce new modifications to the model that can steer its behaviour while requiring fewer trainable parameters to do so.

One of these approaches is Low-Rank Adaption (LoRA) as proposed by Hu et al. (2022). The idea of LoRA is, instead of modifying the weights of a network layer $W \in \mathbb{R}^{m \times n}$ directly, to learn additional parameter matrices $A \in \mathbb{R}^{k \times n}, B \in \mathbb{R}^{m \times k}$ and replace the linear application of the layer to the input x to produce output y,

$$y = Wx$$

by

$$y = Wx + BAx.$$

The reason this does not purely reduce to fine-tuning[5] is that k can be chosen to be small, in effect producing a low-rank matrix BA with fewer trainable parameters and thus more computational efficiency. As k grows larger, LoRA approximates full fine-tuning.

2.2 Transformer-based Language Models

2.2.1 Overview

Over the years, many modifications to plain neural networks have been proposed. Among them, the transformer architecture Vaswani et al. (2017) has proven itself as a powerful approach for function approximation. The main difference between vanilla DNNs and transformers is that while the former is designed to take in vectors of real numbers as input and produce a vector of real number as output the latter is designed to take in a sequence of *tokens* and produce another sequence of *tokens* as output. A token here refers to some arbitrary atomic unit from a set of all possible tokens V, the vocabulary, that can be arranged in a sequence. In the case of language modelling, most often tokens are equivalent to words and punctuation[6] and the sequence is equivalent to a text.

[5] It holds that $Wx + BAx = (W + BA)x$, which begs the question as to why not just adapt W directly.

[6] Stricly speaking, most modern language models use tokens that represent parts of words to prevent the vocabulary size from growing too large. A common way to automatically obtain a good tokenization of a given text is the Byte-Pair encoding (BPE) (Gage, 1994) algorithm that works by iteratively merging common character combinations.

In this Section we will look at a variant of the transformer relevant to our use case: the decoder-only transformer with a language modelling head and a next token prediction training objective designed to do autoregressive language modelling as originally proposed by Liu et al. (2018). The main difference to other types of transformers is that here we take in a sequence of tokens as input and *continue the existing sequence*, instead of generating a new one. In order for the model to do this, it is trained to always predict the next token w_{n+1} given $w_1 \dots w_n$ as input. We can produce another new token w_{n+2} by concatenating the newly predicted token to the end of the existing sequence and running the model again. Repeating this allows us to infinitely extend the given sequence. Because it does so by passing its own output back to itself as the input, this is called autoregressive language modelling.

2.2.2 High-level Architecture and Usage

A decoder-only transformer as concieved of by Liu et al. (2018) with a language modelling head consists of three parts:

1. **Embedding Layer** First, the sequence of input tokens $w_1 \dots w_n$ are converted into a sequence of vectors $e_i \in \mathbb{R}^E$ called *embeddings*, where E is the embedding dimension. Each token in V gets mapped to its own distinct embedding. The values of each vector is learned alongside the other parameters of the model during training. Additionally, a positional embedding is applied on base embedding that contains information about the position of the token in the sequence. Often this is based on applying multiple trigonometric functions like sin to the position i of w_i and concatenating their results to the base embedding.

2. **Decoder Layers** Second, analogously to a regular DNN a number of so-called decoder layers are applied to the embeddings one after the other. Each decoder layer D_i takes in the sequence of current embeddings $e^{(i-1)}$ and produces an updated sequence $e^{(i)}$ of the same length. Intuitively, each layer "refines" each embedding further. We refer to the number of decoder layers as $|D|$.

3. **Language modelling head** Finally, a linear transformation as well as a function called the softmax[7] is applied to all embeddings in the sequence separately.

[7]The function softmax : $\mathbb{R}^n \to \mathbb{R}^n$ maps an input vector $x \in \mathbb{R}^n$ to another vector representing a

This maps the embedding $e_i^{(|D|)}$ to a probability distribution across all tokens in V, $P(\cdot|w_1 \dots w_{i-1})$. This is used to represent the probability that a given token follows the sequence prefix $w_1 \dots w_{i-1}$.

After one pass through the transformer, an input sequence $w_1 \dots w_n$ is thus converted to a sequence of equal length of probability distributions across the next token. Using the distribution of the last input token w_n, we can then either sample a token at random from this distribution or take the most likely one and use that as our prediction of the next token w_{n+1}. Then, as described above to generate even more tokens, we can repeat the process by passing in $w_1 \dots w_{n+1}$ as input to the transformer again.

When we always pick the most likely token as the next one, we refer to this process as *greedy decoding*. When we randomly sample the next token we refer to it just as *generation*. Note that for the latter case we can change the temperature parameter t for the softmax function to vary how "random" the generations are going to be. Higher values result in more unlikely but creative completions, while low values approximate greedy decoding. Which way to use to generate a sequence continuation from the model depends on the task at hand.

2.2.3 Low-level Architecture

Each decoder layer D_i consists of two sub-components that are applied in sequence:

- **Attention Block** Intuitively, the attention block in a decoder layer is responsible for exchanging information between the embeddings $e^{(i-1)}$. As the name suggests, this works by making the embeddings pay attention to each other to extract information. For instance, for an input like "His name is Richard", the embedding originating from the token "Richard" might pay attention to "His" and incorporate information

probability distribution $y \in \mathbb{R}^n$, i.e., the values of y are all positive and sum to 1. Specifically we have that

$$\text{softmax}_i(x) = \frac{e^{\frac{x_i}{t}}}{\sum_{j=1}^{n} e^{\frac{x_j}{t}}}.$$

Here, $t > 0$ refers to the temperature, a hyperparameter that controls how sharp the resulting distribution in y is. When t approaches 0, $\text{softmax}_k(x) \to 1$ where x_k is the largest entry in x. As t approaches infinity softmax converges to a uniform distribution.

pertaining to that token as one refers to the other. In this way, the embeddings become *contextualized*. Mathematically, this is represented as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(M_c \circ \frac{QK^T}{\sqrt{k}}\right) V.$$

Here Q, K and V are derived from the input embeddings $e^{(i-1)}$ of the component by means of three learnable weight matrices $W_Q^{(i)}, W_K^{(i)}, W_V^{(i)} \in \mathbb{R}^{E \times k}$ as $Q_j = W_Q^{(i)} e_j^{(i-1)}$, $K_j = W_K^{(i)} e_j^{(i-1)}$ and $V_j = W_V^{(i)} e_j^{(i-1)}$. The size of the matrices is determined by the hyperparameter k. M_c is an upper triangle matrix filled with 1s and 0 everywhere else called the *causal mask*.

On a high level what is happening here is akin to a database. Each embedding specifies what information it is looking for in terms of the query Q. Similarly, each embedding marks the information it can provide with a key K as well as prepare the associated information itself V (short for value). QK^T represents the lookup process, matching queries to fitting keys in terms of a high dot product. M_c acts as a filter for inaccessible information. Some lookups are erased by multiplying it pointwise with QK^T, implicitly setting them to 0. These are precisely the lookups where a token would request information from a token *after* itself. As the goal at each position is to predict the next token, this mechanism prevents the model from cheating by directly looking up the answer. After the softmax, this results in a distribution of how much information an embedding would like from all the other tokens (summing to 1). By multiplying with the information represented in V, the actual information is aggregated and serves as the output.

In practice, this becomes more complicated as in one attention block we conduct multiple of these attention calculations in parallel, each with their own set of $W_Q^{(i)}, W_K^{(i)}, W_V^{(i)}$. These are called the attention heads. Each head produces some output vector for each input embedding, which are then aggregated either by taking the average or by some additional linear transformation $W_{proj}^{(i)}$ to serve as the final output of the entire attention block.

In summary then we can denote the output of the attention block as

$$\hat{f}_{ATTN^{(i)}}(e) = W_{proj}^{(i)}(\text{Attention}_{H_1}(e); \ldots; \text{Attention}_{H_{|H|}}(e)),$$

with $\text{Attention}_{H_i}(e)$ as the attention computed for head H_i (with the corresponding Q, K and V derived from e) and ; as the concatenation operator.

- **Feedforward Block** The feedforward block is comparatively simple and consists of a small vanilla DNN applied to each embedding separately, typically with a ReLU activation function. We define the output as

$$\hat{f}_{FF^{(i)}}(e) = (f_{DNN}(e_1); \ldots; f_{DNN}(e_n))$$

.

We denote the transformations implemented by the two components with a hat as for each of them two more modifications are made to obtain the true output:

1. Before the transformation takes place, the embeddings are normalized to have a mean of 0 and standard deviation of 1, denoted as LayerNorm(e).
2. A *residual connection* is added, i.e., the input to the component is added to the output.

This means that the final transformation associated with a decoder layer D_i is

$$f_{D^{(i)}}(e) = e_{\text{ATTN}} + \hat{f}_{FF^{(i)}}(\text{LayerNorm}(e_{\text{ATTN}}))$$

with

$$e_{\text{ATTN}} = e + \hat{f}_{ATTN^{(i)}}(\text{LayerNorm}(e))).$$

The full transformer then just sequentially applies the embedding layer, the $|D|$ decoder layers, and the language modelling head, i.e.,

$$f_{TF}(w_1, \ldots, w_n) = f_{LM}(f_{D^{(|D|)}}(\ldots f_{D^{(1)}}(f_{EMB}(w_1, \ldots, w_n)) \ldots)),$$

where f_{EMB} transforms the input tokens to embeddings and f_{LM} is the final language modelling head that produces the next token probabilities using the softmax function.

2.2.4 Training

In order to train a decoder-only transformer for autoregressive language modelling using next token prediction (Radford et al., 2019) all that is required is ...

- ... a dataset D consisting of training sequences.

- ... a loss comparing the probability distribution across the next token to the true next token in the sequence, i.e., we model the prediction task as classification where the possible classes are all tokens in V and the correct class is the true next token (for instance cross-entropy). In order to maximize computational efficiency this loss is calculated for all positions in a training sequence at once (each predicting the token directly after it) and then summed to produce the final loss for the sequence that can then be used with gradient descent or other optimization algorithms to find a good parameterization.
- ... an optimizer, e.g., Adam (Kingma and Ba, 2015).

2.2.5 Role as Foundation Models

Decoder-only transformers with a next token prediction objective are both efficient (as all positions in a training sequence can be used at once for learning) as well as easy to collect training data for (as only unlabelled text is required which is abundantly found online). This makes them suitable to be scaled up both in terms of model size as well as training data.

Over the last few years this fact has come to fruition and today we have a range of models based on this architecture that have shown general capabilities on many tasks relating to language modelling (Brown et al., 2020; Touvron et al., 2023; Jiang et al., 2023). Generally, they are known as *large language models* (LLMs). In this work, we focus on precisely this kind of model and try to reduce their large scale while preserving their capabilities.

2.3 Prompting using Autoregressive Language Models

2.3.1 Overview

In order to solve a task t using a pretrained autoregressive language model M, the traditional approach is to fine-tune M on task data for t or even to retrain the model from scratch. This is slow and requires access to large amounts of task data. In recent years *prompting* has emerged as an alternative that is both efficient and does not require a lot of task data to work.

The idea of prompting is to use an autoregressive model M "as is" in conjunction with a specially crafted input sequence called the prompt to solve instances of t by continuing this sequence. No additional training is necessary. This is motivated by the fact that many modern language models are already quite general and the knowledge and abilities to address many tasks are already implicitly encapsulated in the pretraining next token prediction pretraining objective in conjunction with the large scale of the generic pretraining data. For instance, an LLM trained on tweets is likely to be able to identify the sentiment of a post, because this helps it to predict the replies (a positive post will likely have more positive replies than a negative post). Thus, there is no need to fine-tune M further, as all abilities needed for t are already present. Instead, we only need to provide some way to "access" it, i.e., the prompt.

In this Section we will explore prompting for autoregressive language models, first covering how a prompt is constructed and used to address a task and then briefly illuminating two further advanced concepts to improve upon the basic approach.

2.3.2 Basic Approach

To solve an instance x of a task t using a model M with prompting, we use a prompt $\text{prompt}_t(x) = w_1 \dots w_n$ that gets passed as input to M. From this we obtain either a continuation of the prompt $w_{n+1} \dots w_{n+m}$ or a probability distribution across the next token $P_M(w_{n+1}|w_1 \dots w_n)$, depending on what we need for the task. We can then read off the solution to x from this output. There are multiple approaches to do so, but the most common for ...

- ... generative tasks such as question answering (the goal is to generate some sequence) is to define the completion $w_{n+1} \dots w_{n+m}$ as the returned answer.
- ... classification tasks such as sentiment analysis (the goal is to select the correct class) is to choose some token w_c to represent each class c and interpreting the probabilities $P_M(w_c|w_1 \dots w_n)$ as the class probabilities. For sentiment analysis with the classes positive and negative, for instance, the tokens "positive" and "negative" could be used respectively.

While in theory a prompt could be an arbitrary sequence of tokens that for some reason makes the answer extracted from M to x using prompting represent the correct solution, in

practice the prompt is carefully engineered to ensure this behavior. We achieve this through a prompt template $\text{prompt}_t(\cdot)$, i.e., a function mapping a task instance to a prompt. It is called a template, as the form this function takes is a template text into which x is only inserted to produce the final prompt used for solving x. A prompt template typically consists of three parts[8]:

1. **Context** At the beginning of the prompt template, the goal is to steer the model towards writing documents relating to the task at hand and of adequate quality. Depending on the model how this is best achieved varies but common strategies include (1) direct instructions (for model fine-tuned specifically to follow them), (2) roleplay, e.g., "You are a famous movie reviewer who is renowned for his sentiment analysis skills." or (3) exploiting common formats in the training data, e.g., top 10 lists on the internet. For instance we could exploit this commonly seen structure to generate movie titles by prompting a model with: "You won't believe how amazing these ten movies were:".

2. **Examples** It is often helpful to provide one or more examples of a problem and its solution for the given task. In the case of classifying movie sentiments, this would be pairs of movie reviews and their sentiments. Note that the amount of examples used in a prompt template is typically much smaller than the amount needed to do fine-tuning.

3. **Input** Finally x is inserted into the prompt. This is typically done in a similar format to the other demonstrations. For instance the format "Question: <x> Answer:" as introduced by Brown et al. (2020) is commonly used for question answering.

This structure ensures that completions or next token probabilities coincide with correct solutions for t.

We present an example of a prompt template for the sentiment analysis of movie reviews in Figure 2.1. We denote the place where x is inserted into the template as "<x>".

[8] As outlined by Dong et al. (2022), however, we choose to present the formalism in an adapted, simplified manner.

The following is a list of movie reviews followed by their sentiment.

Review: I loved this film. It was really good.
Sentiment: positive

Review: The actors were so bad!
Sentiment: negative

Review: <x>
Sentiment:

Figure 2.1: Example of a prompt template used for sentiment analysis.

2.3.3 Advanced Prompting

Basic prompting works almost as an accidental side-product of the general next token prediction pretraining objective. As such, great care has to be put in engineering a prompt template that works consistently to ensure the pretraining objective and the task at hand always coincide. Naturally, the question arises: Can we do better?

Instruction Tuning. One approach to improve the consistency of prompting is to fine-tune the underlying model on instruction following datasets, i.e., texts containing a task represented as instructions and their solutions. Because commonly used prompt templates exhibit this structure (in the context section), this naturally improves their ability to do next token prediction on these types of prompts and this by implication also improves accuracy on the task corresponding to the prompt. This procedure is known as *instruction tuning* (Zhang et al., 2023a).

Going even further research by Ouyang et al. (2022) has found that the performance of instruction-tuned models can be improved even more after the fine-tuning step:

1. They ask the fine-tuned model to solve tasks and provides multiple solutions. Human raters compare these against each other and rate which ones are better and which ones are worse.

2. They then use the rankings created by the humans to train a separate model, referred to as the reward model, which learns to represent the human preferences in the answers of the language model.

3. Finally, they further trained the fine-tuned model to optimize the scores it receives from the trained reward model using reinforcement learning techniques (Mousavi et al., 2018).

This is known as *reinforcement learning from human feedback* or RLHF for short.

Constrained Generation. For generative tasks such as question answering, often there are some constraints that we want to hold for the answers. Imagine, for instance, a task where the goal is to answer date questions like "In which year was the book 1984 written?". With basic prompting, the language model might complete "It was written in 1948." as the answer. However, if our goal is to only extract the date, this is insufficient. While it may be possible to build a parser that extracts the answer for us here, this is brittle and prone to failure. Similar problems arise when we want the language model to reply in a specific format, such as valid Python code.

Constrained generation addresses these issues by making the generation of answers that do not match our constraints a priori impossible. To do so constrained generation approaches modify the probabilities $P_M(w_{n+1}|w_1 \dots w_n)$ such that the probability of tokens that do not fit the criteria are 0. When generating a completion the modified ones are used.[9] It is then straightforward to see how this works in case we only want to generate years as our answer: We set the probability of all non-number tokens to 0. More complex constraints are also possible to represent by dynamically modifying which tokens are allowed to be generated at each point in the generation. We can represent them using tools such as regexes or context-free grammars (Wang et al., 2023).

One particular application of constrained generation is to "fill in the gaps" in a prompt template to solve a task step by step. This is explored by the Python library Guidance (Lundberg and Ribeiro, 2023). For instance to make a model compute $3 \cdot 5 + 3 \cdot 4$, their approach would force the output to look like "3·4 : <BLANK>, 3·5 : <BLANK>, 3·5+3·4 :

[9] This is a slight simplification of real approaches such as Wang et al. (2023)We can, for instance, optimize this procedure and speculatively decode multiple tokens at once and check afterwards if the sequence was valid for our constraints.

<BLANK>". Each <BLANK> is forced to be generated as a number, while the other tokens are fixed (the probability of the forced token is set to 1 and the probability of all other tokens to 0 until the next <BLANK> is reached during generation).

In our work we use a similar methodology to generate fake task data used for importance estimation.

2.4 Model Pruning

2.4.1 Overview

Accelerating neural networks is of high practical relevance. Both during training and inference, a faster, smaller model is desirable both due to time as well as hardware constraints. However, most often it is the case that bigger models perform better than smaller ones. This is especially true in the case of large language models where recent models boast upwards of 100 billion parameters (Brown et al., 2020), making them prohibitively expensive to deploy. One approach to address this tension between requiring both a bigger model for better performance as well as a smaller model for fast inference is *model pruning*. The main idea of pruning is to start with a large model M with high accuracy and then remove parts of M, chosen in such a way that the initial accuracy is preserved resulting in a pruned model $\tilde{M} \subset M$. We then say that M is sparse in the removed model parts $M \setminus \tilde{M}$. This process results in a smaller and faster model that retains the performance of the original bigger model. What constitutes a part of the model, how the model is divided up into these parts and how $f_{\tilde{M}}$ is defined, differs from approach to approach. However, we can already say that pruning has been successfully applied (Frantar and Alistarh, 2023) at the level of individual weights all the way up to entire model layers (Peer et al., 2022).

Pruning makes the implicit assumption that not all parts of the model are equally important and that only a smaller subnetwork of the original model is necessary to solve a particular task, i.e., that sparsity exists. Depending on the models parts chosen to be removed, it is not obvious that this holds in general. Therefore, we experimentally test for the existence of this sparsity as part of our experiments.

2.4.2 Common Components of Pruning Approaches

While individual approaches to pruning can differ, a few general components common to most approaches seem to exist (as noted in prior work (Blalock et al., 2020)):

1. **Importance measure** To differentiate between important and unimportant model parts an approximate measure of importance $\text{Imp}_M(p)$ with $p \in M$ a model part is often used.

2. **Removal strategy** Given the available information, such as the importance measure for each part p_i, we need a strategy that decides which parts to remove.

3. **Repair mechanism** Often it is not possible to identify sparsity in M that does not lead to accuracy degradation. In these cases a repair mechanism can be used that restores the accuracy and combined with the pruning itself still leads to a smaller, sparse model.

We will now take a brief look at common approaches for each component.

Importance Measure. There are many possible ways to approximately measure the importance $\text{Imp}_M(p)$ of a model part p to the accuracy/performance $\text{perf}(M)$ (either with respect to a task or in general) of M. In general, however, we want the following criterion to hold:

$$\text{Imp}_M(p) \approx 0 \implies f_M \approx f_{M \setminus \{p\}}.$$

That is to say, when the importance of p is small, the pruned model $M \setminus \{p\}$ should approximate the unpruned model.

Below we list a collection of example criteria:

- **Magnitude** A single weight W_{ij} or a row of weights W_i of a regular linear layer L correspond to the connection between an input and an output neuron and an entire neuron respectively. When we want to prune other of these two types of model parts, we can define importance as $|W_{ij}|$ or $||W_i||_2$ respectively. This fulfills the criteria we defined above, since removal of these parts is equivalent to setting $W_{ij} = 0$ or $W_i = 0$.

- **Activation** Similarly, we can take the average output magnitude of p on a calibration dataset D as the importance score. This works in case the removal of p is equivalent to the component outputting 0. This is particularly the case when residual connections are used, i.e., $f_p(x) = x + h(x)$, with h representing the non-trivial computation associated with p.

- **Oracle** A more accurate approach is

$$\text{Imp}_M(p) = |\text{perf}(M) - \text{perf}(M \setminus \{p\})|$$

with perf$(\cdot)$ being a performance measure such as accuracy on some test set of a task. This definition of importance measures the impact of removal, i.e., the true difference in accuracy when removing p. Since perf is often slow to compute and we want to obtain importance scores for all $p \in M$, estimating importance in this way is slow.

- **Sensitivity** In case perf is differentiable and removal of p is equivalent to setting a mask parameter $\alpha_p = 0$ (and operation as normal is equivalent to $\alpha_p = 1$), we can approximate oracle importance as

$$\begin{aligned}
\text{Imp}_M(p) &= |\text{perf}(M) - \text{perf}(M \setminus \{p\})| \\
&= |\text{perf}(M) - \text{perf}(M_{\alpha_p=0})| \\
&\approx |\text{perf}(M) - (\text{perf}(M) - \nabla_{\alpha_p}\text{perf}(M))| \\
&= |\nabla_{\alpha_p}\text{perf}(M))|,
\end{aligned}$$

where $M_{\alpha_p=0}$ is the model with α_p set to 0. Since we can calculate the gradients with respect to all parameters efficiently using backpropagation in one backwards pass, we can estimate the importance for all model parts p at the same time, making for a much more efficient way to estimate the same effect.

Assuming that perf is calculated as the sum of an inner loss $\mathcal{L}_d$ on a dataset D, it holds that

$$\text{Imp}_M(p) = |\nabla_{\alpha_p} \sum_{d \in D} \mathcal{L}_d(\theta))| = |\sum_{d \in D} \nabla_{\alpha_p} \mathcal{L}_d(\theta))|.$$

Empirically, prior work has found that calculating the importance instead as

$$\text{Imp}_M(p) = \sum_{d \in D} |\nabla_{\alpha_p} \mathcal{L}_d(\theta))|$$

is a more effective way of assessing the importance (Michel et al., 2019; Molchanov et al., 2017). Intuitively, this prevents the failure case that the derivative is large for different datapoints but in different directions, i.e., positive and negative, so that they cancel each other out.

Additionally, this sum of losses is often divided by the dataset size $|D|$, such that the importance corresponds to the expected value of $\nabla_{\alpha_p}\mathcal{L}_d$ on D. When using importance purely to compare different model parts to each other, this is equivalent to the formulation above.

Removal Strategy. The main step of pruning is to find a minimal $\tilde{M} \subset M$ such that $\text{perf}(\tilde{M}) \approx \text{perf}(M)$ holds by removing unnecessary model parts, often under the additional constraint of a fixed size $m < |M|$. In general, finding such a minimal pruned model is computationally hard (Molchanov et al., 2017) and therefore, any efficient approach to pruning will only be approximate. However, given importance scores, two commonly used, efficient approximations are possible:

- **Importance pruning** $\tilde{M}$ is chosen such that the $|M| - m$ least important parts are removed (for instance as in Bansal et al. (2023); Michel et al. (2019)).
- **Greedy pruning** $\tilde{M}$ is chosen such that the $|M| - m$ least important parts are removed, however after each removal importance is reassessed on the newly pruned model. Alternative exit conditions for pruning are also possible in this setting, such as fixed degree in accuracy degradation on some task.

  Greedy pruning is more accurate than importance pruning in case the importance of model parts is not independent of each other, i.e., the removal of one model part has an effect of the importance of the remaining ones. In some settings it can be shown that the sparse submodels found by this approach are close to the theoretically optimal ones (Peer et al., 2022).

Repair mechanisms. Repair is most commonly done by further training the final pruned model on additional data, i.e., regular training (or alternatively sometimes interleaved with pruning by pruning some parts, repairing and then pruning some more). Which parameters are optimized differs from approach to approach. In some cases, we

might prefer to re-initialize the pruned model and train from scratch using a random initialization. This is the case for approaches building on the *lottery ticket hypothesis* (Frankle and Carbin, 2019) that states that trained models contain "lottery ticket" subnetworks, that when trained from scratch, perform equally well or better than the original model.

2.4.3 Taxonomy of Pruning Approaches

While sharing many commonalities, pruning approaches also differ substantially in other respects. Below we list some dimensions in which pruning approaches vary. For each we contextualize how our approach fits in.

Task-specific and Task-agnostic Pruning. One of the most fundamental distinctions one can make is between task-specific and task-agnostic approaches. As the name suggests, the former attempts to find submodels that perform well on a specific task (for instance Bansal et al. (2023)), not trying to retain the full generality of the model but only performance for a specific task. On the other hand task-agnostic approaches (for instance Ma et al. (2023)) try to prune in such a manner that a general model remains generally as good on a large number of task. For this distinction to make sense, the model used for pruning has to be sufficiently general in the first place, e.g., a foundation model.

In this work, we focus on task-specific pruning, where the task is specified implicitly through the prompt template. The models we prune are large general language models.

Static and Dynamic Pruning. Commonly a model is pruned after it has finished training but before it is deployed to be used for inference. However not all pruning approaches fit into this schema and are instead applied earlier or later in the model lifecycle.

At one end of the extreme, one can prune models at training time, jointly pruning a model as well as training it at the same time. For instance Liu et al. (2017) employ a special loss function that achieves this (this is also one of the few instances where an importance score is not commonly used as the loss implicitly deals with importance). Additionally, it is also possible to prune untrained neural networks at initialization, a more recently emerging paradigm that offers both well-performing subnetworks as well as accelerated training (Weissteiner et al., 2022).

At the other end of the extreme, pruning can be delayed to the point of actual inference with a given input x. This is advantageous because research has shown that for a specific input x the levels of sparsity are large compared to pruning levels otherwise achievable that have to maintain performance on a whole distribution of possible inputs (Liu et al., 2023). The drawback to this style of pruning however, is that the importance of different model parts has to be assessed during inference as well, which results in additional overhead. To keep this efficient, special considerations have to be made with regards to the hardware the model will run on. For instance Liu et al. (2023) employ inference-time pruning for large language models and use a complex parallel processing algorithm to compute importance in parallel to the actual inference of the model.

Pruning approaches that prune before inference-time are known as *static pruning* while ones incorporating the additional input information are known as *dynamic pruning*[10]. Our approach can be seen as a mix between static and dynamic pruning as we can precompute the information necessary for pruning (using the prompt template) but then perform the actual pruning during inference.

We focus on pruning model parts for which pruning does not require special hardware considerations to be efficient, but instead, for which the removal is naturally easy to do just-in-time.

Structured and Unstructured Pruning. There are many possible ways to represent a model M as a set of distinct model parts. For instance, one way is to model M as a set of neurons and connections between them, the connections corresponding to a singular weight in the network. However, it is also possible to regard M on a coarser scale such as being made up of a set of layers.

Mathematically, as long as we can represent the removal of model parts this is irrelevant for the calculation of importance or the identification of a performant submodel $\hat{M}$.

[10]Note that dynamic pruning has large overlap with a field called *conditional computation* that aims to only use a subset of model parts during inference by deciding on the fly which to include and which not. While this is in general an idea similar to dynamic pruning, some research trends are more often specified as conditional computation. For instance, Mixture of Experts-based models (Shazeer et al., 2017), where certain model parts, the experts, are present multiple times in a model and only a single version is used during inference which is decided based on a small additional gating network. In this way the number of parameters in the model can be much higher (different experts can handle different inputs which present other challenges) while computational cost can remain low (only few experts are used during inference).

However, it does play a role once it comes to implementing a model in physical hardware. At the level of individual connections between neurons, it is hard to achieve substantial space or time reduction of the model even when pruning away comparably large fractions of the model (Wang et al., 2020). This is because in practice, neurons and their connections are represented in matrices (or more generally tensors). Removing a connection between neurons corresponds to setting a weight to 0. But as it is part of a larger matrix, we cannot manifest a substantial space or time gain. Unless specialized algorithms for sparse matrices are employed, we still have to represent the 0 on the physical machine and it still has to be multiplied with during matrix multiplication. However if we choose to remove rows of the matrix (representing neurons) or even the entire matrix (representing an entire layer of a DNN), we do not run into this issue.

Pruning at the level of connections between neurons is therefore known as *unstructured pruning*, while pruning at the level of bigger model parts that correspond more directly to how the model is represented in physical hardware is known as *structured pruning*. This is not a completely binary distinction, as there are different levels of difficulty associated with efficiently representing pruning. For instance, while both considered structured pruning, removing rows of a matrix still has overhead as compared to whole layer pruning. This is because for the former, to remove a row, the matrix typically has to be reallocated as such modifications are not possible in-place. This is not the case if we remove the entire matrix.

While this type of overhead is often not relevant, it is if the pruning process is time-critical, as is the case for dynamic pruning and, by extension, our approach. The model parts we choose to prune require next to no additional overhead.

Chapter 3

Related Work

3.1 Overview

In this Chapter we describe the works most closely related to this one subdivided into four directions of research.

3.2 Pruning Transformers

While model pruning is a general technique applicable to many kinds of neural network architectures, a number of approaches have emerged that focus specifically on pruning transformer-based models similar to the ones we focus on.

This includes unstructured pruning approaches such as SparseGPT (Frantar and Alistarh, 2023), which proposes a new algorithm to jointly identify unimportant weights as well as adapt the unpruned non-zero ones efficiently by representing pruning as a set of large-scale sparse regression instances. This allows them to scale up pruning to very large model sizes. They further extend their setup to the semi-structured pruning case (unstructured pruning but with regularities where the sparsity occurs). As part of their work, they find that they can prune up to 60% of the models OPT-175B (Zhang et al., 2022) and BLOOM-176B (Le Scao et al., 2023) without significant accuracy degradation.

Furthermore, there is a number of structured pruning approaches like LLM-Pruner (Ma et al., 2023) that focus on on pruning model parts at a level that can induce size reduction and speedup without additional considerations. LLM-Pruner uses a sensitivity-based

pruning approach to identify unimportant parts and removes them. They identify these parts using an algorithm that discovers dependent structures in models automatically. Afterwards, they restore performance using LoRA as an efficient alternative to fine-tuning applicable to large models. Bansal et al. (2023) prune away feedforward blocks and individual attention heads using a similar sensitivity-based pruning approach on OPT-66B (Zhang et al., 2022). They focus on a prompting setting and aim to prune and identify model parts both unimportant and important independent of the concrete prompt template used. They find that many important attention heads correspond to induction heads, i.e., attention heads that allow the model to successfully repeat patterns in its input sequence. Unlike our work they do not investigate at a level of individual prompt templates to understand if important and unimportant model parts differ between them nor do they consider the lack of available data that is common when using prompting.

Approaches that identify sparsity at a level that is applicable just-in-time also exist like Men et al. (2024) and Gromov et al. (2024). Both of them prune whole transformer layers, a type of model part we identify as suitable for just-in-time pruning as part of our approach (however they do not actually prune just-in-time) and score the importance of layers by measuring the similarity of the input and output of each. Then they remove the least important ones. They find that a substantial degree of sparsity is possible across multiple models. Gromov et al. (2024) further explores repair, however, not in a just-in-time manner like we do. Further, neither focuses on sparsity at the level of individual prompt templates.

3.3 Data-free Knowledge Distillation

One of the aspects we want to investigate with regards to prompt template specific sparsity is identifiability even without access to additional data. This kind of constraint has not seen substantial research with regards to model pruning (all the approaches outlined above require access to additional data to estimate importance), however has seen use in the related subfield of knowledge distillation. Instead of making an existing model smaller, the idea of knowledge distillation is to distill the information present in a big model T, the teacher, into a smaller model S, the student. This works by training S to mimic the behaviour, i.e., output values of T on a set of input datapoints.

Data-free knowledge distillation removes the constraint of requiring additional datapoints by exploiting the properties of T. For autoregressive T we can do so by using it to

generate data points on its own instead of using manually created ones. For instance Ye et al. (2022) exploit this to train task-specific small student models from a general GPT-3 teacher by using prompting to generate training data. West et al. (2022) employ a similar approach more specifically suited to commonsense reasoning and even find that the trained student outperforms its teacher in this case. Hou et al. (2023) go one step further and combine both data-dependent traditional knowledge distillation with its data-free counterpart (only unlabelled data is required for their approach) in one by making a single student mimic both a traditional teacher using few labelled data points as well as another teacher using model-labelled data.

In our approach we employ a similar strategy of generating data using the model itself, however we do not use it for distillation. Instead, we use it to estimate importance of model part to remove unimportant ones.

3.4 Scaling-based Model Repair

Another aspect we explore is the repair of a pruned submodel, i.e., restoration of performance to a level similar to that of the unpruned model. We do so by scaling the output features of the remaining model parts, i.e., we modify their interaction.

Prior work has examined similar strategies as part of pruning in other settings. For instance Kwon et al. (2022) study the efficient pruning of smaller transformer models at the level of attention heads (among others) and as part of this process learn a binary multiplicative mask that masks out unimportant heads. Applying the mask has the effect that the model is ”virtually“ pruned in the sense that the output will be the same as if the heads were physically removed. To improve performance they then propose to relax this mask to allow the unpruned heads to have values other than 1, in effect scaling their contributions to the model output. A similar strategy is proposed by Lian et al. (2022) and presented as a parameter-efficient fine-tuning alternative (their approach does not prune) and find that this kind of adaption can effectively improve the task-specific performance of multimodal models.

Chapter 4

Methods

4.1 Overview

In this Chapter we detail our approach to identify sparsity in large autoregressive language models. We design our approach so that we can address the three challenges outlined in the introduction:

1. Existing approaches do not investigate sparsity with respect to a fixed prompt template. We want to understand if it exists and differs between prompt templates.
2. Even if this sparsity exists, it is unclear if it can be identified without additional data. We want to understand if this is possible.
3. The sparsity we find should be able to be applied just-in-time. We want to prune at a level of model part types that allows for this.

In this Chapter we first look at how to choose a model part type to prune to address (3). We present a sufficient condition for the choice of model parts to prune such that they are easy to remove just-in-time. This condition leads us to propose that one should prune attention blocks, feedforward blocks or whole transformer layers.

We then showcase our process for pruning. It takes in a model to prune M modelled as consisting of parts p_i as well as a prompt template prompt_t for task t as input. The approach is made up of four individual steps, which we additionally visualize in Figure 4.1:

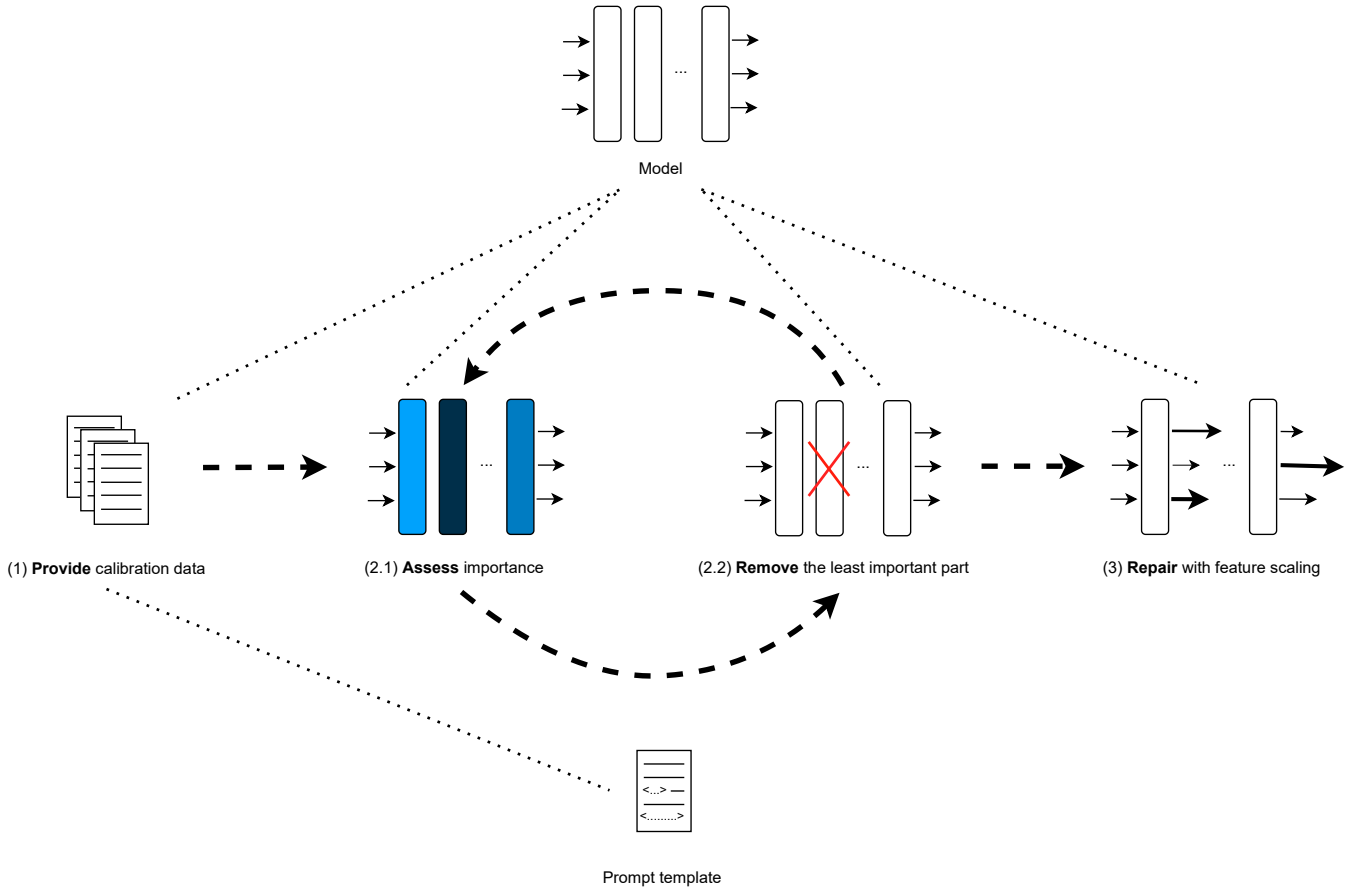


Figure 4.1: Overview of our approach. We first generate data necessary for pruning and repair using the prompt and the model. Then we repeatedly assess the importance of all model parts and remove the least important one. Once we decide to stop pruning, we repair the pruned model using feature scaling between model parts.

1. **Provide calibration data** For the other steps in our approach we need access to task data, both to identify unimportant model parts (pruning), as well as to minimize the impact their removal has (repair). We propose two ways to provide this data: (a) Use real task data, or (b) Use data generated using M and prompt_t using constrained generation. With (a) we can show that sparsity exists at the level of just-in-time prunable model parts in principle and with (b) we can test the hypothesis that sparsity is also identifiable using a natural combination of the information in M and prompt_t in the form of generated data. Using (a) we can understand what shape prompt template specific sparsity takes and address challenge (1). Using (b) we can understand how much of this is identifiable without additional data addressing challenge (2). We call the data we obtain from this step the calibration dataset. In Figure 4.1 this step is reflected on the leftmost side. The dotted lines refer to the input each step requires, in this case both the model and the prompt template.

2. **Assess importance** We assign importance scores to all model parts. Importance for a model part p_i is measured as the approximate average sensitivity of the loss on the calibration dataset when removing p_i. Low sensitivity implies that p_i is unimportant.

3. **Remove the least important part** After assigning importance, we remove the least important model part p_{min}. After this step is complete we go back to assessing importance on the pruned submodel $M \setminus \{p_{min}\}$ if we have not yet reached a level of sparsity we are satisfied with. In Figure 4.1 this is reflected by the two arrows leaving this step, indicating both possibilities.

4. **Repair with feature scaling** Finally, as an optional step, we can repair the final pruned model. We propose to do so not by modifying the remaining model parts but instead scaling the hidden states that gets passed between them. This is highly parameter efficient and has low runtime overhead. As such it allows us to preserve just-in-time application (of both pruning and this repair step).

4.2 Choice of Model Part Types to Prune

In our setting, we focus on sparsity that can be induced just before running M with input x, i.e., we remove unnecessary model parts just-in-time. We then unprune the removed parts directly afterwards and apply a different sparsity for the next input to M. In general

this can be difficult as the pruning of many model parts requires precise modification of the underlying representation of a model M on real hardware. For instance, as linear layers are commonly represented as matrices, pruning a neuron from this layer in this representation is equivalent to removing a row or column of the matrix (rows represent output neurons and columns input neurons). Commonly this removal cannot be done in-place and instead requires the reallocation of a new smaller matrix. Further, to unprune the neuron later, storing the original is necessary as well. This results in both a memory as well as speed overhead.

We identify a class of model part types where removal is trivial to apply just-in-time: Model part types equipped with a *residual connection*. For such a model part p the corresponding transformation f_p applied to the input is of the form $f_p(x) = x + h(x)$, where h is some non-trivial transformation such as an internal neural network. Then removal of p in M is trivially done by not doing the computations associated with h during inference and instead passing the input to p directly as the output. For a removed part p it then holds that $f_p(x) = x$. In some sense this allows us to "virtually" remove p temporarily without having to do anything to the model itself.

Residual connections also give us another necessary property for prunabilty for free: The input dimension of f_p matches the output dimension. If this were not the case it would be categorically impossible to prune M at the level of p because then the function implemented by $M \setminus \{p\}$ is no longer well-formed as the output dimensions of the previous parts do not necessarily match the input dimensions of the part after p. This is a problem, since the part responsible for changing the dimensions to fit, p, has been removed.

For the transformer-based large language models we study, we identify three types of model parts that possess a residual connection: Transformer layers, attention blocks and feedforward blocks. A transformer-decoder consists of a list of transformer layers applied to the embedded input one after the other. Each one consists of an attention block (internally composed of multiple attention heads) followed by a feedforward block of fully connected linear layers (not to be confused with transformer layers). Both attention blocks and feedforward blocks have residual connections. This means they can be trivially pruned. By pruning both together we can also implicitly prune the entire transformer layer.

Therefore, we prune *transformer layers* (further referred to as just layers), *attention blocks* and *feedforward blocks*.

4.3 Pruning Process

4.3.1 Provide Calibration Data

Both the importance assessment as well as the repair step of our approach require a dataset containing task data D to identify unimportant model parts or repair the interaction between those remaining in the pruned model respectively. As we want to answer both if sparsity *exists* in principle as well as if it is *identifiable* using just the model M and the prompt template prompt_t (for the task t), we employ two different ways to obtain D.

- **Real data** For this setting, we sample 256 real datapoints from an existing training dataset for our task. We refrain from sampling from the test dataset to prevent overfitting and identifying a sparse submodel that performs well on the seen test data but fails to perform well generally across all possible task instances and not only those in the test dataset.

- **Generated data** For this setting, we generate 256 fake datapoints using M and the prompt template prompt_t.

  The generation process to create task data for t is a simple generalization from the process to solve an instance of t. When solving a task instance x using M and prompt_t, we first create the instance-specific prompt $\text{prompt}_t(x)$ and then generate the solution to x (or compute the likelihood of multiple competing completions, representing possible classes for a classification task). To generate not only the solution but also a task instance x', we use a prefix of the prompt template up to the point where x is inserted and instead of generating a solution generate x' in this way. For instance, if the prompt template is "Movie review: <sentence> Sentiment:", we would use "Movie review: " as the prompt to generate movie reviews. To determine if the instance is fully generated or if there are still tokens missing, we check if the instance generated so far ends with the suffix of the prompt template and finish if it does, e.g., " Sentiment:".

  If it is the case that x is composed of multiple parts, e.g., multiple sentences, we always generate each individual part using the prompt prefix containing all previously generated parts. For instance for the prompt template "Sentence 1: <sentence1> Sentence 2: <sentence2> Do these sentence mean the same thing:", we would first generate sentence 1 using "Sentence 1:" and then sentence 2 using "Sentence 1: I

am a generated sentence Sentence 2:" where "I am a generated sentence" was just generated. If t is a multiple choice task where an instance is a set of multiple answers, we generate multiple answers as well (equally many as is used for real task instances).

Once we have generated all parts of x', we can treat it like a regular datapoint and predict a solution (since we do not have access to ground truth as it does not exist here). Together, this forms an instance, solution pair we use as part of D.

4.3.2 Assess Importance

To decide which model part to prune next we calculate an importance score for each candidate. In this context, importance refers to the impact of removal on the task-specific accuracy. We choose to approximate this impact using the gradient-based method introduced in Section 2.4.2. That is to say we define the importance as

$$\text{Imp}_M(p_i) = \mathbb{E}_{(x,y)\in D}|\nabla_{\alpha_i}\mathcal{L}(x,y)|.$$

Here α_i refers to a new mask parameter we introduce for each model part p_i. We modify the computation of $f_{p_i} = x + h(x)$ to $f_{p_i} = x + \alpha_i h(x)$. As a value of 0 for α_i is equivalent to the removal of p_i, and 1 is identical to leaving the model part in, we can capture the importance of p_i in the gradient with respect to α_i, as previously explained. For this to work the value during inference for all α_i is set to 1.

We define the loss $\mathcal{L}$ as

$$\mathcal{L}(x,y) = -\sum_{i=1}^{n} \log(P_M(y_i|\text{prompt}_t(x); y_{1:i-1})),$$

where ; is the concatenation operator and P_M the probability distribution spanned by M across the likely next token. This formulation is just the negative log-likelihood loss on the correct completion y that solves x (or class representation in the case of a classification task).

We choose this way of calculating importance, as it has been empirically shown to closely resemble precise measures of importance like oracle importance while being efficient to compute at the same time (Molchanov et al., 2017).

This way of calculating importance is equivalent to that found in Bansal et al. (2023), except for the fact that we keep the prompt template identical across data instances.

4.3.3 Remove the Least Important Part

Once importance is calculated for all $p_i \in M$, we remove the model part with the lowest importance p_{min}. Then we start again from the importance assessment step but on the pruned model $M \setminus \{p_{min}\}$ until we are satisfied and have reached a sufficient amount of model parts or hit some other predefined exit condition. As part of our experiments, this exit condition is something we often vary in order to show how sparsity looks like in different conditions and settings.

The formulation of pruning as presented here is known as greedy pruning, where we always remove the currently least important model part. Past work has shown that this kind of pruning finds sparsity close to the true optimum (Peer et al., 2022), which in general is computationally hard to identify (Molchanov et al., 2017).

If we were to not recompute importance on the pruned model but instead reuse the scores from the original, i.e., only compute importance once at the start and then remove model parts sorted by importance up some amount of desired sparsity, then this would be equivalent to the approach to pruning employed by Bansal et al. (2023) as well as Michel et al. (2019).

4.3.4 Repair

As an optional step we propose to repair the final pruned submodel to restore some performance by scaling the output features of the remaining model parts. Specifically, we suggest repurposing the mask variables introduced to calculate importance α_i for each remaining model part p_i. The key insight here is that the values for α_i can be relaxed from always being 1 like for importance estimation. Instead, we can replace each α_i by a vector equal to the size of the output dimension of f_{p_i} and allow its values allowed to vary. That is to say we replace $f_{p_i} = x + h(x)$ with $f_{p_i} = x + \alpha_i \circ h(x)$ where $\alpha_i \in \mathbb{R}^H$ is now a vector and H is the size of the output dimension of f_{p_i}.[1] We can then learn good values for each α_i by means of gradient descent on the loss function $\mathcal{L}$ (for dataset D) also introduced for importance assessment. We call this method of repair feature scaling as it scales the output features of $h(x)$ for each remaining model part before they get passed to the next

[1] As we work on transformers, we implicitly mean the output dimension of each returned embedding, not their totality. The multiplications used are also applied separately to each embedding.

one. Because the amount of additional parameters and multiplications is small compared to other more substantial operations in the model (such as matrix multiplication), the overhead of this relaxation is small and as such this repair mechanism can also be easily applied just-in-time.

We hypothesize that this repair mechanism is enough to substantially increase the size of sparsity possible before accuracy begins to deteriorate as it is able to attenuate features erroneously increased by the removal of a model part as well as amplify features that have similarly been lessened because of missing parts. For instance, it could be that while in the original model two different parts contributed to a feature relevant for sentiment analysis that only a single one remains in the pruned deteriorated model. Then this repair mechanism can restore the size of the original feature by doubling the scale of the contribution of the remaining model part (assuming that the degree of contribution of the parts to this feature are mostly uncorrelated to that of the rest of the model parts in M).

Finally, we would like to note that the presented repair mechanism is reminiscent of similar approaches such as the relaxation of a pruning mask for attention heads (Kwon et al., 2022) or as part of a general mechanism for parameter-efficient fine-tuning (Lian et al., 2022), however is not applied in the same manner (but instead just-in-time).

Chapter 5

Implementation Details

5.1 Overview

In this Chapter we provide a brief overview of how we implement some aspects of our approach in practice. We first showcase how our approach implements prunable models and then briefly outline how we we evaluate them on a set of tasks. Finally, we also go over how we count floating point operations (used for RQ 4).

5.1.1 Model Implementation and Pruning

We use the HuggingFace Transformers library (Wolf et al., 2020) built on top of PyTorch (Paszke et al., 2019) as the basis of implementation for the models we prune. In order to modify the default behaviour we subclass the main implementation of each model and overwrite the method responsible for implementing the forward pass, i.e., taking in a sequence of tokens and producing probabilities over the next ones.

We modify the code of the model in two ways:

1. We introduce mask variables α_i^t for each model part p_i^t of model part type t that are multiplied with the non-trivial computation component associated with p_i^t, i.e., $h_i^t(x)$ before being added to the residual.

2. We introduce boolean variables $pruned_i^t$ similarly to the mask variables. Then we introduce a new if-condition in the code that checks if the variable is true. If not, the

computation for p_i^t is carried out as normal. If yes, the computation is skipped and the residual is passed on to the rest of the code.

In combination we modify the code as shown in Figure 5.1.

Unmodified

```
def forward(x):
    for each model part p_i^t:
        res = x
        x = h_i^t(x)
        x = x + res
    return x
```

Modified

```
def forward(x):
    for each model part p_i^t:
        res = x
        if not pruned_i^t:
            x = α_i^t * h_i^t(x)
            x = x + res
        else:
            x = res
    return x
```

Figure 5.1: Pseudocode that shows the modifications made to the forward pass of a model by our approach. Note that for simplicity we intentionally do not provide any specifics in the order of the p_i^t in the for-loops as it is irrelevant to understand our modifications. In practice, the ordering of course is carefully chosen and fixed in advance. Left: Unmodified code. Right: Modified code with pruning and repair logic.

We introduce code modification (1) in order to calculate gradients with respect to the mask variables, both to assess importance as well as to repair the model. For the latter, we then simply substitute each α_i^t using a vector while PyTorch takes care of the proper broadcasting of the multiplication in the background. We introduce code modification (2) in order to physically prune the model: When $pruned_i^t$ is true the computation is skipped. To then prune a model just-in-time all that needs to happen is the updating of each $pruned_i^t$ variable to signal the model what it should compute and what not. This has minimal overhead and does not require any modification of the underlying model representation.

5.1.2 Evaluation

In order to evaluate our modified models we use LM-Evaluation-Harness (Gao et al., 2023), a library specifically designed to evaluate language models. We manually modify the library to include additional options we need for evaluation. Namely, the ability to fix a prompt used for all instances of a task (instead of a new one being sampled for each by default) among others used for testing.

5.1.3 Measuring FLOPs

As part of our experiments, we need to calculate the number of floating point operations (FLOPs) used as part of the model inference. We aim to be as accurate as possible and therefore we use an approach based on PyTorch tracing that tracks at the level of individual PyTorch primitives (such as matrix multiplication) how many times each one occurs. Then, for each of these primitives, we have a fixed formula that derives the number of floating point operations necessary. This is implemented in the Fvcore library (Wu, 2024).

In order to interface with LM-Evaluation-Harness, we design a tiny wrapper that can wrap any HuggingFace model to automatically track FLOPs used in this manner when calling the model for inference. We use this version of a model instead of the default one during evaluation when tracking is required.

Chapter 6

Experiments

6.1 Overview

In this Chapter we conduct experiments to investigate prompt template specific sparsity at the level of just-in-time prunable model parts.

We break down our investigation into two fundamental aspects: (a) existence of prompt-specific sparsity and (b) identifiability of prompt-specific sparsity. For (a) we want to show that sparsity exists at all, even with the help of additional data. For (b) we want to show we can find this sparsity even without additional data, i.e., the realistic setting for prompting. We hypothesize that the natural way of combining the underlying model with the prompt template in the form of using both together to generate data on its own is sufficient for this purpose.

Furthermore, we investigate two additional considerations that arise as a consequence of our setting: (c) given that the model parts we prune are big, it is likely that performance deterioration after some threshold of sparsity is not necessarily due to some parts of M stricly required to solve t being removed. Instead, it could just be that the interaction between the remaining layers is damaged, which when repaired together can still address t. We hypothesize that we can mitigate this in a just-in-time manner by scaling the individual output features of the remaining model parts. This additionally serves as a lower bound for more invasive repair mechanisms like full fine-tuning: If feature scaling can repair the model, more powerful mechanisms should be able to do so as well. (d) While pruning at the level of some types of model parts might yield higher levels of possible sparsity than others

this has to be contrasted against the fact that removal of different types of model parts yields differing amounts of speedup. This is because the amount of FLOPs used might differ between part types. For instance a full transformer layer trivially always uses more FLOPs than each of its subcomponents on their own. Moreover, a repair mechanism such as (c) introduces further overhead in terms of FLOPs that has to be traded-off against a potentially larger sparsity it brings. Therefore, in addition to investigating the size of the sparsities found with pruning and repair across different model part types, we also want to compare them in terms of the speedup they bring, as measured by FLOPs.

In summary, we investigate the following four research questions:

- **RQ 1** Does prompt-specific sparsity at the level of just-in-time prunable model parts exist (as measured by greedy pruning using sensitivity-based importance)?
- **RQ 2** Is prompt-specific sparsity at the level of just-in-time prunable model parts still identifiable using model-generated data?
- **RQ 3** Is scaling of features between model parts in a badly-performing pruned model sufficient to restore accuracy to a level similar to the base model?
- **RQ 4** How does prompt-specific sparsity induced by different types of model parts as well as repair mechanisms compare to each in terms of speedup as measured by FLOP reduction?

6.2 Task, Prompts and Models

We answer our research questions for two models on a set of five tasks, each represented three times as three distinct prompt templates.

6.2.1 Tasks

We evaluate on a set of five tasks:

- **COPA** is a *commonsense causal reasoning* task. Each instance consists of a premise and two alternatives. The model has to determine which alternative is more likely to be causally related to the premise (Roemmele et al., 2011).

- **MRPC** is a *paraphrase detection* task. Each instance consists of two sentences. The model has to determine if they are paraphrases of each other (Dolan and Brockett, 2005).

- **PIQA** is a *commonsense reasoning* task. Each instance consists of a physical goal and two solutions. The model has to choose which solution most likely accomplishes the goal (Bisk et al., 2020).

- **SST-2** is a *sentiment analysis* task. Each instance consists of a sentence from a movie review. The model has to determine if the sentence has a positive or negative sentiment (Socher et al., 2013).

- **Web Questions** is a *question answering* task. Each instance is a question which concerns factual information. The model has to generate the correct answer (Berant et al., 2013).

We choose these tasks as they cover a broad spectrum of different topics such as question answering, commonsense reasoning as well as more basic operations for language modelling such as sentiment analysis and paraphrase detection. Furthermore, none of them has a test set of more than 5000 instances which makes it feasible to evaluate on them repeatedly and for many pruned submodels. Finally, they cover three different types of tasks: classification tasks (SST-2, MRPC), generation tasks (Web Questions) and multiple choice tasks[1] (COPA, PIQA).

In line with previous work (Zhang et al., 2022; Bansal et al., 2023), we evaluate each task in terms of accuracy. For SST-2 and MRPC it would be possible to compute an F1 score instead, however this is largely unnecessary as the classes for both tasks are relatively balanced with around 52% of instances in the majority class for SST-2 and 68% for MRPC respectively. Thus accuracy and F1 will have similar values.

For multiple of our research questions, the concept of *relative accuracy degradation* is relevant to define when a pruned model is no longer performing approximately as well as the original model. For the accuracy of the pruned model $\text{acc}_t^{\text{pruned}}$, the accuracy of the

[1]The way we extract an answer using prompting for multiple choice tasks is identical to the strategy used for classification tasks described in Section 2.3.2 except that the "classes" differ for each instance of the task.

original model $\text{acc}_t^{\text{original}}$ as well as the trivial performance $\text{acc}_t^{\text{majority}}$ as measured by the majority class baseline[2] for the test set of t, we define the relative accuracy degradation as

$$deg = \frac{\text{acc}_t^{\text{original}} - \text{acc}_t^{\text{pruned}}}{\text{acc}_t^{\text{original}} - \text{acc}_t^{\text{majority}}}$$

with the assumption that $\text{acc}_t^{\text{original}} > \text{acc}_t^{\text{majority}}$ holds. Then, if the pruned model performs equally as well as the original, the degradation is 0, if it performs like the majority baseline, the degradation is 1. For instance if the original model has an accuracy of 0.9 on some task t and a pruned model has an accuracy of 0.89, and the majority baseline has an accuracy of 0.6, then the relative accuracy degradation would be 0.025 or 2.5%.

6.2.2 Prompts

For each of the tasks we use multiple prompt templates to represent it. This way we can understand if the prompt template used affects the model parts the model is sparse in. Following the default layout in LM-Evaluation-Harness (Gao et al., 2023), we choose to adopt the meta-template in Figure 6.2. This template consists of representations of instance inputs and outputs for a set of examples $e_1, \ldots, e_n$ as well as the representation of the instance input x, all separated by two newlines each. We then pick, for each task, three prompt templates as parameterized by the examples used, by randomly selecting five examples at a time from the training dataset of each task. While it would also be interesting to evaluate on prompt templates with different numbers of examples, we refrain from doing so out of computational constraints. Moreover, we do not expect large differences in results anyways based on prior findings (Bansal et al., 2023).

In total we create 15 prompt templates in this way, three per task. Figure 6.1 shows an example template for SST-2.

[2] A predictor that always predicts the class with the most elements.

if you give a filmmaker an unlimited amount of phony blood
Question: Is this sentence positive or negative?
Answer: negative

lax
Question: Is this sentence positive or negative?
Answer: negative

to the edge of your seat
Question: Is this sentence positive or negative?
Answer: positive

take (its) earnest errors and hard-won rewards over the bombastic self-glorification of other feel-good fiascos like antwone fisher or the emperor's club any time
Question: Is this sentence positive or negative?
Answer: positive

with second helpings of love, romance, tragedy, false dawns, real dawns, comic relief, two separate crises during marriage ceremonies, and the lush scenery of the cotswolds
Question: Is this sentence positive or negative?
Answer: positive

<Real input>
Question: Is this sentence positive or negative?
Answer:

Figure 6.1: Example of a prompt template we use for SST-2. We first have 5 input-output examples and then the space for the real input.

```
<Example 1 input><Example 1 output>

<Example 2 input><Example 2 output>

...

<Example n input><Example n output>

<Real input>
```

Figure 6.2: The prompt metatemplate used by LM-Evaluation-Harness. Examples are seperated by two newlines each.

6.2.3 Models

We evaluate on the following two models:

- **Mistral-7B** A 7 billion parameter generative language model trained by Mistral AI based on the transformer-decoder architecture. Across an array of benchmarks Jiang et al. (2023) demonstrate that this model outperforms many similarly sized models such as LLama 7B and even outperforms larger models such as LLama 13B on some tasks. As such it represents a state-of-the-art level model practically relevant for current uses of prompting. It features a collection of small architectural tweaks over the basic architecture such as grouped query attention and sliding window attention in the attention blocks (irrelevant during pruning as we remove the block as a whole, however relevant for increasing the performance and throughput of the model). Mistral is trained on an undisclosed collection of web text.

  In order to fit the model into memory, we quantize it (reduce the precision of the weights) using the LLM.8bit() algorithm (Dettmers et al., 2022).

- **GPT2-XL** As part of the original GPT-2 Series of models as introduced by Radford et al. (2019), this model serves as a smaller model (1.7 billion parameters) to compare against a state-of-the-art system like Mistral. The training data for GPT-2 is

WebText, a high-quality webscrape. The architecture used is a regular decoder-only transformer architecture.

6.3 RQ1: Existence of Sparsity

6.3.1 Overview

We answer the question whether or not sparsity exists by applying our pruning methodology outlined in Chapter 4 on Mistral-7B and GPT-2-XL for the five tasks, each represented by three prompt templates. For each of them we prune each of the three model part types we study (layers, attention blocks and feedforward blocks). In total we prune $2 \cdot 6 \cdot 3 \cdot 3 = 108$ configurations in this way. Note that we use *real task data* for pruning for this RQ as we only want to show existence in principle (RQ 2 will use *generated data* instead). Also: We do not repair for this RQ.

For each configuration we stop pruning once the pruned model is no longer approximately as accurate as the unpruned model. This is operationalized by a relative accuracy degradation of at least 5% measured on the task-specific test dataset, i.e., real task data. We choose relative accuracy degradation as our metric here to make the configurations comparable across different models and prompt templates, for which the base accuracy and majority baselines might differ.

This approach can ascertain if sparsity exists at all for a configuration and if yes in how many parts the model is sparse (the special case that the model is sparse in at least one part implies that sparsity exists). Under the assumption that our pruning approach reveals sparsity close to the optimal possible size before exiting it can also tell us that sparsity does not exist in case it does not find it. In reality, it is probable that this assumption is violated to an extent, however in any case we can always demonstrate a *lower bound* for possible sparsity size. After answering this research question we conduct some additional analyses to understand the extent to which this is the case.

6.3.2 Results

We present the results in Table 6.1. The table shows the percentage of sparsity (number of removed parts divided by number of all parts of that type) achieved before a 5% relative

Table 6.1: Percentage of model parts that can be removed without impacting accuracy for Mistral-7B and GPT-2-XL when pruning with real data. For each task the the percentages are averaged across running the same setup for three prompt templates. Values in bold indicate that for at least one prompt template no sparsity could be found, i.e., no model parts could be removed without accuracy deterioration. Non-bold values show us that sparsity exists across all prompt template for that setting.

		Task				
Model	Model Part Type	COPA	MRPC	PIQA	SST-2	Web Questions
Mistral-7B	Attention Blocks	38	**6**	31	34	21
	Layers	7	4	4	23	3
	Feedforward Blocks	9	**5**	5	22	3
GPT-2-XL	Attention Blocks	12	–	39	32	**17**
	Layers	**9**	–	13	**2**	3
	Feedforward Blocks	36	–	16	**2**	5

accuracy drop is introduced for the models, model part types and tasks. We call these values maximum sparsity. The percentages for the three different prompt templates per task are averaged. Values in bold indicate that at least one of them is 0, i.e., no sparsity could be found. GPT-2-XL has no results for MRPC as the base model does not perform better than the majority baseline. In this case the degradation is already 100% before any pruning even occurs so pruning does not make sense.

We can see that some task-specific sparsity is possible across almost all models, model part types and tasks. In particular for each possible configuration of model, part type and task there exists at least one prompt template that allows for sparsity. Furthermore there are only 2 and 4 settings for Mistral-7b and GPT-2-XL respectively where the models are not sparse at all for some prompt template (corresponding to the bold values). Aside from this however, results vary strongly in the amounts of sparsity achieved. This is true across models and part types. Still, some trends exist:

- Attention blocks appear less important than feedforward blocks or entire layers for both models. For Mistral-7B maximum sparsity of attention blocks is higher than the other two types of model parts for all five tasks and for GPT-2-XL the same holds

true for three out of four tasks (excluding MRPC).

- Sparsity for feedforward blocks and layers often is of similar size even though one contains the other, however not always, e.g., the results for GPT-2-XL on COPA with a 27% difference in average maximum sparsity size between the two.

Aside from these patterns, results vary a lot. To some degree this is to be expected between models (different training data) and tasks (different necessary skills), however they also vary between prompt templates for the same task. For instance, sparsity on Web Questions when pruning attention blocks from GPT-2-XL varies from 0% (for two templates) all the way to 58% of attention blocks being removable without deterioration of accuracy (which averages out to 17% shown in the table). This either indicates that there are model parts sparse with respect to a specific prompt template (and not the others) or that our approach was not able to identify sparsity due to its limitations.

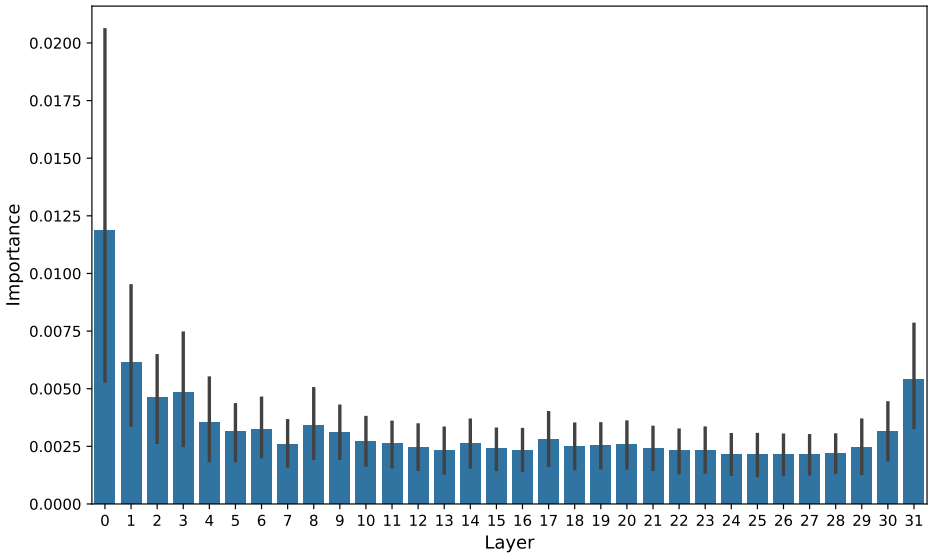
6.3.3 Analysis

We further investigate with regards to which actual model parts were able to be removed without causing accuracy deterioration as well as the importance of the remaining model parts during the pruning process on Mistral-7B. Additionally, we compare our sparsity results to a random baseline and a strictly more powerful, but computationally prohibitive, version of our pruning approach. This allows us to better understand how well our pruning approach performs in comparison.

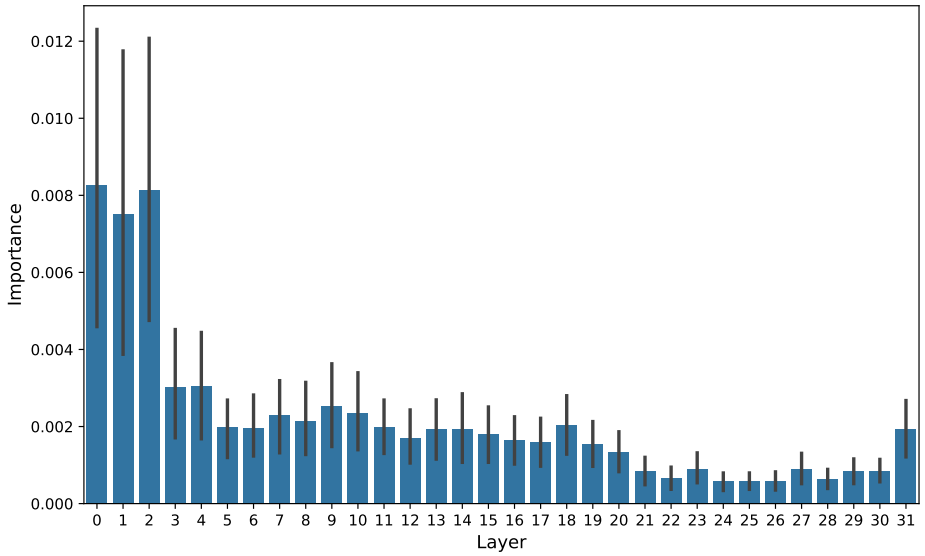
Detailed view on Identified Sparsity. We make the following observations when analyzing the identified sparsity on Mistral-7B:

1. **Parts near the middle of a model are less important than those at the beginning and end** We investigate if there are general trends across both tasks and prompt templates in the importance of model parts. To do so we compute the importance of each part (on the original unpruned model) for all tasks and prompt templates for those tasks.

   Averaged results of this procedure on layers as the model parts are presented in Figure 6.3a. As we can see overall the importances of model parts is much larger near



(a) Importances of whole transformer layers.



(b) Importances of attention blocks.

Figure 6.3: Importances of attention blocks and transformer layers averaged across all tasks and prompt templates for Mistral-7B. Early and late layers are more important on average than middle layers. The black bars indicate the standard deviation.

the beginning of the model and right at the end (however the standard deviations are also larger for both). In comparison importance of the middle model parts is relatively evenly distributed. Results for the pruning of feedforward blocks are similar. When investigating the same data but for attention blocks we again make similar findings however discover that additionally there appears to be a collection of blocks consistently unimportant across all tasks and prompt templates near the end of the model (less important than the rest of the intermediate model parts). We show this in Figure 6.3b.

2. **Importance of model parts is stable for some tasks as parts are removed and not for others** We also take a look at the importance scores of the model as it is being pruned. This allows us the see if the importance scores of the submodels that are obtained by progressively removing parts are similar to that of the full model. We do this to understand if the removed model parts independently contribute to good performance on the task, or if there are dependencies, e.g., two different model parts fill the same role so removing one is not an issue, however removing both is. We expect to see strong shifts in importance scores if the latter is true.

   We present exemplary results for one prompt template on COPA when pruning layers in Figure 6.4. We can see that the removal of layer 3 makes the surrounding layers more important (however otherwise the removal of other layers plays a small role). In general, we find that for some tasks (such as SST-2) model parts do indeed seem to contribute independently as importance stays similar as parts are removed, while for others we can see a dependency of the type in the example.

3. **Obtained sparsity is similar for some tasks, model part types and prompt templates but not others** In addition to our investigation on the importance scores, we also aim to understand the actual model parts Mistral-7B is sparse in, i.e., all the parts removed until we exit because of deterioration. Namely, we want to ascertain if they are similar when pruning for different tasks and prompt templates or not. This allows us to find out if the sparsity our approach finds, i.e., the irrelevant removed model parts, is just task-agnostic sparsity that gets discovered for all settings, or if the sparsity is unique to a specific task or prompt template (so that it preserves accuracy for a specific prompt template but not others). We approach this

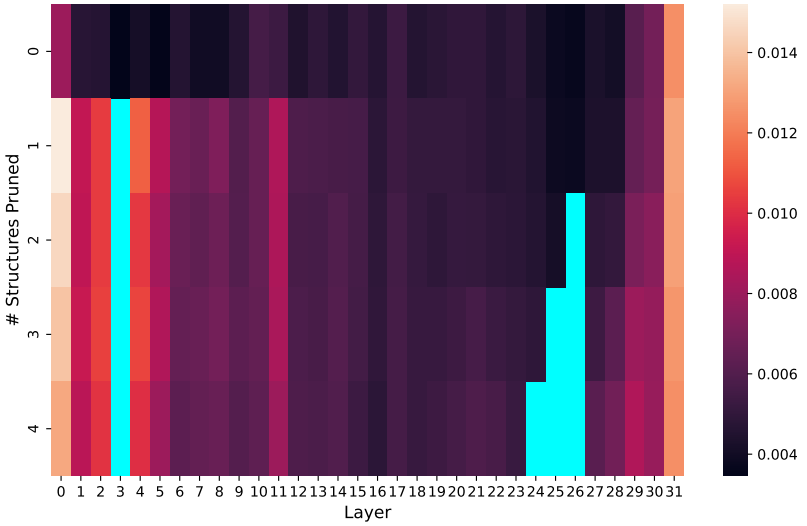


Figure 6.4: Importance scores of the different layers of Mistral-7B on one of the prompt templates for COPA as layers are removed during pruning until deterioration (always the least important). Removed layers at each step are marked in cyan, other layers are colored according to their importance score (the color bar to the right of the plot). We can see that the removal of layer 3 strongly changes the importance scores of the surrounding layers.

Table 6.2: Jaccard similarity of the maximum sparsity, i.e., the collection of all model parts we can remove without accuracy deterioration, for Mistral-7B. Each entry corresponds to the average similarity of the sparsity identified for the three prompt templates per task and model part type. For "All Tasks", we average across combinations of sparsity for all tasks (not just among the three prompt templates for individual tasks).

		Task					
Model	Model Part Type	COPA	MRPC	PIQA	SST-2	Web Questions	All Tasks
Mistral-7B	Attention Blocks	0.40	0.00	0.77	0.74	0.90	0.43
	Layers	0.17	0.33	0.67	0.34	1.00	0.10
	Feedforward Blocks	0.28	0.08	0.44	0.44	0.33	0.15

by calculating the Jaccard similarity[3] between the different sparsities for each task and prompt template. To answer if it generally differs between prompt templates for a task, we take the average Jaccard similarity of the pruned model parts between the three prompt templates (3 templates lead to 3 comparisons). For instance if we were able to prune layer 2 and 3 for one prompt template, 3 and 4 for another, and just 1 for the third, we obtain Jaccard similarities of 0.5, 0 and 0. Therefore the average is 0.17. Additionally, we compute the average Jaccard similarity between all combinations of prompt templates of all tasks to understand if there also is similarity across tasks. As the jaccard similarity of two sparsities is undefined if they are empty we exclude comparisons of this nature from the averages.

The results are presented in Table 6.2. In each column we can see the average Jaccard similarity across the three prompt templates per task as well as the average of the Jaccard similarity by comparing all prompt templates across all tasks in the final column. As we can see sparsity for attention blocks is consistently higher than for the other two model part types. This holds both per task (except for MRPC) as well as generally across tasks. By manually looking at the obtained sparsities, we find that for many tasks and prompt templates attention blocks are mostly removed between layer 21 and 28, which indicates that these are more universally unimportant. This also matches the low average importance with small standard deviations of attention blocks in Figure 6.3b. For layers and feedforward blocks the found sparsity is more dissimilar. For some tasks this can be explained by the fact that often only a single model part can be pruned and when this differs between prompt templates the Jaccard similarity is 0, significantly lowering the average result in the process. However also when more sparsity is possible for layers and feedforward blocks such as for COPA or SST-2 (see Table 6.1), the similarity can remain low (as is the case for COPA on layers).

It could be that the sparsity found might differ here between prompt templates, but is still usable for another prompt template, i.e., it does not cause accuracy deterioration. When we apply the maximum sparsity found across the three prompt templates for layers on COPA and use it in conjunction with the prompt template with the lowest identified sparsity (not necessary the one it was found for), we find that the sparsity is not transferrable: The performance deteriorates. Repeating the same procedure for

[3]The Jaccard similarity between two sets X and Y is defined as $\text{Jaccard}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$.

Table 6.3: Change in percentage of model parts that can be removed without deteriorating accuracy using random pruning instead of pruning using task data for both Mistral-7B and GPT-2-XL. For each task and model, i.e., each value in the table we average over the change for each of the three prompt templates. For GPT-2-XL, the results are further conducted using three random seeds for each setting and we average over them as well. We highlight values in bold where random pruning identifies sparsity of bigger size than pruning using task data.

		Task				
Model	Model Part Type	COPA	MRPC	PIQA	SST-2	Web Questions
Mistral-7B	Attention Blocks	**+1**	**+2**	−7	−8	−18
	Layers	−4	−4	−1	−14	−3
	Feedforward Blocks	−5	−5	0	−8	−3
GPT-2-XL	Attention Blocks	**+2**	–	−22	−29	−13
	Layers	−2	–	−4	−1	−1
	Feedforward Blocks	−30	–	−1	−1	−2

SST-2 yields no such deterioration. Here the sparsity is transferrable.

Comparison against Random Baseline. We repeat the entire experiment for this research question but instead of pruning the least important model part, we randomly remove one during each pruning step. Pruning in this manner is completely task-agnostic and serves as a baseline for what is identifiable without any additional information from the task data. As GPT-2-XL is much smaller than Mistral-7B we can and do conduct three random pruning runs with different seeds instead of just one run for each configuration and average the results.

We provide our results in Table 6.3. The table shows the difference in size of sparsity identified between our approach using task data and random pruning (we subtract the size of sparsity identified using task data from that identified using random data). We highlight values where random pruning identifies sparsity of the same or bigger size compared to our approach. We can see that it is almost always less than what is found using our importance-based approach using task data except in a handful of cases. Even then the difference is

always below 1-2%. This baseline shows us that the task data is indeed relevant and useful to identify sparsity of bigger sizes. This gives further motivation for RQ 2, in the sense that we would like to replicate the usefulness of task data without having access to it directly.

Testing if a Deteriorated Pruned Model Contains Undeteriorated Submodels. If it holds that a pruned model with deteriorated accuracy does not contain any non-deteriorated submodels, this raises our confidence that our approach identifies sparsity close to the optimal size. This is because we would know that no further pruned and yet well-performing submodel exists in the final pruned model we obtain from pruning until accuracy deterioration.

We repeat our main experiment for this research question with a computationally more expensive version of our approach that identifies such smaller non-deteriorated submodels even if the above assumption does not hold. This way we get a measure for how close our approach is to this stricly more powerful version and how strongly this assumption is violated.

The modified approach works by not stopping pruning once the pruned model shows deteriorated performance (instead we prune until trivial performance is reached). Then we can identify the sparsest submodel that still performs well. Running this in general for all research questions is infeasible as the evaluation of all these submodels would take too much time. This is why we only prune GPT-2-XL here.

We present our results in Table 6.4. The table shows the change in maximum sparsity from the original approach to this one. Maximum sparsity for this analysis is defined as the largest collection of pruned model parts that do not lead to deteriorated accuracy (instead of the last sparsity we obtain before stopping due to deterioration in the original experiment). The table shows the average change across the three prompt templates. We can see that for over half the settings, i.e., entries in the table, the assumption is only mildly violated: Sparsity size is only marginally larger. For SST-2 and COPA, it is violated more strongly: Deteriorated pruned models contain much smaller undeteriorated submodels. However this does not have a big impact on answering this research question: In the table we mark settings as bold, where at least one prompt template fails to identify any sparsity, i.e., the same as for the main experiment and the central tool to answer existence. We find that relaxing the stopping criterion only allows finding sparsity in a single additional

Table 6.4: Change in percentage of model parts on GPT-2-XL that can be removed without deteriorating accuracy when relaxing our original stopping criterion: Stop once performance deteriorates. Instead we prune until trivial performance is hit and then trace back to the latest point where accuracy is not deteriorated. Or in other words, even once deterioration occurs during pruning we keep going in case one of the further pruned models turns out to be non-deteriorated. Settings where we cannot find any sparsity using this approach for at least one prompt template are marked in bold.

		Task				
Model	Model Part Type	COPA	MRPC	PIQA	SST-2	Web Questions
GPT-2-XL	Attention Blocks	+31	–	+3	+19	**+2**
	Layers	+25	–	+2	**+12**	0
	Feedforward Blocks	+15	–	+5	**0**	0

setting (Pruning Layer for COPA) and is consistent what we find with the main experiment otherwise, i.e., the other bold entries are the same.

6.4 RQ 2: Identification of Sparsity

6.4.1 Overview

To answer if we can identify sparsity without additional data, we employ a similar approach to RQ 1. We prune Mistral-7B and GPT-2-XL on the five tasks for three different types of model parts until accuracy begins to deteriorate but instead of using *real task data* to estimate importance and prune the model we prune using *generated data* instead. The dataset we generate is of the same size as the calibration dataset consisting of real data, i.e., 256 samples. This makes the two settings comparable to each other as we only test the impact of replacing the data, while everything else is fixed. An identified sparsity of size greater zero identified in this manner, means that generated data can discover sparsity. Further, we can compare the identified sparsity against that identified using real data.

Like for RQ 1, we use three prompt templates to represent each task. The generated data used for pruning is always generated using the same prompt template as the one used

Table 6.5: Change in percentage of model parts that can be removed without deteriorating accuracy when pruning using generated data (instead of task data like for RQ 1) for both Mistral-7B and GPT-2-XL. The results are averaged across three prompt templates. We highlight values greater or equal to 0 in bold as they imply that on average more or equal amounts of sparse model parts are identified as compared to pruning with real data.

		Task				
Model	Model Part Type	COPA	MRPC	PIQA	SST-2	Web Questions
Mistral-7B	Attention Blocks	−9	**+7**	**+5**	**+1**	−4
	Layers	−2	**+4**	**0**	−4	−3
	Feedforward Blocks	−4	−1	**+1**	−1	−2
GPT-2-XL	Attention Blocks	−5	–	−1	**+15**	−9
	Layers	**+14**	–	**+1**	**+3**	**+1**
	Feedforward Blocks	−15	–	**+2**	**+10**	−3

to represent the task (and therefore evaluation). We fix the temperature for the generation process to 1.

6.4.2 Results

We present the results for RQ 2 in Table 6.5. The table shows the change in sparsity size when changing real data to generated data. The changes for each task are averaged across three prompt templates. We can see that the pruning results are similar, i.e., many differences are small and hover between only a 1-4% difference in achieved sparsity size. In fact, for Mistral-7B there is only one setting where sparsity is, on average across the prompt templates, worse by over 4% in size (for attention blocks on COPA). For GPT-2-XL there are three in total. It makes sense that this number is higher as GPT-2-XL is on average a weaker model and thus less able to generate quality data for itself. Moreover, we find that across both models there are a decent number of settings where pruning using generated data *outperforms* pruning using real data. This is especially true for Mistral-7B on MRPC and for GPT-2-XL on SST-2.

These insights indicate that we can also identify sparsity using generated data to a

similar extent as when additional task information in the form of task data is given.

6.4.3 Analysis

We conduct two further analyses.

Examples of Generated Data. We present a side-by-side comparison of real task data examples and those generated by the model itself for Mistral-7B in Table 6.6. The generated examples chosen are not cherry-picked (instead randomly chosen). We can see that for all tasks the examples are coherent with only minor grammatical errors such as in "how to stop chicken to stick to nugget maker" for PIQA. For Web Questions the generated question in the Table is correct however the answer is slightly incorrect: Melissa Joan Hart stars as the lead character in the Sabrina the Teenage Witch series however is not the execute producer, which is Paula Hart, Melissa's mother. For the example for SST-2 in the Table we can observe an interesting effect, in that part of the generated data recycles an example given in the prompt "if you give a filmmaker an unlimited amount of phony blood", but continues it with additional content.

Similarity to Sparsity Identified using Real Task Data. We want to compare the sparsity identified using the generated data with that obtained using the real task data for each setting. This allows us to identify if they are similar or if there are differences in the actual pruned parts, for instance, even if the size of sparsity is the same. To do so we compute the Jaccard similariy between the pruned model parts when using real and generated data on the same task and prompt template for each model part type. We present the results in Table 6.7. Again, the results are averaged across the three prompt templates[4] for each entry. We can see that for attention blocks on Mistral-7B, the similarity is high (except for MRPC) between the identified sparsities. To some extent this can be explained by the fact that on average sparsity in attention blocks is generally similar across all tasks and prompt templates. It appears that which attention blocks are irrelevant does not depend strongly on the task or prompt template. For the other settings there is moderate to little similarity. This is especially true across both models for Web Questions.

[4]In case we cannot identify sparsity using neither real data nor generated data for a prompt template, the Jaccard similarity is undefined. Then, we only average over the remaining two prompt templates.

Real Example	Generated Example
SST-2	
Sentence: and that leaves a hole in the center of the salton sea. **Sentiment**: Negative	**Sentence**: if you give a filmmaker an unlimited amount of phony blood, a castle, and the sense that all of britain has slipped into a three- **Sentiment**: Negative
MRPC	
Sentence 1: In fiction: Edward P. Jones ("The Known World") and Scott Spencer ("A Ship Made of Paper"). **Sentence 2**: The fifth nominee for fiction is Scott Spencer, for A Ship Made of Paper. **Label**: Not Paraphrases	**Sentence 1**: Thanx and Fiat praises its reliability, economy, and passenger space. **Sentence 2**: praised Thanx's reliability, says Kim Olson, 67, of East Patchogue, New York, who gets 33 **Label**: Not Paraphrases
PIQA	
Goal: Remove seeds from strawberries **Continuation**: Blend the strawberries, pour the mixture through a fine-mesh strainer with a bowl underneath to catch the pulps and strain out the seeds	**Goal**: How to stop chicken from stick to nugget maker **Continuation**: Spray oven door with cooking spray, as you use the first side of the Nugget Maker, and spray that side with cooking spray.
Web Questions	
Q: when was the last time the toronto maple leafs made the playoffs? **A**: 1949 Stanley Cup Finals	**Q**: who was executive producer of the sabrina series in the 90's? **A**: Melissa Joan Hart
COPA	
Premise: I twisted the cap off the soda bottle. **Effect**: The soda fizzed.	**Premise**: The girl climbed the mountain. **Cause**: the girl wanted to see the sunrise.

Table 6.6: Real and generated examples (by Mistral-7B using our approach) for the five tasks we evaluate on.

Table 6.7: Jaccard similarity between the maximum sparsity identified using real data and the one identified using generated data for Mistral-7B and GPT-2-XL. The results are averaged across three prompt templates.

		Task				
Model	Model Part Type	COPA	MRPC	PIQA	SST-2	Web Questions
Mistral-7B	Attention Blocks	0.75	0.29	0.68	0.89	0.81
	Layers	0.07	0.73	0.33	0.37	0.00
	Feedforward Blocks	0.42	0.38	0.28	0.52	0.00
GPT-2-XL	Attention Blocks	0.41	—	0.48	0.50	0.28
	Layers	0.20	—	0.26	0.38	0.17
	Feedforward Blocks	0.29	—	0.20	0.39	0.08

6.5 RQ 3: Repair of Pruned Models

6.5.1 Overview

To answer this question, we again employ a similar approach to RQ 1. We prune Mistral-7B and GPT-2-XL on five tasks for our three different types of model parts until accuracy begins to deteriorate. However instead of measuring accuracy deterioration on a pruned model directly, we first repair it using task data with feature scaling, i.e., after each pruning step we repair the pruned model and only then evaluate to see if we keep pruning.[5] We use *real task data* for importance estimation.

This experiment reveals how much more sparsity is possible if we repair after pruning. Specifically, if the model is sparse in more model parts after repair than without (the main results from RQ 1), we know that repair is able to restore accuracy on deteriorated models (as it does so for the model parts the model is now newly sparse in thanks to repair). Note that for this research question we prune only for one prompt template per task out of computational constraints for Mistral-7B (for GPT-2-XL we use all three and average).

We train the parameters introduced by feature scaling using Adam (Kingma and Ba,

[5]The repair is discarded after each evaluation and we continue to prune on the unrepaired version. Then the next model part is removed and the remaining submodel is repaired and evaluated and so on.

Table 6.8: Change in percentage of model parts that can be removed without deteriorating accuracy when pruning with repair using feature scaling versus when no repair is used.

		Task				
Model	Model Part Type	COPA	MRPC	PIQA	SST-2	Web Questions
Mistral-7B	Attention Blocks	+12	+6	+9	+9	+31
	Layers	+6	0	0	+12	+9
	Feedforward Blocks	+6	+6	+3	+6	+6
GPT-2-XL	Attention Blocks	+12	–	+15	+39	+49
	Layers	+17	–	+1	+51	+38
	Feedforward Blocks	+7	–	+3	+72	+41

2015) on the same dataset we use for importance estimation for one epoch with a learning rate of $3 \cdot 10^{-2}$ and 4 as the batch size.[6] The parameters in the scaling vectors are all initialized to 1, i.e., without further training the model is equivalent to the unrepaired one. The hyperparameters are chosen based on small preliminary experiments such that the training loss can be consistently minimized.

Note that similar to RQ 1, this shows a lower bound on how far sparsity can be extended by this kind of repair but not necessarily an upper bound as it could be that the optimization procedure we use was just not able to find a parameterization that successfully restores accuracy, even if it exists. We believe this possibility to be unlikely due to the empirical success of Adam (Kingma and Ba, 2015), however it remains possible that our choice of hyperparameters is not optimal (although also unlikely based on our preliminary experiments).

6.5.2 Results

We present the results for RQ 3 in Table 6.8. The table shows the change in size of sparsity before deterioration of relative accuracy beyond 5% occurs (averaged for GPT-2-XL). We find that repairing the model by relaxing the scaling of the features is indeed able to increase the size of sparsity possible across almost all tasks and model part types with the

[6]Except for MRPC, where we pick a batch size of 3 out of memory constraints.

sole exception of layers on COPA and MRPC for Mistral-7B. This shows that repair using only feature scaling is indeed capable of restoring accuracy on a deteriorated pruned model in most settings. For Mistral-7B we can see that additional sparsity ranges from around 6%-12% across all settings except when pruning attention blocks on Web Questions where a sizable increase of 31% is possible. For GPT-2-XL feature scaling is even more potent and can yield sparsity that is up to 72% larger than without repair.

6.5.3 Analysis

As a further analysis we repeat the above experiment but repair using LoRA.[7] As it approximates full fine-tuning (Hu et al., 2022), we expect it to perform better than feature scaling and serve as an upper bound. On the other hand, it is no longer easy to apply just-in-time as the number of parameters for LoRA as well as the operations necessary to apply have a more significant overhead (additional matrix multiplications are introduced unless the LoRA is merged into the model). The hyperparameters we use are identical except for the learning rate we choose to fix at $3 \cdot 10^{-5}$ based on preliminary testing. LoRA is applied to all the weight matrices used in the attention blocks as suggested by Hu et al. (2022). We choose $k = 8$ for the intermediate dimension.

We show our results in Table 6.9. The table shows the change in sparsity size repair brings. As we can see the results are similar to what is achieved just using feature scaling for Mistral-7B (differences range just from 3% to 6%), the only exception being MRPC where LoRA strongly overtakes feature scaling. For GPT-2-XL, LoRA performs worse than feature scaling across all settings. Overall these results are unexpected as LoRA generally has many more tunable parameters and should be able to adapt the model behaviour more consistently. More work is needed to understand why this is not the case. It might be for instance that our internal dimension size was chosen to be too small.

[7] Out of computational constraints we also limit ourselves to only one prompt template for GPT-2-XL here. It is always the same as the one used for Mistral-7B.

Table 6.9: Change in percentage of model parts that can be removed without deteriorating accuracy when pruning with repair using LoRA versus when no repair is used.

		Task				
Model	Model Part Type	COPA	MRPC	PIQA	SST-2	Web Questions
Mistral-7B	Attention Blocks	+9	+12	+9	+16	+19
	Layers	+3	+22	0	+9	+6
	Feedforward Blocks	+3	+19	+6	+3	+6
GPT-2-XL	Attention Blocks	0	–	+2	0	+44
	Layers	0	–	0	+17	+12
	Feedforward Blocks	+2	–	0	+54	+19

6.6 RQ4: Speedup Achieved

6.6.1 Overview

For this experiment we compare the speed of the smallest pruned model identified (so just before accuracy deterioration) using *real task data*, either with or without repair added. We compare the speedup these sparsities result in by measuring the average floating point operations (FLOPs) used by the pruned model on a randomly selected subset of test data of 16 instances for each task. This allows us to compare all these settings using a common metric and identify which performs best, i.e., has the highest FLOP reduction.

We run this experiment for each of the five tasks and three model part types on Mistral-7B as well as GPT-2-XL. For each we use the single prompt template also used for RQ 3. We count FLOPs with the Fvcore library (Wu, 2024).

6.6.2 Results

We provide the results in Table 6.10. In this table we report the relative FLOP reduction for each setting when compared to the base model on the same evaluation data. FS stands for repair with feature scaling and NR stands for no repair. For each task the configuration with the identified sparsity leading to the highest relative FLOP reduction over the corresponding base model on the task is highlighted in bold. We can see that across all

Table 6.10: Relative FLOP reduction in percentage of the final pruned submodel (just before accuracy deterioration) over the corresponding base model, i.e., Mistral-7B or GPT-2-XL. FS stands for the setting where we prune with repair (RQ 3) and NR stands for no additional repair (RQ 1). For each Task and model, we highlight which setting results in the biggest FLOP reduction across type of repair as well as pruned model part type. Like for RQ 2 we limit ourselves to a single prompt template per task. The setting with the highest relative FLOP reduction for each task is highlighted in bold.

		Task									
		COPA		MRPC		PIQA		SST-2		Web Questions	
Model	Model Part Type	FS	NR	FS	NR	FS	NR	FS	NR	FS	NR
Mistral-7B	Attention Blocks	**10**	8	5	4	**9**	7	7	5	10	4
	Layers	9	3	3	3	6	6	31	18	**12**	3
	Feedforward Blocks	10	5	**15**	10	7	5	**32**	27	7	2
GPT-2-XL	Attention Blocks	5	1	–	–	**17**	12	25	19	22	0
	Layers	**40**	24	–	–	10	10	42	6	**43**	2
	Feedforward Blocks	37	35	–	–	10	10	**46**	1	28	1

tasks when directly compared by FLOPs, the difference between the different model part types shrinks. While in absolute percentage of removable structures attention blocks are on average more sparse than the other two model part types, we can see that in terms of computations saved they are similar to the other two types. We can also compare among part types to prune and repair or no repair for a given task to determine which setting is optimal in terms of reduced compute without accuracy deterioration. In terms of model part type to prune this seems to depend on the task and we cannot generalize. However we can note that pruning using repair always yields a better overall FLOP reduction, even taking into account the overhead introduced from feature scaling. The optimal choice in terms of FLOP reduction for each task is highlighted in bold.

6.6.3 Analysis

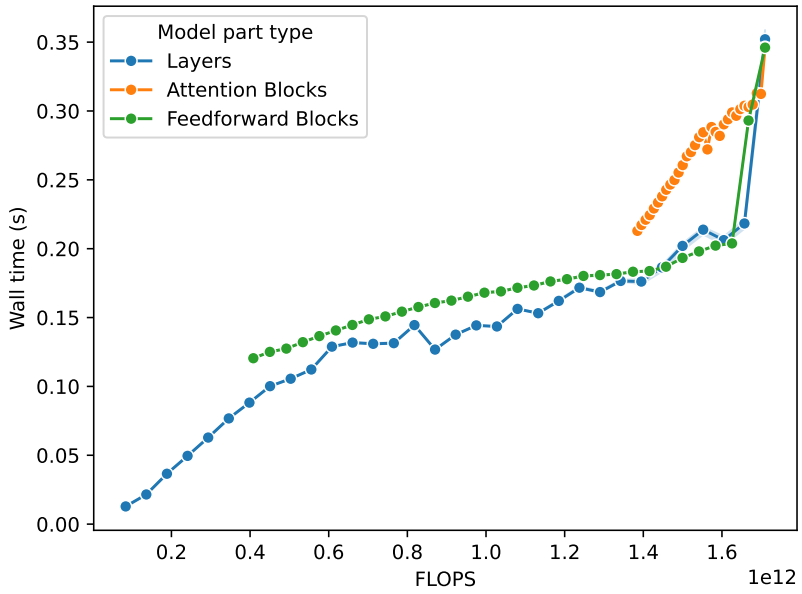
It might be that FLOP reduction by itself is not a good measure for actual speedup as we incur some overhead in terms of non-FLOP operations like additional logic in the code

to handle the pruning as we apply it in a just-in-time manner during each inference (we change the pruning variables as outlined in the Implementation Chapter). We conduct a small follow up experiment on real hardware to test if real clock speed up correlates well with FLOP reduction. We refrain from using it as a direct measure in the full experiment as FLOPS serve as a more consistent measure in general.

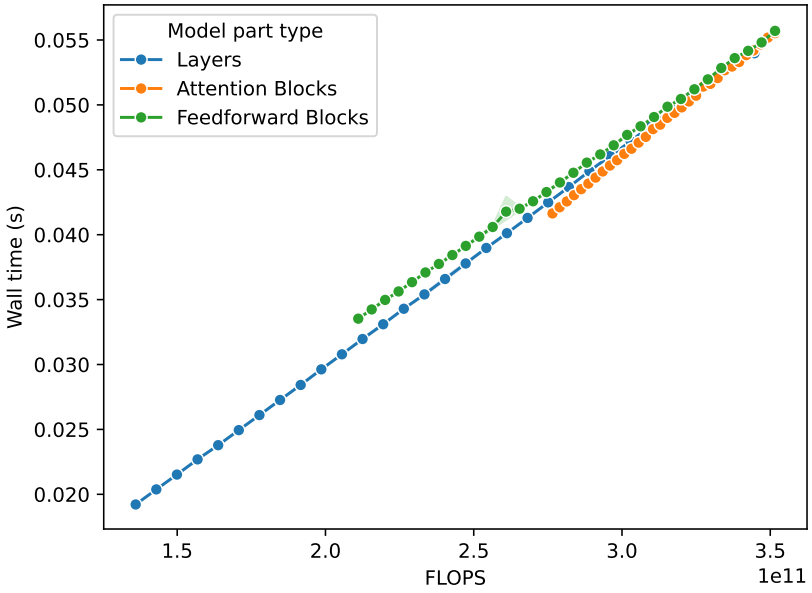
For each of the three model parts and both models, we record the average wall time for inference as well as FLOPs used on SST-2 with a fixed prompt template. We do not expect the specific task or prompt template to play a significant role. We do this for 32 submodels, each consisting of one model part less than the previous, i.e., we start with the full model and end with the empty model consisting just of embedding layer and language modelling head. We then compare FLOPs used against actual time (as measured by running the model on a Nvidia RTX A6000 GPU).

We present the results in Figure 6.5. We can see that wall time and FLOPs do in fact correspond to each other for all model part types in a direct linear relationship for GPT-2-XL and still in a mostly monotone way for Mistral-7B.[8] This confirms that our just-in-time implementation does not induce significant overhead and can be practically applied to achieve speedup. Additionally, we also see the stark difference in FLOPs saved per removed model part between the three types we study more visually.

[8]We hypothesize the non-linear relationship is caused the fact that we use a quantized version of the model and inference speed for the specific technique used depends on the number of ”large“ elements during matrix multiplication (Dettmers et al., 2022). As the values might change as we remove layers, so might the inference time to some degree.



(a) Mistral-7B



(b) GPT-2-XL

Figure 6.5: Comparison of FLOPs and real wall time as more and more model parts are pruned from Mistral-7B and GPT-2-XL for the three different types of model parts we investigate. For each type, the line consists of 32 individual data points, each with a different amount of model parts pruned, for which we measure both FLOPs and wall time.

Chapter 7

Discussion and Conclusion

In this work we investigate prompt template specific sparsity, i.e., irrelevant model parts, at the level of just-in-time removable model parts of large language models. Specifically, we study the pruning of attention blocks, feedfoward blocks and whole transformer layers. We choose these parts as they have residual connections we can exploit for efficient just-in-time pruning. For our experiments we aim to understand four different aspects of this sparsity: Existence, Identifiability without additional data, Repair using feature scaling and resulting Speedup. We answer these by pruning two models (Mistral-7B, GPT-2-XL) on five tasks (COPA, MRPC, PIQA, SST-2, Web Questions) each represented by three randomly chosen prompt templates. We find that:

1. **Sparsity at the level of just-in-time prunable model parts exists** Across almost all tasks and prompt templates there are model parts that can be removed with only small accuracy degradation. The amount of these parts we can remove this way range up to over 50% in some settings.

   We find that on average more attention blocks can be removed than layers or feed-forward blocks. This relates to some prior work (Bansal et al., 2023) on the pruning of attention heads (although not entire blocks), that shows that a higher percentage of them can be pruned when compared to feedforward blocks. Further, we find that model parts early or late in the model are on average more important than than ones near the middle, mirroring findings by Ma et al. (2023); Men et al. (2024). Finally, we also investigate with respect to the similarity and difference in sparsity across prompt

templates and tasks, something as yet unexplored in depth by previous work. We find that irrelevant model parts can be similar across tasks and prompt templates (such as for attention blocks on Mistral-7B) but also different enough in others so that pruning model parts irrelevant for one prompt template leads to deterioration of accuracy on another for the same task (such as for COPA on Mistral-7B).

2. **Sparsity can be identified with generated data** We find that generated data instead of real data is sufficient for our pruning approach to still work well for most settings. This means that it is identifiable using only model and prompt template itself (relevant for a practical setting where data is hard to come by).

   In some cases this even outperforms pruning using real data and can reveal sparsity of bigger sizes. This is similar to findings by West et al. (2022) that show that training a small model on data generated by a large one, can even outperform the original, larger model for specific tasks. In some sense, both their approach as well as ours, extract some meaningful aspect of an original general model, which allows us to improve upon it (either by creating a more accurate model for some tasks in their case or a smaller model with equivalent performance in ours).

3. **Repairing the pruned model with feature scaling boosts sparsity size** We find that scaling the output features is sufficient to restore performance of a deteriorated pruned model in almost all settings. In doing so we can prune further before accuracy deterioration occurs. In some cases we can even prune over 70% in additional model parts.

   Our initial hypothesis was that accuracy deterioration during pruning can be traced back to mostly problems in the interaction of remaining model parts. These findings verify this to some extent. As such it appears that modifying the intermediate representations passed between model parts is a powerful way to adapt model behaviour during pruning. This is backed by previous work in other research areas that works with these representations. For instance they can be used for parameter-efficient fine-tuning (Lian et al., 2022) and even for steering model behaviour with minimal intervention (adding a fixed steering vector suffices to elicit certain behaviours) (Subramani et al., 2022).

4. **In terms of speedup the type of model part removed is less important** We find that the amount of FLOPs saved when pruning either attention blocks,

feedforward blocks or entire layers does not differ strongly. The optimal choice in terms of FLOPs saved while still retaining accuracy varies per task. We also find that just-in-time pruning of these parts still results in speedup on actual hardware in terms of pure wall time.

These findings suggest three fruitful directions for future work:

1. The fact that pruning is possible at all and even more so when repairing, suggests that there are large amounts of redundancy in the models. Could we find a way to directly train smaller language models that are not redundant but still as performant as larger ones when trained on the same data?

2. Our results show that sparsity differs between prompt templates. But it is unclear which qualitative features contribute to this. What aspects of a prompt template makes it so a model requires some model part that another does not when addressing the same task?

3. Repair using feature scaling performs well, so does pruning using generated data. Additionally, prior work has shown that an approach similar to feature scaling can also be used to enhance model performance (not just prevent deterioration) (Lian et al., 2022). Instead of pruning the model, it might be interesting to purely inquire if an approach similar to ours can be used to improve performance for a prompt template. This could work for instance by training feature scales using generated data on the base model directly.

Bibliography

Hritik Bansal, Karthik Gopalakrishnan, Saket Dingliwal, Sravan Bodapati, Katrin Kirchhoff, and Dan Roth. Rethinking the role of scale for in-context learning: An interpretability-based case study at 66 billion scale. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 11833–11856, 2023.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, 2013.

Christopher M. Bishop. *Neural networks for pattern recognition.* Oxford University Press, USA, 1995.

Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the 34th AAAI conference on Artificial Intelligence*, pages 7432–7439, 2020.

Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. What is the state of neural network pruning? In *Proceedings of the 3rd Machine Learning and Systems (MLSys) Conference*, pages 129–146, 2020.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al.

Language models are few-shot learners. In *Proceedings of the 34th Conference on Neural Information Processing Systems*, pages 1877–1901, 2020.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, December 1989.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. In *Proceedings of the 36th Conference on Neural Information Processing Systems*, pages 30318–30332, 2022.

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

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*, 2019.

Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *Proceedings of the 40th International Conference on Machine Learning*, pages 10323–10337, 2023.

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

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, December 2023. URL `https://zenodo.org/records/10256836`.

Andrey Gromov, Kushal Tirumala, Hassan Shapourian, Paolo Glorioso, and Daniel A. Roberts. The unreasonable ineffectiveness of the deeper layers. *arXiv preprint arXiv:2403.17887*, 2024.

Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, 1949.

Boyu Hou, Chengyu Wang, Xiaoqing Chen, Minghui Qiu, Liang Feng, and Jun Huang. Prompt-distiller: Few-shot knowledge distillation for prompt-based language learners with dual contrastive learning. In *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5, 2023.

Edward J. Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *Proceedings of the 10th International Conference on Learning Representations (ICLR)*, 2022.

Yukun Huang, Yanda Chen, Zhou Yu, and Kathleen McKeown. In-context learning distillation: Transferring few-shot learning ability of pre-trained language models. *arXiv preprint arXiv:2212.10670*, 2022.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Eric R. Kandel, John D. Koester, Sarah H. Mack, and Steven A. Siegelbaum. *Principles of Neural Science*. Elsevier, 1991.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 26th Conference on Neural Information Processing Systems*, pages 1097—-1105, 2012.

Woosuk Kwon, Sehoon Kim, Michael W. Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. A fast post-training pruning framework for transformers. In *Proceedings of the 36th Conference on Neural Information Processing Systems*, pages 24101–24116, 2022.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2303.01911*, 2023.

Dongze Lian, Daquan Zhou, Jiashi Feng, and Xinchao Wang. Scaling & shifting your features: A new baseline for efficient model tuning. In *Proceedings of the 36th Conference on Neural Information Processing Systems*, pages 109–123, 2022.

Peter J. Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Lukasz Kaiser, and Noam Shazeer. Generating wikipedia by summarizing long sequences. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, 2018.

Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *In Proceedings of the International Conference on Computer Vision (ICCV)*, pages 2755–2763, 2017.

Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. Deja vu: Contextual sparsity for efficient LLMs at inference time. In *Proceedings of the 40th International Conference on Machine Learning*, pages 22137–22176, 2023.

Scott Lundberg and Marco Tulio Correia Ribeiro. Guidance-ai/guidance: A guidance language for controlling large language models. `https://github.com/guidance-ai/guidance`, 2023.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. LLM-pruner: On the structural pruning of large language models. In *Proceedings of the 37th Conference on Neural Information Processing Systems*, pages 21702–21720, 2023.

Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

Xin Men, Mingyu Xu, Qingyu Zhang, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. Shortgpt: Layers in large language models are more redundant than you expect. *arXiv preprint arXiv:2403.03853*, 2024.

Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? In *Proceedings of the 33rd Conference on Neural Information Processing Systems*, pages 14014–14024, 2019.

Marvin Minsky and Seymour Papert. *Perceptrons*. MIT Press, 1969.

Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

Seyed Sajad Mousavi, Michael Schukat, and Enda Howley. Deep reinforcement learning: An overview. In *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016*, pages 426–440, 2018.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning*, pages 807–814, 2010.

Meenal V. Narkhede, Prashant P. Bartakke, and Mukul S. Sutaone. A review on weight initialization strategies for neural networks. *Artificial Intelligence Review*, 55(1):291–322, 2022.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Proceedings of the 36th Conference on Neural Information Processing Systems*, pages 27730–27744, 2022.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd Conference on Neural Information Processing Systems*, pages 8026–8037, 2019.

David Peer, Sebastian Stabinger, Stefan Engl, and Antonio Rodríguez-Sánchez. Greedy-layer pruning: Speeding up transformer models for natural language processing. *Pattern Recognition Letters*, 157:76–82, 2022.

Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12:145 – 151, 1999.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Eric Roberts. Neural networks - history. `https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html`, 2024. Accessed: 2024-04-23.

N. Rochester, J. Holland, L. Haibt, and W. Duda. Tests on a cell assembly theory of the action of the brain, using a large digital computer. *IRE Transactions on Information Theory*, 2(3):80–93, 1956.

Melissa Roemmele, Cosmin Adrian Bejan, and Andrew S Gordon. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *2011 AAAI Spring Symposium Series*, 2011.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

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

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, 2013.

Nishant Subramani, Nivedita Suresh, and Matthew Peters. Extracting latent steering vectors from pretrained language models. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 566–581, 2022.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st Conference on Neural Information Processing System*, page 6000–6010, 2017.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. Grammar prompting for domain-specific language generation with large language models. In *Proceedings of the 37th Conference on Neural Information Processing Systems*, pages 65030–65055, 2023.

Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured pruning of large language models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pages 6151–6162, 2020.

Jakob Weissteiner, Chris Wendler, Sven Seuken, Ben Lubin, and Markus Püschel. Fourier analysis-based iterative combinatorial auctions. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 549–556, 2022.

Peter West, Chandra Bhagavatula, Jack Hessel, Jena Hwang, Liwei Jiang, Ronan Le Bras, Ximing Lu, Sean Welleck, and Yejin Choi. Symbolic knowledge distillation: from general language models to commonsense models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4602–4625, 2022.

Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *1960 IRE WESCON Convention Record, Part 4*, pages 96–104, 1960.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam

Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, 2020.

Yuxin Wu. Fvcore library. `https://github.com/facebookresearch/fvcore/`, 2024.

Jiacheng Ye, Jiahui Gao, Qintong Li, Hang Xu, Jiangtao Feng, Zhiyong Wu, Tao Yu, and Lingpeng Kong. ZeroGen: Efficient zero-shot learning via dataset generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11653–11669, 2022.

Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, et al. Instruction tuning for large language models: A survey. *arXiv preprint arXiv:2308.10792*, 2023a.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zhengyan Zhang, Zhiyuan Zeng, Yankai Lin, Chaojun Xiao, Xiaozhi Wang, Xu Han, Zhiyuan Liu, Ruobing Xie, Maosong Sun, and Jie Zhou. Emergent modularity in pre-trained transformers. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 4066–4083, 2023b.

Appendix A

Results of Base Models

For our research questions we evaluate two models across five tasks using three different prompt templates per task. In this Appendix we provide the results (in terms of accuracy) in these settings for the base models, i.e., unpruned models which we use to derive relative accuracy degradation together with the majority baseline. We show them in Table A.1. We show the results for the three prompt templates separately. They are specified by the random seed we use to derive them in our modified version of LM-Evaluation-Harness (Gao et al., 2023) (equivalent to the random prompt chosen for evaluation of the first instance of the test set in the unmodified version, however reused for all others).[1] The prompts with seed 42 are the ones used for RQ 3 and RQ 4.

Table A.1: Accuracy of the two base models we prune for the tasks we evaluate on. Each represented by three prompts generated using a random seed.

		Task				
Model	Prompt Seed	COPA	MRPC	PIQA	SST-2	Web Questions
Mistral-7B	42	0.90	0.74	0.82	0.94	0.33
	43	0.89	0.78	0.82	0.95	0.35
	44	0.93	0.69	0.82	0.94	0.36
GPT2-XL	42	0.72	0.68	0.72	0.71	0.06
	43	0.71	0.68	0.70	0.60	0.06
	44	0.74	0.68	0.71	0.78	0.08

[1] This information should also be sufficient to rederive them for replication.