Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Adaptive Layer Skipping during Large Language Model Inference

Master Thesis

2026
Gabriela Chutňáková, Bc.

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Adaptive Layer Skipping during Large Language Model Inference

Master Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Applied Informatics
Supervisor:	Mgr. Vladimír Boža, PhD.

Bratislava, 2026
Gabriela Chutňáková, Bc.

62054592

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Gabriela Chutňáková
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Adaptive layer skipping during large language model inference
Adaptívne preskakovanie vrstiev počas predikcie veľkých jazykových modelov

Anotácia: Predchádzajúce práce [1] ukázali, že je možné odstrániť celé vrstvy z už natrénovaného veľkého jazykového modelu bez zníženia presnosti. Cieľom tejto práce je vytvoriť metódu, ktorá umožňuje veľkým jazykovým modelom adaptívne preskočiť niektoré vrstvy počas výpočtu na základe aktuálneho vstupu a potenciálne skrátiť ča predkcie bez zníženia presnosti.

[1] Gromov, Andrey, et al. "The unreasonable ineffectiveness of the deeper layers." arXiv preprint arXiv:2403.17887 (2024).

Vedúci: Mgr. Vladimír Boža, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 01.11.2024

Dátum schválenia: 08.11.2024 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

62054592

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Gabriela Chutňáková
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Adaptive layer skipping during large language model inference

Annotation: Previous works [1] have shown that one can remove whole layers from an already trained large language model without sacrificing accuracy. The goal of this work is to create a method that allows LLM to adaptively skip some layers during computation based on the current input and potentially reduce inference time without sacrificing accuracy.

[1] Gromov, Andrey, et al. "The unreasonable ineffectiveness of the deeper layers." arXiv preprint arXiv:2403.17887 (2024).

Supervisor:	Mgr. Vladimír Boža, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. RNDr. Tatiana Jajcayová, PhD.
Assigned:	01.11.2024
Approved:	08.11.2024

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Declaration of Honour: I declare that I have written the entire diploma thesis, including all its appendices and figures, independently, using only the literature listed in the attached bibliography and with the assistance of artificial intelligence tools. I declare that the use of AI tools was in accordance with applicable legal regulations, academic rights and freedoms, ethical and moral principles, and that I have maintained academic integrity throughout the process. The following AI tools were used: ChatGPT, Claude, and NotebookLM — for text reformulation, grammar corrections, formatting of LaTeX tables and diagrams, assistance with coding, and source summarisation. The author is responsible for the correctness of the final form of the text.

Acknowledgments: I am grateful to all who supported me throughout this work. First, I would like to thank my supervisor Vladimír Boža for his valuable advice and willingness to help whenever needed. I am also thankful to everyone who proofread my work, offered ideas, or asked questions that helped along the way. Finally, I wish to thank my family and friends for their love and encouragement.

Abstrakt

Inferencia veľkých jazykových modelov si vyžaduje veľké množstvo výpočtov, pričom nie všetky musia byť potrebné. Predpokladáme, že rôzne tokeny potrebujú rôzny rozsah spracovania, a preto je možné preskočením niektorých dopredných sietí (FFN) znížiť výpočtovú náročnosť bez výraznej straty výkonnosti modelu.

V tejto práci skúmame možnosti adaptívneho preskakovania vrstiev počas predikcie na základe aktuálnej vektorovej reprezentácie tokenu. Sústredíme sa na dôležitosť vrstiev a na vzťah medzi veľkosťou ich výstupu a ich vplyvom na kvalitu modelu. Smerovanie na úrovni tokenov porovnávame so statickým odstraňovaním celých vrstiev.

Predstavujeme dva spôsoby smerovania: (1) preskakovanie vrstiev s malým výstupom na základe odhadu pomocou osobitne trénovaných klasifikátorov a (2) použitie smerovacích vrstiev trénovaných spoločne, s cieľom priblížiť výstupy upraveného modelu k pôvodnému. Smerovaním tokenov dokážeme vynechať tretinu výpočtov dopredných sietí, za cenu zvýšenia perplexity zo 7.3 na 15.3 na dátach WikiText-2.

Kľúčové slová: veľký jazykový model, efektivita inferencie, orezávanie modelov, preskakovanie vrstiev, adaptívne výpočty

Abstract

Large language model inference demands substantial computation, yet not all of it may be necessary. We assume that tokens vary in the amount of processing they need, therefore skipping some feed-forward network (FFN) layers may reduce computation without significantly degrading model performance.

In this work, we investigate the possibilities of adaptive layer skipping during inference, based on the current token representation. We focus on layer importance and the relationship between the magnitude of layer outputs and their impact on model quality. We compare token-level routing to static layer removal.

We propose two routing methods: (1) skipping layers with small estimated output based on separately trained classifiers, and (2) using gating layers trained jointly to approximate the outputs of the modified model to those of the original. Token routing allows us to omit one third of FFN computation, at the cost of an increase in perplexity from 7.3 to 15.3 on WikiText-2.

Keywords: large language model, inference efficiency, model pruning, layer skipping, adaptive computation

Contents

List of Figures

List of Tables

Introduction

In recent years, large language models (LLMs) have demonstrated remarkable capabilities across diverse tasks. Nevertheless, their increasing scale has introduced significant computational demands: larger models require more memory, longer inference times, and greater energy consumption, often calling for expensive hardware or cloud infrastructure. This not only increases their environmental footprint but also limits their accessibility, particularly for smaller research groups and organizations with constrained resources. The ability to run LLMs locally, faster and at lower cost, has therefore become a key motivation driving model compression research.

In this work, we investigate layer pruning in transformer-based LLM architectures. We focus specifically on feed-forward network (FFN) layers, as they account for three quarters of model parameters. Our goal is to identify redundancies in these layers and remove them to eliminate unnecessary computation.

Our main contribution is an adaptive pruning method that assigns a different computational budget to each token during inference, based on its current vector representation. Rather than permanently removing layers for all tokens, we allow individual tokens to follow different paths through the model. The method operates without modifying the LLM's weights — instead, lightweight routing mechanisms are trained to direct each token's trajectory.

We compare two such methods. The first is a magnitude-based routing approach that estimates the importance of each layer's contribution by examining its output magnitude. Routing decisions are then made based on these estimates, skipping computations that are likely to produce only minor changes in token representations. The routers are trained as separate predictors, each targeting a fixed skip ratio per layer.

The second approach employs learned gating layers, trained end-to-end with all other LLM parameters frozen. Each gate outputs a binary decision for each incoming token — whether it needs to be processed by that layer — using a continuous relaxation during training. Unlike the first method, this approach does not enforce a uniform skip ratio across layers, allowing layers of varying importance to naturally process different numbers of tokens.

As a secondary contribution, we analyse layer importance under various local output magnitude measures and compare these against each layer's actual impact on model

performance. We show that local measures often do not align with true global importance. We further analyse the routing decisions of well-performing models, revealing which layers are critical and which are redundant.

The thesis is structured into five chapters. The first chapter introduces the foundational concepts needed to understand our methods, covering LLM architecture and functioning, the principles of layer-skipping, and the aspects of model evaluation.

The second chapter surveys related work in the field of model compression. We focus primarily on pruning, describing existing methods for skipping transformer blocks or individual layers, and discussing the nuances that distinguish them. Particular attention is paid to adaptive routing approaches that process tokens selectively based on input complexity, as these are the most relevant to our approach.

The following chapters present the practical side of our work, starting with the third chapter, which covers an experimental study of model behaviour. We first introduce our setup — the language model and datasets used — then analyse layer importance both locally and globally, and examine the effect of different static pruning strategies. We also introduce token-level oracle skipping, which represents the target for routing decisions.

The fourth chapter proposes our two layer-skipping methods: the magnitude-based routers and the gated layers. We describe the principles behind each approach in detail, covering their architectural design and training procedure.

The final chapter presents the experimental results of our two methods. We evaluate each method individually and compare them against static layer removal methods. The chapter closes with an analysis of the routing decisions made in each approach, illustrating the apparent impact of individual layers.

Chapter 1

Preliminaries

This chapter establishes the theoretical foundation for the rest of this thesis, organized around three core concepts: large language models, neural network pruning, and LLM evaluation.

We begin with large language models, focusing on autoregressive text generation and the Transformer architecture, paying particular attention to how information flows through their layers. Next, we define pruning in neural networks and show how residual connections can be leveraged to remove components with minimal impact on model behaviour. Finally, we survey the evaluation metrics used to measure model quality, focusing especially on perplexity, which serves as the primary metric throughout this work.

1.1 Large Language Models

The term **large language model** (LLM) refers to a class of deep neural networks designed for natural language processing. They model language typically by learning to predict the next word in a sequence given the preceding context [1]. A defining characteristic of large language models is their scale: modern architectures can contain hundreds of billions to over a trillion parameters [2,3], particularly when using mixture of experts [4] designs. Greater scale enhances a model's ability to capture complex linguistic patterns and long-range dependencies.

In practice, input text is first decomposed into discrete subword units called **tokens**, over which the model defines its predictions. Given a sequence of tokens $x_1, x_2, \ldots, x_T$, the model estimates a probability distribution $P(x_{T+1} \mid x_1, \ldots, x_T)$ over a fixed vocabulary, and generates text autoregressively — sampling each token from the distribution conditioned on all preceding tokens, including the previously generated ones.

LLMs are developed through self-supervised pre-training on massive text corpora, requiring no human annotation — the training signal is derived directly from the

data itself. The pre-training objective is either next-token prediction or masked language modeling [5], where portions of the text are withheld and the model learns to reconstruct them. In both cases, training minimizes cross-entropy loss over token predictions, measuring the divergence between the model's output distribution and the ground truth label.

As a result, LLMs acquire extensive knowledge of both language and the world. This makes them capable of solving a variety of tasks, including text generation, summarization, question answering, and reasoning, often without any task-specific training [1, 6].

1.1.1 Encoders and Decoders

There are three common language model architectures: encoder-only, decoder-only, and encoder-decoder [1]. The **encoder** maps input token embeddings to contextualised vector representations that capture the meaning and structure of the input. Encoders are typically trained as masked language models: random tokens in the input are masked out, and the model learns to predict them using the surrounding tokens. This bidirectional context makes encoder models suitable for text comprehension tasks, such as sentiment or topic classification.

For text generation, a **decoder** is more suitable. It takes a sequence of tokens as input and generates the output sequence one token at a time. Masking in the decoder attention prevents the model from seeing subsequent tokens, ensuring that the prediction at position t depends only on tokens at positions less than t [1, 7] — making generation autoregressive, as defined above. In this work, we focus on a decoder-only model.

Encoder-decoder models, described in the original Transformer paper [7], combine these two components. First, the encoder processes the input token embeddings into contextualised representations. The decoder then uses these representations to generate the output sequence one token at a time, with each output token attending to all input tokens. Since encoder-decoder models can map between different types of token sequences, they are well suited for tasks such as machine translation, where input and output tokens belong to different languages, or speech recognition, where the input represents audio and the output represents text [1].

1.1.2 Transformer Architecture

Modern LLMs are based on the Transformer architecture [7]. They consist of an **embedding layer**, which maps each input token to a d-dimensional vector representation. These vectors are then passed through multiple **transformer blocks**. The final layer — the **language model head** — produces logits representing the probability distribution over possible next tokens.

Each block consists of a **multi-head attention** mechanism (MHA) and a fully connected **feed-forward network** (FFN), each with their own trainable parameters. The FFN processes each token representation independently, while the attention mechanism captures dependencies between tokens across the sequence. In this work, only the FFN layers are considered for pruning. The MHA modules remain unchanged across all experiments and are treated as fixed components of the architecture.

One way to think about the processing is through the notion of a **residual stream**: a single stream of d-dimensional representations for each token, starting with the original input vector [1]. The components within each block read from this stream and add their output back into it. In earlier blocks, the residual stream represents the current token. Later, it gradually shifts toward the following token, reflecting the model's next-token prediction objective.

This is implemented via **residual connections**, where for an input x, the representation after each layer is computed as $x + \text{Layer}(\text{LayerNorm}(x))$. **Layer normalization** stabilizes training by keeping activations in a range suitable for gradient-based optimization. In the formula above, it is applied before the layer, corresponding to the prenorm variant. Most modern architectures prefer prenorm over postnorm, where normalization follows the layer instead.

1.2 Intuition behind Pruning of Residual Networks

Within residual networks, including transformers, individual blocks typically induce only slight changes to the hidden representation, meaning that token representations in layer $\ell + 1$ are similar to those in layer ℓ [8].

Gromov et al. [9] explain that the representation at layer $\ell + 1$ can be expressed by the following residual equation

$$x_{\ell+1} = x_\ell + f(x_\ell, \theta_\ell),$$

where x_ℓ is the input at layer ℓ and θ_ℓ are its parameters. Each layer thus adds its transformation to the previous representation. Expanding this recursion, the output after L layers is

$$x_L = x_0 + \sum_{\ell=0}^{L-1} f(x_\ell, \theta_\ell).$$

Removing layer ℓ means the subsequent layer receives $x_{\ell-1}$ instead of x_ℓ, modifying the equation to

$$x_{\ell+1} = x_{\ell-1} + f(x_{\ell-1}, \theta_\ell).$$

If the transformation $f(x_\ell, \theta_\ell)$ is small, this substitution introduces only a minor change to the subsequent representations, and the layer can be removed with little impact on the network output.

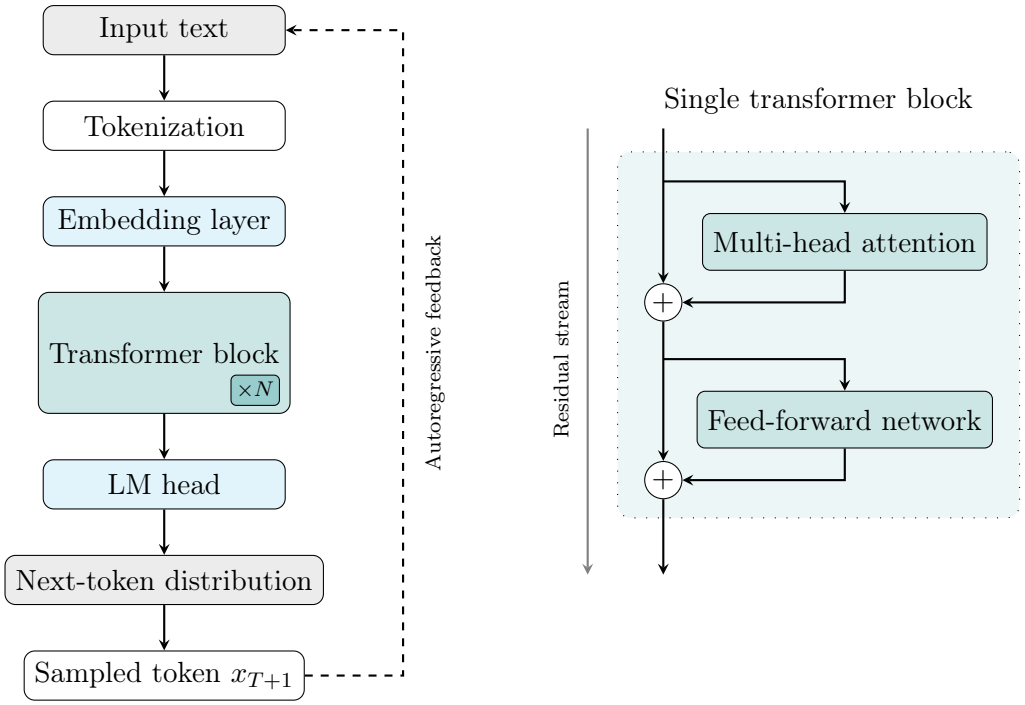

Figure 1.1: Architecture of a decoder-only LLM. The left shows the full forward pass through the embedding layer, N stacked transformer blocks, and the language model head, which projects the final representations to a probability distribution over the vocabulary. The right zooms into a single transformer block: each sublayer can be either preceded or followed by layer normalization, and its output is added back into the residual stream.

Layer removal has a cascading effect, as the change propagates through all subsequent layers. Consequently, pruning an earlier layer is expected to have a greater impact on the network output than pruning a later one [9].

1.3 Evaluation of LLMs

There are two ways of evaluating LLMs. The more complex approach evaluates how the model performs within an application, measuring the output quality of the entire system. This end-to-end evaluation is called **extrinsic**. It typically involves measuring performance on downstream tasks using standardized benchmarks — for example, having models answer multiple-choice questions and measuring their accuracy against the ground truth.

Extrinsic evaluation is costly in both time and computation, which makes it impractical in settings where the model must be evaluated repeatedly — such as during the development of a pruning method. A more suitable alternative is **intrinsic evaluation**, which measures model performance independently of any specific application.

A standard intrinsic metric for quickly evaluating a language model is **perplexity**. It measures how well the model predicts the test data by examining the probabilities it assigns to each token. A good model assigns high probabilities to the tokens that actually appear in the text. Perplexity normalizes these probabilities by the number of tokens, making it comparable across texts of different lengths. This is the metric we use to evaluate the pruned models in this work.

Besides output quality, other factors also matter when evaluating language models. Training and inference duration, as well as model size, play an important role, as time and memory are typically limited resources. Larger models also have higher energy demands, raising concerns in both environmental and financial terms. All these aspects should be considered when comparing models [1].

The following subsections discuss downstream tasks as an extrinsic evaluation approach, and perplexity as an intrinsic one.

1.3.1 Downstream tasks

Evaluating the quality of an LLM requires considering many aspects of model behaviour across a wide range of downstream tasks. Chang et al. [10] provide a comprehensive overview of these, which we briefly summarize here.

The main focus is on **natural language processing** abilities, covering both understanding and text generation. Typical understanding tasks include sentiment analysis, text classification, and natural language inference. Reasoning tasks go further, requiring the model to not only comprehend the provided information but also draw conclusions when explicit answers are absent. Language generation is evaluated through tasks such as text summarization, dialogue, translation, and question answering.

Beyond language abilities, evaluation also considers **factuality** — the consistency of model outputs with real-world facts and the model's tendency to avoid hallucination. **Fairness** is another concern, as models may inherit biases and stereotypes from training data, leading to toxic or ethically problematic outputs [1]. Finally, **robustness** measures model stability when facing challenging inputs, such as adversarial attacks or shifts in data distribution [10].

Another important aspect is **multilinguality**. Although LLMs are often trained on mixed-language corpora, English remains the dominant language for both training and evaluation. Assessing understanding and generation across different languages provides valuable insight into model quality, particularly for low-resource languages that are often underrepresented in training data.

Each of these aspects has dedicated datasets and benchmarks, such as multiple choice question sets, which allow for standardized comparison across different language models [1].

Model evaluation can be performed automatically or by humans. Automatic evaluation saves time and reduces the impact of subjective factors [10], typically by comparing model outputs to reference answers using metrics that measure overlap or matching between texts. In some cases, another LLM can serve as an automated judge.

1.3.2 Perplexity

Perplexity is a metric grounded in information theory, formally defined as the exponential of the cross-entropy H of the model on a word sequence W:

$$\text{Perplexity}(W) = e^{H(W)}$$

Cross-entropy $H(W)$ measures the divergence between the true probability distribution of a language and the distribution learned by the model. The more accurate the model, the lower the cross-entropy. For a sequence of N words, it is estimated as the negative log probability of the sequence, normalized by the number of words:

$$H(W) = -\frac{1}{N} \log P(w_1, w_2, \ldots, w_N)$$

The base of the logarithm can be either e or 2, with the base of the exponent in the perplexity formula matching accordingly.

Perplexity can also be interpreted as a branching factor — the average number of tokens the model considers likely to follow at each position. Lower perplexity therefore indicates a better model, one that is more confident and accurate in predicting upcoming tokens.

Improvement in perplexity does not automatically guarantee improvement on downstream tasks, although the two tend to correlate [1]. Models with lower perplexity generally outperform those with higher perplexity, underscoring its value as a proxy for overall model performance [11].

This correlation is further supported by the GPT-2 paper [12], which uses perplexity as the primary evaluation metric. Its results show that as model capacity increases, perplexity decreases and performance on downstream tasks consistently improves, suggesting that lower perplexity reflects genuine improvement in language understanding. Nevertheless, perplexity alone is not always a sufficient indicator of model quality, and results should ideally be confirmed by end-to-end downstream evaluation [1].

Chapter 2

Advances in Model Compression

The demands of large-scale deployment have motivated growing interest in model compression: reducing model size while preserving as much of the original performance as possible. The resources saved by using a smaller model can be reinvested — into longer training or greater capacity — within the same budget.

Compact models are especially suitable for on-device deployment [13], eliminating dependence on external infrastructure and making them valuable in privacy-sensitive applications. They also bring practical advantages in terms of reduced hardware requirements and lower inference costs, broadening their applicability across a wide range of deployment scenarios.

This chapter surveys recently proposed strategies in model compression, with a particular focus on pruning and input-dependent adaptive routing, as these form the core of our work. We review a selection of relevant works, examining the approaches they employ and the results they achieve.

2.1 Model Compression Methods

The main approaches to model compression include knowledge distillation, low-rank decomposition, quantization, network pruning [9,14], and conditional computation [15].

In **knowledge distillation**, a compact student model is trained to replicate the behaviour of a larger teacher, capturing the relevant information with fewer parameters and requiring fewer resources for inference. The student either imitates the logits and hidden states of the teacher, known as a white-box approach, or only has access to the output tokens generated by the teacher, known as a black-box approach. A drawback of distillation is that it remains computationally expensive, as the teacher must process large amounts of data during the student's training.

Low-rank decomposition replaces the large weight matrices of a neural network with lower-rank approximations which store the essential information in fewer dimen-

sions and thereby reduce the number of parameters.

Parameter quantization reduces storage and computational requirements by lowering the numerical precision of model parameters. High-precision floating-point numbers (e.g. 16-bit) are converted to lower-precision integers (e.g. 8-bit), while maintaining prediction accuracy at an acceptable level.

Unlike the previous methods, **pruning** directly identifies and removes unnecessary components, either individual weights or structural parts such as attention heads, FFN layers, or entire transformer blocks. The resulting model is smaller and less computationally demanding, and may even outperform the baseline, as reducing model complexity can mitigate overfitting and improve generalization [14].

Conditional computation activates different parts of a model depending on the input, resulting in greater capacity for capturing knowledge without a proportional increase in computational costs [4, 15]. The computation pathways are dynamically adjusted through adaptive routing based on the complexity of the input. Unlike static compression methods, which irreversibly shrink the model, adaptive routing retains the model's full capacity while introducing dynamic efficiency by skipping computations for certain inputs. This is particularly useful for large language models, where different inputs may require different levels of computation to achieve optimal results.

Model compression affects various aspects of a network, and evaluating a compression algorithm therefore requires considering multiple metrics: compression rate, model sparsity, computational complexity measured in floating point operations (FLOPs), training and inference time, and crucially, the accuracy of the compressed model.

2.2 Fundamentals of Network Pruning

Research in this area addresses several key aspects of the pruning pipeline: the granularity, the criteria for selecting which components to remove, the timing relative to training, and whether to prune all at once or iteratively [14].

2.2.1 Pruning Granularity

Pruning methods differ in the granularity at which they remove parameters, which has implications for both the achievable compression ratio and practical inference speedup.

Unstructured pruning removes individual parameters without altering the overall model structure, typically by setting those with small magnitudes to zero. While this fine-grained approach can significantly reduce the number of non-zero parameters, the resulting irregular sparsity patterns make it difficult to translate into real inference speedups on current hardware, except for limited low-batch inference cases [16].

Structured pruning targets entire groups of parameters — such as channels and filters in convolutional networks, or whole layers — and removes them as a unit. This results in a more regular architecture that can be accelerated using standard GPU hardware.

Applied to transformers, these methods target a variety of components: multi-head attention layers or individual attention heads, feed-forward network layers, entire transformer blocks, or the hidden dimension [9, 14].

2.2.2 Criteria for Pruning

When deciding which blocks to prune, the natural goal is to remove those that contribute least to the model's output. We can either evaluate the impact on final performance directly, which we refer to as global importance, or estimate it using local information at lower computational cost.

Determining a block's **local importance** typically involves measuring the similarity between its input and output hidden states — blocks whose output closely resembles their input contribute little to the overall transformation. In contrast, evaluating the perplexity increase when a given block is skipped provides a more direct measure of its global importance [11]. ShortGPT [17] showed a positive correlation between these two measures, confirming that local importance can serve as a computationally cheap proxy for global importance.

Several local importance measures have been proposed, differing in how they quantify a block's contribution using its hidden states alone. For both measures below, the reported value is averaged over tokens and inputs.

The **block influence** (BI score), introduced by ShortGPT [17], is defined for block i as

$$\mathrm{BI}_i = 1 - \frac{x_i^\top x_{i+1}}{\|x_i\|\|x_{i+1}\|},$$

where x_i denotes the hidden state entering block i and $\|\cdot\|$ is the L_2 norm. A lower BI score indicates high cosine similarity between the inputs to consecutive blocks, suggesting that the block causes minimal transformation and is therefore less important. Blocks with the lowest BI scores are pruned.

Local importance can also be measured through **relative magnitude** (RM) [8], defined as

$$\mathrm{RM}_i = \frac{\|f_i(x_i)\|}{\|x_i + f_i(x_i)\|},$$

where f_i denotes the transformation applied by block i. It captures how large the block's output is relative to the residual stream, with a smaller value indicating a less influential block.

A local importance approach is also employed in FFN-SkipLLM [18], though the importance score is not computed for every block individually. Instead, it tracks the cumulative magnitude of changes caused by subsequent blocks, and once a set threshold is reached, the remaining blocks are assumed to be unimportant based on the expectation that block influence decreases monotonically.

While these strategies achieve perplexity comparable to the original model at lower pruning ratios, removing more layers results in a sharp performance decline [11]. This suggests that local metrics alone may not fully capture a block's importance.

BlockPruner [11] takes a different approach by directly measuring how much each block contributes to the model's output, defining the **global importance** of block i as

$$\mathrm{GI}_i = \mathrm{PPL}(\mathcal{M} \setminus \{i\}),$$

where $\mathcal{M} \setminus \{i\}$ denotes the model with block i removed. Rather than evaluating all blocks independently in a single pass, the process is applied iteratively — after each removal, the GI values of the remaining blocks are recomputed, capturing how the role of a block may change as others are pruned.

2.2.3 Pruning Schedule

The traditional pruning pipeline consists of three stages: pre-training a dense model, pruning it to obtain a sparse model, and fine-tuning the pruned model to recover accuracy [14]. However, more recent work has introduced pruning at initialization or during training, motivated by the lottery ticket hypothesis [19]. It states that a dense, randomly-initialized network contains subnetworks — so-called winning tickets — that can be trained in isolation to reach comparable test accuracy.

Pruning before or during training has the advantage of accelerating not only inference but also training itself, avoiding the need to train a large dense network and subsequently fine-tune the pruned model [14].

Each of these approaches can be further varied. For example, pruning at initialization may or may not require training data. Pruning during training can either start with a randomly initialized sparse network and dynamically prune and redistribute weights, or start with a dense network and gradually prune it to obtain a sparse model.

Another aspect to consider is whether to prune iteratively or all at once. The iterative approach makes new pruning decisions after each removal, while one-shot pruning simply drops the k blocks with the lowest importance scores. BlockPruner [11] compared both strategies, with the iterative approach achieving better results, suggesting that it better captures codependencies between blocks. This makes iterative pruning an important component of the pruning pipeline and a potential avenue for improving other methods.

2.3 Review of Layerwise Pruning Methodologies

The previous section outlined the key dimensions along which pruning algorithms can vary. In this section, we survey a selection of recent layerwise pruning methods, examining their underlying mechanisms, the strategies they employ, and the results and insights they provide. The methods are grouped by their primary approach to pruning.

2.3.1 Pruning of Sequences of Transformer Blocks

Gromov et al. [9] propose a layer-pruning strategy that removes a contiguous sequence of n transformer blocks, treating the block as the pruning unit rather than the individual FFN or attention layers. The pruning decision is based on the condition that the output of the removed block should be similar to its input:

$$x_i \approx x_{i+n} + \epsilon.$$

As a measure of block importance they use angular distance, defined for a single sequence of length T as

$$d\left(x_i, x_{i+n}\right) = \frac{1}{\pi} \arccos\left(\frac{x_i^{(T)} \cdot x_{i+n}^{(T)}}{\left\| x_i^{(T)} \right\| \left\| x_{i+n}^{(T)} \right\|} \right),$$

where the inner product is taken over the hidden dimension of the model for the final token T of the sequence, $\| \cdot \|$ denotes the L_2 norm, and the factor $1/\pi$ normalizes the result to the range $[0, 1]$ by convention. The authors focus on the final token since its embedding is the only one that depends on the entire sequence, due to the causal attention mask.

Angular distance is closely related to cosine similarity — while cosine similarity measures the cosine of the angle between two vectors, angular distance converts this to the actual angle.

Using a pre-training dataset, they compute the angular distance between the inputs to layers i and $i + n$ for each i, identifying $i^\star$ as the starting layer that minimizes this distance. The contiguous block of n layers starting at $i^\star$ is then pruned. A small amount of fine-tuning is applied to compensate for the induced mismatch, which has a significant impact on the resulting perplexity.

With this approach, up to 20–55 % of layers can be pruned without any significant decrease in performance, depending on the model family and size. Beyond this threshold the model's performance drops sharply, with predictions collapsing to random guessing. Notably, layer pruning appears more robust for larger and deeper models. This could be related to the fact that smaller models are more overtrained, making their parameters less redundant, or simply that deeper models can afford to lose more layers in an absolute sense.

2.3.2 Consecutive FFN Layers Pruning

Rather than pruning entire transformer blocks, FFN-SkipLLM [18] focuses specifically on the FFN layers, which account for approximately two-thirds of the total parameter count in the architectures studied. The method tracks the cosine similarity between the input and output of each FFN layer. Once a certain threshold is reached, the method greedily decides to skip the FFN layers in the following k blocks.

The cosine similarity, referred to as FFN saturation, typically increases monotonically in the middle layers, signalling redundant computations. The earliest and deepest layers, where the impact of the FFN is higher, are called cold regions. Unlike the middle layers, FFN layers in cold regions are not redundant, and removing them significantly decreases performance.

The first few tokens play a crucial role in understanding the input context, a phenomenon known as attention sink. To account for this, the initial tokens are processed fully through all layers, while for the remaining tokens, skipping is applied once the mean similarity across tokens reaches a certain threshold. This distinct treatment is possible because, unlike attention, the FFN layer processes each token independently.

With this approach, 25–30 % of FFN layers can be skipped with marginal impact on performance across knowledge-intensive tasks.

2.3.3 Individual MHA and FFN Layers Pruning

BlockPruner [11] offers further flexibility by considering both MHA and FFN layers — referred to as blocks in this method — as pruning candidates, without requiring them to be consecutive. The decision of which layers to remove is based on their global importance, measured using the model's perplexity.

The pruning is performed iteratively. For each layer, a modified model is created by masking it out, and the block importance is computed as the resulting perplexity. The method then prunes the layer with the lowest importance score and repeats the process until the desired number of layers is removed, greedily minimizing performance degradation on the calibration dataset.

BlockPruner outperforms several competing methods across most benchmarks, even at higher pruning ratios. These include block influence [17] and relative magnitude [8], discussed earlier, as well as SliceGPT [20] and LaCo [21]. SliceGPT compresses the model by removing rows or columns corresponding to smaller principal components in weight matrices, while LaCo reduces model size by merging layers, maintaining the overall model structure.

As with the previous methods, performance holds up better on larger models, further supporting the observation that bigger models tend to have more redundancy and can therefore tolerate more aggressive pruning.

The authors also analyse the impact of two key design choices: the iterative pruning strategy and the fine-grained layer pruning. As an ablation, the iterative approach is compared to one-shot pruning, where all importance scores are computed by masking a single layer at a time and the lowest-scoring layers are removed directly. Additionally, pruning individual MHA and FFN layers is compared against pruning entire transformer blocks. Both modifications result in performance degradation. This suggests that the iterative search better captures interactions between layers, and that model redundancies are fine-grained, making individual layer pruning more effective.

The authors also compare pruning only MHA layers, only FFN layers, and both together. Pruning MHA layers alone initially causes smaller performance degradation, matching the outcomes of mixed pruning. However, once a pruning ratio of just 17 % is surpassed, model quality drops steeply. This suggests that while MHA layers tend to be more redundant than FFN layers, the majority of them remain critical.

2.4 Adaptive Routing Strategies

Traditional language models assign a fixed computational budget to every token, regardless of the input's complexity or the specific task — an inherent inefficiency that adaptive routing aims to address [22]. Instead, it selectively activates the necessary parts of the model, reducing computational cost and memory usage per query.

Routing decisions can be based on the estimated difficulty of the input, the available computation budget, or learned criteria [15]. We survey these approaches in two groups: methods that route tokens among experts of varying capacity, and methods that skip layers or computations entirely.

2.4.1 Expert-Based Routing

In expert-based routing, the model maintains multiple specialized components, called experts, and dynamically selects which ones to activate for each token. Unlike straightforward pruning, this first expands the model and then reduces computation dynamically — directing each token through the most suitable expert, allowing the model to scale its capacity while keeping computational costs low.

Mixture of Experts

The Mixture-of-Experts (MoE) layer [4] is one of the most established implementations of conditional computation. It consists of up to thousands of feed-forward sub-networks and a trainable gating network that routes each token to a sparse subset of them. Individual experts tend to specialize based on syntax and semantics, each processing

tokens with similar contextual roles or semantic meanings. This specialization greatly increases the model's ability to capture diverse knowledge.

The gating network may converge to selecting the same few experts for the majority of tokens, leading to a self-reinforcing imbalance where favoured experts are trained more and thus selected even more often. To counteract this, an additional loss term is introduced to encourage a more uniform distribution of tokens across experts.

MoE is commonly applied to transformer-based language models by replacing the FFN layer in each transformer block. Large language models that utilize MoE are referred to as MoE-LLMs.

MoE with Adaptive Number of Experts

In traditional MoE, every token is processed by a fixed number of experts. However, not all tokens require the same level of processing for sufficient feature abstraction. AdaMoE [23] addresses this by introducing null experts — lightweight placeholders that require no computation — allowing the model to adaptively select fewer real experts for simpler tokens. A load-balancing loss encourages this adaptive behaviour, resulting in a variable number of real experts per token.

AdaMoE can be applied to both pre-trained LLMs and MoE-LLMs by adding experts and augmenting the router with corresponding weights, outperforming both in terms of accuracy.

Routing to FFN Modules of Different Sizes

Building on the MoE principle, Duo-LLM [22] extends this idea by introducing two interchangeable FFN modules per layer — a smaller and a larger one — alongside the option to bypass the FFN entirely. Each token is routed to one of these three paths based on its complexity, with the attention layer shared across all of them.

During training, tokens are routed randomly to ensure both modules are trained equally. Unlike in MoE models, the router is not trained alongside the modules, as doing so would introduce bias: the routing strategy would influence the training of the modules it is supposed to evaluate.

To determine the optimal routing strategy, an oracle evaluates the perplexity of all feasible routing paths for a given computational budget — for example, only considering routes that use the large module 30% of the time. Since exhaustive search is far too expensive in practice, the oracle's routing is approximated by a learned routing strategy.

The total loss consists of two parts: a cross-entropy loss that compares the predicted routing decisions to those of the oracle, and a budget loss that regulates computational resources across all layers for each sequence. Using a global budget loss rather than

a per-layer load balancing loss allows the model to allocate more compute to certain layers and discover optimal routing patterns.

The results show that with optimal routing, a small additional FFN can significantly improve model performance. Interestingly, the oracle achieves lower perplexity by selectively using the large module than by using it for all tokens. However, the learned routing strategy falls short of the oracle, achieving perplexity comparable to selecting the best out of 100 random paths.

2.4.2 Layer and Computation Skipping

Rather than routing tokens among different experts, a complementary approach is to skip computations entirely. Depending on the input, certain layers or operations can be bypassed, reducing the computational cost for tokens that do not require full processing.

Mixture of Depths

Mixture-of-Depths (MoD) [24] builds on the intuition that not all tokens require the same amount of processing, and introduces a method of dynamically allocating compute across tokens. As a result, individual tokens may pass through a different number of layers. The method draws from MoE logic, with the key difference that instead of routing tokens among multiple expert FFNs, each MoD layer has a single expert — a block consisting of self-attention and the subsequent FFN — which can be skipped entirely.

For each block, a number k is chosen to limit how many tokens it processes. A per-block router assigns a weight to each token, reflecting its preference to be processed by the block, and the k tokens with the highest weights are selected to pass through. Fixing the number of tokens per block to k yields a useful property: the computational graph remains static, as the number of tokens processed by each block is always the same — only the identity of the chosen tokens varies. This allows the model to be compiled and optimized by hardware accelerators more efficiently.

From a token's perspective, at each block it faces two options: pass through, participating in attention and FFN computation, or bypass the block via the residual connection at no computational cost. The token routing decision is determined by the router weight r_i, computed as a linear projection $r_i = w_\theta^T x_i$, where w_θ are the learned projection weights and x_i is the hidden state of the token at block i.

The output of block i for the token is

$$x_{i+1} = \begin{cases} r_i f_i(\tilde{X}_i) + x_i, & \text{if } r_i > P_\beta(R_i), \\ x_i, & \text{otherwise.} \end{cases}$$

Here, $P_\beta(R_i)$ is the β-th percentile of router weights across all tokens, with $\beta = 1 - \frac{k}{S}$, where S is the sequence length. This ensures that k tokens participate in each block's computation. The function $f_i(\tilde{X}_i)$ represents the result of self-attention and FFN using the top-k tokens $\tilde{X}_i$.

When a token passes through the block, the output is scaled by its router weight, as seen in the first case of the equation. This ensures the router weight is included in the gradient path, allowing the router to be optimized during training.

A limitation of the top-k routing mechanism is its non-causal nature — to determine whether a token belongs among the top-k, the router needs access to future tokens, which are unavailable during autoregressive inference.

To address this, the authors propose two solutions. The first introduces an auxiliary loss that encourages the router to produce outputs above 0.5 for tokens selected among the top-k and below 0.5 for the rest, using binary cross-entropy with the top-k selections as targets. The second introduces an auxiliary MLP predictor that receives the same inputs as the router and learns to predict whether a token will be among the top-k. This turns out to be a relatively easy auxiliary task that quickly achieves 99 % accuracy.

The results were compared against a vanilla transformer and a control routing scheme that randomly selects which blocks to skip. While the control scheme significantly underperforms the vanilla transformer, the best MoD variants outperformed it, suggesting that a standard transformer uses its compute budget inefficiently. For the same computational budget, performance can be improved by employing MoD and redirecting the saved compute toward a larger model or longer training.

Conditional Heavy and Light Processing

While MoD routes tokens around entire transformer blocks, CoLT5 [25] makes routing decisions at the level of individual layers. Each layer contains both a lightweight and a heavyweight variant of the FFN and attention modules, with tokens routed to one or the other based on their importance, reserving expensive operations for key tokens while filler words and punctuation are handled by the lighter variants.

All tokens are first processed by the lightweight modules at each layer. The routing modules then select the most important tokens for further processing by the heavyweight variants. Each CoLT5 layer contains three independent routers: for the FFN, attention queries, and attention key-values.

Routing analysis reveals that in the last layer, tokens belonging to the question or placed close to the correct answer are most likely to be selected as important. In the early layers, these tokens are selected only moderately — which makes intuitive sense, as the model has not yet developed a clear picture of which tokens are relevant.

Adaptive Layer Skipping

In FlexiDepth [26], the authors study how computational demands vary across the generation of different tokens. They propose plug-in components — routers and adapters — that can be employed in pre-trained LLMs to dynamically bypass unnecessary layers without modifying other model parameters.

The router makes binary decisions about whether to skip each layer, while the adapter resolves representation misalignments caused by skipping. Each token therefore has two options: to be processed by the full transformer block, or to bypass both the attention and FFN layers and pass through the lightweight adapter instead. The outputs of both paths are then combined to produce the layer output.

To optimize the router and adapter, the authors introduce an additional loss term that penalizes the number of layers used during generation. Combined with the standard next-token prediction loss, this encourages layer skipping while preserving generation quality. After training, the model dynamically adjusts its layer usage based on the input.

Skipping the attention module poses a challenge: in autoregressive generation, each token contributes key and value vectors to the cache, which subsequent tokens rely on for context. A skipped token never produces these vectors, effectively making it invisible to future tokens. The authors resolve this by still computing the keys and values for skipped tokens, while omitting the query computation.

The authors evaluate FlexiDepth across a diverse set of benchmarks using task-specific metrics, applying the method to models of 3B, 8B, and 13B parameters. The method achieves strong results across all tested models, even outperforming the baseline on certain benchmarks, with an average retention of 100.7 % across both single-token and multi-token generation tasks.

Consistent with previous observations [9, 11], larger models exhibit greater redundancy. This is reflected in the number of layers skipped: the larger models skip 6.42 and 6.25 layers on average, compared to only 1.49 out of 36 for the smallest model.

The authors further investigate the relationship between layer-skipping patterns and task complexity, analysing layer usage across two domains: language modeling and math reasoning. Tasks requiring deeper contextual understanding, such as continuation, summarization, and arithmetic, consistently utilize more layers than simpler tasks like copying.

Attention-Based MoD

A recurring limitation of token routing methods is the need for a dedicated router — an additional component that must be trained, adds parameters, and introduces computational overhead. Attention-based Mixture-of-Depths (A-MoD) [27] addresses this

by deriving routing decisions from information already present in the model, namely the attention maps.

Specifically, A-MoD aggregates attention maps from the previous layer, averaging each token's interactions with others, and uses the result as an importance measure for routing. The method outperforms standard MoD and consistently selects important tokens, strongly correlating with the leave-one-out importance — measured as the change in model loss when a token is removed at an MoD layer. Being parameter-free, it can be applied to pre-trained transformers without any additional training.

Chapter 3

Empirical Analysis of Model Behavior

This chapter presents an empirical analysis of the selected language model under different modification strategies. We compare methods for estimating importance at the level of layers and token representations, and examine how these choices affect performance when parts of the network are removed.

In all experiments, modifications are applied exclusively to the MLP layers, while attention layers remain unchanged. This design choice reflects their different roles: attention enables information exchange between tokens, while MLP layers operate on each token independently. Consequently, modifying attention would alter how contextual information is shared across the sequence, whereas modifying MLP layers primarily affects per-token transformations.

Skipping the attention mechanism can significantly disrupt information flow in autoregressive generation. Attention computes query, key, and value representations for each token, caching the key-value pairs for efficient decoding. If attention is skipped, these representations are not produced, which prevents subsequent tokens from attending to the affected positions [26].

The experiments in this chapter therefore focus on how MLP computations contribute to model performance. We evaluate local importance criteria and analyse how well these measures align with the global impact of skipping MLP layers. The goal is to identify which local criteria reliably approximate this effect and can be used to filter low-magnitude token-level outputs. The resulting method is later used as the basis for a decision oracle.

3.1 Experimental Setup

In the first section, we outline the experimental setup used throughout the thesis. We introduce the language model, describe its architecture, and present the datasets used for calibration and evaluation.

3.1.1 Language Model

For our experiments, we use the **SmolLM** language model [28], which serves as the basis for analysing model behaviour and evaluating the proposed methods.

SmolLM is available in three configurations: 135M, 360M, and 1.7B parameters. Compared to typical large language models, the number of parameters is much smaller, which makes it easier to work with and reduces computational requirements. This allows for efficient experimentation while keeping the results relevant for larger models.

Architecturally, SmolLM is a decoder-only transformer. The variant used in this work, **SmolLM2-1.7B**, consists of 24 blocks. Each of them comprises a self-attention module and a multilayer perceptron (MLP), corresponding to the FFN described in earlier chapters. Residual connections and layer normalization are applied after each component.

3.1.2 Datasets

This work relies on two datasets: the C4 [29] dataset for calibration and WikiText-2 [30] for evaluation. This combination is standard in language modeling, as the datasets serve complementary roles, and has been widely adopted in prior work, including model compression studies [31, 32].

The **C4** (Colossal Clean Crawled Corpus) dataset consists of text extracted from web pages after removing markup and other non-text content. Since raw web data contains a substantial amount of low-quality or non-linguistic content, C4 applies additional filtering to retain primarily natural language text. Due to its scale and diversity, it is commonly used for pretraining LLMs and for calibration.

The **WikiText-2** dataset is a smaller corpus containing clean, high-quality text from Wikipedia. The articles are written and curated by human editors, resulting in well-structured and reliable text. This dataset is commonly used for language model evaluation and benchmarking.

3.2 Layer Importance Analysis

Assuming that not all layers contribute equally, we analyse their behaviour and their effect on token representations within the model. We measure the importance by the impact of individual layers on perplexity, together with the magnitude of their outputs and its relative significance.

3.2.1 Global Importance

Global importance of a layer can be assessed as measure of model perplexity after removing said layer while keeping all other layers unchanged. In this setup, each token passes through the model along the same path, skipping the layer. A higher resulting perplexity indicates greater disruption, and thus higher importance for the removed layer.

We use the measured values to guide further experiments. In one approach, we gradually remove layers according to their global importance, starting with the least important ones. The order is established at the beginning and remains fixed throughout the process.

Another option is to use an iterative method that re-evaluates layer importance after each removal and then selects the next layer to prune. We adopt this strategy from BlockPruner [11], with minor adjustments. In contrast to the original method, which considers both attention and FFN layers, we restrict pruning to FFN layers. The method can be viewed as a greedy search for layer subsets that minimize the increase in perplexity.

3.2.2 Local Importance

To avoid repeatedly computing model perplexity, we consider local metrics to estimate layer importance. These are based on output magnitude, assuming that layers with larger effects on token representations are more important.

For each token, a layer produces an output vector matching the input dimensionality, which is added through the residual connection. To measure the layer output magnitude, we compute the norm of the output vector for each token and then average these values across all tokens for a given layer.

The magnitude of MLP output vectors, denoted as y, is estimated using the following norms:

$$\textbf{L1 norm:} \quad \|y\|_1 = \sum_{i=1}^{n} |y_i|,$$

$$\textbf{L2 norm:} \quad \|y\|_2 = \sqrt{\sum_{i=1}^{n} y_i^2},$$

$$\textbf{L}\infty \textbf{ norm:} \quad \|y\|_\infty = \max_{1 \le i \le n} |y_i|.$$

We further introduce and evaluate two derived measures that capture additional aspects of layer behaviour:

Relative norm accounts for the magnitude of the input by comparing the norms of the input and output vectors. The computation can be based on L1, L2, or L∞

norms.

$$\| \text{MLP}(x) \| \rightarrow \frac{\| \text{MLP}(x) \|}{\|x\|}$$

Relative weighted norm extends this idea by weighting vector components according to their importance, estimated via gradient flow analysis. This formulation assumes that neurons vary in importance, and that those receiving larger updates during training are more influential.

The weights w are obtained by briefly fine-tuning the model on the C4 dataset and capturing the gradient flow with a backward hook. For each hidden dimension i, the corresponding weight w_i is computed as the accumulated mean squared gradient at that position across all processed tokens. Dimensions that consistently receive stronger gradients are therefore assigned higher weights, indicating a greater influence on the training objective.

$$\| \text{MLP}(x) \| \rightarrow \frac{\| \text{MLP}(x) \odot w \|}{\|x\|}$$

To estimate the importance of an entire layer, we use the standard L2 norm. For token-level pruning, we later consider and compare all previously introduced norms.

3.2.3 Importance Comparison

We compute global and local importance metrics on the C4 dataset. Figure 3.1 indicates that the two types of measures are not well aligned. The basic L2 norm and relative L2 norm exhibit negligible rank correlation with global importance (Spearman $\rho = -0.023$ and $\rho = 0.037$, respectively), whereas the weighted relative L2 norm shows a moderate positive correlation ($\rho = 0.563$).

Local importance, measured via the L2 norm and relative L2 norm of layer outputs, increases toward later layers, with the final layer showing the largest output magnitude. In contrast, global importance is strongly concentrated in the early layers.

Specifically, removing the first two layers causes a major drop in performance, with perplexity rising to several hundred thousand. The 7th layer also has a strong effect, increasing perplexity to about 300 when removed. Two other layers cause a moderate increase to around 30. For most of the remaining layers, the impact is small, with perplexity staying close to 13, compared to the baseline value of 11.78 on C4.

The weighted relative L2 norm appears to partially reflect both local and global behavior, assigning higher importance to the first two layers, the 7th layer, and the final layer.

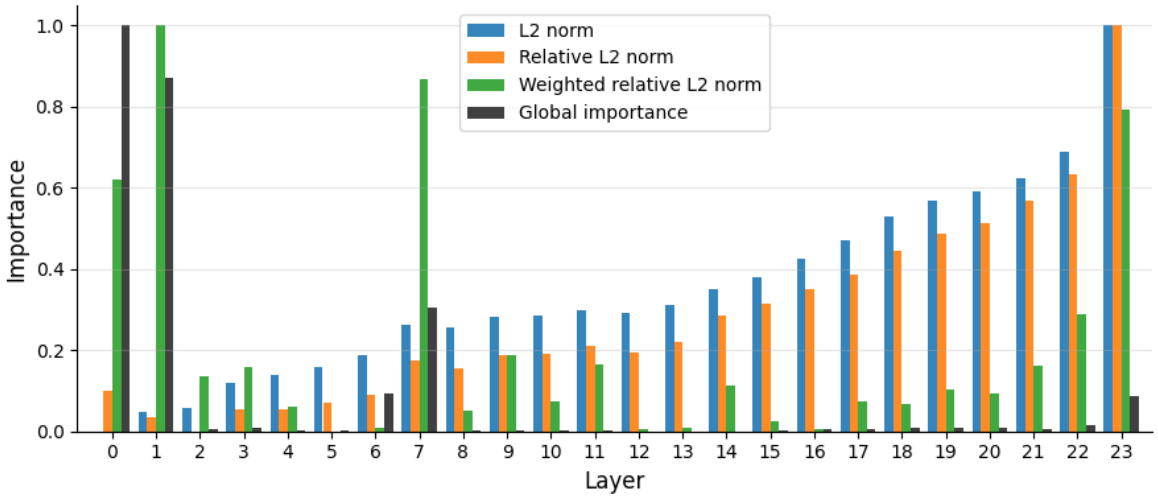

Figure 3.1: Comparison of layer importance across different metrics. Norm-based measures capture local importance via average output magnitude, while global importance is defined by the change in model perplexity after removing a layer. All metrics are log-transformed and min-max normalized to the range $[0, 1]$ for comparability.

3.3 Layer Removal Strategies

This section evaluates multiple approaches for selecting layers to prune. In each case, we progressively remove layers from the model by replacing the MLP block with a module that returns zeros. The model's performance is measured after each step to track the impact of pruning.

Different strategies define the removal order using different criteria. As a baseline, we use a random order, which we compare with strategies based on local and global importance. Additionally, we analyse the effect of removing the deepest layers first.

Figure 3.2 shows how perplexity evolves for each strategy as layers are progressively removed. The best results are achieved using methods based on global importance. The iterative greedy BlockPruner approach allows the removal of up to eight layers (one third of the MLP layers) while keeping the perplexity on the evaluation dataset below 30. In contrast, the strategy based on local importance measured by the L2 norm leads to model collapse immediately after removing the first layer.

We resolve this issue by keeping the first two layers unchanged, after which the model remains relatively stable and performance improves. Similarly, experiments with deepest-first removal indicate that the final layer is also critical. Therefore, it is reasonable to keep those layers intact.

Even with this adjustment, the local importance strategy based on the L2 norm still performs worse than both the deepest-first removal strategy (excluding the final layer) and the random baseline. This suggests that local importance based on a sim-

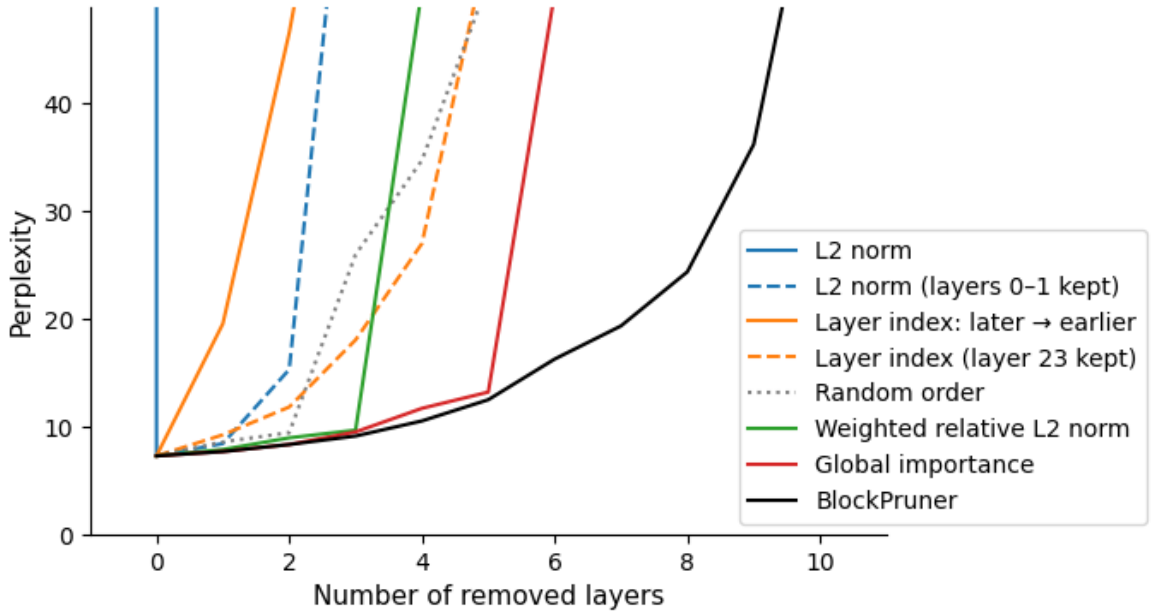

Figure 3.2: Layer skipping strategy comparison. Perplexity on the evaluation dataset as MLP layers are progressively pruned under each strategy.

ple norm may not be a reliable measure of a layer's impact on model performance. With a weighted relative L2 norm, it is possible to remove up to three layers without a significant increase in perplexity, even without protecting any layers.

3.4 Token-Level Pruning

We develop a method where the network path taken by each token is determined by the input itself, rather than by a predetermined set of removed layers. Understanding which layers contribute substantially to each token is central to this approach. Consequently, we also examine the effect of deleting insignificant layer outputs at the token level.

Since this method zeros out layer outputs after they are already computed, rather than actually skipping layers, we treat it as oracle-style pruning. We aim to establish an upper bound on the performance achievable by token-level sparsity, relying solely on local information.

3.4.1 Pruning Method

During inference, the outputs of each MLP block are reviewed and the vectors with small magnitude are replaced by zero vectors. We measure magnitude using the vector norm and compare it to a predefined threshold. Each layer uses its own threshold, computed as a quantile of norms collected on the C4 dataset.

We evaluate the modified model on the WikiText-2 dataset across different quantile

Table 3.1: Perplexity (PPL) and skip ratio for token-level pruning of small outputs using different threshold quantiles and norm types.

	L1 norm		**L2 norm**		**L∞ norm**	
Quantile	**PPL**	**Skip ratio**	**PPL**	**Skip ratio**	**PPL**	**Skip ratio**
0.1	19.8	0.15	12.3	0.08	14.6	0.09
0.2	1270.2	0.43	84.1	0.23	49.3	0.15
0.3	62036.2	0.51	929.3	0.40	389.7	0.21

thresholds and norm types, assessing how well each norm captures the significance of MLP outputs and its effect on model performance.

The distribution and behaviour of tokens in the evaluation dataset may differ from those in the calibration data. As a result, the proportion of removed outputs does not exactly match the chosen quantile, making comparisons between different settings less straightforward.

3.4.2 Results

The initial results using the simple norm are poor, with a substantial increase in perplexity even for small quantile values, as shown in Table 3.1. This suggests that the model may be highly sensitive to modifications in certain layers and that these layers are critical to model performance.

Table 3.1 also shows that the differences between the actual skip ratios and the quantiles used as thresholds are quite large. This discrepancy indicates that pruning alters how tokens are processed, changing the distribution of their internal representations significantly.

To address the issue of critical layers, we evaluate different combinations of layers to keep unchanged. To minimize perplexity, while keeping the number of protected layers low, we choose layers 0, 1, and 23 to be consistently excluded from modification. These layers were also identified as critical in the previous section when pruning entire layers. This adjustment leads to a significant improvement in results and a better alignment between the observed skip ratio and the target quantiles, as presented in Table 3.2.

Keeping the three selected layers unchanged, we evaluate different norm types — basic, relative, and weighted—using L1, L2, and L∞ norms. Figure 3.3 shows the best-performing configuration for each type.

Using local information based on output magnitude, token-level pruning achieves results comparable to BlockPruner, which relies on final perplexity. By making prun-

Table 3.2: Perplexity (PPL) and skip ratio for token-level pruning of small outputs using different threshold quantiles and norm types, with selected layers kept unchanged.

	L1 norm		L2 norm		L∞ norm	
Quantile	**PPL**	**Skip ratio**	**PPL**	**Skip ratio**	**PPL**	**Skip ratio**
0.1	14.2	0.11	9.0	0.07	9.1	0.10
0.2	39.8	0.26	15.1	0.21	12.8	0.18
0.3	96.6	0.38	33.1	0.33	21.7	0.25

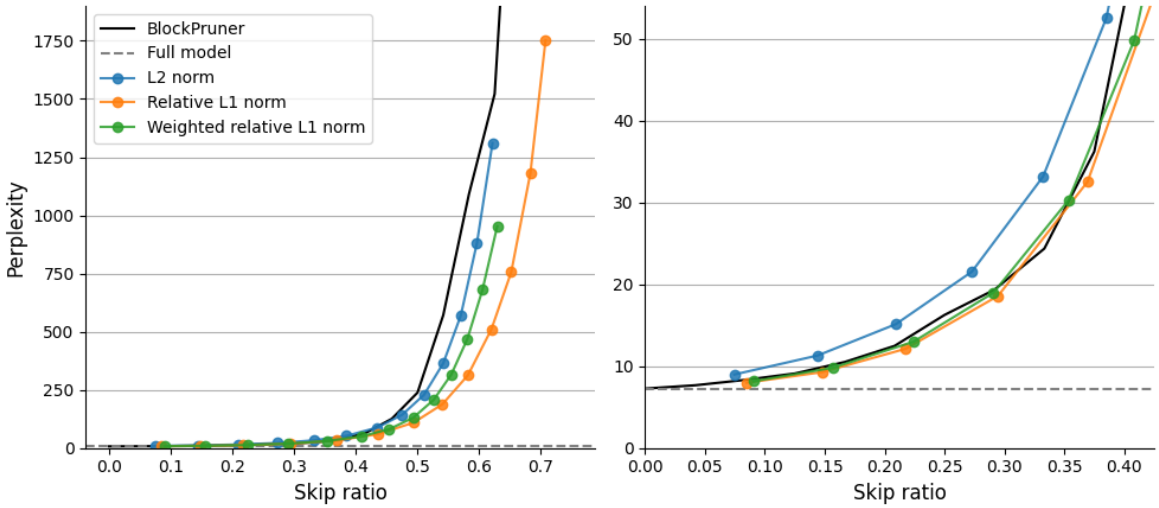

Figure 3.3: Comparison of token-level pruning methods using basic, relative, and weighted norm formulations with L1, L2, and L∞ norms, showing the best-performing configurations for each variant. The figure includes an overview plot and a zoomed-in view for better detail.

ing decisions at the level of individual tokens, this method compensates for the lack of global information. At higher skip ratios, token-level pruning can even outperform BlockPruner; albeit, the high perplexity values for both methods make this improvement practically irrelevant.

The performance differences between the norm variants are small. Considering both effectiveness and simplicity, we choose the relative L1 norm as the preferred option. This approach, which removes small outputs based on the relative L1 norm, is adopted as the base decision oracle.

Chapter 4

Design of Routing Mechanisms

In this chapter, we propose two approaches to adaptive layer-skipping in LLMs. Both methods introduce a routing mechanism into each transformer block that can bypass the MLP layer via the residual connection, as illustrated in Figure 4.1. The goal is to reduce computational cost by selectively skipping transformations that have minimal impact on model performance.

Routing decisions are made at the token level, meaning different tokens can take different paths through the network. As illustrated in Figure 4.2, each token first passes through the router, which decides whether it should be processed by the MLP layer or bypass it via the residual connection. In that case, a zero vector is added to the token representation in place of the MLP output, leaving it unchanged.

In both methods, the router uses the current vector representation of each token to make its decision. In the first approach, routing decisions are based on estimating the importance of the MLP output for each token. If this contribution is expected to be small, the computation is skipped. Since directly computing the MLP output to measure its magnitude would defeat the purpose of skipping it, the ideal skipping decisions — referred to as oracle decisions — must be approximated. Therefore, each router employs a lightweight predictor trained on previously observed oracle behaviour.

In the second approach, we introduce gating mechanisms trained end-to-end to minimize perplexity. These gates learn to control the flow of tokens through the MLP layers directly during training, without relying on a predefined oracle.

Both strategies aim to enable efficient token-level routing while preserving model quality. The predictor-based method does so by approximating explicit importance signals, whereas the gating approach learns routing decisions implicitly. Together, they represent two distinct ways of balancing computational efficiency and model performance.

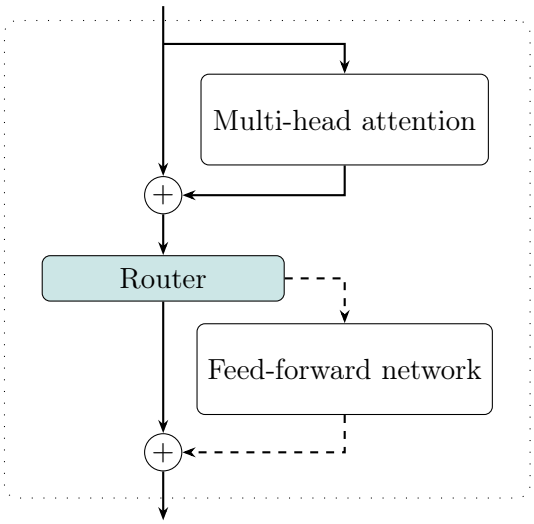

Figure 4.1: Transformer block with token-level MLP routing. For each token, the router determines whether to execute or skip the feed-forward network computation (dashed path).

4.1 Magnitude-Based Predictors

Building on the token-level magnitude thresholding introduced in Section 3.4, we develop predictors that estimate the magnitude of MLP layer outputs to determine which layers can be bypassed for a given token. We refer to this magnitude-based routing signal as the oracle.

We construct an independent predictor for each layer; excluding those chosen to remain fully active. Two types of predictors are considered: **regressors**, which estimate the magnitude of change, and **classifiers**, which predict whether the magnitude exceeds a predefined threshold. We compute the R^2 score for regressors and the AUC score for classifiers to evaluate and compare their performance. However, due to the limited predictive performance of the regressors, we do not use them for routing.

To ensure efficient computation, the predictors use simple architectures: linear regression, logistic regression, and neural networks with one hidden layer, which can capture more complex patterns. The input to each predictor is the current token representation, i.e., the input to the MLP.

The magnitude of change is defined as the relative L1 norm of the MLP output with respect to its input. For classification, we use a threshold corresponding to the precomputed 30th percentile of these values. We obtained the thresholds from statistics collected on the C4 dataset, with a separate threshold defined for each layer.

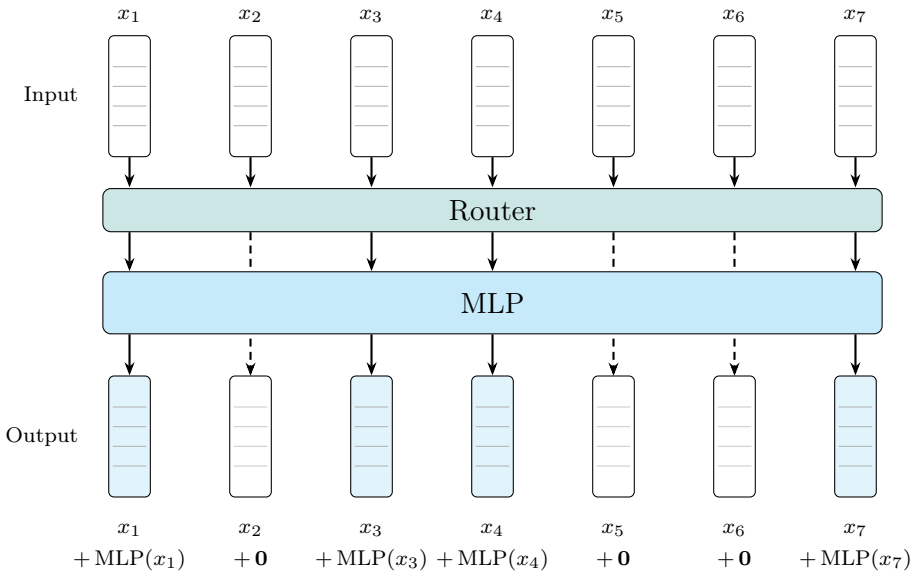

Figure 4.2: Illustration of token-level routing through a single MLP layer. The router processes all tokens and decides which are routed through the feed-forward network and which are skipped. Skipped tokens receive a zero vector in place of the MLP output, leaving their representation unchanged.

4.1.1 Predictor Development

The training data for the predictors are collected using the C4 dataset, which is partitioned into training, validation, and test splits. For each layer, we store the current token representation (MLP input) together with the corresponding MLP output magnitude and the associated binary label for each token. Tokens may appear multiple times across different contexts.

During data collection, we apply the oracle in each MLP layer (except for the three excluded ones), removing outputs with magnitude below the 30th percentile. We chose this threshold based on previous experiments, as it provides a balance between the amount of pruning and a relatively small drop in performance. Using oracles in the preceding layers during data collection is intended to simulate how token representations would evolve in the modified model with routers. However, this approximation does not fully capture the distribution shift introduced by routing, as shown in later experiments.

We use 8 192 tokens (one sample) for the linear and logistic models, given their smaller parameter count of 2 048. In contrast, the neural network–based predictors require significantly more data to train effectively due to their higher capacity, using over 130 000 tokens (sixteen samples).

All models are trained in PyTorch with GPU support. The training objective is to minimize the error on oracle decisions, without directly considering the final perplexity.

To prevent overfitting, we monitor validation performance during training and apply early stopping. For each layer we save the predictor achieving the best AUC score.

4.1.2 Router Integration

We integrate trained predictors directly into the language model, where they serve as routing mechanisms that replace the original MLP layers. Rather than always executing MLP computations, these routers determine on a per-token basis whether the corresponding MLP should be executed or skipped.

Each router operates by applying a classifier to the MLP input and producing a binary decision. When the decision is negative, the router returns a zero tensor of the appropriate shape, effectively bypassing the MLP layer entirely and leaving the residual stream unchanged. When the decision is positive, the standard MLP computation proceeds as usual.

By embedding the routers directly within the model, we are able to evaluate the method in terms of its effect on overall model performance.

4.2 Gated Layers

In the second approach, we replace predictors with gating mechanisms that are trained end-to-end. Instead of approximating the magnitude of layer outputs, the gates learn directly which computations are necessary, with the objective of preserving model performance while reducing computation.

Only the gate parameters are optimized during training, with the rest of the model kept frozen throughout. As in the previous method, we train on the C4 dataset and evaluate on WikiText-2, keeping the experimental setup consistent.

We replace each MLP layer with a `GatedMLP` module, which combines the original MLP with a gating layer. Unlike the predictor-based approach, no layers require special handling, as the model learns which parts are important during gate training.

Before entering the MLP, each token representation passes through the gate, implemented as a simple linear layer followed by a sigmoid function that produces a value between 0 and 1. During training, the gate operates in a continuous regime, where the gate output $g(x)$ scales the MLP output directly. At inference time, the gate values are rounded to obtain binary decisions $\hat{g}(x)$, enabling hard routing. Only selected tokens are processed by the MLP, while the remaining tokens receive a zero vector in place of

the MLP output, effectively skipping the computation:

$$\text{training:} \quad y = g(x) \cdot \text{MLP}(x),$$

$$\text{inference:} \quad y = \begin{cases} \text{MLP}(x), & \text{if } \hat{g}(x) = 1, \\ \mathbf{0}, & \text{if } \hat{g}(x) = 0. \end{cases}$$

To reduce the gap between training and inference, the gates are encouraged to produce values close to 0 or 1. Achieving stable training and good performance, while also targeting the desired skip ratio and near-binary decisions, requires careful choice of initialization, training procedure, and hyperparameters.

4.2.1 Gate Binarization

The desired gate outputs are binary. However, discrete variables are non-differentiable and therefore difficult to optimize. To address this, we use continuous relaxation, which enables stable gradient-based optimization. The resulting soft gate values can be interpreted as probabilistic binary decisions and are later binarized during inference.

The gate forward pass follows the Binary Concrete reparameterization [33], which allows gradients to propagate through stochastic gating decisions during training. The key components of this mechanism are the addition of logistic noise to the gate logits and the use of a temperature-controlled sigmoid function.

The gate output is computed from the linear layer output (logit ℓ) as

$$g = \sigma\left(\frac{\ell + \epsilon}{\tau}\right),$$

where $\epsilon \in \text{Logistic}(0, 1)$ represents logistic noise, τ is the temperature, and σ denotes the sigmoid function.

Logistic noise ϵ added to the gate logits introduces stochasticity, which, together with the sigmoid temperature, encourages the gates to produce outputs closer to 0 or 1 while remaining differentiable. It is computed as

$$\epsilon = \ln u - \ln(1 - u),$$

where u is sampled from a uniform distribution. The noise has a stronger effect in the early stages of training and is gradually reduced, becoming negligible after 70% of training. It is not used during inference.

Temperature τ controls how sharp the gate outputs are, scaling the logits before applying the sigmoid function. A higher temperature produces softer, more uncertain decisions, which is useful during the early stages of training. As the temperature is lowered over time, the outputs become more discrete and confident, approaching hard decisions.

We also incorporate the binarization objective directly into the training loss, as described in a later subsection.

4.2.2 Initialization

We initialize the gate weights to set a controlled starting point for the soft skip ratio. The goal is to start with uncertain, soft decisions that scale down the MLP outputs without removing them entirely.

To achieve this, the bias term is given the main importance. We initialize it to a logit value that, after applying the sigmoid, corresponds to the desired initial skip ratio:

$$\text{init\_logit} = \ln \frac{1 - \text{init\_skip}}{\text{init\_skip}}.$$

For example, an initial skip ratio of 0.3 corresponds to a logit of approximately 0.847.

The remaining weights are initialized with small random values, using a standard deviation of 0.1. Consequently, the initial gate decisions are dominated by the bias. During training, the bias develops much larger variance than the weights in order to produce effective gating decisions. To account for this difference in scale, we use separate learning rates for the bias and the weights.

4.2.3 Loss Function

Training with standard cross-entropy loss proved unstable in our experiments. To address this and to encourage the pruned model to mimic the outputs of the dense model, we use the Kullback-Leibler (KL) divergence [34], following the standard teacher-student framework used in knowledge distillation [35].

KL divergence measures the dissimilarity between two distributions $P(x)$ and $Q(x)$. It is non-negative and equals zero only when the two distributions are identical. For discrete distributions, it is defined as:

$$D_{\text{KL}}(P\|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)},$$

where i indexes the possible outcomes. In our setup, $P(x)$ corresponds to the dense model (teacher) outputs, and $Q(x)$ to the pruned model outputs. Minimizing KL divergence encourages the pruned model to reproduce the output distribution of the dense model.

In addition to matching predictions, we also consider the proportion of skipped token-layer pairs and the degree of binarization of the gate decisions.

The ratio loss measures the squared difference between the actual and target skip ratio:

$$\text{ratio\_loss} = (\text{skip\_ratio} - \text{target\_skip})^2.$$

For this purpose, we estimate the skip ratio as the mean value of all soft gate outputs which are stored for each layer. The ratio is measured across the entire model rather

than individually per layer. This allows some layers to be skipped frequently while others are rarely skipped — reflecting their relative importance, without enforcing uniformity.

Since we compute the ratio from soft gate outputs, it may differ from the ratio of hard skip decisions after rounding. For example, if all gates output 0.7, the soft ratio would meet the target, but rounding would keep all MLP computations. To address this, we encourage continuous gate outputs to move toward the endpoints using a binarization loss.

The binarization loss penalizes intermediate values, promoting confident decisions that are closer to 0 or 1. This helps ensure that the actual skip ratio aligns more with the intended target. We calculate it as:

$$\text{bin\_loss} = \text{mean}\big(g \cdot (1 - g)\big),$$

where g are the token-layer gate outputs. The loss reaches its maximum at 0.5, and is minimized when outputs are near the endpoints.

The total loss combines the prediction quality and gate behaviour objectives:

$$\text{total\_loss} = \text{lm\_loss} + \lambda_{\text{ratio}} \cdot \text{ratio\_loss} + \lambda_{\text{bin}} \cdot \text{bin\_loss}.$$

The coefficients λ_{ratio} and λ_{bin} control the contribution of the ratio and binarization terms. Balancing these weights allows us to optimize all objectives simultaneously during training.

Chapter 5

Results

This chapter presents evaluation results for the two routing approaches developed in this work: magnitude-based routers and gated layers. Both are assessed in terms of their ability to reduce MLP computation while preserving language model quality. We compare both approaches against the established baselines: the oracle and BlockPruner.

5.1 Magnitude-Based Routers Evaluation

We evaluate the magnitude-based routers in two stages. First, we analyse individual predictors based on their ability to approximate oracle routing decisions, and select the best-performing model for each layer.

Subsequently, the selected predictors are integrated into the language model and evaluated in an end-to-end setting. Overall model performance is more informative of their practical usefulness than standalone predictor accuracy. We analyse the resulting model behaviour to assess the effectiveness of magnitude-based routing in practice.

5.1.1 Individual Predictor Performance

Regression-based predictors are found to be insufficient, exhibiting high variability across layers and model types, as shown in Figure 5.1. This indicates that directly predicting output magnitude is a difficult task for predictors of this form, including both linear regression and shallow neural networks. Due to their consistently poor performance on the collected data, regressors are excluded from further analysis.

In contrast, classifiers yield better results. Figure 5.2 presents the AUC scores of the best-performing classifiers for each layer. The results show that predictions correlate well with token importance, particularly in the early and final layers, while performance degrades in the middle layers. The difference between logistic regression

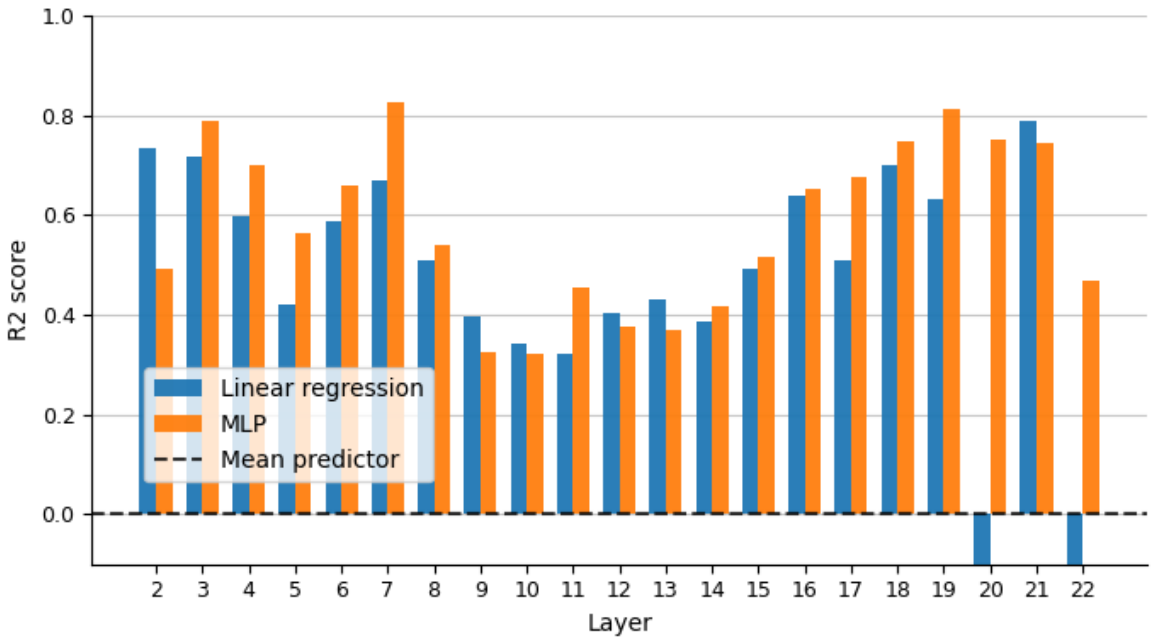

Figure 5.1: R^2 scores of regressors for output magnitude prediction across layers. Negative R^2 values indicate performance worse than always predicting the mean of the data. Two predictors yield strongly negative values (-6.1 and -0.4), which are partially truncated in the figure for readability.

and neural network classifiers is minimal, with neural models providing only marginal improvements.

5.1.2 Performance of the Router-Augmented Model

In Table 5.1, we report the perplexity of pruned models using different router types, together with the corresponding ratio of skipped token-layer pairs, after applying the improvement attempts described in this subsection. Nonetheless, due to differing skip ratios, router performance is not directly comparable from the table alone. For this reason, Figure 5.5 additionally provides a visual comparison of router performance against the BlockPruner method and the oracle.

Basic router

Initial evaluation of the pruned model with integrated routers on the WikiText2 dataset shows poor performance for both logistic regression and neural network variants. The overall skip ratio is over 40 %, compared to a target value of 30 %.

Routing behaviour is highly uneven across layers and router types, with a strong bias toward skipping later layers (excluding the final two layers), which are bypassed for 70–90 % of tokens (Figure 5.3). This shows that the basic routing strategy does

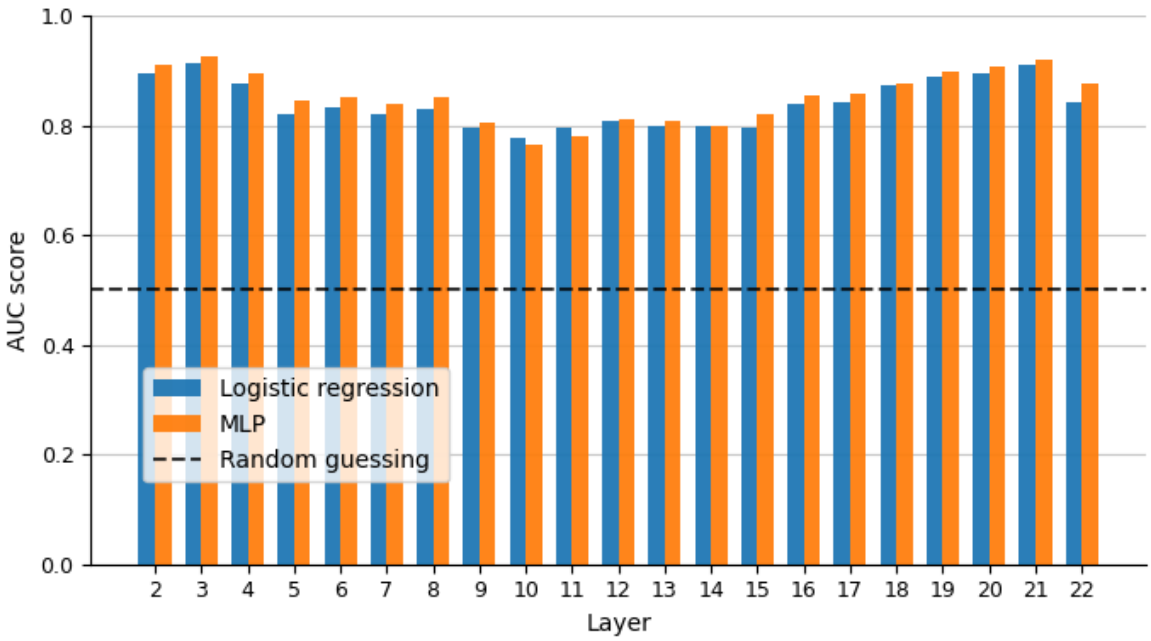

Figure 5.2: AUC scores of classifiers of output magnitude across layers, compared to the performance of random guessing.

not produce a balanced distribution of computation across the model. This imbalance is likely caused by a distribution shift between training and inference: predictors are trained on representations produced by oracle-based routing, yet at inference time they operate on representations shaped by their own decisions. We address this later with an iterative training scheme.

AUC-filtered router

As the first modification, we apply routers only in layers where predictors achieve an AUC above 0.85. This is intended to exclude unreliable predictors that perform poorly even on the training data.

This change leads to lower perplexity; nevertheless, the improvement is primarily attributable to a reduced skip ratio rather than improved routing quality.

Single-Skip router

The Single-Skip router modifies the basic routing scheme by preventing any token from skipping two consecutive MLP layers. The motivation is to avoid long uninterrupted blocks of skipped computation. Each router takes into account the decision from the previous layer: tokens that were skipped in the preceding layer are automatically routed through the MLP, while all other decisions are determined by the predictor as usual.

In order to evaluate the contribution of the learned predictors, we also introduce

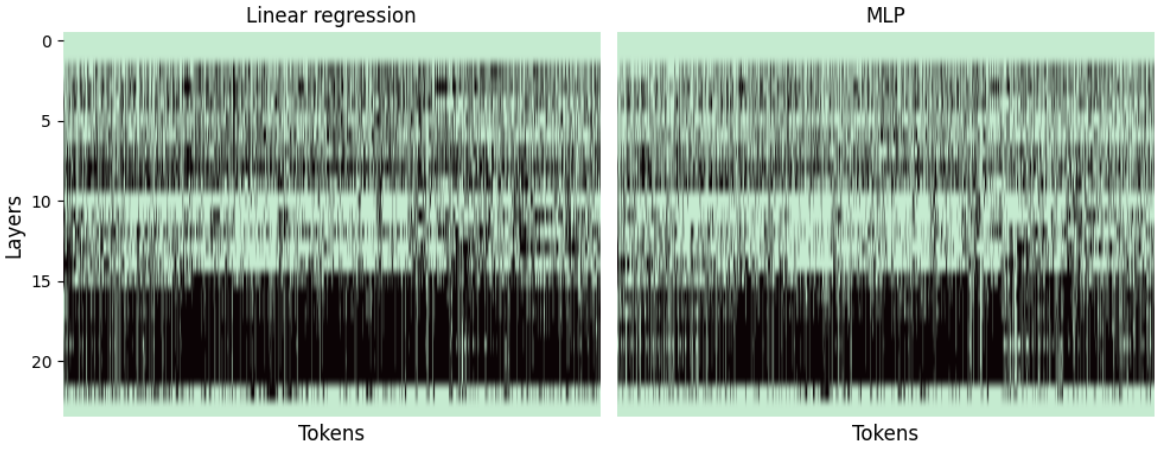

Figure 5.3: Routing decisions of the basic router. Light green indicates tokens passing through the MLP, while black indicates skipped computation. Tokens traverse the layers from top to bottom.

a Random Single-Skip variant. It enforces the same no-consecutive-skip constraint, but assigns routing decisions randomly otherwise. While this baseline is not entirely degenerate, its performance is consistently worse than that of the predictor-based Single-Skip router.

Iterative router

A key issue observed in earlier experiments is the mismatch in skip ratios across layers. In an experiment, we left earlier layers (up to layer 15) without routing, which resulted in acceptable skip ratios in subsequent layers but only delayed the underlying imbalance. This behaviour is likely caused by differences between oracle-based decisions used during training and predictor-based decisions at inference time, which lead to a distribution shift in token representations.

To address this, we modify the data collection procedure for predictor training to an iterative scheme. Predictors are trained layer by layer, while already-trained routers are applied in all preceding layers during data collection. This ensures that the inputs to each layer reflect the actual routing behaviour that will be used at inference time. Once a predictor is trained for a given layer, we immediately integrate it into the model and use it for collecting data for subsequent layers.

This iterative procedure improves the consistency of skip ratios across layers. In particular, it reduces cases where large blocks of layers bypass almost all tokens, and leads to skip ratios that match the target distribution more closely, as illustrated in Figure 5.4.

We apply the same iterative training strategy to the Single-Skip router variant, following its constrained routing design during data collection.

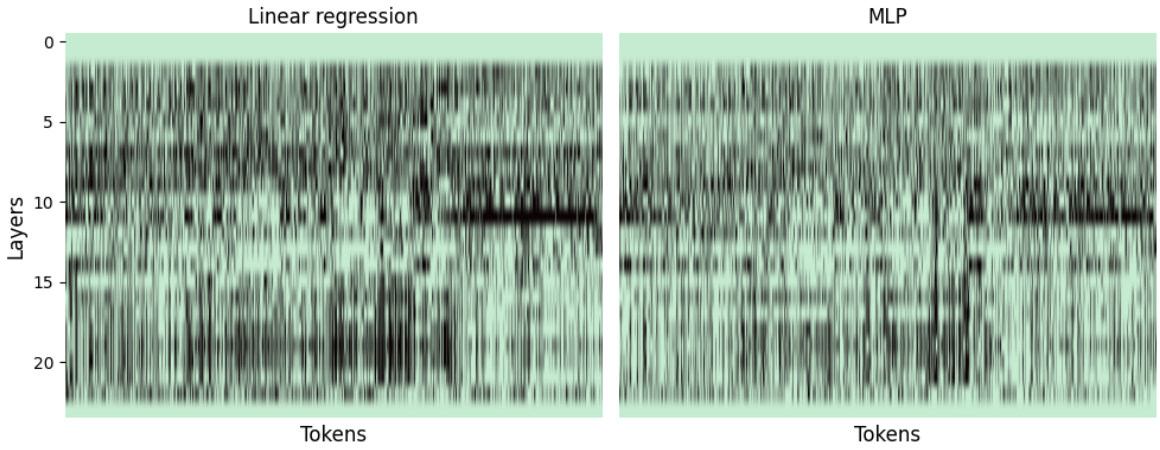

Figure 5.4: Routing decisions of the iterative router, using the same layout and colour encoding as Figure 5.3. Compared to the basic router, skipping is distributed more evenly across layers, with no large contiguous blocks of bypassed computation.

Table 5.1: Performance of the pruned model with different router types.

Router	**Predictor**	**Perplexity**	**Skip ratio**
Basic	LR	255.03	0.47
	NN	143.12	0.42
AUC-filtered	LR	10.41	0.13
	NN	20.49	0.23
Single-Skip	LR	16.95	0.25
	NN	14.24	0.23
Random Single-Skip	–	40.51	0.20
	–	113.07	0.25
Iterative	LR	52.03	0.35
	NN	33.84	0.31
Iterative Single-Skip	LR	13.53	0.22
	NN	11.43	0.19

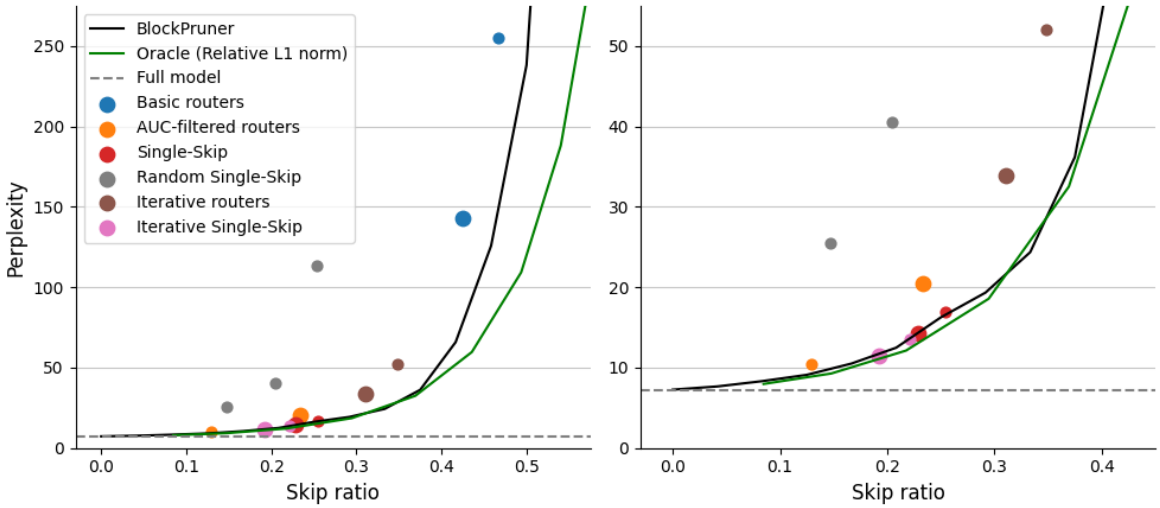

Figure 5.5: Comparison of router types with the oracle and BlockPruner baselines. Markers indicate predictor type: small for logistic regression, bigger for neural network. The right panel shows a zoomed-in view of the low perplexity region.

Router Type Comparison

In general, the magnitude-based routers did not outperform the BlockPruner baseline. Figure 5.5 shows that routers achieve perplexity comparable to the BlockPruner and the oracle only at lower skip ratios, up to approximately 25 %.

Basic routers trained with the naive data collection procedure exceed the target skip ratio of 30 %, which leads to a noticeable increase in perplexity. This issue is mitigated either by filtering routers based on AUC or by enforcing the Single-Skip constraint, both of which reduce the skip ratio.

The Single-Skip approach achieves perplexity below 17 while reducing MLP computation by approximately 25 %. These methods approach oracle-level performance, however, they introduce additional constraints into the routing logic. Iteratively trained routers provide a more stable solution. They produce skip ratios slightly above 30 % and achieve perplexities of 34 and 52 for the two variants.

We do not observe any clear benefit from using a more complex classifier, as logistic regression performs comparably to a neural network with one hidden layer. Differences in perplexity between classifier types are consistent with corresponding differences in skip ratio.

5.2 Gated Model Evaluation

We evaluate gated models trained across a range of hyperparameter configurations, achieving skip ratios between 20 % and 45 %. Training is performed on subsets of

Table 5.2: Best gated model perplexity at each skip ratio, across different numbers of training samples.

128		**256**		**512**	
Perplexity	Skip ratio	Perplexity	Skip ratio	Perplexity	Skip ratio
12.35	0.22	11.84	0.20	10.52	0.21
11.61	0.25	11.14	0.26	10.66	0.25
29.50	0.32	16.91	0.33	15.32	0.33
21.34	0.35	17.86	0.36	15.96	0.36
41.99	0.43	32.23	0.43	38.55	0.42
46.90	0.45	37.16	0.45	38.46	0.46

the C4 dataset of sizes 128, 256, and 512 samples. The models are compared against the BlockPruner, the oracle, and magnitude-based routers in terms of perplexity. We additionally analyse their routing decisions to understand which layers are skipped more frequently and how this behaviour changes with the amount of training data.

5.2.1 Evaluation of Top-Performing Models

Figure 5.6 presents the best-performing model configurations for each combination of training set size and skip ratio level, with colour denoting the number of training samples. For each skip ratio threshold — ranging from 0.15 to 0.46 in increments of 0.01 — we selected the two lowest-perplexity gated models from each training size category whose skip ratio exceeded the threshold by no more than 0.05.

The gated models achieve considerably lower perplexity than both the BlockPruner baseline and the local-importance-based oracle. Training set size has a modest but consistent effect: models trained on more data tend to reach lower perplexity, with differences typically on the order of a few units. This trend is most clearly observed in the middle of the skip ratio range (25–40 %).

Table 5.2 provides a closer comparison of selected results. For clarity, we show only the single best-performing model from each training size category across skip ratio levels from 0.20 to 0.45 in steps of 0.05. We allow each model's actual skip ratio to fall up to 0.035 above the target threshold, ensuring that models across training categories remain truly comparable.

5.2.2 Decision Analysis

Prior to analysing layer-wise skip patterns, we examine how gate outputs evolve during training. Initially, gate values are concentrated around intermediate values — a mean

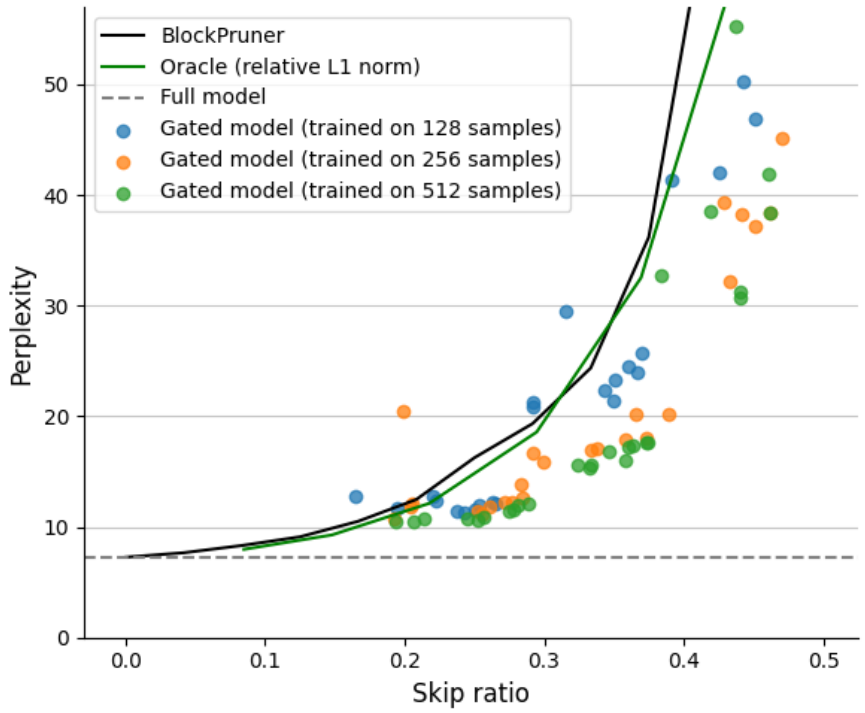

Figure 5.6: Comparison of the best-performing gated models at different skip ratios. Colours indicate the amount of training data used (128, 256, or 512 C4 samples). Results are compared with the BlockPruner baseline and the oracle.

of 0.68 — reflecting uncertain routing decisions due to the chosen initialization. As training progresses, the distribution shifts toward 0 and 1, and the gating behaviour becomes increasingly binary, as illustrated in Figure 5.7.

To compare routing decisions across models trained on different amounts of data, we select models from each training category (128, 256, and 512 samples) with similar skip ratios and perplexities — specifically, those achieving a skip ratio of approximately 0.26 and perplexity between 11 and 12.

Figure 5.8 shows the routing decisions for the same 2048 tokens across all three strategies. The strategies show partial agreement in which layers they choose to skip. In particular, the first layer and the last two layers are rarely skipped, while layers 12, 13, 20, and 21 are skipped most frequently, as further illustrated in Figure 5.9.

Models trained on more data tend to skip layers 1 and 2 more often. In contrast, layer 8 is increasingly preserved as the amount of training data grows.

Occasionally, all three strategies agree on skipping the same layer for certain tokens, even when that layer is not among the most frequently skipped overall. This can be observed in the central part of Figure 5.8, where layer 14 — skipped for approximately 25% of tokens — shows a similar skipping pattern across all three strategies.

The most distinct layer-wise separation is observed in the model trained on only 128 samples. As the amount of training data increases, routing decisions become

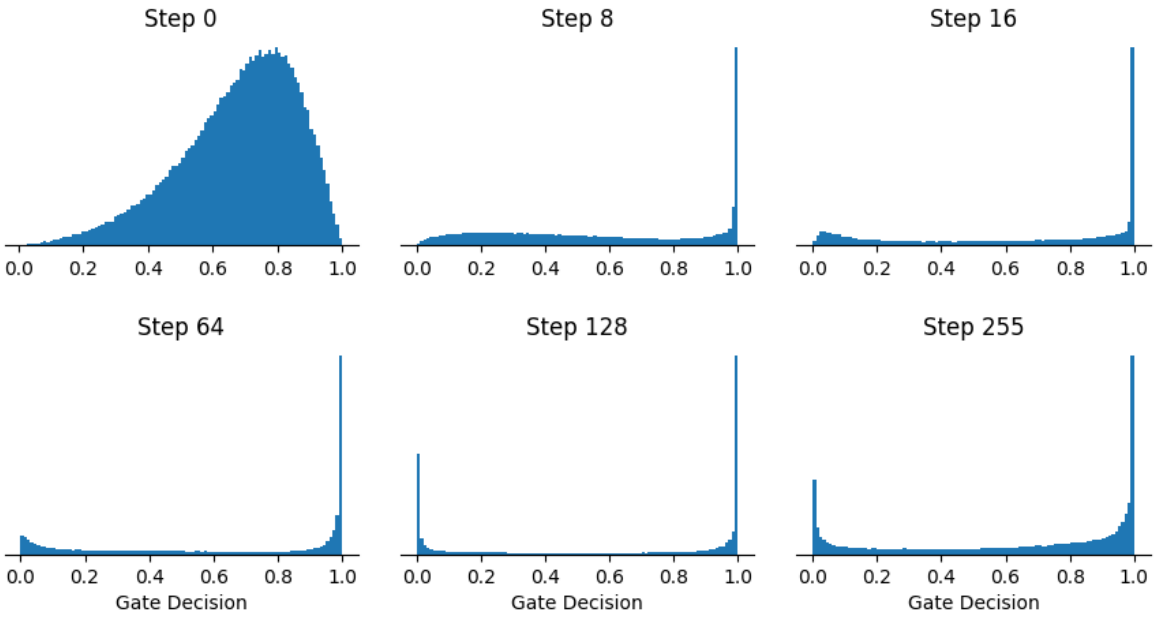

Figure 5.7: Distribution of gate outputs during training, gradually shifting from intermediate values toward binary outputs.

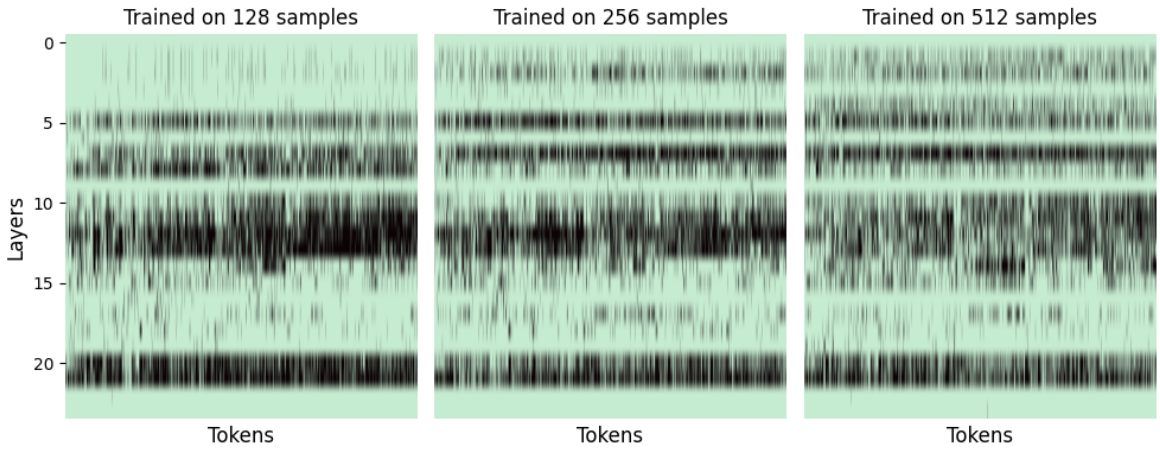

Figure 5.8: Gate decisions. Light green indicates tokens passing through the MLP, while black indicates skipped tokens. In this visualization, tokens traverse the layer from top to bottom.

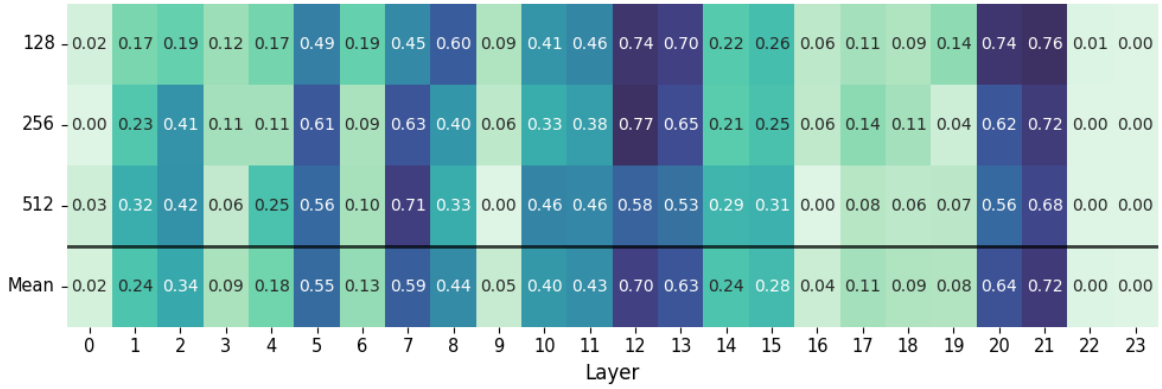

Figure 5.9: Per-layer skip ratio for models trained with different numbers of training samples. The heatmap also shows the mean skip ratio across methods for each layer.

Table 5.3: Average per-layer variance of decisions across gated models trained on different numbers of samples.

Training samples	Mean gate variance
128	0.0373
256	0.0427
512	0.0556

more dispersed across layers and less aligned with full-layer skipping patterns. This is reflected in the per-layer variance reported in Table 5.3.

Models trained on more data likely capture more informative patterns, leading to more refined routing decisions. As a result, their routing becomes more adaptive at the token level, deviating from coarse layer-wise skipping strategies and achieving lower perplexity.

Comparison to Layer-Skipping Methods

Figure 5.10 compares the layer skipping decisions of gated models with those of Block-Pruner, global importance, and local importance-based methods. For the local metrics, we include the weighted relative norm — the best-performing L2-based variant in the gradual layer-skipping experiment, though one that led to model collapse beyond three removed layers — as well as the basic L2 norm, which performed worse.

For comparability, we consider the six layers assigned lowest importance by each method, corresponding to a skip ratio of 25 %. Two additional layers that would be removed next, representing an increase to a 30 % skip ratio, are also shown in a lighter

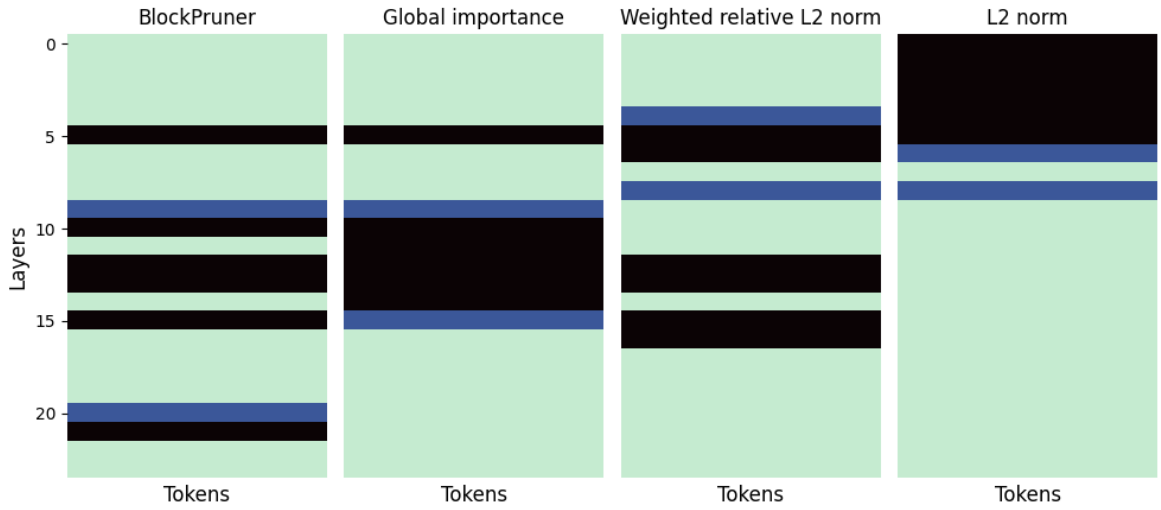

Figure 5.10: Layers skipped by BlockPruner, global, and local importance methods. Darker shading indicates the six least important layers pruned at a 25 % skip ratio; lighter shading marks two additional candidates extending the ratio to 30 %.

colour.

Both BlockPruner and the global importance metric produce decisions that resemble those of the gated models, primarily skipping layers in the middle of the model. The removal of the fifth layer is consistent across all layer-skipping methods and the gated models. The 20th and 21st layers, frequently skipped by the gated models, are also identified by BlockPruner but not by the other methods. Interestingly, both global importance methods suggest skipping the 9th layer, even though the gated models almost never do so.

In contrast, local importance metrics show poor agreement with the gated models, particularly the basic L2 norm, which tends to skip the earliest layers.

In Figure 3.1 in an earlier chapter, the 7th layer was assigned high importance by both the global method and the local metric based on the weighted relative L2 norm of the layer output. BlockPruner implicitly reflects this by keeping this layer in the model the longest. However, the gated models skip this layer for an average of 59 % of tokens, making it the fifth most frequently skipped layer.

Conclusion

In this work, we investigated the importance of layers and their contribution to token processing and model outputs. We introduced two adaptive layer-skipping strategies, employing magnitude-based routers and gated layers. We now summarize our main findings.

Our experiments show that the average output magnitude does not perfectly correlate with the actual influence of a layer on model performance. However, making skipping decisions for individual tokens based on the layer's output magnitude achieves performance comparable to BlockPruner [11], even slightly exceeding it. The result is notable since BlockPruner iteratively removes layers using direct information about their impact on model performance, while our method relies solely on output magnitude.

A possible explanation is that some layers are crucial for processing certain tokens, even if this does not surface when averaging output magnitude across all tokens. The freedom of tokens to take different routes outweighs the disadvantage of not having direct information about model performance.

Taking this token-level layer skipping method based on output norm as an oracle, we find that predicting whether a layer's output will be of significant magnitude is a fairly doable task, even with a simple 1–2 layer classification model. Although the routers do not make perfect predictions, integrating them into the LLM achieves performance matching the oracle up to a skip rate of 25 %. These results were, however, obtained with some further improvements, such as only applying the router in more predictable layers or ensuring that no token skips two consecutive layers.

Plain iteratively trained routers with no further modification achieved perplexity of 34 and 52 at skip rates of 31 and 35 % of FFN computation, differing from the oracle by approximately 15 and 25 points, respectively. We did not find these results satisfactory. They indicate that output magnitude alone is not sufficient information about a layer's contribution to model performance, even at the granularity of individual tokens.

A possible reason for the poorer performance of magnitude-based routers is the fixed skip ratio applied uniformly to every layer. The classifiers were trained to route 30 % of tokens around each layer, regardless of its importance. As the gated model routing decisions revealed, layers differ substantially in importance, with some passing

almost all tokens through and others skipping more than 70 %. Allowing a varying skip ratio per layer could improve results, though this would require a reliable strategy for determining the appropriate ratio for each layer.

As an alternative, we developed a second method leveraging global information about model output quality. We trained gated layers jointly, end-to-end, minimizing the difference in output logit distribution compared to the original model. This approach proved more effective, for example achieving perplexity of 15.3 at a skip ratio of 33 %, and consistently outperforming both BlockPruner and the magnitude-based oracle across all tested skip ratios. Gated models trained on more data achieved slightly better results, though the difference was small.

A notable challenge in this work was getting the gates to train properly. The standard cross-entropy loss proved insufficient and unstable. We therefore minimized KL divergence as the training objective, forcing the gated model to imitate the behaviour of the full model rather than directly optimizing next-token prediction performance.

We analysed the routing decisions made by the gated models and compared them to methods that skip entire layers. Although the models were free to make routing decisions at the token level, we observe clear layer-wise patterns, with significant differences in routing behaviour across layers.

Certain layers were almost never skipped: the first, the last two, and the ninth. The ninth layer is especially interesting, as it showed no signs of importance by either output magnitude or the effect of individually removing it from the model. On the other hand, layers skipped for more than half of all tokens included layers 5, 7, 12, 13, 20, and 21, spanning early, middle, and deep parts of the network. This suggests that importance cannot be attributed to any single region of the model — it must be assessed at the level of individual layers.

Several directions remain open for future work. Although the results are promising, it would be informative to evaluate the pruned models on downstream tasks, beyond perplexity alone. Another interesting direction would be to apply these methods to larger LLMs. The language model used throughout this thesis is considerably smaller than typical LLMs, and both intuition and prior work [9, 11, 26] suggest that larger models tend to be more redundant, leading us to expect that our methods could perform even better at scale.

Analysing which tokens require less computation and whether this can be interpreted intuitively would also be valuable, and might shed light on whether certain layers specialize in processing certain groups of tokens.

Some methods applied additional finetuning after pruning to recover lost performance [9, 14]. In our setting this was not directly applicable, as the pruning decisions are made dynamically for each token during inference, leaving no fixed pruned model to finetune. However, with gated layers it might be beneficial to also adjust the model pa-

rameters alongside or after training the gates. Alternatively, as done in Flexidepth [26], a lightweight adapter layer could be added to process tokens routed around the FFN, minimizing the impact on model outputs.

Finally, despite the computational savings, our method is not expected to achieve wall-clock speedup without further work. This is due to irregular routing, which does not align well with existing GPU architectures. Further hardware optimizations would be needed to fully exploit the potential of adaptive layer skipping and enable actual speedup.

We leave these limitations and open questions as inspiration for future work.

Bibliography

[1] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, with Language Models*. 3rd ed., 2026.

[2] A. Grattafiori, A. Dubey, A. Jauhri, *et al.*, "The Llama 3 Herd of Models," *arXiv preprint arXiv:2407.21783*, 2024.

[3] A. Ben Abacha, W.-w. Yim, Y. Fu, *et al.*, "MEDEC: A Benchmark for Medical Error Detection and Correction in Clinical Notes," *arXiv preprint arXiv:2412.19260*, 2024.

[4] N. Shazeer, A. Mirhoseini, K. Maziarz, *et al.*, "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer," in *International Conference on Learning Representations*, 2017.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019.

[6] W. X. Zhao, K. Zhou, J. Li, *et al.*, "A Survey of Large Language Models," *Frontiers of Computer Science*, 2026.

[7] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[8] M. Samragh, M. Farajtabar, S. Mehta, *et al.*, "Weight Subcloning: Direct Initialization of Transformers Using Larger Pretrained Ones," *arXiv preprint arXiv:2312.09299*, 2023.

[9] A. Gromov, K. Tirumala, H. Shapourian, *et al.*, "The Unreasonable Ineffectiveness of the Deeper Layers," in *International Conference on Learning Representations*, 2025.

[10] Y. Chang, X. Wang, J. Wang, *et al.*, "A Survey on Evaluation of Large Language Models," *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 3, 2024.

[11] L. Zhong, F. Wan, R. Chen, *et al.*, "BlockPruner: Fine-grained pruning for large language models," in *Findings of the Association for Computational Linguistics: ACL 2025*, 2025.

[12] A. Radford, J. Wu, R. Child, *et al.*, "Language Models are Unsupervised Multitask Learners," tech. rep., OpenAI, 2019.

[13] Hugging Face, "SmolLM: Blazingly Fast and Remarkably Powerful," 2024. Blog post.

[14] K. Zhu, F. Hu, Y. Ding, *et al.*, "A Comprehensive Review of Network Pruning Based on Pruning Granularity and Pruning Time Perspectives," *Neurocomputing*, 2025.

[15] D. Lepikhin, H. Lee, Y. Xu, *et al.*, "GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding," in *International Conference on Learning Representations*, 2021.

[16] V. Macko and V. Boža, "MACKO: Sparse Matrix-Vector Multiplication for Low Sparsity," *arXiv preprint arXiv:2511.13061*, 2025.

[17] X. Men, M. Xu, Q. Zhang, *et al.*, "ShortGPT: Layers in Large Language Models are More Redundant Than You Expect," in *Findings of the Association for Computational Linguistics: ACL 2025*.

[18] A. K. Jaiswal, B. Hu, L. Yin, *et al.*, "FFN-SkipLLM: A Hidden Gem for Autoregressive Decoding with Adaptive Feed Forward Skipping," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024.

[19] J. Frankle and M. Carbin, "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks," in *International Conference on Learning Representations*, 2019.

[20] S. Ashkboos, M. L. Croci, M. G. do Nascimento, *et al.*, "SliceGPT: Compress Large Language Models by Deleting Rows and Columns," in *The Twelfth International Conference on Learning Representations*, 2024.

[21] Y. Yang, Z. Cao, and H. Zhao, "LaCo: Large Language Model Pruning via Layer Collapse," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024.

[22] K. Alizadeh-Vahid, S. Iman Mirzadeh, H. Shahrkokhi, *et al.*, "Duo-LLM: A Framework for Studying Adaptive Computation in Large Language Models," in *Proceedings of The 4th NeurIPS Efficient Natural Language and Speech Processing Workshop*, vol. 262 of *Proceedings of Machine Learning Research*, 2024.

[23] Z. Zeng, Y. Miao, H. Gao, *et al.*, "AdaMoE: Token-Adaptive Routing with Null Experts for Mixture-of-Experts Language Models," in *Findings of the Association for Computational Linguistics: EMNLP 2024*.

[24] D. Raposo, S. Ritter, B. Richards, *et al.*, "Mixture-Of-Depths: Dynamically Allocating Compute in Transformer-Based Language Models," *arXiv preprint arXiv:2404.02258*, 2024.

[25] J. Ainslie, T. Lei, M. de Jong, *et al.*, "CoLT5: Faster Long-Range Transformers with Conditional Computation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[26] X. Luo, W. Wang, and X. Yan, "Adaptive Layer-skipping in Pre-trained LLMs," in *Second Conference on Language Modeling*, 2025.

[27] A. Gadhikar, S. K. Majumdar, N. Popp, *et al.*, "Attention Is All You Need For Mixture-of-Depths Routing," *arXiv preprint arXiv:2412.20875*, 2024.

[28] L. B. Allal, A. Lozhkov, E. Bakouch, *et al.*, "Smollm - Blazingly Fast and Remarkably Powerful," 2024.

[29] C. Raffel, N. Shazeer, A. Roberts, *et al.*, "Exploring the Limits of Transfer Learning With a Unified Text-To-Text Transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, 2020.

[30] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer Sentinel Mixture Models," in *International Conference on Learning Representations*, 2017.

[31] E. Frantar and D. Alistarh, "SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot," in *Proceedings of the 40th International Conference on Machine Learning (ICML)*, vol. 202 of *Proceedings of Machine Learning Research*, 2023.

[32] V. Boža, "Fast and Effective Weight Update for Pruned Large Language Models," *Transactions on Machine Learning Research*, 2024.

[33] C. J. Maddison, A. Mnih, and Y. W. Teh, "The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables," in *International Conference on Learning Representations*, 2017.

[34] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[35] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," in *NIPS Deep Learning and Representation Learning Workshop*, 2015.

Appendix A: GitHub Repository

All relevant source code is available in a public GitHub repository: `https://github.com/chutnakova5/Adaptive-layer-skipping`. The repository contains implementations of the proposed methods, experiments, analyses, and evaluation scripts, together with results, tables, and visualizations.

The core files are listed below:

- `layer_importance.ipynb` — analysis of layer importance, comparison of layer removal strategies, and oracle-based token-level skipping;
- `predictor_definitions.py` — definitions of lightweight prediction models used in magnitude-based routing;
- `predictor.ipynb` — data collection, training, and evaluation of simple predictors estimating layer output magnitude;
- `predictor_v2.ipynb` — iterative data collection, training, and evaluation of predictors, where each iteration uses predictors in previous layers to collect training data for subsequent layer;
- `predictor_v3.ipynb` — iteratively trained predictors designed for use in SingleSkip routing;
- `router_evaluation.ipynb` — integration of predictors into the language model and evaluation of the pruned LLM, including AUC-filtered and SingleSkip routing;
- `router_v2_evaluation.ipynb` — integration of predictors trained on iteratively collected data and evaluation of the pruned LLM;
- `router_v3_evaluation.ipynb` — integration of SingleSkip predictors trained on iteratively collected data and evaluation of the pruned LLM;
- `gates_kl.ipynb` — design, training, and hyperparameter tuning of gated models using a KL-based objective with varying numbers of training samples;
- `gates.ipynb` — earlier version of gated model training using cross-entropy loss;

- `gates_evaluation.ipynb` — training and evaluation of gated models with the best-performing configurations, including analysis of routing decisions;
- `routing_results.ipynb` — analysis and visualization of results for magnitude-based routing and gated models, including predictor performance and comparison of both methods against BlockPruner and oracle-based pruning;
- `block_pruner.ipynb` — implementation and evaluation of the BlockPruner baseline;
- `results/` — directory containing tables with results of all experiments.