Informatics

Towards Resource Efficient Code Generation through Dynamic Early Exits in LLMs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Lukas Florian Briem, BSc.
Matrikelnummer 11828347

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof.in Mag.a rer.soc.oec. Dr.in rer.soc.oec. Ivona Brandić
Mitwirkung: M.Tech. PhD Shashikant Shankar Ilager

Wien, 18. Jänner 2025

Lukas Florian Briem | Ivona Brandić

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Informatics

Towards Resource Efficient Code Generation through Dynamic Early Exits in LLMs

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Lukas Florian Briem, BSc.
Registration Number 11828347

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Mag.a rer.soc.oec. Dr.in rer.soc.oec. Ivona Brandić
Assistance: M.Tech. PhD Shashikant Shankar Ilager

Vienna, January 18, 2025

Lukas Florian Briem	Ivona Brandić

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Lukas Florian Briem, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 18. Jänner 2025

Lukas Florian Briem

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my thesis supervisors, Dr. Shashikant Ilager and Prof. Ivona Brandić, for giving me the opportunity to work on such an interesting topic. Their guidance, encouragement, and support have been instrumental throughout this thesis. I am particularly grateful to Shashikant for his exceptional mentorship, constructive feedback, and patience. I sincerely appreciate his promptness, constant availability, and willingness to accommodate meetings and provide valuable guidance whenever needed.

Additionally, I would like to thank my colleagues and friends at TU Wien for their support and camaraderie. The shared experiences, collaboration, and memorable moments made my time at TU Wien truly special.

Lastly, I am deeply grateful to my family for their unwavering support and encouragement throughout my studies. Their constant belief in me and support in every endeavor have been the foundation of both my academic and personal achievements.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kurzfassung

Große Sprachmodelle (Large Language Models, LLMs) sind zum Stand der Technik in der Verarbeitung natürlicher Sprache (Natural Language Processing, NLP) und im Verstehen natürlicher Sprache (Natural Language Understanding, NLU) geworden. Neben Anwendungen wie Textzusammenfassung, Übersetzung und Texterstellung gewinnen LLMs auch in der Softwareentwicklung zunehmend an Bedeutung. Tools wie GitHub Copilot und AmazonQ bieten intelligente Codevervollständigung und unterstützen Millionen von Entwicklern weltweit.

Trotz ihrer Effektivität in vielen Bereichen sind LLMs extrem ressourcenintensiv und benötigen erhebliche Rechenressourcen für Training und Einsatz in der Praxis. Während die Ressourcenineffizienz des Trainings von LLMs umfassend untersucht wurde, wurde den Ineffizienzen während der Inferenz weniger Aufmerksamkeit geschenkt. Aufgrund ihres kontinuierlichen Charakters wird die Inferenz jedoch mit der Zeit immer ressourcenintensiver. Dies gilt insbesondere für Code-Vervollständigungsaufgaben, bei denen Vorschläge oft nach jeder Dateiänderung ausgelöst werden.

Daher ist die Verbesserung der Ressourceneffizienz der Codekomplettierung während der Inferenz von entscheidender Bedeutung für die Verbesserung der Nachhaltigkeit. Es wurden mehrere ressourceneffiziente Methoden in wissenschaftlichen Arbeiten vorgestellt, von denen jede ihre eigenen Vorteile und Grenzen hat. Darunter ist Early Exiting eine Technik, die dynamisch zur Laufzeit bestimmt, wann Berechnungen beendet werden sollten, so dass Vorhersagen gemacht werden können, ohne alle Schichten des LLM für jede Eingabe zu verwenden. In dieser Arbeit wird ein Framework vorgestellt, das Early Exiting durch spezielles fine-tuning von LLMs ermöglicht und das Early Exiting Problem durch Reinforcement Learning (RL) formuliert. Der RL-Agent bestimmt dynamisch, wann er während der Inferenz vorzeitig abbrechen soll, indem er den Trade-off zwischen Genauigkeit und Effizienz lernt.

Dieser Ansatz wurde in zwei modernen LLMs, OPT-2.7B und Llama3.2-3B, prototypisch implementiert. Evaluierungen mit den PY150- und JavaCorpus-Datensätzen mit verschiedenen NLP-, code-spezifischen und effizienzbezogenen Metriken zeigen, dass unsere Methode im Durchschnitt 20-50% Energie einsparen kann, bei geringen bis moderaten Verlusten in den Genauigkeitsmetriken.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Large language models (LLMs) have become the state-of-the-art in natural language processing (NLP) and natural language understanding (NLU). Beyond applications such as summarization, translation, and text generation, LLMs are increasingly essential in the software engineering domain. Tools such as GitHub Copilot and AmazonQ provide intelligent code completion, assisting millions of developers worldwide.

Despite their effectiveness in many domains, LLMs are highly resource-intensive, demanding significant computational resources in terms of both energy consumption and latency for training, deployment, and practical usage. While the resource inefficiency of LLM pre-training has been widely studied, less attention has been paid to the inefficiencies during inference. However, inference becomes more resource-intensive over time due to the high number of continuous user requests in real time. This is particularly true for code completion tasks, where suggestions are often triggered after each change to the file.

Therefore, improving the resource efficiency of code completion during inference is essential to enhance sustainability, provided that the methods maintain acceptable levels of accuracy. Several resource-efficient methods have been proposed, each with its own advantages and limitations. Among them, early exiting is a technique that dynamically determines, at runtime, when to stop computations, allowing predictions to be made without utilizing all layers of the LLM for every input. This thesis introduces a framework that enables early exiting through specialized fine-tuning of LLMs and formulates the early exiting problem through reinforcement learning (RL). The RL agent dynamically determines when to exit early during inference by learning the trade-off between accuracy and efficiency.

This approach has been implemented with a prototype system using two state-of-the-art LLMs, OPT-2.7B and Llama3.2-3B. We carried out extensive experiments. Our evaluations on the PY150 and JavaCorpus datasets in various NLP, code specific, and efficiency-related metrics show that our method can save 20-50% energy on average, with small to moderate accuracy losses.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 1

Introduction

1.1 Motivation

Large Language Models (LLMs) are rapidly becoming an essential part of everyday life, with tools such as ChatGPT [56] serving hundreds of millions of users every month [11], illustrating the vast potential of these technologies across multiple domains. Almost every industry is now using generative AI to improve efficiency, from customer support over development to creative content creation.

Since the introduction of the Transformer architecture in 2017 [85], LLMs have redefined the state of the art in natural language processing (NLP) and natural language understanding (NLU), outperforming traditional algorithms such as recurrent neural networks (RNNs) and long-short-term memory (LSTM). They are used for tasks such as text translation [45], summarization [8], and context-based text generation, among others.

Beyond natural language processing, LLMs are also becoming increasingly popular in software development by assisting in tasks such as code completion, modification, bug fixing, or translation into other programming languages. Tools such as GitHub Copilot [28] and AmazonQ [6] are already used by many software engineers, facilitating their workflows and automating a lot of tasks with high precision [95]. GitHub Research reports that Copilot not only speeds up development by smartly suggesting code but also frees engineers to focus on more creative and complex problems, reducing the time spent on routine tasks such as boilerplate code or searching for solutions to small problems [42]. With more than 1.3 million paying users at the beginning of 2024, according to Microsoft's earnings announcement [55], Copilot proves the impact LLMs already have on the software development sector.

Although LLMs have achieved impressive performance in a wide range of tasks, a significant drawback is their high resource consumption. Since the introduction of transformer

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

architectures by Vaswani et al. [85], the trend in LLM development has primarily focused on increasing the number of parameters to enhance the model capabilities and improve predictions by collecting increasing training data. This often led to models that outperform smaller models, i.e., to some extent, the size of a model also indicates its performance. Earlier models like BERT [19] and GPT-2 [62] contained hundreds of millions of parameters, but the size of these models has increased in recent years. Today's best-performing models, such as the larger versions of LLaMA 3 [39] and GPT-4 [57], have hundreds of billions of parameters. Some studies [43] also suggest that this upward trend in parameter count is likely to continue, leading to even more computing resource requirements and, thus, resource consumption for those models.

Due to the immense number of parameters in these models, increasingly powerful and energy-hungry GPUs are required to train and run them. For instance, a large LLaMA version with 405 billion parameters requires over 200 GB of VRAM, meaning multiple TPUs are necessary for execution. Numerous studies have explored energy consumption's (in)efficiency in deep neural networks, including LLMs. For example, training GPT-3, a model with 175 billion parameters, consumed approximately 1,287 MWh of energy [51], while MetaAI reported that training LLaMA 3.1 resulted in approximately 11,400 tons of $CO2eq$ emissions [54].

The resource-intensive nature of LLM training is often highlighted. At the same time, the energy demands of inference and the process of using the models in a deployed setting after pre-training are less frequently discussed in detail. However, inference is, after some usage, more resource-hungry than training since model pre-training is typically done only once, while inference is done continuously. De Vries [16] estimates that GPT-3 consumes 564 MWh daily during inference, using as much energy in 2.3 days as its entire pre-training process (1,287 MWh). Google reports [60] that $\frac{3}{5}$ of its use of machine learning energy is for inference, while Facebook AI states [88] that 70% of its hardware is dedicated to inference. For the code completion task, this efficiency challenge is critical. Code generation is especially a computationally demanding application of LLMs since tools that rely on code generation models, such as Copilot, are used continuously during development, e.g., suggesting code completions after every change in a file. Therefore, efficient methods tailored to this domain are much needed.

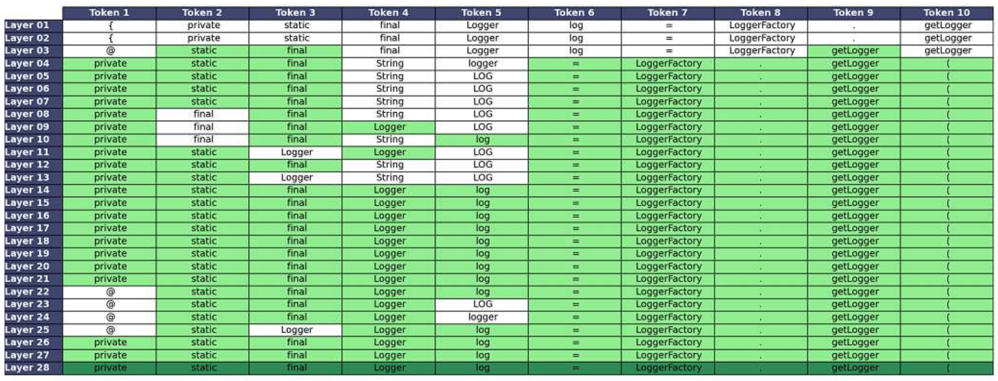
Various methodologies have been investigated to make inference more efficient, including techniques such as quantization [47, 98, 18, 48], knowledge distillation [40, 91, 79, 14, 92], and pruning [52, 26, 13, 53, 86]. Quantization reduces the precision of model weights, lowering memory and computational costs, while knowledge distillation transfers knowledge from a large model to a smaller one, maintaining performance while reducing size. Pruning removes unnecessary connections or weights in the model to make it lighter. Another promising approach is **early exiting**, where not all layers of the transformer are used; instead, the model exits at specific layers based on smart decisions, reducing computation by skipping deeper layers.

Furthermore, while all the previously mentioned efficient inference methods offer unique advantages and disadvantages, this thesis will focus on the *early exiting* method. Early

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

exiting, unlike other methods, can be done in a fully dynamic manner during runtime. This dynamic behavior allows the decision of when to exit early to be made in real-time; for example, early exit can be based on a local scoring function [84, 68, 20, 89] or other criteria to decide when to exit while processing a sample. In contrast, techniques like distillation, pruning, and quantization typically involve pre-processing steps before inference and remain fixed during runtime, lacking the adaptability to dynamically adjust to varying computational or performance requirements. Therefore, we aim to adopt early exiting as the approach in this thesis.

We can examine Figure 1.1 to further illustrate the potential for early exit. This figure shows the outputs from the intermediate and final layers of a 3B-parameter Llama 3.2 model, fine-tuned with aggregated loss (as introduced later in this thesis) and generating predictions through a single LM head. The model was given a context with standard `import` statements and the start of a Spring Boot endpoint class.



	Token 1	Token 2	Token 3	Token 4	Token 5	Token 6	Token 7	Token 8	Token 9	Token 10
Layer 01	{	private	static	final	Logger	log	=	LoggerFactory	.	getLogger
Layer 02	{	private	static	final	Logger	log	=	LoggerFactory	.	getLogger
Layer 03	@	static	final	final	Logger	log	=	LoggerFactory	getLogger	getLogger
Layer 04	private	static	final	String	logger	=	LoggerFactory	.	getLogger	(
Layer 05	private	static	final	String	LOG	=	LoggerFactory	.	getLogger	(
Layer 06	private	static	final	String	LOG	=	LoggerFactory	.	getLogger	(
Layer 07	private	static	final	String	LOG	=	LoggerFactory	.	getLogger	(
Layer 08	private	final	final	String	LOG	=	LoggerFactory	.	getLogger	(
Layer 09	private	final	final	Logger	LOG	=	LoggerFactory	.	getLogger	(
Layer 10	private	final	final	String	log	=	LoggerFactory	.	getLogger	(
Layer 11	private	static	Logger	Logger	LOG	=	LoggerFactory	.	getLogger	(
Layer 12	private	static	final	String	LOG	=	LoggerFactory	.	getLogger	(
Layer 13	private	static	Logger	String	LOG	=	LoggerFactory	.	getLogger	(
Layer 14	private	static	final	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 15	private	static	final	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 16	private	static	final	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 17	private	static	final	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 18	private	static	final	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 19	private	static	final	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 20	private	static	final	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 21	private	static	final	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 22	@	static	final	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 23	@	static	final	Logger	LOG	=	LoggerFactory	.	getLogger	(
Layer 24	@	static	final	Logger	logger	=	LoggerFactory	.	getLogger	(
Layer 25	@	static	Logger	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 26	private	static	final	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 27	private	static	final	Logger	log	=	LoggerFactory	.	getLogger	(
Layer 28	private	static	final	Logger	log	=	LoggerFactory	.	getLogger	(

Figure 1.1: Example of Llama3.2 3B (fine-tuned with aggregated loss) generating 10 tokens, outputs of intermediate layer decoding. Green cells mean that the output of Layer l is aligned with the output of the last layer.

The figure demonstrates that even shallow layers can predict many tokens correctly (light green in the table means that layer l has predicted the same token as the last layer). For example, layer 14 produces the entire sequence identically to the final layer. Furthermore, shallow layers, such as layer 4, still achieve high accuracy, correctly predicting 8 out of 10 tokens. Additionally, token 5, which represents the `Logger` variable name, is challenging for any model to predict consistently. Layer 4, for example, predicts `logger`, a plausible choice that, while different from the final layer's prediction (`log`), must not be considered incorrect.

Although this example is relatively straightforward, it effectively highlights the potential of early exiting in code completion with LLMs, as correct predictions can often be made from shallow layers. However, determining the optimal layer to exit during runtime

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

remains challenging. As the figure exemplifies, there is no linear relationship between the number of layers and prediction accuracy. Although many tokens can be accurately predicted using very early layers, some predictions still require processing through deeper layers, underscoring the complexity of making dynamic exit decisions.

This work aims to address several challenges and research gaps in early exiting for LLMs. First, while many studies focus on encoder-only architectures [20, 89, 71, 72], such as BERT, there is limited exploration of decoder-only models like GPT, LLaMA, and Vicuna, which dominate modern NLP applications and differ in architecture. This work prioritizes decoder-only architectures, given their prevalence and relevance in state-of-the-art autoregressive tasks. Second, early exiting techniques have been predominantly evaluated in general NLP tasks, with little focus on code-related tasks, despite the critical importance of efficient code completion in real-world software engineering. Third, determining optimal exit points remains a significant challenge, as existing metric-based [84, 68, 20, 89] or classifier-based [68, 73, 72, 90] methods often lack adaptability to contextual changes. To address this, reinforcement learning is proposed as an alternative offering greater flexibility and optimization potential. Fourth, while additional language model heads have been widely used for early exits [20, 68, 90, 73], they substantially increase model complexity; alternative methods, such as leveraging specialized loss functions [84] for intermediate layers, warrant further exploration. Finally, most studies focus on the number of layers skipped as an efficiency metric, neglecting energy consumption, which is critical for real-world applications and cost estimation. This work emphasizes energy consumption as a key metric, in conjunction with traditional measures, to provide a more comprehensive evaluation of efficiency.

1.2 Problem Statement

LLM-based code completion is inherently energy-intensive because it needs to perform on-the-fly completions in response to frequent file changes, requiring repeated and fast inference across large models. This constant demand for real-time predictions significantly impacts computational resources, especially in professional development environments where responsiveness and costs are critical. At the same time, losing too much accuracy negatively affects the developer experience. Developers rely on precise and contextually relevant suggestions to maintain productivity and flow. Inaccurate or inconsistent completions can lead to increased manual corrections and reduced trust in the tool.

Thus, deciding on the optimal early exit layer in such scenarios is particularly challenging. Different tokens in the same sequence require varying model depth levels for accurate sequence predictions. Finding a balance between exiting early to save resources and continuing to deeper layers for complex predictions demands a smart & dynamic approach that can dynamically adapt to the token and context.

Therefore, the research objective of this thesis is the following:

"To develop an efficient early exiting mechanism for LLM-based code generation that

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

minimizes computational costs (in terms of energy consumption and latency) without significantly impacting the model performance."

We articulate this research objective with the following research questions (RQs).

1.3 Research Questions

RQ1: *What factors influence the trade-off between model performance and efficiency in LLM inference with early exiting techniques about the code generation task?*

We aim to address this research question by analyzing how various factors, such as inputs, number of layers, and context, affect the inference quality of an LLM. The goal is to identify key considerations that will guide the development of methods to address the other *RQs*.

RQ2: *What are suitable methods to determine early exit points in LLM inference, considering the trade-off between performance and efficiency?*

With this research question, we aim to analyze recent studies on early exiting in large language models, providing an overview of state-of-the-art techniques and methodologies. Additionally, we seek to identify gaps in the research or underexplored techniques that could inform the development of a novel approach for our early exiting strategy.

RQ3: *How can we incorporate and implement the early exiting methods in the inference pipeline to optimize the trade-off between performance and efficiency of the code generation task?*

After gaining an understanding of the influencing factors (*RQ1*) and reviewing current state-of-the-art techniques (*RQ2*), we then aim to design & implement our methodology within the inference pipeline of modern autoregressive LLMs. This involves specialized fine-tuning and the training of RL agents to optimize performance for early exiting.

RQ4: *How can we evaluate our proposed method, and how can its efficacy be measured and validated against baseline methods?*

Finally, we address the evaluation of our methodology to assess our approach based on (i) accuracy in code completion and (ii) efficiency metrics, such as energy consumption. Our method will be compared against baseline models, primarily showing the impact compared to the full model to benchmark improvements in resource usage while trying to minimize accuracy deductions.

1.4 Thesis Structure

The structure of this thesis is as follows: Chapter 2 lays out the fundamental concepts necessary for this thesis, including an introduction to LLMs and a summary of efficient

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

inference methods for LLMs. Chapter 3 delves deeper into these topics by exploring related work relevant to this thesis. Chapter 4 provides a detailed explanation of our approach, while Chapter 5 elaborates on applying reinforcement learning to address the early exit problem. Chapter 6 presents our experimental findings and results. Lastly, Chapter 7 summarizes the thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Background

This Chapter will introduce the concepts and techniques necessary as background for this thesis.

We provide the technical background of LLMs, focusing on the transformer architecture. This section covers concepts such as self-attention and briefly discusses various types of LLM, including encoder-based and decoder-based models. We then give an overview of how LLMs are applied in software engineering tasks, such as code completion.

We then examine the sustainability concerns associated with LLMs and discuss their environmental impact due to the significant energy required to train and deploy these models.

Next, we discuss post-training optimization techniques and efficient LLM inference methods, including pruning, early exiting, and quantization. The following chapter will detail these optimization strategies, focusing on more specific methods.

Lastly, given that this thesis uses reinforcement learning, we introduce RL's basic concepts and notation, such as Markov decision processes, rewards, and environments. We also introduce the Proximal Policy Optimization (PPO) algorithm, a popular RL approach used in the method proposed in this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

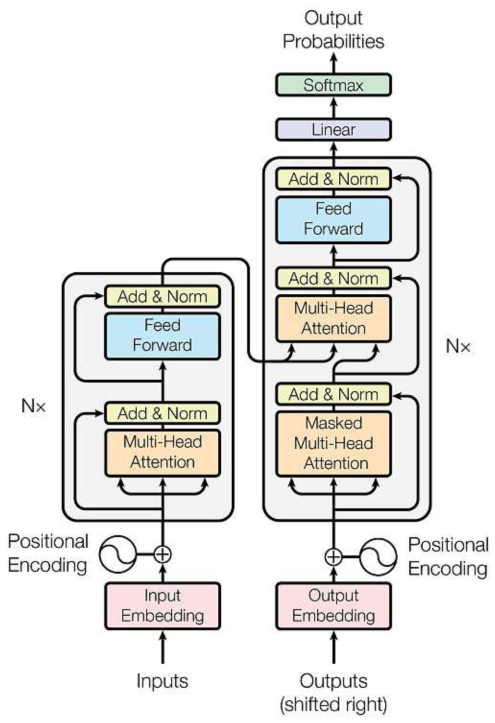


Figure 2.1: Transformer model architecture [85].

2.1 Large Language Models

In 2017, Vaswani et al. [85] from Google Research introduced the paper "Attention Is All You Need," which presented the transformer architecture. This architecture forms the backbone of modern LLMs like GPT and LLaMA.

Figure 2.1 illustrates the transformer architecture. This architecture consists of an *encoder* and a *decoder* stack. In simple wording, the encoder processes the input sequence while the decoder generates the output. RNNs were used in similar settings but suffered from a critical bottleneck [82]. Specifically, the encoder in RNNs could only access the final hidden state of the encoder, which meant that this single state had to encapsulate the meaning of the entire input sequence, leading to a loss of detail for long sequences [82].

This limitation has been overcome by introducing the *attention mechanism*, where the encoder stack generates a series of hidden states for each input token, which are then selectively attended to by the decoder. This attention mechanism assigns varying weights to the hidden states, effectively deciding which parts of the input sequence are most relevant at any given time step. The model can capture complex relationships between input and output tokens by learning these weights, allowing it to align words more effectively across languages or tasks [82].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

One limitation of traditional attention mechanisms was that they operated sequentially, which hindered parallelization. The transformer addressed this by introducing *self-attention*. This mechanism computes the attention scores for all tokens in the input sequence rather than sequentially, as in RNN architectures. This shift enabled much greater efficiency and scalability.

In language models, tokens (e.g., words or characters) are typically represented as *encodings*, which map tokens to unique numerical identifiers. These encodings are then transformed into *embeddings*, vector representations in a lower-dimensional space. The key innovation of self-attention lies in computing a weighted average of these embeddings. For a given input sequence $x_1, \ldots, x_n$, self-attention generates a new sequence of embeddings $x'_1, \ldots, x'_n$, where each embedding is calculated as:

$$x'_i = \sum_{j=1}^{n} w_{ji} x_j$$

Here, w_{ji} represents the attention weight between tokens x_j and x_i, which are normalized to ensure they sum to one. This mechanism allows the model to dynamically focus on different parts of the input sequence at each layer, capturing dependencies between tokens, regardless of their distance.

There are many ways to implement self-attention in an architecture. Vaswani et al. [85] used the so-called scaled-dot-product attention. The token embeddings are projected onto three vectors, called *query, key* and *value*. The query and key vector are then compared using a dot product to determine how much they relate to each other. This is called attention score. Typically, for n input tokens, this step yields a $n \times n$ attention matrix. This matrix is then normalized using a softmax function so that the columns of the matrix sum up to one, which gives the attention weights w_{ji}. These weights are then multiplied with the value vector to obtain an updated embedding x'_i. So, to summarize, scaled-dot-product attention can be formulated as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \quad [85]$$

In practical settings, projections of self-attention are made multiple times, using the so-called *multiheaded attention*. The results of the attention heads are then concatenated to yield the resulting attention. This is done so that the model can simultaneously focus on several aspects of similarity. In addition, the attention layer output is also fed through a normalization layer and passed through a feedback network, which outputs the *hidden state*. Often, FFNs in LLMs are called *position-wise feed forward*, since instead of processing the whole sequence, it processes each embedding alone. Finally, the output is put through a normalization layer to have zero mean and unit variance. The process of applying self-attention followed by normalization and a feedforward network explains the functionality of an *encoder* layer [82, 85], as depicted on the left side of Figure 2.1.

In contrast, the decoder layers function slightly differently, generating new tokens. The attention mechanism in the decoder employs *masked* attention, ensuring that the current

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

token being generated is only influenced by previously generated tokens. This means the prediction of each token depends solely on the preceding known outputs, while all future tokens are masked to prevent the model from "looking ahead."

Additionally, the decoder includes a multi-head attention layer that attends to the output of the encoder stack. This lets the decoder focus on relevant parts of the input sequence while generating the output. Finally, the output of the transformer is mainly forwarded into a fully connected linear layer, known as *language model head* (LM head), that predicts the token out of the final hidden state of the model [82].

In summary, the encoder stack is responsible for encoding the input into hidden representations by capturing relationships and context, while the decoder predicts tokens sequentially based on those representations.

This outlines the basic functionality of modern LLMs. However, since the release of the transformer, many models have evolved, and the traditional encoder-decoder architecture has often been split. New models tend to focus on either the encoder (for example, BERT and its variants) or the decoder (e.g., GPT, OPT, or LLaMA), each tailored for different use cases and strengths [82].

2.1.1 LLMs for Code & Software Engineering

Model	**Type**	**Example of SE tasks**
Encoder-only	Understanding	Code understanding Bug localization Vulnerability detection
Encoder-decoder	Understanding and generation	Code summarization Code translation Program repair
Decoder-only	Generation	Code generation Code completion Test case generation

Table 2.1: Summary of LLMs used as support for software engineers based on Hou et al. [34].

In their survey on LLMs for software engineering, Hou et al.[34] categorize models based on their architecture, as summarized in Table 2.1. Since they recognized, similar to natural language-related tasks, that specific architectures of LLMs decide their primary usage.
Encoder-based models, such as BERT and its variants, utilize a bidirectional context. Adaptations of BERT, like CodeBERT and BERTOverflow, are primarily used for tasks involving code understanding, such as code review or bug localization. These tasks typically require full context from both directions to make accurate predictions, making

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

encoder models well-suited for software engineering tasks at the beginning of the software lifecycle - like reviews, bug reporting, or named entity recognition (related to code) [34].

In contrast, decoder-only models, such as GPT, generate output tokens using masked attention, focusing solely on the "left" side of the context (past tokens). This makes them ideal for tasks involving code generation and completion. Fine-tuned versions of decoder-only models, such as GPT and Google's Gemini, have gained significant momentum as commercial products, excelling in code-related tasks [34].

Lastly, encoder-decoder models combine both components, making them well-suited for tasks like code translation (e.g., from one programming language to another). Similar tasks to translation are corrections, where incorrect code is transformed into corrected or more meaningful code [34].

Hou et al. [34] further state that for code-related tasks, in the last years (2023-2024), mainly decoder-only architectures have been widely explored in research. Furthermore, they state that more than 70 LLM architectures are currently used for SE tasks.

Further applications in software engineering settings are, for example, upgrading to higher language levels (i.e., migrating from Java 8 to Java 17) or modernizing from legacy code - as AmazonQ reports on their landing page [1].

In addition to code-related tasks such as code completion and translation, LLMs have proven valuable throughout the software development lifecycle [36, 25]. For example, in requirements engineering or testing and documentation, LLMs can help SEs in their daily lives.

Prominent tools such as GitHub Copilot [28] have become widely used in software development. According to a report by GitHub [42], experiments were carried out in which participants were divided into two groups: one group used Copilot, and the other did not. The study found that the Copilot group had a higher task success rate, completing their tasks in about half the time. A survey was also conducted to assess the impact of Copilot on the developers' workflow. Of the 2,000 developers surveyed, around 90% stated they were significantly more productive and completed tasks faster, while 60% reported they felt less frustrated when coding. In addition, almost all developers (96%) said that Copilot helped them complete repetitive tasks faster, highlighting its potential to increase efficiency.

In summary, it is easy to see that LLMs are already widely adapted in many ways related to the field of software engineering and that they are already used in extensive settings, with Copilot being one of the most significant commercial tools, having already more than a million paying users [55]. Therefore, LLMs code completion tasks, which we will investigate in this thesis, are a highly relevant and impactful area.

[1]https://aws.amazon.com/de/q/developer/, last accessed 20-11-2024

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.2 Sustainability and LLMs

Since their introduction in 2017, LLMs have grown exponentially in size and complexity, requiring vast computational resources to train and deploy. Figure 2.2, as assessed by Desislavov et al. [17], illustrates the pattern of energy use by leading NLP models for forward passes over various years, depicted in the Figure as a dashed line. Therefore, modern models such as the LLaMA-3.1 and GPT variants, which contain hundreds of billions of parameters, demand powerful GPU-based accelerators with increasing memory capacities. For example, NVIDIA GPUs have advanced from the 32GB V100 in 2018 to the 80GB A100 by 2021 [88]. Training these huge models comes at a significant environmental cost. The 405-billion-parameter version of LLaMA-3.1 required an estimated 31 million GPU hours. It emitted approximately 8,930 tons of CO2eq [2], comparable to the annual carbon footprint of 900 individuals in Germany, based on data from the German Federal Environment Agency [83]. Strubell et al. [70] highlight that even smaller models, such as BERT, have energy demands comparable to a trans-American flight. Furthermore, Strubell et al. caution that new models often bring minimal gains while incurring substantial energy costs. For example, training neural architecture search (NAS) to improve BLEU scores by just 0.1 required enormous computational resources. Schwartz et al. [69] highlight the contrast between Red AI, which prioritizes accuracy through increased computational resources, and Green AI, which seeks to balance strong performance with resource efficiency, whereas 80-90% of top studies are in the Red AI field.

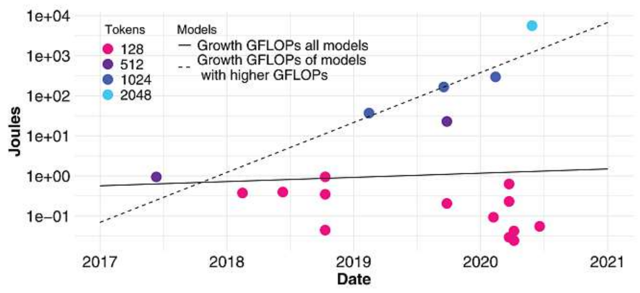


Figure 2.2: Estimated Joules of an NLP forward pass over years with the logarithmic y-axis. From Desislavov et al. [17]

Although the energy-intensive nature of LLM training is well documented, the challenges do not end there. Deploying LLMs for task-specific applications often requires retraining or fine-tuning, further contributing to resource consumption. Alex de Vries [16] estimates that OpenAI's inference infrastructure requires 564 MWh per day, exceeding the training energy consumption of GPT-3 after only 2.3 days of deployment. Moreover, individual inference requests are energy-intensive. For example, a single ChatGPT request in 2023 consumed approximately 2.9 Wh, almost ten times the energy of a Google search. As

[2]https://huggingface.co/meta-llama/Llama-3.1-405B, accessed 8-10-24

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

future applications such as AI-powered search tools emerge, this gap is expected to grow, potentially reaching 9 Wh per request [16]. Reports from Google [60] and Facebook AI [88] indicate that up to 70% of their hardware resources are allocated to inference, underscoring its growing importance as a sustainability challenge in the deployment of LLM.

Inference, the process of using trained models to generate outputs, poses unique challenges that often exceed those of training. Unlike training, inference is a real-time process that must handle a wide-scale deployment for millions of users with varying demands and does this continuously. This creates a dynamic need for high-quality service (QoS), where latency, cost, and accuracy are critical. Inference is also a sustained cost, as opposed to the one-time cost of training. This is particularly relevant for the code generation task, as the LLM is typically prompted with each file modification, such as when a variable is renamed or when a new line's beginning is written. LLM-powered code generation tools, such as GitHub Copilot, are widely used, indicating a significant environmental footprint. According to GitHub Customer Research Blog [27], developers at major corporations, such as Accenture, engage with Copilot about 3.4 days a week. This tool has over a million subscribers [55] and has been used by 50,000 companies, showing the impact it already has.

2.3 Post Training Optimization Methods

In the following section, we give an overview of *post-training optimization methods*, that is, methods that can be applied after the pre-training of a LLM. Here, we focus on those methods that aim to achieve more resource-efficient inference.

A widely used method to improve the sustainability of LLM is *knowledge distillation*. This technique can be applied as a specialized fine-tuning process or during the full training of a model [82]. Originally introduced for *ensemble methods* [10], knowledge distillation gained significant traction in the neural network community after the work of Hinton et al. [33].
In knowledge distillation, a smaller model called the *student* is trained to replicate the behavior of a larger and more complex model called the *teacher*. This allows the student model to learn from the teacher's output in addition to the standard training data and, therefore, aims to reproduce the teacher's outputs. The distillation process typically augments the ground-truth labels with a "soft" probability distribution that conveys information about the teacher model predictions. Learning from this output, the student model captures not only the target labels but also additional insights from the larger architecture of the teacher [82].
There are several approaches to implementing knowledge distillation. A common strategy is to train the student model with these soft probabilities, allowing it to gain more information than would be available from the labels alone. While some student models are fine-tuned through this process, others are entirely pre-trained through distillation from scratch. Although significantly smaller than their teacher counterparts, the distilled

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

models have shown impressive performance in various tasks, often achieving competitive results while reducing model parameters and resource requirements a lot [82].

Another widely used optimization method for reducing model size is called *quantization*. Unlike knowledge distillation, which transfers knowledge between models, quantization focuses on reducing the precision of a model's weights and activations to reduce its size since weights with smaller precision require less memory. For example, instead of using 32-bit floating point representations, quantization converts them to 8-bit integers [82], thus saving 24 bits per weight, which can be quite a lot given that modern models have millions to hundreds of billions of such weights. Therefore, this reduction in precision results in a smaller model footprint, which requires less memory and allows faster computation on modern hardware, especially for operations such as matrix multiplication. Despite reduced precision, quantized models often perform similarly to their full-precision counterparts.

From a mathematical perspective, quantization maps the floating-point values, initially in a range from f_{min} to f_{max} to a fixed precision range from q_{min} to q_{max}. These values are linearly distributed throughout the range, allowing a representation with fewer bits [82]. Neural networks work well with this approach because their weights and activations fall within small ranges. As a result, not all possible values from a 32-bit floating point range need to be represented in an 8-bit integer space ($2^8 = 256$ possible values) [82].

Quantization techniques are often divided into three subcategories [82]:

1. *Dynamic quantization*: The model weights are converted just before inference in this method. It is named *dynamic* because the quantization occurs on the fly, with matrix multiplications performed using optimized INT8 operations. However, activations are still stored and processed at floating points, which can create a bottleneck, as frequent conversions between formats slow down computation.

2. *Static quantization*: Unlike dynamic quantization, static quantization precomputes the quantization scheme by observing a representative sample of the data before inference. This eliminates the need for on-the-fly conversions between floating- and fixed-point numbers, leading to faster execution and less overhead during runtime.

3. *Quantization-aware training (QAT)*: The model is trained with simulated quantization effects in QAT. Floating point values are *faked* during training by rounding them to mimic the lower precision of quantized values. This approach tends to bring better results than dynamic or static quantization, as the model learns to adapt to the reduced precision during training, resulting in improved accuracy during inference.

Another common approach to decrease the size of LLM is *pruning*. Like quantization and distillation, pruning aims to compress a model by decreasing its size, specifically by eliminating weights or connections within the neural network. The goal is to remove

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

less important parameters to the model's overall performance, resulting in a more sparse model [82] with fewer weights and, therefore, can reduce resource consumption.
Pruning can be performed in multiple ways, one of the most common being *weight pruning*, where the k most important weights are retained and the least important ones are removed. The challenge lies in determining which weights are non-essential and how many can be pruned without significantly impacting performance. A popular strategy is *magnitude pruning*, where the weights with the smallest magnitudes (absolute values) are first pruned, assuming that they contribute the least to the prediction of the model. Other advanced methods include *movement pruning*, in which the weights are gradually pruned during the fine-tuning process. As training progresses, the model becomes progressively sparser, allowing for more efficient computation while aiming to preserve performance. This dynamic approach to pruning enables the model to adapt as the weights are pruned, making it a popular choice for fine-tuning large models [82].

Further, model-level optimizations focus on improving the efficiency of the structure or processing of the model. This includes optimizing feedforward networks (FFNs) or attention layers to reduce computational costs. For example, a mixture of experts' methods dynamically allocates different computational resources to various input tokens according to their needs [104]. A specialized form of model-level optimization that is dynamic is known as *Early Exiting*. Early exiting allows an LLM to skip computations based on the input and the model's current state, bypassing entire layers, thus saving computations - in the optimal case, without sacrificing performance. Although techniques like mixture-of-experts require extensive training, early exiting typically involves only some modifications, such as training small additional components to determine when to exit. This often requires attaching additional language model heads to enable intermediate layer decoding, as the original LM head cannot predict from intermediate hidden states. There are various strategies for when to exit, such as at the token level or the sample level [104], which will be discussed in detail in the next Chapter.

In addition to methods that focus on reducing the model size, many more methods aim to enhance the inference efficiency in LLMs. One broader category is data-level optimization, which alters the input or output rather than modifying the model. This includes input compression, where the input prompt is compressed or summarized so that the model processes only a smaller portion of it [104]. Another technique, output organization, involves parallelizing parts of output generation to accelerate the sequential generation of tokens [104].

Finally, system-level optimizations focus on improving the surrounding infrastructure rather than the model or inference process. These techniques include offloading tasks, caching, and optimizations such as graph optimization or efficient task scheduling.

In Table 2.2, we summarize the optimization methods discussed here, categorizing them based on their applicability during the runtime. Each technique presents specific advantages and limitations, depending on the use case and computational requirements. Although all approaches aim to reduce latency and computational overhead, their performance varies across different scenarios.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Optimization Type	**Behavior at Runtime**	**Phase**	**Efficency Gain**	**Overhead**
Knowledge Distillation	Static	Fine-tuning & Pre-training	Smaller model	Learning student
Quantization	Static & Dynamic	All phases, depending on type	Model in lower precision	Quantizizing weights
Pruning	Static	Pre-training, fine-tuning	Smaller model	Remove weights, layers...
Early Exiting	Dynamic	Inference	Explicit by adapting inference at RT	Depending on method: train additional components etc.
Data-Level Optimization	Dynamic	Inference	Implicit (e.g. smaller prompts)	Depending on method

Table 2.2: Summary of Efficient LLM Methods.

Given the dynamic nature of early exiting, which adapts the inference process by predicting the exit points for tokens during runtime, we chose this method as the optimization technique throughout this thesis. Early exiting not only allows for a flexible trade-off between accuracy and efficiency, which other methods often can not, but is also applicable to most model architectures.

2.4 Reinforcement Learning

After discussing post-training optimization methods, such as early exiting, we introduce the fundamental concepts of reinforcement learning (RL). This sets the stage for understanding how the RL-based approach presented in this thesis can be leveraged as an alternative framework for early exiting, offering a more dynamic and adaptive approach than traditional methods.

Reinforcement learning is often considered the third pillar of machine learning discipline, alongside supervised and unsupervised learning. RL is used to model complex environments and optimize processes. By modeling accurate state, action, reward, and environment dynamics, agents are enabled to learn strategies for decision-making and aim to achieve optimal outcomes in diverse applications.

In reinforcement learning, an agent learns based on the *feedback* it receives given its *action* on an *environment* (or on a model of the environment). The goal of reinforcement learning is to obtain rewards based on an observation to learn a policy that is as optimal as possible. In RL, the agent typically knows neither the reward function nor the concrete environment; it learns by interacting with the environment and receiving rewards. Any method suitable for solving such problems can be considered an RL method [74]. Figure 2.3 illustrates the interaction between a reinforcement learning environment and an agent. RL problems are typically framed as *Markov Decision Processes* (MDPs), which provide a mathematical and idealized structure for modeling RL tasks. MDPs offer a simple yet powerful way of framing the problem of learning from interaction to achieve a specific goal. In RL, the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

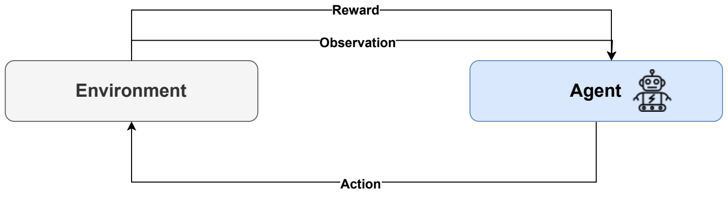


Figure 2.3: RL agent-environment interface, based on [74].

entity trying to solve the problem is referred to as *agent* [74].

The agent interacts with an *environment* by taking actions, and in response, the environment provides feedback in the form of new observations and rewards. Feedback is usually a numerical value (the reward), and over time, the agent's goal is to maximize the cumulative reward it receives. This interaction generates a trajectory of states, actions, and rewards: $S_0, A_0, R_1, S_1, A_1, R_1 \ldots$ where S_i represents the state of the environment at the time step i, A_i the action taken by the agent and R_i the reward received after action A_i [74].

In finite MDPs, the sets of possible states S, actions A, and rewards R are finite, and the probabilities of transitioning between states and receiving rewards are well defined, depending on the preceding states and actions [74]. However, a finite definition of these sets is not always possible.

So, in RL, the main goal is to design an agent that optimizes an unknown reward function through interaction with the environment. For example, a reward for a robot that needs to leave a labyrinth is often -1 for every passing period - so the agent learns to leave the maze as fast as possible [74]. However, in principle, reward functions can be additive complex. Negative rewards are often called *penalties*.

Training and evaluations are often done in RL in *episodes*. While traditional ML uses *epochs*, i.e., passes through a dataset, episodes denote the sequence from a beginning to a final state in the interaction with the environment. In traditional RL settings, an episode can, for example, be a round of a game. The subsequent episode typically does not depend on the last episode. An agent should, therefore, learn to maximize the reward in a given episode; in the most straightforward settings, those can be just the sum of individual rewards [74].

Reinforcement learning encompasses a variety of algorithms and techniques, each with its own strengths and weaknesses. Some are relatively simple, like the Multi-Armed Bandit, while others, such as deep reinforcement learning algorithms like deep-Q-learning or Proximal Policy Optimization (PPO), are more intricate. A common feature among these algorithms is their reliance on *value functions*, which are functions (or algorithms, networks) designed to predict the potential performance of an agent in a particular state by estimating expected future rewards. Moreover, numerous algorithms incorporate policies. A policy is a strategy that maps states to the probabilities of selecting each

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

possible action. For instance, if an agent is following policy π at time t, then $\pi(a|s)$ denotes the probability of the agent taking action a given the state s [74].

A common problem of RL algorithms is known as *Exploration-Exploitation Dilemma*, which describes the trade-off between greedily choosing an action that worked in the past (exploit), but to discover actions that lead to high rewards, an agent needs to explore the search space.

A widely used RL algorithm that has gained significant popularity across various domains is *Proximal Policy Optimization* (PPO), introduced by OpenAI in 2017 [67]. PPO belongs to the family of deep reinforcement learning algorithms. The key innovation of PPO lies in its ability to combine the strengths of *policy gradient methods* and *trust region methods*, offering a more stable and efficient approach to policy optimization. Policy gradient methods work by directly adjusting the policy (a deep neural network) in the direction that maximizes the expected cumulative reward. Although powerful, these methods can sometimes lead to too large updates, destabilizing the learning process [67]. To address this, PPO incorporates ideas from trust region methods, which constrain the size of policy updates to ensure that each new policy remains close to the previous one and stays stable. However, traditional trust region methods, such as Trust Region Policy Optimization (TRPO), can be computationally expensive due to the need for second-order optimization techniques. PPO overcomes this challenge by introducing a *surrogate objective function* that allows using first-order gradient optimization, such as stochastic gradient descent while ensuring that policy updates are not too much. This is achieved using a *clipping mechanism*, which restricts the update to a certain threshold, preventing large deviations from the current policy. This allows for more efficient training while maintaining a balance between exploration and exploitation [67].

In this chapter, we introduced the necessary background for this thesis. We discussed the basic properties of LLMs and their applications in the context of software engineering, such as code generation. Furthermore, we examined the resource footprint that modern LLMs produce and introduced efficient LLM methods at a high level. Finally, we discussed the basic properties of reinforcement learning. In the next chapter, work related to this thesis is discussed, and an overview of current state-of-the-art efficient LLM inference techniques is given.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

Related Work

In this chapter, we discuss efficient LLM inference techniques in detail by presenting studies related to this thesis. This includes early exiting techniques and discusses some studies on pruning, quantization, etc. We conducted a literature study on SOTA of efficient inference methods. In particular, this includes techniques that (i) enable more efficient inference, (ii) studies that discuss contributing factors to inference quality, and papers (iii) that discuss the efficient utilization of LLMs. Our search was conducted in several online libraries while focusing on high-tier studies, namely we searched for papers in the following online libraries:

- Scopus (`https://www.scopus.com`)
- ACM Digital Library (`https://dl.acm.org`)
- IEEE Explore (`https://ieeexplore.ieee.org`)
- arXiV (`https://arxiv.org/`)

Although we did not conduct a complete *structured literature review*, which is beyond the scope of this thesis, we still aim to search for online libraries exhaustively and well-organized. An example search string can be seen in Listing 1. While many studies focus on efficient inference on deep neural networks, we solely focused on methods proposed in the LLM/transformer field.

In the following, we present the papers found through this search.

3.1 Pruning Methods

Gromov et al. [29] present a pruning method where they see a model's hidden states as a function that slowly changes with the layer index. The change between two layers

```
TITLE("efficient*" OR "Resource efficient*") AND  TITLE("Inference") AND
TITLE-ABS-KEY(("LLM" OR "Large Languge Model" OR "Transformer")) AND
LANGUAGE(english) AND (LIMIT-TO(SUBJAREA , "COMP"))
```

Listing 1: Example Search String for literature study.

can be seen as adding a residual to the previous layer, so they formulate the pruning problem through a *residual network*. They hypothesize that pruning layers might be more successful on deeper parts of the model, and pruned layers should have outputs close to their inputs, i.e., pruned layers do not change the hidden state significantly and, therefore, should not affect model capabilities too much. Using this hypothesis, they introduce an algorithm that drops these layers and connects the remaining layers based on the smallest angular distance between a pair of layers $(l, l+n)$. In addition, they *heal* the damage caused by pruning with a bit of additional fine-tuning. The authors evaluated the Llama-2, Mistral, and Qwen model families. They showed that they could prune up to 50% (Llama) and 20-25% (Mistral, Qwen) of layers without losing too much inference quality, but after these thresholds, the models behave somewhat randomly.

The LLM-Pruner, presented by Ma et al. [52], is a structured pruning method that aims to tackle compression in a task-agnostic manner to preserve the multitasking capability of LLMs. The LLM pruner consists of three steps: (1) discovery, the goal is to find structural dependencies in the LLM; (2) estimation, where the contribution of each group to performance is estimated and pruned based on those metrics; and (3) recovery stage, where post training is performed to heal the damage caused by pruning. The importance is calculated using the *Fisher information matrix*. Experiments show that performance can often be maintained; for example, a 20% parameter reduction without post-training can still achieve 89% of the performance of the original model, and with post-training, even 94% can be achieved; this holds for both models evaluated (Llama, Vicuna).

Another approach using pruning is presented by Fan et al. [26]. The authors propose a methodology similar to feature selection in traditional ML algorithms such as SVM. Based on *mutual information*, a measure that quantifies the amount of information from one random variable to another, they prune the most correlating neurons by preserving neurons with higher values of MI. They conducted iterative and greedy pruning experiments on models of different sizes and showed higher speed and performance than traditional weight-based pruning methods.

LLM-Streamline [13] is an approach that prunes layers in LLMs based on redundancy and replaces the removed layers with simpler models. The redundancy is calculated based on cosine similarity by randomly sampling from pre-training data and computing the cosine similarity of the input hidden state and the output after passing through a single layer. Layers are then pruned based on the highest cosine similarity, i.e., layers that do not significantly change the hidden state are considered redundant. After removing the most redundant layers, a simpler network (FFN or transformer layer) is trained based

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

on the removed layers that were part of the original LLM. Thus, the pruned part of the network is replaced by a more lightweight one. They show that they outperform other pruning methods with an accuracy of up to 15% and better stability. Furthermore, they report that performance correlates linearly with the number of parameters of the model.

Similarly, ShortGPT [53] introduces a method that prunes layers based on importance using *block information*, which can be seen to be quite similar to LLM-Streamline since the block information also defines how much a transformer block changes a given hidden state. They delete layers based on this score and evaluate it on Llama-2 and Baichuan-2 models, showing a pruning ratio of 20% without significantly changing the inference quality.

Yang et al. [93] propose a method in which they do not directly prune the layers but merge them based on a few calibration samples so that the model behaves as similar as possible to the original model. Again, they use input-output cosine similarity as a metric and collapse *most similar* layers and then evaluate and adjust the network. If the merge passes an evaluation threshold $\mathcal{T}$, the merge is considered successful, and the model is updated. Although the overall results are good, they report having to prune at a higher rate than other pruning methods.

A further structured pruning method is Shortend LLaMa, proposed by Kim et al. [44], where the goal is again to prune complete transformer blocks. The authors argue that the problem with width pruning is that for scenarios with limited batch size, width pruning can only achieve a small speedup; therefore, depth pruning is a more interesting method. Like the other methods discussed above, they use an importance score to decide what to prune. The metric of choice here is block-level importance based on perplexity and a Taylor series that incorporates the loss. They then perform one-shot pruning, i.e., prune all less important blocks in one step and then heal with some retraining. They evaluated LLaMa and Vicuna models and different pruning ratios. They also compared different retraining scenarios. The results show the same accuracy as many width pruning methods but with a more significant speedup. They also show that even heavily pruned models (e.g., reducing the size of a model by 60%) produce valid and fluent text where other methods struggle to create meaningful output.

The SpAtten framework introduced by Wang et al. [86] follows a different pruning approach. Instead of pruning the network's weights (e.g., layers), SpAtten prunes heads and tokens. They propose cascade pruning; i.e., once a token or head is pruned, it is removed from all remaining layers; therefore, similar to other pruning methods, the deeper the layer, the more it is pruned. *Token pruning* refers to removing tokens because human language is highly redundant and removes irrelevant tokens, such as prepositions, from the input, thus saving DRAM access and computation. Furthermore, the authors prune attention heads based on the impact of the heads on the output. Token pruning is based on accumulated attention scores, and head pruning is based on the absolute value of `attention_out`. They conducted experiments on the BERT and GPT-2 models and reported a maximum speed-up of 2.3 while maintaining moderate performance.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In summary, pruning methods aim to reduce a model's size and computational requirements by removing potentially unnecessary components, such as individual weights, neurons, or even entire layers, from a LLM. Therefore, it relies on the hypothesis that a model has redundant weights. Various pruning techniques have been explored [105], each with its strategies to identify and eliminate redundancies. Despite their differences, these methods share a key limitation: pruning is typically non-recoverable, meaning that any accuracy loss incurred after pruning cannot be fully restored. Further, pruning often needs additional fine-tuning. However, pruning methods usually achieve significant efficiency gains, such as reduced memory usage and faster inference speeds without introducing any overhead at runtime, while maintaining high levels of accuracy in many scenarios.

3.2 Knowledge Distillation

TinyBERT, introduced by Jiao et al. [40], is a model compression technique designed for BERT models and was one of the first studies on knowledge distillation for LLMs. The distillation process has two stages: general distillation, where TinyBERT learns from the teacher model during pre-training, and task-specific distillation, which is focused on fine-tuning for specific tasks. TinyBERT leverages attention-based and hidden state distillation to efficiently compress the model while maintaining performance.

WizardLM [91] uses KD to train a Llama 7B model to mimic the behavior of ChatGPT, which they call *WizardLM*. Using the ChatGPT API, the authors constructed a dataset of 250K samples, which they used to fine-tune the Llama model. In some cases, WizardLM outperformed ChatGPT, while in others, it lost slightly. While this paper was not particularly interested in efficient inference, it is interesting because it shows how rather small models like Llama 7B can still perform similarly to much larger models like GPT4 by mimicking their input, thus saving on inference costs.

Tian et al. [79] introduce TinyLLM, which can be seen as a different distillation paradigm, as it learns its LLM from multiple teacher models. They report that existing methods suffer from limited knowledge diversity and lack of rich contextual information - because these approaches use only a single-teacher model. Therefore, the authors propose a model that learns from multiple teacher models, which they call co-advisors. They report superior performance to single-teacher distillation methods.

The mixed distillation approach introduced by Li et al. [14] combines the methods of *chain of thoughts* (CoT) and *program of thoughts* (PoT). CoT refers to teaching LLMs to reason step-by-step so that, for example, complex numerical problems can be solved. PoT is mainly used for code-related tasks and often transfers the computation to a program interpreter that can execute the generated program. They argue that most distillation methods focus only on CoT and ignore essential properties of the PoT prompts. Therefore, their method presents mixed distillation, which combines both ideas by distilling the reasoning paths of CoT and PoT from LLMs into smaller models. Their mixed approach outperforms single-distilled models, focusing only on CoT or PoT.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Knowledge distillation is a growing field of research [92] focused on reducing resource consumption by training smaller student models to replicate the behavior of larger teacher models. Distilled models excel in cases where the student only needs to learn specific aspects of the teacher's knowledge or imitate the behavior of closed-source models with inaccessible weights. However, these smaller models have limitations, as they may struggle to fully replicate the teacher's performance and, therefore, have problems similar to pruning methods.

3.3 Quantization

Lin et al. [47] present the Activation-Aware Weight Quantization (AWQ) method. This approach focuses on protecting the most important weights instead of quantizing every weight. The authors state that the weights in LLMs are not equally important, instead there are a few weights in an LLM, which they call *salient*, that have more impact on the performance of the model. They calculate the importance of the weight using the L_2 norm. Further, they state that weights with high activation magnitude seem more critical. Based on this, they kept between 0.1% and 1 % of the weights in FP16, while others were quantized as integers. Their method is a quantization-aware training method. The method is compared with several other quantization methods and outperforms them in terms of perplexity in most settings. They also show performance similar to that of the full precision baselines.

Although most quantization methods use integers of 8 or more bits, there are more extreme examples, such as binary quantization. For instance, PB-LLM [99] uses partial binary quantization. The authors freeze part of the weights while binarizing the other weights using an optimal scaling factor. In forward passes, the values are binarized using the *sign* function - while backward propagation is much more complex due to the non-differentiable property of the sign function and destroys the gradient chain. So, they use the straight-through estimator, which approximates the derivative of the whole binarized neural network. They also use magnitudes to compute salient weights but mention that other criteria give similar results. The authors use post-training quantization and show with OPT-1.3B that even without fine-tuning, the model loses only a small language capacity with 50% binarized weights. Other experiments suggest that around 30-50% salient weights seem to be a good compromise, where much of the original LLM capacity can be retained.

One of the most prominent LLM quantization techniques is QLoRA, introduced by Dettmers et al. [18]. First, they introduce a new data format, a *4-bit NormalFloat*, which can be seen as an information-theoretically optimal type that ensures that each quantization bin is assigned an equal number of values from the input tensor. Furthermore, they use *double-quantization*, i.e., they quantize the quantized constants, which leads to additional memory savings. Through these innovations, they introduce QLoRA, a fine-tuning approach that significantly reduces model size. For example, QLoRA reduces memory so that a 65 billion parameter model can be fine-tuned on a 48 GB GPU,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

requiring approximately 780 GB if the full 16-bit floating-point numbers were used. Using their approach, the authors state that their 4-bit normal float fine-tuning achieves similar results to full 16-bit and "normal" 16-bit LoRA fine-tuning on academic benchmarks. Furthermore, the 4-bit NF outperforms standard 4-bit quantization, and there is no degradation in performance due to double quantization. The approach using LoRA is called parameter-efficient fine-tuning (PEFT), which retrains only a subset of the original model weights - these subsets are significantly smaller than retraining all the parameters [35].

The QServe framework [48] takes a different approach, arguing that SOTA of 4-bit integer quantization fails in performance for large-batch cloud inference while performing well in low-batch edge environments. This is because INT4 quantization can lead to significant run-time overheads of up to 90%. They present a method to overcome this using 4- and 8-bit quantization: 4-bit weights, 8-bit activations, and 4-bit KV caches. In doing so, the authors show significant speedups over other methods using their quantization serving framework QServe. They evaluated the Llmaa and Qwen model families.

In Summary, quantization is a widely studied technique for reducing the computational and memory requirements of LLMs models by representing weights with lower-precision data types, such as int8 instead of float32. This approach is particularly practical in scenarios where resource efficiency is critical, such as deploying models on edge devices or mobile hardware with limited power and storage. Quantized models can retain much of the original performance in many scenarios. However, quantization has limitations, as it can lead to significant performance degradation for tasks requiring high precision, similar to other techniques that aim to reduce the overall model size.

3.4 Early Exiting

Elbayad et al. [20] conducted one of the first studies on early exits in LLMs. They introduced transformers with multiple output classifiers, an approach that is often used in follow-up works. In addition to the standard LM head, which is used to predict a token from the last hidden state, additional output classifiers $\mathcal{C}$ are introduced, which learn to predict from an intermediate hidden state h^l of layer l. Thus, dynamic decisions can be made to use an intermediate classifier to predict from an early hidden state. They propose several training approaches for these intermediate classifiers based on the availability of hidden states. Furthermore, they propose different types of exit: (i) *sequence-specific depth*, where the decoding of the output tokens is done on the same transformer block, and (ii) *token-specific depth*, where a separate exit is chosen for each token individually. Furthermore, Elbayad et al. experiment with different methods for deciding when to exit, including likelihood-based, correctness-based, or confidence-based methods. They conducted experiments which showed that up to a maximum of $\frac{3}{4}$ layers can be omitted without a significant loss of accuracy.

Another earlier work on early exit in transformer architecture is DeeBERT [89]. Similarly to [20], their method introduces additional off-ramps (i.e., LM heads) at each layer and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

trains them on a fine-tuning dataset. They used entropy to decide when to exit, using the output distribution of an intermediate LM head and comparing it to a threshold S. Thus, S is a hyper-parameter controlling the trade-off between efficiency and accuracy. They also implemented *ensembles* instead of simple classifiers as an off-ramp, which they report does not lead to very significant improvements. They show up to 45% less time consumption while only sacrificing up to five points of accuracy in their tests with BERT and RoBERTa. They extended their work with BERxiT [90], where they introduced additional learn-to-exit modules, which perform better than its predecessor, DeeBERT, and apply to a broader range of tasks.

The *Confident Adaptive Language Modeling* builds upon Elybayad et al. work [20] and was introduced by Schuster et al. [68]. Their method involves making an exit decision based on a local confidence score to achieve global consistency. The higher the local confidence, the more likely it is to exit early. Thus, they opt for an early exit predicated upon a locally determined threshold. Their method included a weighted loss that favored higher layers, which was the best way in their experiments. They compared three different approaches to decide to exit: (1) soft-max response ($\text{Softmax}(W_i d_t^i)$), (2) hidden state stabilization based on cosine similarity, and (3) training a linear classifier that decides to exit. In addition, they calibrated local early exits from global constraints so that global consistency is approximated. They report that calculating the local softmax score and the classifier-based exiting brought the best results, while the most efficient state saturation fell slightly. Evaluations were performed on the T5 encoder-decoder model. Furthermore, they implemented state copying into their method to enable KV caching even though not all hidden states are computed.

SkipDecode [15] is another early exit technique. The main focus of this study was to overcome the limitations of KV caching and batch inference. Batch inference refers to processing multiple input instances simultaneously to use available hardware efficiently. SkipDecode does this by assigning fixed exits to tokens in a batch at a particular position in the sequence so that computations of tokens at the same position in the sequence are complete simultaneously. Furthermore, they overcome problems in KV caching: if a token exits later than the preceding token, KV caches have to be recomputed, so SkipDecode uses monotonically decreasing exits, which means that preceding tokens use at least as many layers as succeeding tokens. The rationale is that they state that predicting the next word is more difficult at the beginning of sequences and becomes more manageable with more tokens. They also use a computational budget, and based on this, later tokens get fewer layers by specifying maximum and minimum exit points. They show that a speedup of up to 2 can be achieved without losing too much performance compared to the base model, i.e., more speedup leads to decreased performance.

Another interesting approach to early exiting is discussed by Elhoushi et al. [21]. They combine several techniques to perform early exit with verification and correction in their approach. First, they introduce a training approach with dropout and early exit loss. Dropout is a technique often seen in DNN in which a layer is skipped during training with probability p. They also use a special loss function so that the original LM head is

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

capable of intermediate layer decoding. Speculative decoding is a technique in which a smaller model generates tokens fed into a larger model to verify the predictions. In their method, they use self-speculative decoding, i.e., reuse the same model for generation and verification, thus overcoming the problem of training two separate models for both phases. Therefore, they combined layer dropout, early exit loss, and self-speculative decoding, resulting in a speedup of up to 1.86.

Din et al. [96] introduced a completely different approach and proposed a method to cast hidden states of intermediate layers to use later as predictions. They use an intermediate hidden state to approximate a future (deeper) layer. More formally, using linear regression, they take a hidden state $l < L$ (where L is the final layer) and cast it up to a layer l' with $L \geq l' > l$. So, they skip the calculations between layer l and l'. They learn the regression model over sampled hidden states from an input data set. Then, they multiply the regression matrix by the hidden representation. The sampled hidden states show superior results compared to skipping layers without using the proposed regression method. They, therefore, encourage other researchers to use such methods for more advanced early exit techniques.

One of the first papers to introduce early exit with reuse of the original LM head, and therefore without additional parameters for intermediate prediction, is *Losses from the Intermediate Layers* (LITE) [84]. To do this, they introduce the weighted aggregation of losses to train the entire model during fine-tuning, that is, $Loss = \frac{\sum_{i=1}^{N} w_i \cdot Loss_i}{\sum_{i=1} w_i}$, where N is the number of layers and $Loss_i$ is the cross-entropy loss introduced by an intermediate layer i, weighted by the parameter w_i. Therefore, the fine-tuning process should teach the model to decode tokens from intermediate representations. The experiments show that through the fine-tuning process, about 40%-60% of the shallow layers' output matches the final layer's output. Furthermore, the authors present that the quality of the final-layer predictions is not significantly degraded. They introduce a confidence-based early exit based on the softmax logit value of the LM head. Although they show very good efficiency and quality metrics, they only evaluated their accuracy with the quality guided by the Claude model without introducing other NLP metrics.

The only method that has analyzed early exits in the code completion realm, to the best of our knowledge, is the *Stop & Exit Controller* (SEC) introduced by Sun et al. [73]. This paper is, therefore, one of the most closely related to this thesis. The authors provide a method that introduces additional LM heads at each exit level while using token-level exits. Furthermore, similar to [20, 68], they train a classifier to decide when to exit during the computation by skipping layers after an exit decision. As a novelty, they introduce a third option *stop*, which aims to stop the inference when the tokens already generated might not lead to a meaningful inference. They evaluated their method on code datasets using Rouge-L and also used the acceptance rate, which indicates how likely a developer would accept the generated code. The evaluation was conducted on small decoder models, GPT-2 and CodeGen, with a maximum of 350 million parameters.

Since our proposed method uses RL, additional related work is the ConsistentEE frame-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

work introduced by Zeng et al. [100]. ConsistentEE introduces policy networks and additional LM heads at each layer of LLMs. Their reward function introduces the *hardness* of an instance. They train an additional layer, which they call the *memorized layer*, which should remember more complex instances and rewards (penalties) are handed to the policy based on the output of this layer. Their fine-tuning is done in three steps: (1) fine-tuning model, (2) freezing model and training LM heads, and (3) training policies. They evaluated the encoder-only BERT and decoder Llama models and showed superior performance to baselines such as DeeBERT [89]. To the best of our knowledge, this is the only method that uses RL to make exit decisions.

T. Sun et al. [71] perform a study on learning instance difficulty. They argue that models have a different view of difficulty from humans and that it is touThey define difficulty as d, which denotes the exit level for an instance x. Their method HashEE is introduced based on the hypothesis that if a training instance x_i is predicted to exit at layer l, then a similar instance x_j should use the same layer l. Therefore, their method assigns instances to buckets based on hash functions on a token-level basis. These buckets are used during inference to decide when to exit. The technique shows superior methods on some tasks but fails on others; evaluation was carried out using BERT models.

A more advanced method introduced by T. Sun is early exit based on ensembles [72]. Instead of relying on the information of a single classifier that the decision to exit is a good choice, the method depends on the voting of subsequent classifiers. An exit decision is made if a certain number of classifiers have decided with a certain confidence threshold that it is good to exit. Later layers have a more significant impact on the decision. Although the method shows some speedup loss compared to other early exit methods due to the overhead of handling multiple votes, it outperforms others in accuracy.

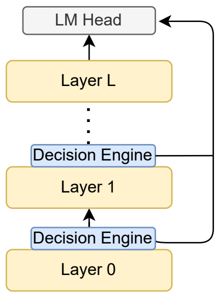
This study introduces an approach to early exit in large language models. Therefore, we presented several relevant early exit studies. Early exit differs from many other efficient methods because it is naturally dynamic—i.e., the decision to exit computation at an intermediate layer l is made at runtime and must not be made beforehand. Although it does not reduce the overall memory size, it can speed up inference by not using all layers of the model.

3.5 Comparison of Early Exiting Techniques

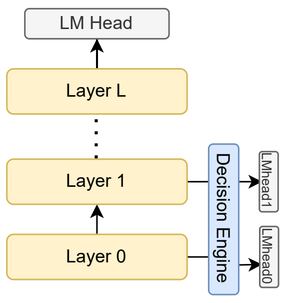
In Section 3.4, we discussed various early exiting mechanisms employed in LLM inference. Despite their differences, all these methods share a common goal: to reduce inference costs while minimizing loss of accuracy. However, they differ significantly in how early exiting is implemented.

As illustrated in Figure 3.1, there are two primary architectural approaches to early exiting in modern language models. Both methods rely on a central component called a *decision engine*. This engine determines when to stop inference at an intermediate layer $l < L$ rather than continuing to the final layer.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Early Exiting with single LM head.



(b) Early Exiting with multiple LM heads.

Figure 3.1: Schematic representation of Early Exit Architectures.

The decision engine can employ different strategies to make this determination. It might use simple metrics such as confidence scores, entropy, or more sophisticated machine learning models that learn when to stop forward passes. Regardless of the complexity of the decision-making process, a crucial factor is ensuring that the overhead introduced by these methods remains minimal. The cost of early exiting should not outweigh the benefits; otherwise, the overhead would make early exiting more expensive than full-model inference. Thus, maintaining a balance between efficiency and performance is essential for the effectiveness of early exiting methods. Furthermore, some techniques impact the full-layer abilities; again, such frameworks mustn't impact the original model too much.

Figure 3.1b illustrates one of the most common approaches for implementing early exiting in our analyzed studies. In this approach, additional linear layers, at specific *exit points*, are introduced at various layers of the model: In a typical language model, the LM head generates predictions from the final hidden state. However, it is not designed or trained to make intermediate-level predictions. To enable early exiting, each layer where an exit is possible uses its own linear layer, allowing predictions to be made at multiple points during inference. This modification introduces a significant number of additional parameters. For example, in the case of Llama-3 8B, where the hidden state has a dimension of 4096 and the tokenizer maps to 128256 different tokens, the LM head is defined as `linear (in_features = 4096, out_features = 128256)` (in PyTorch notation). This translates to $4096 \times 128256 = 525.336.576$ parameters—approximately 0.5 billion parameters per head. These parameters must either be trained or recomputed for each additional exit point, leading to a considerable increase in the overall parameter count and an apparent increase in memory.

Due to the memory overhead introduced by the multi-head approach, another strategy, shown in Figure 3.1a, aims to reuse the original LM head without adding additional

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

parameters. In this approach, instead of introducing new heads for intermediate layers, the original LM head and the model itself are fine-tuned to effectively decode from these intermediate layers. This method avoids the parameter bloat of multiple heads. Still, special fine-tuning is required to ensure the LM head can handle predictions from earlier stages in the model. Therefore, it changes the original model weights, which can lead to performance losses.

Although fewer studies have focused on this single-head early exit technique, it has demonstrated promising results in some cases. While the multi-head approach may offer more robust predictions due to its layer-specific heads, the single-head approach remains attractive because of its simplicity and reduced memory overhead. Finally, we summarize

Method	**Exit Type**	**Architecture**	**Model**	**Overhead**
Depth Adaptive Transformers [20]	Metric, LTE	Mult. Heads	Transformer	Training classifier & LM heads
DeeBERT [89]	Entropy	Mult. Heads	Encoder-Only (BERT)	Training LM heads
BERxIT [90]	LTE	Mult. Heads	Encoder-Only (BERT)	Training classifier & LM heads
CALM [68]	Metric, LTE	Mult. Heads	Encoder-Decoder (T5)	Training classifier & LM heads
SkipDecode [15]	Monotonic decreasing exits	Origin Model	Decoder-Only (OPT)	None
LayerSkip [21]	Fixed (with spec. decoding)	-	Decoder-Only (Llama2)	Speculative Decoding
MAT [96]	Upcast layers	Origin model	Encoder & Decoder (BERT, GPT2)	Regression
LITE [**varshney2023**]	Metric	Single Head	Decoder-Only (Llama2)	Calculating confidence
SEC [73]	LTE	Mult. Heads	Decoder-Only (GPT2, CodeGen)	Training classifier & LM heads
ConsistentEE [100]	RL-Policy	Mult. Heads	Encoder-Only & Decoder-Only (BERT, Llama)	Training policies & LM heads
HashEE [71]	Hash bins	-	Encoder-Only (BERT)	Calculating hash
Ensembles [72]	LTE	Mult. Heads	Encoder-Only (BERT)	Training classifiers & LM heads, votings

Table 3.1: Summary of different Early Exiting methods proposed.

related early exit studies in Table 3.1. We report the exit type, where LTE refers to Learn-To-Exit, and Metric refers to metrics like confidence, the architecture type, and the overhead on which model experiments were performed. Key insights of this are (i) most studies used early exiting with multiple heads; (ii) there are fewer studies on decoder-only models, and most studies on encoder-only models; (iii) a lot of studies used LTE or metrics as existing paradigms.

Therefore, we want to address these gaps in the thesis by presenting an approach that uses reinforcement learning on modern decoder-only models for the code completion task on real-world code samples using a single LM head.

3.6 Other Methods

Other techniques exist to enable efficient inference, such as Orca [97]: a method that attempts to solve scheduling problems for autoregressive LLMs. Decoder-only models generate token-by-token on an iteration-based schedule. After one token is generated, the next token is generated, and so on. Typically, requests to an LLM are based on a complete generation, so Yu et al. propose a method for iteration-level scheduling and a specialized batching method. They report a 36x speedup compared to another framework with faster transformers.

Agrawal et al. [1] discuss their method SARATHI. The proposed techniques aim to make LLMs' prefill and decoding phases more efficient by exploiting better GPU utilization. They use *chunked prefills* and further optimize decoding, where a batch consists of a single

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

prefill chunk, and the rest is filled with decodes. They report a best-case throughput of 1.33x for Llama-13B models. Furthermore, the proposed method reduces pipeline bubbles.

Another GPU optimization approach is LightSeq [87]. Here, the authors implemented more efficient techniques; for example, they implemented the encoder layer using general matrix multiplications with cuBLAS, a Nvidia library. They also optimized the top-k sampling by providing a hierarchical search, using an approach where they write intermediate results to GPU memory so that not everything needs to be recomputed, and parallelizing the top-k selection by partitioning the logits.

A completely different approach is BlockSkim [30]. In this study, the authors present a strategy in which they predict the relevance of the input tokens using attention weights. They show a joint training paradigm so that it can filter tokens from the original input during inference. They evaluated it only on QA datasets and reported significant speed-ups of up to 3x.

Another technique that tries to make the inference more efficient is H20 [102], which aims to reduce the size of the KV cache. Based on observations, they report that only a few tokens are responsible for most of the attention scores, called H2 (heavy hitters). When these heavy hitters are removed, the model performs significantly worse. The authors propose an oracle that controls the H2s in the KV cache and suggest a cache eviction strategy. They report substantially higher throughput than other similar methods.

This chapter explored various techniques to accelerate inference and enhance resource efficiency. We presented methods from the fields of *knowledge distillation*, *quantization*, *pruning*, and *early exiting*, alongside briefly mentioning other approaches. Although the primary focus was early exiting, we also examined various related studies using different methods. Building on this SOTA overview, the following section introduces the methodology employed in this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

System Model and Methodology

In this chapter, we introduce our proposed approach and the overall methodology of this thesis. We start by presenting the system model of the approach in this thesis and then turn attention to individual aspects of the methodology.

We describe the datasets selected for this study and the rationale behind their choice. Then, we discuss the LLMs used to evaluate our approach. Next, we explain the post-training fine-tuning process designed to enable early exiting.

4.1 System Model & Methodology Overview

The methodology of this thesis is illustrated as a system model in Figure 4.1, which outlines the key steps in developing an early exiting approach for LLMs in code generation. Our approach is conceptually divided into two phases: (i) offline RL agent training and evaluation and (ii) online energy-aware code generation.

In the first phase, we identify appropriate datasets and pre-trained models aligned with real-world code completion requirements. The datasets, representative of practical use cases such as GitHub codebases, undergo preprocessing to ensure compatibility with the LLM's input format, including tokenization, normalization, and splitting into training, validation, and testing sets. Details on datasets and model selection are discussed in Sections 4.2 and 4.3, respectively.

Then, the pre-trained LLM is fine-tuned using a specialized aggregated weight-based loss function inspired by current research [84]. This function enables early exiting by learning the model to decode hidden states of the intermediate layers, enabling early exits. Since pre-trained models do not inherently support early exiting, this fine-tuning step is essential and is detailed in Section 4.4.

Next, the early exit problem is formulated through reinforcement learning (RL). An RL agent is trained to dynamically predict when to exit, balancing computational resources

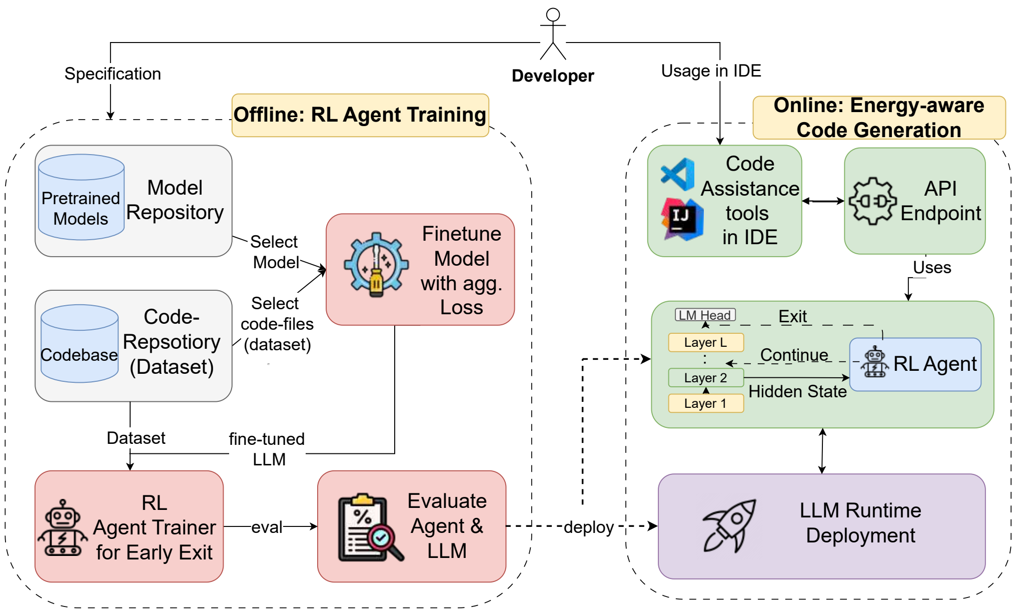


Figure 4.1: High-level Methodology Overview as system Model.

with output quality. The RL environment and reward structure are designed to achieve this trade-off, as described in Chapter 5.

After training, the entire system is evaluated to assess its performance. Evaluation metrics include code-specific and general NLP metrics and measures of resource consumption. This ensures a comprehensive understanding of the framework's efficacy before deployment - details on evaluation setup and results are presented in Chapter 6.

In the second phase, the fine-tuned LLM and trained RL agent are deployed together to enable resource-aware early exiting during code completion. A deployment endpoint integrates the RL agent and LLM, facilitating real-time user requests and supporting use cases such as code completion in integrated development environments (IDEs). This deployment setting, discussed in Section 5.4, provides a prototypical implementation of the feasibility of the framework as an efficient code-generation tool for software engineers.

4.2 Data Selection & Preparation

In current LLM research, there are several definitions of code-related tasks. It can be differentiated between *text-to-code* and *code-to-code* completions. Text-to-code refers to the paradigm of generating code in a programming language from natural language, for example, translating specifications into an executable program. This can also involve instruction models, which allow a user to input an instruction in natural language that leads to code. In contrast, *code-to-code* refers to processes where the model receives code written in a programming language as input and produces code as output.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We only consider *code-to-code* related code completion as a task. Given some code context, that goal is to predict the next token(s). This task can be further divided [49] into sub-tasks, for example, token-level or line-level completion.

There are many different datasets available for the code completion task. For example, HumanEval [12] is a dataset with the task of predicting a function body from a docstring and a signature, with samples comprising "traditional" computer science problems. CodeSearchNet [38] is a huge data set consisting of function definitions and docstrings, totaling 2 million functions scraped from publicly available code.

Most code-completion benchmarks and datasets have in common that the primary task is often to predict a single function from a given docstring or method/function signature. Although this is certainly an interesting problem, it does not capture the entire scope of code completion. Within software engineering projects, code completion involves more than filling in functions with existing docstrings. It also includes dynamically anticipating entire code elements, such as variable declarations, imports, constructors, enumerations, or during the development of a function.

Therefore, we aim to use datasets more closely related to real-world code projects, potentially allowing software engineers to utilize our approach. Consequently, we selected two datasets from the CodeXGlue [49]. CodeXGlue is a benchmark and dataset collection for various code-related tasks. It primarily provides available datasets and assigns tasks to them. In total, it consists of 10 tasks with 14 datasets. We specifically use the datasets designated for the code completion task, that is, the JavaCorpus introduced by Allamanis and Sutton [4] and PY150 [65] from Raychev et al.

Name	**#Samples (per Split)**	**#Tokens**	**Language**	**Source**
JavaCorpus [4]	12.934/7.189/8.268	15.8m/3.8m/5.3m	Java	GitHub
PY150 [65]	100.000/50.000	76.3m/37.2m	Python	GitHub

Table 4.1: Overview of datasets used, as provided in CodeXGlue [49].

Table 4.1 summarizes the two datasets. We follow the train-test split introduced in the CodeXGlue paper [49]. Both datasets are collected from open-source projects on GitHub and include thousands of files. Since these are complete code files, they reflect real software engineering projects, including import statements, classes, and functions, and some files only contain large enumerations, typically for larger projects. Thus, they should provide a more accurate representation of actual software engineering code rather than focusing solely on pure computer science problems. Furthermore, we conducted experiments on two programming languages, Java and Python, which are the most widely used languages.

For preprocessing, both datasets from the CodeXGlue collection are tokenized (i.e., split into words), and all comments are removed. Thus, the datasets are pure code and do not contain any noise in the form of comments or the like. We did not remove literals or constants and replace them with some special tokens as they do in the CodeXGlue

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

GitHub repository [1] to not simplify the task, since we believe that predicting literals can be pretty important; at least from a developers perspective.

In the following two subsections, we present results from fundamental data analysis on these two datasets and discuss the datasets in more detail.

4.2.1 Java Corpus

The JavaCorpus was introduced by Allamanis et al. in 2013 [4] and aimed to be a high-quality dataset of Java files. To do this, the authors filtered the GitHub archive, using only projects that had been forked at least once; this filtered out projects with a bad reputation, which should result in higher-quality code. Furthermore, they removed duplicates and projects with very similar commit hashes, which indicate that an additional user uploaded the same project.

Thus, the JavaCorpus can be seen as a collection of real-world Java code consisting of open-source, publicly available archives from GitHub. A downside of this data set is that it was collected by the authors about 10 years ago. However, since Java syntax and semantics change very little in newer versions, this should not be a disadvantage, except that it might lack new versions of libraries or language features introduced later. For example, Listing 2 shows a very short example from the JavaCorpus, showing a class from the `sqlproc.dsl.ui` package. All the examples are structured similarly: they are complete Java files from an openly available project. These files also contain package and import statements. Comments and documentation have been removed. Tabs or newlines in the data set are also removed, as the Java syntax defines statement endings with `;` and block endings with } and, therefore, would rather be noise for an LLM.

For convenience, we use the version of the dataset provided in Huggingface [2]. Here, the dataset is split into tokens in arrays, but we combine the tokens into a single string and use a simple whitespace `" "` as a token delimiter. Also, the version in Huggingface introduces special `<s>` and `</s>` at the beginning and end of a sequence. However, we remove these special tokens according to the end-of-string and start-of-string tokens of the models we use by using the appropriate tokenizers of the models. For example, OPT uses only `</s>` at the start of every sequence.

Figure 4.2 shows the distribution of the number of tokens in each sample capped at a maximum of 5000 tokens in the whole dataset. In order not to introduce a bias through a tokenizer of a model we are using, the number of tokens here is calculated by splitting the tokens by whitespace, i.e., the number of tokens calculated by a proper tokenizer might be higher, since tokenizers will split a long word into more than one single token. What we can see here is that most of the samples are less than 1000 words. However, a reasonable number of samples go way beyond that; the longest sample in the dataset is 264969 tokens long. For example, this outlier represents a file that defines thousands

[1] `https://github.com/microsoft/CodeXGLUE/tree/main`, accessed 04-11-24

[2] `https://huggingface.co/datasets/google/code_x_glue_cc_code_completion_token/viewer/java`, last accessed 04-11-24

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
package org.sqlproc.dsl.ui;

import java.net.URL;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;
import org.eclipse.core.runtime.FileLocator;
import org.osgi.framework.BundleContext;
import org.sqlproc.dsl.ui.internal.ProcessorDslActivator;

public class ProcessorCustomizedActivator extends ProcessorDslActivator {
    protected Logger LOGGER =
    ↪  Logger.getLogger(ProcessorCustomizedActivator.class);

    @Override
    public void start(BundleContext context) throws Exception {
        URL confURL = context.getBundle().getEntry("");

        ↪  PropertyConfigurator.configure(FileLocator.toFileURL(confURL).getFile());
        LOGGER.info("" + FileLocator.toFileURL(confURL).getFile());
        super.start(context);
    }
}
```

Listing 2: Simple Example from the JavaCorpus dataset.

of bitset rules. However, only 2.35% of the samples have more than 5000 tokens. We do not remove those extremely long samples, but we consider this when specifying the maximum context length we use with our chosen LLMs. Since the context size is a critical parameter for both the model and the GPU memory requirements, using too large samples or contexts is not feasible.

Figure 4.3a shows the 20 most frequent tokens in the data set. Here, the most frequent tokens represent tokens necessary for Java syntax. That is, `.` `()` `{}` etc. are all mainly relevant to the syntax of Java code. Furthermore, we can see typical statements like `public`, `new` `return`, and so on, which can be found in almost any Java file. Thus, the main observation is that a model must correctly predict syntactic elements to achieve a meaningful prediction, which we consider when selecting performance-relevant metrics.

Furthermore, Figure 4.3b shows a pie chart illustrating the relative proportion of `package` and `import` statements in each sample. This metric is important, as it is crucial for determining the model context required for completion. A Java file's import/package section may represent the minimum number of tokens needed to predict the rest of the meaningful file. Predicting during the import phase can be pretty random from a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

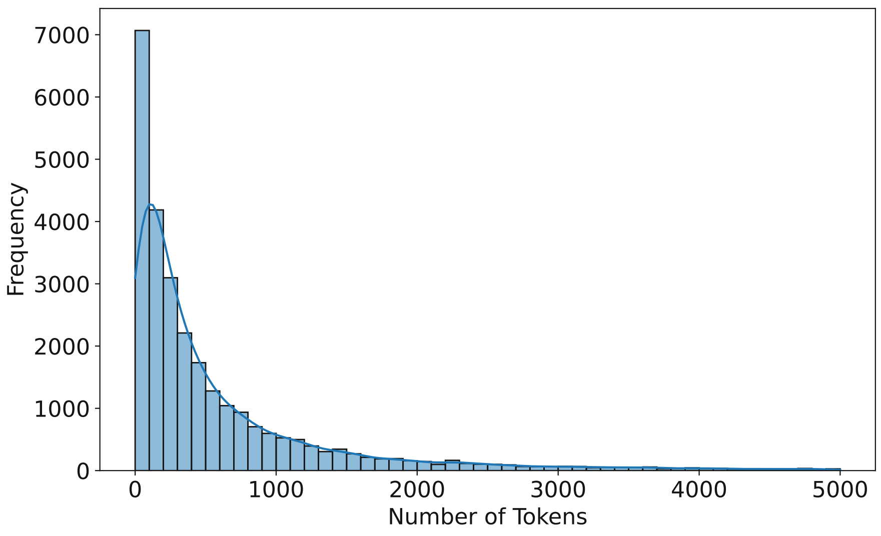
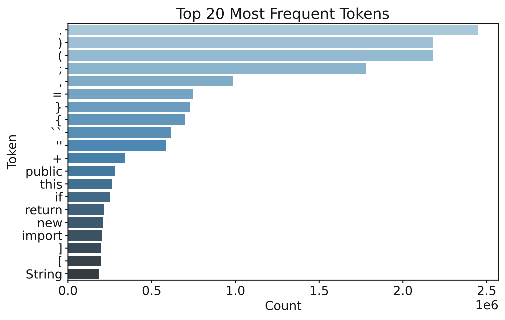
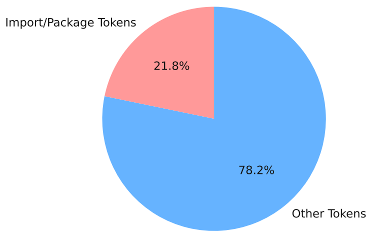


Figure 4.2: Distribution of sample length up to 5000 tokens in the JavaCorpus dataset (in number of word-tokens).



(a) Most frequent tokens in the JavaCorpus.



(b) Fraction of import statements.

Figure 4.3: Most frequent tokens & Fraction of import statements.

developer's perspective; for an LLM, hypothetically, it is generally challenging to predict import statements accurately, as they vary based on the developer and the specific code being written, which can vary widely. In other sections of the file, i.e., in a class, much more information is available for the model, and thus, it should be reasonable to focus more on those parts when doing evaluations.

However, some imports might be predictable based on previous imports. For example, a SQL library might be imported after a gateway to a DBMS. Therefore, we retain all import and package declarations in the samples, recognizing that this introduces some error in real-life settings. Not all import and package statements might be available in a real-world application from the beginning of writing code. Nevertheless, it appears essential for a model to predict these structures, as available tools for code completion (e.g., Copilot) are not limited to specific code sections. Additionally, as indicated in this chart, we set the minimum context length in experiments to a reasonable threshold of at least 20% tokens to avoid staying solely within a code file's import/package section. Note that this slightly differs from that provided in CodeXGlue, which only uses at least 15% of code as context. However, this will lead to some inferences of completions in a file's import/package section.

4.2.2 PY150

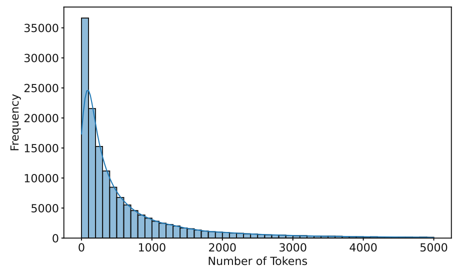
Like JavaCorpus, the PY150 dataset is a collection of code samples from Github, but with Python as a programming language. It was initially introduced by Raychev et al. [65] as an evaluation for a probabilistic code model. In PY150, the authors only included open-source licensed programs (Apache, MIT, BSD). Similarly to JavaCorpus, they incorporated the entire source code file, including import statements. Note that removing comments and documentation from the dataset was not initially mentioned by the authors of PY150 but was done in the CodeXGlue version. Furthermore, as Python does not rely on semicolons and curly braces to mark element and block ends but instead uses newlines and indentation, CodeXGlue includes a unique token `<EOL>` to mark an OS-independent newline, which we add to the tokenizers of the LLMs as a special token. However, no special token exists for any indentation, so a model output learned from this dataset must be post-processed to match the editor's standard indentation. Nevertheless, the level of indentation is usually a developer's preference or a specific company policy, and it may be a good choice not to let a model directly decide on a concrete level of indentation.

As we have already discussed with the other dataset, we can see the length of the samples in Figure 4.4a. Again, we cut samples larger than 5000 tokens from the visualization. Further, we can see that most samples are below the 2000 token length, similar to the Java files. The largest Python sample is "only" about 8000 tokens large. Notice that the Python files contain much less "boilerplate" tokens than the Java files (e.g., no braces, semicolons, etc.). Furthermore, we can see the proportion of import statements in Figure 4.4b; What is interesting is that Python files contain much fewer such tokens (probably because Java imports are often long paths to a class, such as `java.util.List` which consists of 5 tokens). Again, we consider this when setting appropriate contexts for performing LLM evaluations.

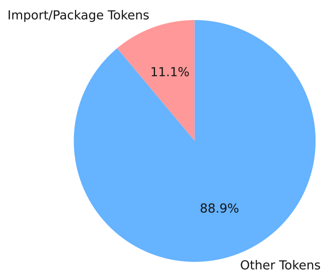
In summary, we use two fairly sophisticated datasets of open-source data collected from Github. Since Java and Python are two of the most widely used programming languages [3], these datasets should be a good choice for evaluating our models. Furthermore, the

[3] `https://www.tiobe.com/tiobe-index/`, accessed 04-11-24

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Average sample length PY150 dataset (up to 5000 tokens).



(b) Fraction of import statements.

Figure 4.4: PY150 sample length & Fraction of import statements.

datasets are quite complex since they are composed of code files that do not follow specific rules.

4.3 Candidate LLM Selection for our Study

Vendors release new LLMs almost daily, making selecting the appropriate models to evaluate our approach challenging. We aimed to choose autoregressive, decoder-only models since these are models of current SOTA, and trends indicate that most modern models are decoder-only. Regarding model size, it is essential that models are suitable for inference and allow full-parameter fine-tuning on our available hardware/GPUs. This limits us from using models with tens or hundreds of billions of parameters, as training and deployment are infeasible with our constraints. While inference can be done relatively cheaply, requiring only a few GB of memory, full-parameter fine-tuning demands significantly more—approximately 16GB per billion parameters [50], compared to around 2GB per billion for inference. This excludes additional factors such as batch sizes and sequence lengths, which can further increase requirements. Although some models are trained exclusively on code, we prioritized flexibility for fine-tuning rather than limiting our selection to code-specific models.

Therefore, we aimed to use models that are (i) modern SOTA models, (ii) decoder only, (iii) open and available as pre-trained versions, and (iv) viable with our hardware limitations.

As a first model architecture, we use Open Pre-trained Transformers (OPT), which Meta Research introduced in 2022 [101]. The OPT model family consists of sizes ranging from 125M to 175B. With their models, Zhang et al. aimed to achieve a performance similar to that of the GPT-3 class of models while requiring only $\frac{1}{7}$ of the carbon footprint to train compared to GPT-3. The OPT models have been trained on publicly available data. The

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

pre-training corpus mainly comes from datasets introduced with the RoBERTa model, i.e., collections of texts from books, stories, and news. Training data from mathematics, Wikipedia, Reddit, and texts from the Web were also included. The authors report that the large OPT models outperform the (back then SOTA) GPT-3 in some settings, while it falls off in other tasks. OPT has been widely used, for example the 1.3B model is among the top 20 most downloaded models in Huggingface [4] as of November 2024. Due to the abovementioned requirements, we use the 2.7B parameter model. This model comes with 32 layers and a maximum sequence length of 2048.

Second, we use a model from the LLaMa model family. LLaMa was also introduced by Meta AI [80] and was initially presented in 2023. The focus when introducing LLaMa was that the authors introduced it as a model trained only on publicly available data but still outperforming the GPT-3 models while being 10 times smaller. Initially, the LLaMa model family ranges from 7 to 65 billion parameters. The training corpus includes different text types, from books and news to GitHub code and even latex files from ArXiv. Compared to traditional decoder models, the Llama architecture is adapted, using an RMS norm to normalize the hidden layers' input and RoPE embeddings instead of absolute position embeddings. The Llama models continue to improve, and in April 2024, Meta introduced the Llama 3 models, which are the successors of Llama and Llama2 [3]. The architecture, while still based on a "standard decoder only" model, was slightly adapted with more efficient tokenizers and *grouped query attention (GQA)*. The training was done on 15 trillion tokens, with similar data to the original architecture but with 4 times more code. In late September 2024, they further improved their models with Llama 3.2 [2], where small-scale Llama versions in sizes 1B and 3B became available. These models were introduced for environments where memory requirements are essential. According to their model card [5] Llama 3.2 was trained on 9 trillion tokens from public data. Furthermore, they used logits from larger Llama 3.1 models in the pretraining phase, meaning that the Llama 3.2 text models are distilled (as introduced in Chapter 2) versions of the larger 3.1 models.

We chose Llama as the second model because it is widely used in many particle and research settings. Llama has become a de facto SOTA among open-source models, showing superior performance in many tasks. We chose the larger 3B version of Llama 3.2.

Model	**Introduced In**	**Parameters**	**#Layers**	**Max Context Window**	**Hidden Size**
OPT	2022	2.7B	32	2048	2560
Llama 3.2	2023-2024	3B	28	8192 (with RoPE:131072)	3072

Table 4.2: Models used within this thesis.

Table 4.2 summarizes the model we chose for the experiments and prototype implemen-

[4]`https://huggingface.co/models?sort=downloads`, accessed 08-11-24
[5]`https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/MODEL_CARD.md`, accessed 08-11-24

tation, furthermore Listing 3 show the detailed PyTorch OPT model. The models are comparable in size (2.7B and 3B) and have 32 and 28 layers, respectively. Since the number of layers is a rather important parameter for early exit, we choose models with quite a reasonable number of layers. In particular, 8B models in the Llama family have only 32 layers but a much larger hidden size (see, for example, Llama-3.1-8B configurations [6]). This matches our requirements for the LLMs we want to use for evaluations. Both models are relatively new, with LLaMa 3.2 released only a few months ago. Furthermore, both are suitable for the task, being auto-regressive text-only models. In addition, both models come in sizes that fit our hardware requirements. We use the Hugging Face version of both models.

In the following subsection, we briefly discuss some limitations of the LLMs we choose and their general limitations in early exit scenarios.

4.3.1 Challenges in LLM Selection for Early Exiting Scenarios

In the selection of LLMs for the code completion task and early exiting scenarios, several critical considerations must be addressed:

Context Window: We introduced the datasets in Section 4.2, where we discussed that the length of the samples in the dataset reaches several thousands of tokens. This leads to problems with the models we chose. For example, OPT-2.7B has a maximum context window of 2048 tokens, which means that many of the samples we use in the data sets are too large to be used with the model, at least with arbitrary contexts. To overcome this, we must set the maximum context length to a predetermined number that is short enough for both models. However, Llama 3.2 has a relatively large context window due to using RoPE extensions, a mechanism to improve performance on long texts [103]. In addition, very long contexts also threaten hardware VRAM limits.

Size of Models: We choose models of about 3B parameters. While these models are usable in their size, current SOTA models have up to hundreds of billions of parameters. Due to memory and time constraints in this work, we cannot apply our method to such large models. In particular, full parameter fine-tuning is not possible with larger models. Therefore, the approach must be adapted to work on larger models, especially given that very large LLMs have many more layers, e.g., Llama-3.1 405B with 126 decoder layers, 4 times more than the models we use.

KV Caching: Most early exit methods have problems with KV caching. KV caching is done in decoders when multiple tokens are generated. During attention computations, at the step of generating token t_i, the decoder's attention relies on all tokens $j < i$. However, the resulting QK matrix of previous tokens can be cached to speed up this computation significantly. This is done at every layer of the model.

[6]`https://huggingface.co/meta-llama/Llama-3.1-8B/tree/main`, accessed 08-11-24

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
OPTEESingleHeadEarlyExit(
  (model): OPTModelWithExit(
    (decoder): OPTDecoderWithExit(
      (embed_tokens): Embedding(50272, 2560, padding_idx=1)
      (embed_positions): OPTLearnedPositionalEmbedding(2050, 2560)
      (final_layer_norm): LayerNorm((2560,), eps=1e-05,
      ↪  elementwise_affine=True)
      (layers): ModuleList(
        (0-31): 32 x OPTDecoderLayerWithExit(
          (self_attn): OPTAttentionWithExit(
            (k_proj): Linear(in_features=2560, out_features=2560, bias=True)
            (v_proj): Linear(in_features=2560, out_features=2560, bias=True)
            (q_proj): Linear(in_features=2560, out_features=2560, bias=True)
            (out_proj): Linear(in_features=2560, out_features=2560,
            ↪  bias=True)
          )
          (activation_fn): ReLU()
          (self_attn_layer_norm): LayerNorm((2560,), eps=1e-05,
          ↪  elementwise_affine=True)
          (fc1): Linear(in_features=2560, out_features=10240, bias=True)
          (fc2): Linear(in_features=10240, out_features=2560, bias=True)
          (final_layer_norm): LayerNorm((2560,), eps=1e-05,
          ↪  elementwise_affine=True)
        )
      )
    )
  )
  (lm_head): Linear(in_features=2560, out_features=50272, bias=False)
)
```

Listing 3: PyTorch OPT model.

In early exiting, however, not all KV caches are available when an exit occurs. Suppose an exit occurs at layer $l < L$ at token t_i. Now, for the next token t_{i+1}, an exit occurs at a layer l' such that $l < l' < L$. The hidden states of all layers $l'' > l' \wedge l'' < L$ do not exist since those computations were skipped in the previous round. Thus, traditional KV caching becomes infeasible. Therefore, there are at least four options to evaluate and design early exiting approaches: (i) do not do KV caching and report results without it, (ii) approximate KV caches, (iii) recalculate missing KV caches,(iv) avoid increasing exits (i.e., always exit on same or more shallow layers).

One approach that approximates KV caches used in other studies [68, 20, 73] is to compute the KV caches based on the hidden state of an exited layer. Schuster et

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

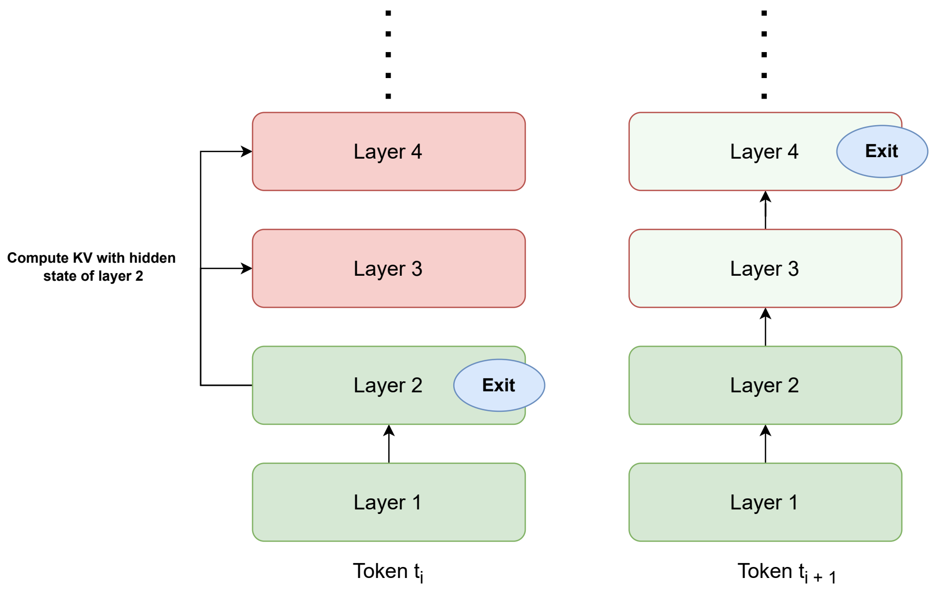


Figure 4.5: Illustration of KV cache calculation of missing caches in early exiting based on [68].

al. [68] note that simply copying the KV caches of an intermediate layer to deeper layers causes significant performance problems regarding accuracy-related metrics. Therefore, they suggest using the projections $W^i_{K/V}$ to compute the KV caches with the hidden state of an existing layer, which does not result in a significant performance loss. To better understand this, we have illustrated it in Figure 4.5. Here we see the first four layers of a decoder-only LLM represented as boxes, the green color means KV caches are available, red that no caches exist. Suppose the model exits at a token t_i in layer two and at some token t_{i+1} in layer 4. The KV caches or layers 3 and 4 are missing (red boxes). Before moving on to the next token, we compute the KV using the hidden states of the exited layer, but using the projections of layers 3 and 4, respectively, so that an approximate cache is available in subsequent generation rounds (illustrated as light-green box with red edges). Notably, such computations could be done in parallel since all deeper layers use the same hidden state of layer 2. Of course, this approach introduces some errors since it does not use the real hidden state. Moreover, it introduces additional overhead but is still faster than a traditional forward computation since it skips point product attention, and expensive MLP forward passes in the decoder. Furthermore, it may introduce unnecessary overhead - because we do not know in advance which layer the next exits will occur: for example, if a model exits only on the same layers for a whole generation, all the additional computations would not be needed.
We present findings without utilizing KV caching and conduct some tests involving KV caching without optimizations to demonstrate its compatibility with this method.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Nonetheless, discussing optimizations like parallelization falls outside the scope of this thesis.

4.4 Finetuning for Early Exiting

After selecting the dataset and model, the next step in the methodology is to determine how to enable early exits in LLMs. Standard LLMs are not inherently usable to support early exiting: during typical inference, the hidden state from the final layer is passed through a language modeling head, generally, a linear, fully connected layer that maps the hidden state to a token prediction within the vocabulary. However, in the context of early exiting, the final layer's hidden state is not always utilized, which necessitates an alternative approach.

As outlined in Chapter 3, there are two primary strategies for facilitating early exiting in LLMs: (i) training additional language modeling heads at each potential exit layer that learns to map hidden states from that layer, or (ii) employing specific fine-tuning techniques to make intermediate layer decoding viable.

Notably, both approaches require specific training to facilitate early exiting. Modifications are necessary, whether training additional language modeling heads or fine-tuning the model. Training typically occurs over multiple rounds when using multiple language modeling heads, as fine-tuning the model for a specific task is often needed before training the LM heads. Additionally, these methods tend to demand significantly higher memory resources by adding more LM heads.

This thesis focuses on the second strategy, implementing specialized fine-tuning to enable effective decoding from intermediate layers.

In traditional fine-tuning, the appropriate loss function is chosen based on the specific task. Cross-entropy loss is commonly used to compare predicted tokens to the ground-truth tokens for text generation. The loss function measures the difference between the model predictions and target outputs. During training, the input data undergoes a forward pass through the model to generate predictions. The loss calculation follows, where the defined loss function evaluates the discrepancy between the predicted outputs and the actual labels. The labels are typically just presented as a shift of the input (for text generation): for example, if the tokenized text is given as $[1, 2, 3, 4, 5]$, the labels, for the first pass (i.e., input is only $[1]$) the labels could be $[2, 3, 4, 5]$. Next, the backpropagation step computes the loss gradients with respect to the model parameters. Finally, an optimization step (using algorithms like Adam or SGD) updates the model's weights to minimize the loss, iteratively improving the model performance. So, the cross-entropy loss in sequence models during fine-tuning is typically noted as visible in Equation 4.1.

$$H(P, Q) = -\sum P(x) log Q(x) \tag{4.1}$$

In this thesis, we use a more specialized fine-tuning procedure. We follow and adapt the approach of Varshney et al. [84], who introduced the LITE (Losses from InTermediate

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

layErs) method. They present a loss formulation so the model learns to generate tokens from intermediate layers. This is similar to other work, in which (multiple) exiting classifiers are trained on similar loss functions [20, 68]. However, Varshney et al. presented it to work with the single original LM head, which does not introduce additional parameters. Furthermore, they note that the original LM head can be used to collect losses from these intermediate layers. Therefore, they present a loss as in Eq. 4.2.

$$Loss = \frac{\sum_{i=1}^{N} w_i \cdot Loss_i}{\sum_{i=1}^{N} w_i} \tag{4.2}$$

Where w_i is the weight given to the loss of the layer i ($Loss_i$) and N is the number of layers. This loss is then propagated through the network. That is, through this loss, the model should be able to learn to decode from intermediate layers. The authors show that this loss formulation can lead to good results for intermediate-layer decoding.

Notably, there are a few things to consider in this loss formulation that are not discussed in great detail by Varshney et al. [84]:

Hyper-parameters w_i: The weights for the losses of the intermediate layers are hyper-parameters that lead to the ability of intermediate layer decoding but also shift the generation capability of the model. The authors mainly report results with equal weights for each layer and very short discussions that give more weight to later layers (as those have more learning capabilities). However, hypothetically, giving more weight to lower layers should improve the results in lower layers even more since more attention is paid to them during fine-tuning, which, on the other side, can lead to an overall performance loss. Since fine-tuning is a resource- and time-intensive activity, finding optimal weights for different layers is challenging and might be interesting to investigate further in future work.

Limitations of lower layer generations: Exiting at intermediate layers will unavoidably lead to some performance loss. Each intermediate layer might have an inherent upper limit to its performance, representing a theoretical point beyond which the next layer becomes necessary to produce accurate or well-aligned predictions. This aspect remains unexplored mainly in current research. Ideally, there should be an optimal point for each w_i where a layer captures as much information as possible without compromising overall performance. However, identifying such optimal values is computationally challenging, as the complexity of hyper-parameter tuning in machine learning is NP-hard. Thus, there is significant potential for further investigation into such fine-tuning methods.

Selection of exit points: Another open-ended question is which layers should be selected to allow early exits. In the LITE paper, the authors choose specific layers, beginning with layer eight as the initial exit point, with exits permitted only on five layers (8, 16, 24, 28, 32). We propose that earlier layers can also effectively predict numerous tokens. Identifying more appropriate or optimal exit points can be

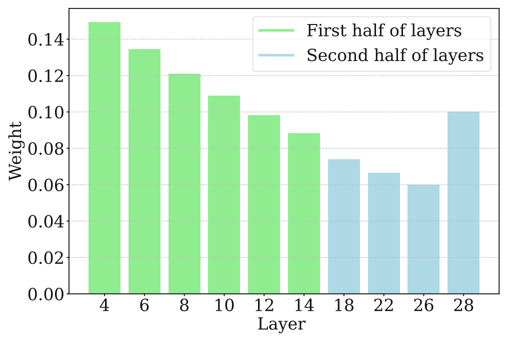
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

considered a distinct research undertaking in determining optimal weights. Moreover, opting for a later layer as the first exit point generally results in higher-quality outcomes. However, this decision also increases the minimal energy requirements, thus introducing a trade-off right when choosing these exits.

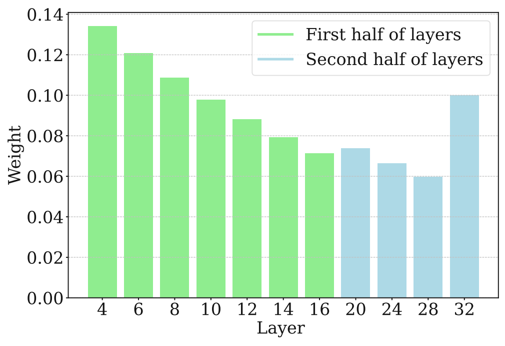
With these considerations in mind, we established our fine-tuning strategy. We chose not to allow early exits at every layer but ensured sufficient flexibility for decision-making. To this end, we implement the following rules to define exit points.

- The earliest exit point for both models is set in layer 4, ensuring that a portion of the full performance of the model is retained in any case.
- To facilitate early exits as soon as possible, we allow exits on alternating layers in the first half of the model, skipping every second layer.
- In the second half of the model, exits are allowed in every fourth layer.

This configuration results in 9 exit points for Llama (28 layers) and 10 for OPT (32). Furthermore, we assign the weights w_i as follows: we define separate budgets for the first



(a) w_i distribution on Llama.



(b) w_i distribution on OPT.

Figure 4.6: Weight distributions of loss calculations.

and second halves of the layers. The budget for the first half is 0.7, while the second half is 0.2. The final layer, initially connected to the LM head, received a fixed budget of 0.1 to maintain the ability to make predictions through the last layer. In our experiments, we explore only settings with decaying weights, assigning the highest weight to the earliest exit and hypothesizing that this would improve performance at shallower layers.

To assign weights computationally instead of manually, we use geometric sequences. The ratio for each layer in a group is calculated as presented in Equation 4.3, with a decay factor $r = 0.9$.

$$\text{ratio} = \left[r^i \mid i = 0, 1, \ldots, n-1 \right] \tag{4.3}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

This results in a decaying sequence, such as [1.0, 0.9, 0.81, 0.729, 0.6561,0.590, 0.531441] for the first half of the OPT model.

As presented in Eq. 4.4, we continue to normalize each ratio and multiply it by the specified budgets (0.7 and 0.2) to compute the weight w_i.

$$w[i] = \frac{ratio[i]}{\sum_{j=1}^{n-1} ratio[j]} \cdot \text{budget} \quad \text{for } i = 0, 1, \ldots, n-1 \tag{4.4}$$

The weights obtained in this way are illustrated in Figure 4.6, where the green bars represent the layers of the first half and the blue bars represent the layers of the second half.

4.4.1 Analysis of fine tuning

In this section, we will present results and analysis of how fine-tuning with aggregated loss affects the performance of the full model and how it enables decoding from the intermediate layer.

The analysis of the fine-tuning process is illustrated in Figure 4.7, showing training loss and gradient norms for both models. OPT was fine-tuned on JavaCorpus for five epochs, compared to 10 for Llama, due to results discussed in this section. Slight differences in the training steps arise from the tokenizer variations.

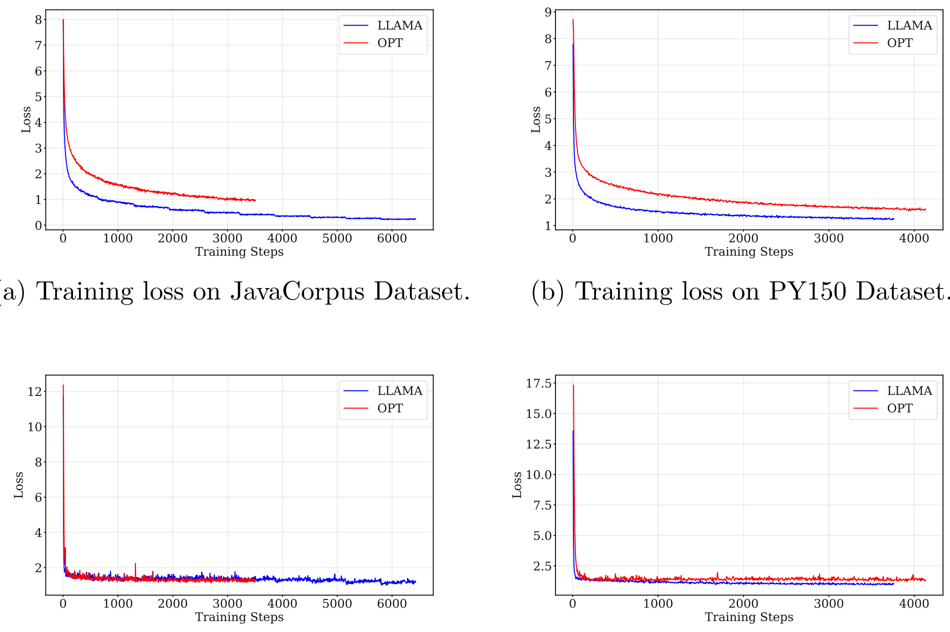
Both models learn quickly from the datasets, with loss converging towards zero, confirming the effectiveness of the aggregated loss function. On JavaCorpus, significant improvements are observed up to 3000 steps, after which the loss plateaus, suggesting that early stopping could be beneficial. For PY150, the loss starts slightly higher but stabilizes faster. Notably, gradient norms for OPT on PY150 increase after 2000 steps, hinting at potential overfitting. This pattern is absent in Llama, possibly due to its larger size (0.3B more parameters) but greater capacity, as reflected in its consistently lower loss.

In summary, the models effectively learn from the loss function, with similar behavior across datasets, although Llama demonstrates slightly better learning efficiency.

To explore the impact of fine-tuning on intermediate layer decoding, we conducted an experiment measuring metrics using fixed exiting across different epochs during training to evaluate how the fine-tuning approach influences over time. Fixed exiting refers to always exiting at a specific layer, irrespective of any mechanisms deciding the exit point. Hypothetically, the model should progressively learn to decode from intermediate layers during training. Initially, decoding from these layers is expected to perform poorly, improving with extended training as the model adapts to intermediate hidden states. The performance loss to the original model in such a setting is discussed later.

Figure 4.8 shows the RougeL and CodeBLEU scores for LLaMA in the JavaCorpus dataset, comparing the performance of the non-fine-tuned base model (black line) with fine-tuned models in 10 training epochs. The base model shows almost no capability

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Training loss on JavaCorpus Dataset. (b) Training loss on PY150 Dataset.

(c) Gradient norm on JavaCorpus Dataset. (d) Gradient norm on PY150 Dataset.

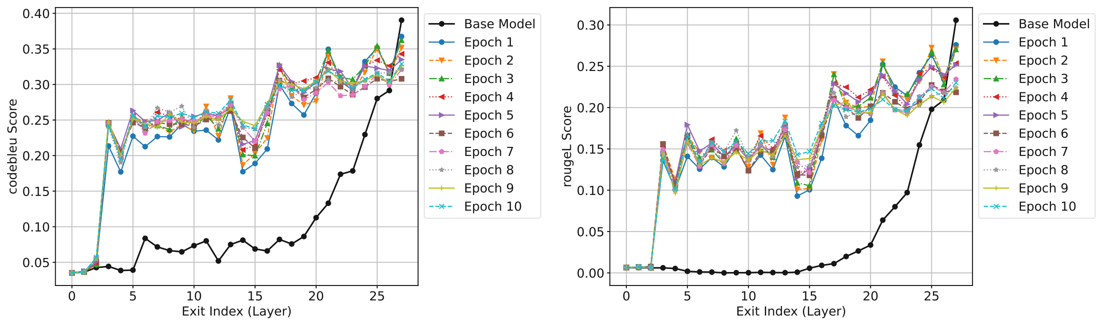
Figure 4.7: Loss & Gradient norm of fine-tuning models with aggregated Loss on both datasets.

for decoding from intermediate layers until layer 25, whereas fine-tuning over epochs reveals interesting patterns. While the overall trends across epochs appear similar, a closer analysis of the plots highlights how extended training affects the generation quality of intermediate layers. For instance, at epoch 1 (blue curve), performance in lower layers is noticeably worse than in subsequent epochs, but it excels at the final layer. In contrast, later epochs demonstrate the opposite trend: the final layers performance worsens while earlier layers improve. Additionally, earlier epochs generally perform better in the second half of the layers. In contrast, later epochs are superior in the earlier layers, meaning that kind of a "shift" in performance can be seen - i.e., the longer the training, the more performance earlier layers show, which, however, also scarifies more from the original model performance on deeper layers.

Therefore, the training duration (in steps or epochs) significantly impacts the model's overall performance and ability to decode from intermediate layers. Given its substantial improvement over the base model, a detailed analysis of the first epoch could offer even more insight.

Figure 4.9 shows the prediction accuracy of the single next token for the LLaMA base models and the fine-tuned models on the JavaCorpus and PY150 datasets. The models generate 15 tokens, each compared to the ground truth for token-level accuracy. A key finding is that base models perform poorly when decoding from intermediate hidden

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) CodeBLEU over epochs. (b) RougeL over epochs .

Figure 4.8: Metrics over training epochs of Llama on JavaCorpus (0.4 context, 1000 samples, max 50 tokens).

states, with near-zero accuracy for the first 12 layers. Accuracy improves significantly in deeper layers but only reaches competitive values in the final layers.

However, fine-tuned models achieve high accuracy even at intermediate layers. For example, at layer 6, LLaMA achieves ∼ 60% accuracy on JavaCorpus and ∼ 70% on PY150. The final layers only improve accuracy by 10–15%, highlighting the potential for early exiting. These findings agree with previous research [73, 68] that utilized multiple LM heads. In particular, PY150 shows higher accuracy in earlier layers than JavaCorpus, potentially due to its simpler syntax or the larger and different data set size, which might leads to differences or overfitting. The plot demonstrates drops in performance at layers

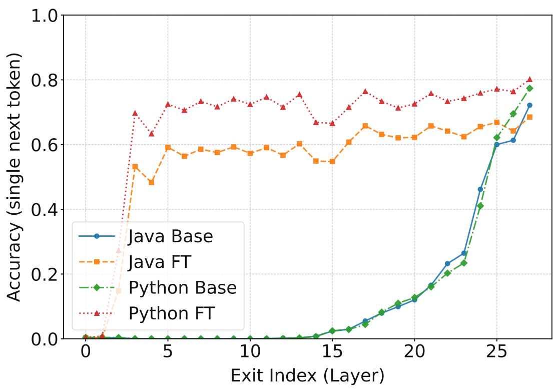


Figure 4.9: Token level accuracy of Llama on both datasets, generating 15 tokens (1000 samples, 0.25 context).

excluded from the aggregated loss during fine-tuning, a trend observed consistently across datasets. For JavaCorpus, the final-layer predictions are slightly less accurate (by a few percentage points) than those of the non-fine-tuned models.

Focusing solely on token-level accuracy can be misleading, as Figures 4.10 demonstrate through advanced sequence-based metrics like ROUGE-L, CodeBLEU, BLEU, energy

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

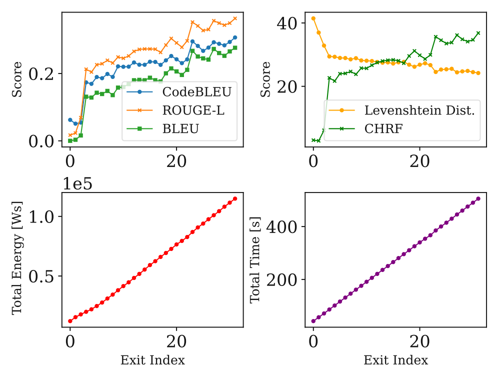
consumption, and time for LLaMA and OPT on the PY150 dataset with fixed exits. Unlike token-level accuracy, these metrics reveal that earlier exits significantly impact overall performance.

Although the differences in token-level accuracy between layers are modest (10–15%), sequence-level tasks are much more challenging. For example, LLaMAs RougeL score at layer 5 is ~ 0.35 compared to ~ 0.5 at the final layer, a much more significant gap than the accuracy of the token level indicates. This highlights the need for dynamic exit strategies, as lower layers can handle many token prediction but struggle with generating full, high-quality sequences.

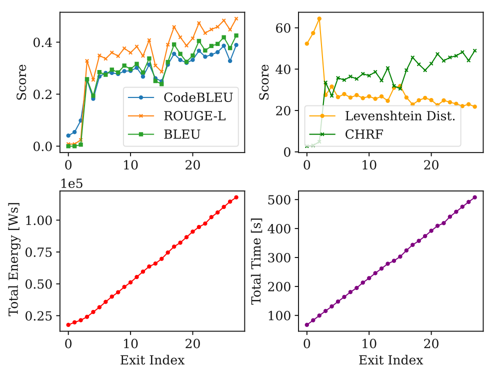
Energy and computation time scale linearly with the number of layers used, emphasizing the trade-offs between resource efficiency and model performance.

Another notable observation is that LLaMA exhibits more pronounced performance drops in non-finetuned layers than OPT. For example, OPT's Levenshtein distance in shallow non-finetuned layers (0-2) is significantly lower (40) than LLaMA's (60).

These findings underscore the importance of balancing accuracy and resource usage. Dynamic strategies offer potential solutions for efficient yet effective inference.



(a) OPT



(b) Llama3.2

Figure 4.10: Metrics, total Energy, and total time of Llama and OPT on PY150 dataset with fixed exiting.

In summary, shallow layers can decode a significant portion of tokens correctly. Still, deeper layers are essential for generating complete and high-quality sequences, as evidenced by metrics like RougeL and CodeBLEU. Fine-tuning with aggregated loss improves intermediate-layer performance but introduces trade-offs, reducing performance in later layers. This effect is more pronounced in LLaMA than OPT, particularly in non-finetuned layers. Energy consumption and computation time increase linearly with the number of layers used. Still, accuracy improvements are non-linear, emphasizing the need for intelligent exit strategies to balance efficiency and performance. The duration of fine-tuning plays a critical role, as extended training enhances shallow layer decoding but can negatively impact later layer quality.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

After introducing the overall methodology, including the selection of the model and dataset, as well as the fine-tuning process, the next step is to develop an exit strategy, which is discussed in the next chapter.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

Resource Efficient Code Generation by Learning Dynamic Early Exits

In this chapter, we present the methodology behind our exit strategy by formulating the early exit problem through reinforcement learning.

Unlike many existing methods that rely on score-based strategies or classifiers, we chose to apply reinforcement learning to the problem to explore its potential effectiveness in early exit scenarios for LLMs since RL is a technique not often used in such settings. The following section presents our formulation of early exiting as a reinforcement learning task, where we formulate the early exiting problem quite directly by using information about layers and token accuracy.

5.1 Formulation as RL Problem

There are several tools and frameworks to define reinforcement learning problems. One prominent tool for designing RL problems through code is OpenAI Gym [9], introduced in 2016. Gym, on the one side, includes RL benchmarks and, on the other hand, introduces a standard class that can be inherited to define custom environments for RL. OpenAI Gym prioritizes this by centering its framework around the environment rather than a specific agent or algorithm. This approach allows for the definition of an environment that can then be used to train an agent. So, Gym provides an API that facilitates communication between learning agents and environments. In our work, we use Gymnasium [81], a recent successor to OpenAI Gym, developed from a Gym fork and maintained by the Farama Foundation. Although environments can also be created manually, using such a framework offers several advantages:

- It is compatible across various training libraries and is not dependent on any single one. Many popular RL libraries and tools support environments defined in this way, such as Stable-Baselines3 [63], PyTorch [77], and RLLib [78], among others. Consequently, creating a Gymnasium environment enables testing with different libraries, RL agents, and algorithms without writing a new, aligned environment for any specific library or method.
- Standardization is an important topic in many ML/AI communities, bringing benefits for reproducibility. Adherence to such a standard allows easier sharing, versioning, and access to created RL models.

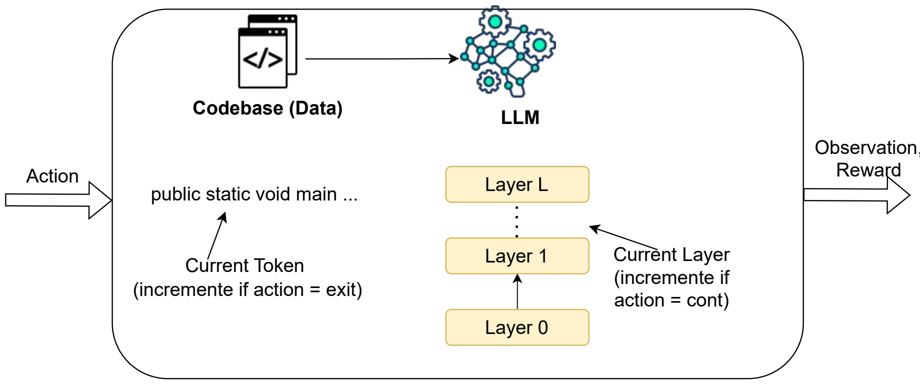
The Gymnasium API introduces two main methods that must be created to adhere to their API, which builds upon basic RL principles as introduced in Section 2.4:

`reset(seed)` : This function is used to start a new episode when the current episode is over. Parameters include a seed that resets any random previous seed and returns the first observation of the new episode just initialized.

`step(action)` : The step function is the primary method of the environment, taking as argument the action of the agent, which is the response to the last observation and the reward it received. The step function performs the interaction with the environment and must, therefore, compute the following observation and reward, given the action chosen by the learning algorithm. Thus, here, the state of the environment changes.

Consequently, utilizing the `step` function allows the agent to engage with the environment, thus obtaining new rewards and observations.

Based on these functions, we introduce our environment. We note the current layer as l and the current token as t in the following as abbreviations.



Codebase (Data)
LLM
Action
Observation, Reward
public static void main ...
Current Token (incremente if action = exit)
Layer L
Layer 1
Layer 0
Current Layer (incremente if action = cont)

Figure 5.1: Illustration environment overview.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Two primary components are at the core of our environment, as visible in Figure 5.1. The first is a codebase, a dataset the LLM uses to generate observations. This codebase could represent, for instance, a personal code repository, allowing the agent to be fine-tuned using a specialized dataset (we use the datasets introduced before). The second component is the LLM, which has been fine-tuned as outlined in the preceding section.

Observation

To explain how the environment functions, we first need to discuss the observations the agent perceives. The choices for defining these observations are limited but must remain computationally inexpensive. During inference, such computations must be performed in real-time and at a minimal cost, which should be less expensive than a forward pass through the next LLM layer.

Table 5.1 summarizes potential observation candidates that meet the relatively low computational expense requirement. For example, using confidence as an observation could be effective if combined with other information, but calculating the confidence of the LLM is not an efficient operation (since it needs at least a forward pass on the language model head plus a soft or argmax calculation). This challenge also applies to using predictions from the final layers. Although analyzing the output of the last layers (e.g., checking for convergence on the last-layer output) could provide valuable insights for an RL agent, it is computationally intensive.

Additionally, using the current layer or token as an observation might not provide significant value, as it could heavily depend on preceding tokens. Therefore, the most viable option for observations during training is the hidden state produced by the last layer of the current token. This hidden state encapsulates the current contextual understanding of the layer and is available without additional overhead.

However, interpreting hidden states in deep learning models is complex, making it difficult to understand why an agent learns specific behaviors solely from observing the hidden state. Despite this, we used the hidden state of the current layer for the current token as an observation for the RL agent.

Observation	**Overhead**	**Potential benefit**
Confidence	Forward pass LM head & Softmax/Argmax	Confidence distribution of current layer on current token
Current layer	None	Less meaningful without other information
Last token	None	Could be meaningful in some settings
Prediction of current layer and token	Forward pass LM head & argmax	Could be meaningful in some settings
Hidden State of current layer	None	High, should encode most information
Prediction of preceding X layers	High (voting)	High: could e.g. converge

Table 5.1: Observation candidates in the early-exit environment.

Resets and Steps in the Environment

First, let us consider how the environment reset process works. A reset occurs when the last decision for the final token generated by the LLM is made. This can happen when

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

an exit action is taken on the last token predicted by the LLM or when the last layer of the final token is surpassed by a continue action (i.e., continuing after the final layer of the final token).

During a reset, the environment samples a code file uniformly from the dataset and generates T tokens using the LLM. The context given to the LLM is determined randomly by uniform sampling from the interval $[0.2, 0.6]$; this interval comes from the observation shown in Section 4.2 such that the predictions do not fall into an import section of a file. In addition, the upper bound of 0.6 ensures that there are still enough tokens to generate since very small samples could cause problems. Furthermore, we limit the maximum context with an upper limit so that extremely large samples do not lead to VRAM overflows and to speed up the computation.

The environment then internally tracks all hidden states for each token across all layers, resulting in a total of $T \cdot L$ hidden states.

In a step, the agent provides the environment with the action *continue* or *exit*. The behavior of the environment for this action is as follows:

exit: If an exit occurs, the token currently processed is considered completed. Therefore, internally, the environment continues to the next token (generated in the last reset if the current sample is not finished yet), and the observation the agent gets is the hidden state of the next token at the first exit point, i.e., the first hidden state of the next token, along with the reward.

continue: If the agent provides a continue signal, internally, the environment remains at the current token but increases the current layer, and the agent gets an observation of the hidden state of the new current layer. To avoid infinite continuing loops, if the environment has already reached the last layer, we mark the current token as finished and proceed with the next token and first layer.

As points for deciding whether the agent should exit, we use the layers we fine-tuned, as discussed in Section 4.4.

In addition, we do not use truncation in the environment. We only finish episodes when a sample is completed (i.e., all tokens are processed) and do not stop episodes after a maximum number of steps. We decided to do so because it is unpredictable how many steps should be allowed at maximum since this depends heavily on sample to sample and from token to token. A simple limit would be $L \cdot T$, but by construction, this is the practical limit anyway. Any shorter definition would lead to problems if the agent encounters samples that need every layer for every token. This means it is expected to see fluctuations in episode lengths since every token can have an arbitrary optimal exit, and therefore, episode lengths might vary.

The steps in our environment are designed to reflect the early exit problem. If an exit action is taken, the internal state of the environment moves to the next token, and rewards

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

are assigned accordingly. The environment advances to the next layer if a continuing action is chosen. This setup reflects how early exiting is used during inference.

The environment functions as a *sequential* environment, meaning that the next state depends on the current state. Specifically, the hidden state of a given layer l is influenced by the hidden states of all the previous layers l' where $l' < l$. Furthermore, the environment can be characterized as a combination of *deterministic* and *stochastic* elements since a learning agent could not fully predict the next observation (i.e., the subsequent hidden state) with certainty.

Moreover, the environment is *dynamic* because the data generated by the LLM, including the tokens within each episode, changes with every reset. This variability ensures the agent encounters different scenarios across episodes, adding complexity to the learning process.

So, in summary, we provide an environment that gives the agent only the hidden state of the current layer & current token. The hidden state is without extra computations available. It does not bias the agent strictly (which confidence or saturation metrics might do). Therefore, agents must learn the complete efficiency-accuracy trade-off solely based on the hidden state and the reward signal.

5.2 Reward Function

The reward function has been iteratively refined based on insights from initial trials, which led to the following observations.

- As discussed in Section 4.4.1, many tokens can be accurately predicted with just a few layers, often over 50% of all tokens. Consequently, agents exploit this property efficiently by leveraging many (too) early exits.
- We initially started defining the reward by using energy and latency to determine a received reward. However, this formulation led to quite some problems, since it heavily depends on NVMLs performance counter update frequency. This often leads to energy measurements becoming 0, which is therefore unusable in a reward function. Some studies [94] also report that NVML or `nvidia-smi` is not a fully trustable source for energy measurements. Consequently, we use the number of layers as the efficiency term in the reward function, as it shows a linear relationship to energy consumption and is a more stable metric.
- Learning to exit at deeper layers becomes significantly more challenging in our environment setup and how early exiting is defined. Since lower layers can often predict tokens effectively and the trajectory to reach deeper layers is much longer, this poses difficulty since only a small fraction of samples needs all layers for correct processing. For instance, consider the exit-point definition of Llama presented in Section 4.4. If the optimal exit for an agent is layer 26, the trajectory to reach

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

this correct exit is C, C, C, C, C, C, E, where C denotes continued and E denotes an exit. This makes learning deeper exits extremely difficult, as the agent must identify and follow these long trajectories during training, avoiding early exits and interacting with the environment to reach the later layers.

- Allowing the agent to explore such late exit scenarios requires enabling exploration for longer continuations. However, rewarding the agent too much for exploring these actions could result in the agent always learning to continue, missing opportunities for early exits. So, this is a non-trivial and non-linear trade-off that the agent needs to learn. Of course, such problems could be addressed by adapting the environment so that the agent always gets equally distributed exits as observation. However, this would lead to the agent not learning from the true problem space but rather an artificially generated one that does not truly reflect the problem since, typically, RL agents need to learn the environment sufficiently by collecting the data on their own.

- A key consideration is whether the agent should learn to exit based on ground truth or alignment with the final layer. In a small number of cases, an earlier layer may make a correct prediction while later layers do not. Nevertheless, using ground truth labels often leads to situations where no layer correctly predicts a token, which raises the question of how such cases should be rewarded. Therefore, we define optimality based on the alignment with the final layer. Since the environment stores the hidden states of all layers, we can efficiently determine the first layer that matches the final layer's prediction (i.e., the optimal exit point).

- Additionally, maximizing cumulative rewards may not always lead to an optimal solution. Similar observations could result in different exit decisions, and because the agent is provided with dynamic samples, it may be infeasible always to make the correct exit decision.

- LLMs process input one token at a time, so using only episodic rewards (i.e., rewards based on the performance over an entire sample) could mislead the agent by relying on inappropriate rewards over the whole sample (e.g., a bad reward introduced only by a single token, while other actions are correct). So, the agent decides based on local properties but optimizes global, sample-level consistencies since rewards are optimized over an episode.

Insights from these observations guided the stepwise reward function. We developed the reward depending on the selected action, exit as represented in Eq. 5.1 and continue as shown in Eq. 5.2, giving feedback to the agent after each step. The reward function was

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

intentionally tailored to align closely with the definition of the early exit problem.

$$r_e = \begin{cases} 1, & \text{if } y_{\text{pred}} = y \text{ and } \ell_{\text{curr}} = \ell_{\text{opt}} \\ -(\ell_{\text{curr}} - \ell_{\text{opt}}) \cdot \alpha, & \text{if } y_{\text{pred}} = y \text{ and } \ell_{\text{curr}} \neq \ell_{\text{opt}} \\ -(\ell_{\text{opt}} - \ell_{\text{curr}}) \cdot \beta, & \text{if } y_{\text{pred}} \neq y \text{ and } \ell_{\text{curr}} < \ell_{\text{opt}} \\ -\epsilon, & \text{otherwise} \end{cases} \tag{5.1}$$

$$r_c = \begin{cases} 1, & \text{if } \ell_{\text{curr}} < \ell_{\text{opt}} \\ -(\ell_{\text{next}} - \ell_{\text{opt}}) \cdot \gamma, & \text{otherwise} \end{cases} \tag{5.2}$$

ℓ_{curr} is the current layer being considered, ℓ_{opt} is the first layer whose prediction matches the prediction of the final layer (that is, the optimal point to exit), and ℓ_{next} represents the subsequent layer. y denotes the token prediction of the final layer, which serves as the ground truth token for the reward calculation, while y_{pred} represents the prediction at ℓ_{curr}. The coefficients α, β, γ are trade-off parameters, constrained by $0 \leq \alpha, \beta, \gamma \leq 1$, which can control the agents' learning behavior. We also scale penalties to the interval $[-1, 0)$ to adhere to best practices in reinforcement learning (as discussed in the blog [41]) to stabilize learning by ensuring rewards are in the range $[-1, 1]$. Note that all layers here are considered for simplicity. Still, we use defined exit points in experiments (i.e., do not allow exits on non-fine-tuned layers) and calculate rewards based on them, e.g., if the optimal exit point is a layer that we do not consider as an exit point, we set it to the next layer that is an exit point.

Let us examine the reward function:

exit:

- The first condition applies a fixed reward if the decision to exit is optimal, meaning the current layer's prediction is correct (i.e., matches the prediction of the final layer) and it is the shallowest layer to do so, so $\nexists \ell' | y' = y \wedge \ell' < \ell_{curr}$.
- The second condition describes the penalty for exiting too late. This occurs when the current layer produces a correct token, but there is a shallower layer ℓ' such that $\ell' < \ell_{curr}$ with $y' = y$. The penalty is proportional to the distance between ℓ_{curr} and ℓ_{opt}, scaling with the number of steps from the optimal layer. Although such late exits still yield correct predictions in practice, overly rewarding them can cause the agent to exploit exiting decisions excessively. Thus, the agent should avoid consistently making suboptimal exits. However, in experiments, we set $\alpha \leq \beta$ so that exiting late is at least as good (or better) than exiting too early.
- The third condition represents the worst-case scenario: exiting at a too-early layer, resulting in an incorrect prediction. The penalty is also scaled based on the distance from the optimal exit layer.
- The final condition for exits, which is rarely encountered, covers cases where $y_{pred} \neq y$ and $\ell_{curr} > \ell_{opt}$. Although such cases occur, they are infrequent

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

and thus considered negligible. We assign a minor constant penalty to this scenario.

continue:
- We reward optimal decisions with a constant value of 1 when continuing. A continuation is considered optimal if $\ell_{curr} < \ell_{opt}$, which means that the agent has not yet reached the optimal layer.
- We penalize all other cases where $\ell_{curr} \geq \ell_{opt}$, scaling the penalty based on the distance between the current layer and the optimal layer. Here, we use $\ell_{next} = \ell_{curr} + 1$ because a continuation is incorrect if $\ell_{curr} = \ell_{opt}$ since the agent should have exited at this point. Again, we introduce a coefficient here to control the penalty.

The coefficients control the way the agent learns: e.g., increasing values for α and γ should lead the agent to learn not to exit too late while increasing β controls the agent not to exit too early.

Therefore, in summary, training aims to maximize the objective function of Eq. 5.3, based on the step rewards r_c and r_e defined above. Note that we define $r_c = 0$ if the action is exit and $r_e = 0$ if the action is continue.

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{T} (r_e(t) + r_c(t))\right] \tag{5.3}$$

Where π_θ is the policy parameterized by θ, T is the number of time steps per episode and t the current time step.

Finally, for completeness, the expected theoretical reward in a given time step can be formalized as visible in Eq. 5.4 and Eq. 5.5, where, however, all probabilities $P(..)$ can only be approximated.

$$\begin{aligned} E[r_e] = &\sum_{\ell_{\text{curr}}=1}^{L} (P(y_{\text{pred}} = y) \cdot P(\ell_{\text{curr}} = \ell_{\text{opt}}) \cdot 1) \\ &- \sum_{\ell_{\text{curr}} \neq \ell_{\text{opt}}} (P(y_{\text{pred}} = y) \cdot (\ell_{\text{curr}} - \ell_{\text{opt}}) \cdot \alpha) \\ &- \sum_{\ell_{\text{curr}} < \ell_{\text{opt}}} (P(y_{\text{pred}} \neq y) \cdot (\ell_{\text{opt}} - \ell_{\text{curr}}) \cdot \beta) \\ &- P(\text{otherwise}) \cdot \epsilon \end{aligned} \tag{5.4}$$

$$E[r_c] = \sum_{\ell_{\text{curr}} < \ell_{\text{opt}}} (P(\ell_{\text{curr}} < \ell_{\text{opt}}) \cdot 1) - \sum_{\ell_{\text{curr}} \geq \ell_{\text{opt}}} (P(\ell_{\text{curr}} \geq \ell_{\text{opt}}) \cdot (\ell_{\text{next}} - \ell_{\text{opt}}) \cdot \gamma) \tag{5.5}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

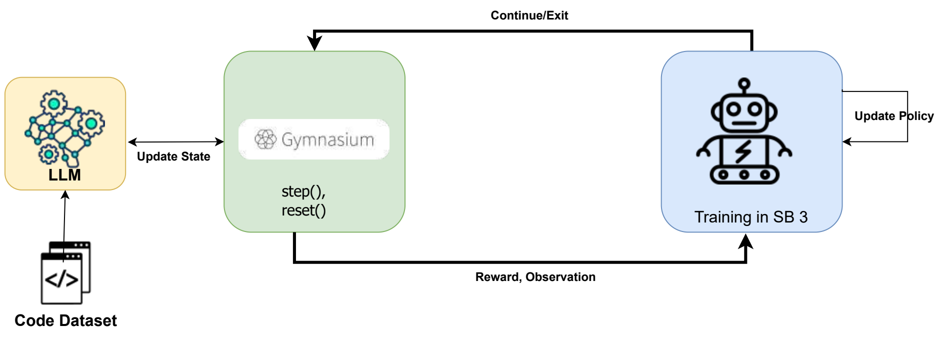


Figure 5.2: RL Agent Training process illustrated.

5.3 Training RL Agent for Early Exit

Figure 5.2 provides an overview of our training setup, including the tools and libraries used. At the center is the LLM, which the Gymnasium environment uses to generate the current problem states. The SB3 agent interacts with this environment, making decisions based on observations to either continue processing or exit. Each decision results in a corresponding reward, facilitating the agent's learning and adaptation. While learning, the agent with the LLM always generates T tokens per sample and randomly chooses the context as described above.

Our choice of algorithms focused on a widely recognized RL method, Proximal Policy Optimization (PPO) [67] This algorithm is selected for its status as a SOTA solution, proven effectiveness, and broad application in various RL tasks [5]. PPO was chosen primarily because of its stability in training. It uses clipping approaches to avoid too large updates. However, every algorithm suited for discrete action spaces and observations can, in principle, be applied to the problem and environment presented before.

Deep RL algorithms are sensitive to hyper-parameters and can require millions of steps to reach an optimal policy. We always trained until convergence was observed in both episodic and stepwise rewards. However, it is essential to note that the policies we report may not be fully trained or optimal, as training was halted after approximately 200k-500k steps on average when we can reach convergence behavior. A detailed description of the hyper-parameter setup and results is provided in Chapter 6. Furthermore, since we use an LLM in the environment directly, due to VRAM and CUDA limitations, running multiple parallel environments simultaneously was not feasible, even though some algorithms, such as PPO, can benefit significantly by learning from multiple environments at the same time [37]. Furthermore, we are restricted to use policies of small size only, such as a shallow network with few neurons; this is because the overhead introduced by the agent must be reasonably small to achieve efficiency gains when doing early exiting.

In the inference phase, we extract the policy from the trained RL agent and convert it into a PyTorch-based DNN using the corresponding policy and action networks from SB3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
(policy_net): Sequential(
   (0): Linear(in_features=3072, out_features=64, bias=True)
   (1): Tanh()
   (2): Linear(in_features=64, out_features=64, bias=True)
   (3): Tanh()
 )
 (action_net): Linear(in_features=64, out_features=2, bias=True)
```

Listing 4: Example of PyTorch PPO model.

Consider an example involving Llama, the corresponding PyTorch PPO model featuring two hidden layers, each containing 64 units, as demonstrated in Listing 4. Here, the input size is 3072 (the hidden size of the Llama model), and the output consists of two actions: exit and continue.

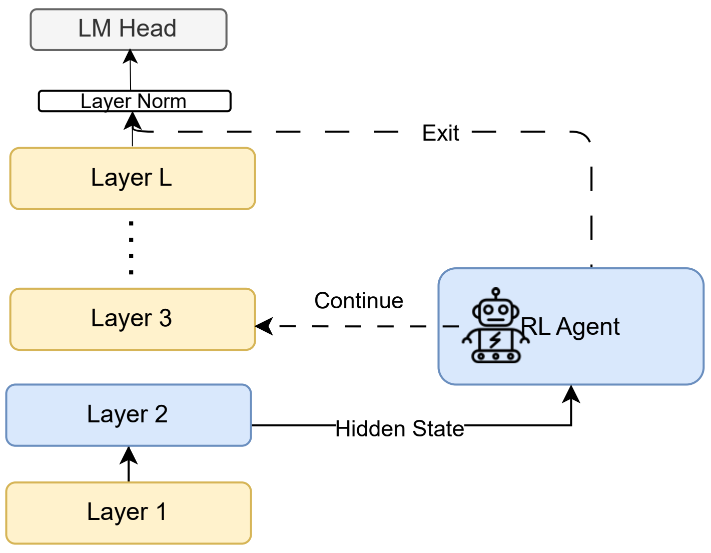


Figure 5.3: Inference on Early-Exit enhanced LLM.

As depicted in Figure 5.3, the inference process operates as follows. At predefined exit points (denoted by the blue layer in the figure) within the LLM, the RL agent receives the hidden state corresponding to the output of the current layer and determines whether to continue processing or exit early. This decision is made by passing the hidden state through the policy network of the RL agent.

The agent's decision is based on the softmax probabilities output by the policy network (or action network). If the agent decides to continue, the LLM proceeds with its

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

standard forward pass, processing additional layers. However, continuing the forward pass introduces additional computational overhead as the agent continues to evaluate subsequent model layers. In particular, the overhead at a continue is the agent forward pass and softmax calculation.

On the other hand, if the agent decides to exit, the language model skips all layers beyond the current exit point. In this case, we copy the key-value caches and hidden states (if caching is enabled) and pass the hidden state from the exited layer through the LLM's layer normalization. Finally, the normalized hidden state is fed into the language model head to generate the prediction. Thus, all computations are saved between the exited and final layers.

We employ a batch size of 1 for inference because, in LLM inference, combining numerous samples into a single batch can significantly affect latency from the user's perspective.

5.4 Implementation & Deployment

To train the agent, we evaluated existing libraries, as this thesis focuses on not developing a new RL algorithm but applying established methods. For this purpose, we utilized the widely used Stable-Baselines3 (SB3) library [63]. SB3 is built on top of PyTorch [59], a highly popular deep-learning framework known for its flexibility. As an enhancement of OpenAI's Baselines, SB3 provides user-friendly interfaces and well-documented access to several state-of-the-art RL methods, making it a suitable choice for training agents within our research context. After training the RL agent, we deploy it along with the LLM using FastAPI [64]. The endpoint adheres to the Huggingface Inference API and enables our EE-enhanced LLMs to be used in the Huggingface VSCode extension [24]. Thus, our setup can be used similarly to Github Copilot or other tools. The Listing 5 shows the configuration file for fine-tuned LLMs and early exit agents, adhering to Huggingface VSCode. A developer can specify LLM-relevant behavior (`num_predict`, `top_p`...) but also RL thresholds to decide how many resources to save (i.e., specify the trade-off).

After presenting our methodology in Chapter 4 and the definition of the RL setup in Chapter 5, in the next chapter will present our method's results and evaluations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
{
    //...

    "llm.attributionEndpoint": "http://localhost:8000/api/generate/",
    "llm.configTemplate": "Custom",
    "llm.modelId": "", // empty, since we use custom endpoint
    "llm.requestBody": {


        "options": {
            "num_predict": 10,
            "temperature": 0.2,
            "top_p": 0.95,
            "RL_agent_thresh": 0.9
        }
    },
    "llm.tokenizer": {
        "repository": "briemelchen/llama-3.2-3b-code-java-ee-w-scaled"
    },
    "llm.url": "http://localhost:8000/api/generate",
    "llm.contextWindow": 1024

    // ...
}
```

Listing 5: Example of VSCode Huggingface extension `config.json` to be usable with our EE-enhanced models.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Results & Performance Evaluation

In this chapter, we outline the evaluation of our method, describe the experimental design, and present the results obtained from our experiments.

We begin by describing the efficiency and accuracy metrics we used. The evaluation setup, including the hardware specifications, software versions, and Python libraries follow this. This section will also cover the specific hyper-parameter settings chosen to fine-tune LLMs and train RL algorithms. Furthermore, we describe how we design our experiments.

Next, we will analyze the convergence patterns and behavior of the RL agents used in our setup.

Subsequently, we present the results of our method evaluation on the test sets of the two datasets introduced earlier. This part details the changes in accuracy and computational efficiency brought about by our early-exiting strategy and examines the extent of overhead introduced by the method.

6.1 Evaluation Metrics

In this work, we analyze metrics related to resource efficiency and model accuracy, showing how our method impacts these metrics. The following sections discuss the metrics used for evaluation.

6.1.1 Efficiency Metrics

Efficiency can be defined in multiple ways. Existing research on early exiting often relies on the following metrics:

Number of layers skipped (in % or absolute numbers) is a commonly used metric that provides a clear overview of model performance, as latency tends to scale linearly with the number of layers used. This metric is hardware-independent and easy to interpret. However, it has a notable limitation: It does not account for any additional overhead introduced by the decision process, which means that inference costs could potentially increase despite exiting at earlier layers.

Latency, Time (in seconds) is a straightforward and essential metric, especially from the perspective of end-users who prioritize prompt response times. Although it is a critical performance indicator, it depends on the hardware used and can be influenced by various other factors.

FLOPS (Floating Point Operations per Second) measures the total number of operations performed, clearly indicating computational savings. However, this metric is primarily useful for comparisons with other algorithms and may not directly reflect real-world performance.

It is difficult to judge a model's efficiency by only examining one of these metrics since they all capture efficiency from different perspectives. For example, time identifies a typical end-user goal; the fraction of skipped layers is a more theoretical and formal way to describe efficiency gains. FLOPs present a view that may not be directly relevant from a user perspective but is valuable for comparing computational complexity.

We report the number of layers as well as latency and use two metrics that have not been considered in current LLM early exiting research that we reviewed:

Energy Consumption (in Ws) Measurement of energy consumption is a valuable indicator because it provides a practical understanding of the real-world cost and impact on the sustainability of running a model. Unlike theoretical measures, energy consumption accounts for the actual power usage during inference, reflecting the efficiency of hardware utilization and any overhead introduced. This metric is critical because energy efficiency directly impacts operational costs.

Throughput (in tokens per seconds) Shows how many tokens a model processes on average. It is a very interesting metric since it can be seen as a more fine-grained analysis than simply measuring the time.

So, we present both (i) hardware-independent and more formal measures, like the number of skipped layers, and (ii) completely practical and hardware-dependent measures, like time, throughput, and energy consumption.

We measure time in seconds and energy consumption in Ws using Zeus-Monitor [76] and calculate throughput and number of layers by counting.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

6.1.2 Accuracy Related Metrics

Evaluating LLMs for code completion requires different considerations compared to general language modeling. Although traditional NLP evaluation often focuses on matching in texts through metrics such as n-gram matching, code generation demands additional syntactic and semantic correctness to ensure functionality. Evtikhiev et al. [23] report that standard NLP metrics often prioritize lexical precision, which may not align with the evaluation of the functional aspects of the code. For example, swapping words in natural language, such as exchanging *angry* and *wild* in the sentence *The cat is wild and angry* may not significantly change the meaning of the sentence. In programming, these rearrangements typically result in a total lack of usefulness as the code becomes non-functional or unable to compile. An example is the snippet `public void main (){}`, where swapping two tokens results in unusable and erroneous code. This should exemplify that evaluating code completions can be very challenging. Furthermore, the authors who described CodeBleu [66] argued that code is always tree-like (by AST programming construction) and, therefore, does not match the sequential structure of natural language.

Standard metrics used in LLM evaluation, such as BLEU, ROUGE, and character-level F-score, have often been applied to assess code generation despite their misalignment with human expectations of code quality. Evtikhiev et al. [23] discuss these metrics for their limitations in accurately reflecting code performance. In our work, we employ these standard metrics to provide a broad performance estimate, supplementing them with additional measures for comprehensive evaluation.

Furthermore, our analysis incorporates metrics from the CodeXGlue benchmark, such as token-level accuracy, exact match, and Levenshtein distance.

The following sections provide an overview of these metrics and explain their roles in assessing code generation performance. Note that all metrics we use have been used to evaluate code-related tasks, so they should yield a good performance overview.

ROUGE

Rouge scores were introduced by Lin et al. [46] as a method to evaluate automatic text summaries. ROUGE, which stands for Recall-Oriented Understudy for Gisting Evaluation, has since become a widely used metric for assessing LLMs. It measures the overlap of n-grams between a reference (or multiple references) and the predictions, focusing primarily on recall.

One of the most commonly used variants is Rouge-L, which evaluates the longest common subsequence between the reference and generated text, thereby capturing sentence-level structure similarity. The formula for Rouge-L is as follows:

$$\text{Rouge-L} = \frac{(1+\beta^2)\cdot \text{Precision}\cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}} \tag{6.1}$$

In this, Precision refers to the fraction of matched n-grams in the generated text compared to the reference. Recall represents the fraction of matched n-grams in the reference text that appear in the generated text. The parameter β is generally set to 1, giving equal weight to precision and recall.

BLEU

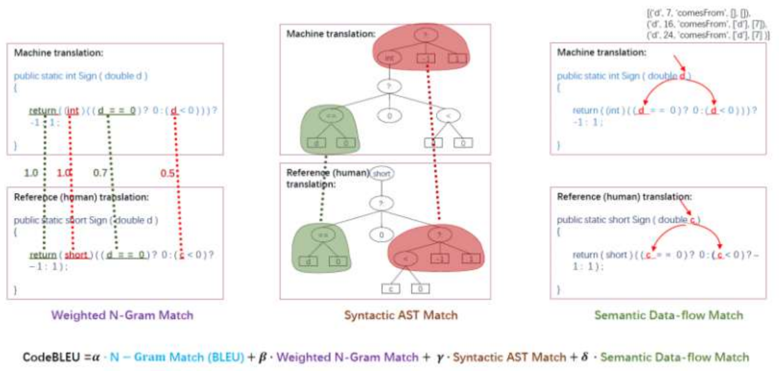
Bilingual Evaluation Understudy (BLEU) is another traditional metric introduced to evaluate machine translations [58]. They designed BLEU to be a metric, which incorporates the fact that good translations share many parts with a reference translation. They define a modified n-gram precision on blocks of text,

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{6.2}$$

Where BP is a brevity-penalty, and p_n and w_n are precision for the n-gram and weights for it. Typically $N = 4$ and the weights are equal $\frac{1}{N}$. Therefore, BLEU calculates the percentage of n-gram matches.

CodeBLEU

Based on BLEU, Ren et al. [66] introduced CodeBLEU, which can be seen as a successor to BLEU that is more tailored for code-related tasks and was initially introduced to evaluate program synthesis.


Machine translation:
1.0
1.0
0.7
0.5
Reference (human) translation:
Weighted N-Gram Match
Machine translation:
Reference (human) translation:
short
Syntactic AST Match
[('d', 7, 'comesFrom', [], []),
('d', 16, 'comesFrom', ['d'], [7]),
('d', 24, 'comesFrom', ['d'], [7])]
Machine translation:
Reference (human) translation:
Semantic Data-flow Match
CodeBLEU = α · N − Gram Match (BLEU) + β · Weighted N-Gram Match + γ · Syntactic AST Match + δ · Semantic Data-flow Match

```
public static int Sign ( double d )
{
    return ( (int ) ( ( d = = 0 ) ? 0 : ( d < 0 ) ) ) ?
    -1 : 1;
}
```

```
public static short Sign ( double d )
{
    return ( short ) ( ( d = = 0 ) ? 0 : ( c < 0 ) ?
    -1 : 1 );
}
```

```
public static short Sign ( double c )
{
    return ( short ) ( ( c = = 0 ) ? 0 : ( c < 0 ) ? -
    1: 1 );
}
```

Figure 6.1: CodeBLEU calculation summary by Ren et al. [66].

Consequently, the authors introduced the metric depicted in Figure 6.1. CodeBLEU consists of four components, each with a typical weight of $\frac{1}{4}$:

1. The initial component mirrors traditional BLEU.
2. The weighted n-gram match assigns greater significance to programming language keywords, as they are often more crucial to predict accurately than ordinary tokens like local variable names.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

3. The syntactic AST match assesses all subtrees within the candidate and reference ASTs of the program, determining the count of matched subtrees. This component includes a metric that checks for syntactic consistency, even if the syntax is reordered in the generated program.

4. Lastly, the semantic data flow analyzes the correspondence of data flows within the generated program.

ChrF

The character n-gram F-score, introduced by Popovic [61], follows the same calculation method as RogueL, except it utilizes character n-grams instead of word n-grams.

$$\text{ChrF} = \left(1 + \beta^2\right) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}} \tag{6.3}$$

ChrF is helpful for code LLMs as it detects subtle variations in code syntax and has been recommended for evaluating code models [23].

Levensthein Distance

The Levensthein distance is a pretty traditional metric in information systems and describes how many changes in a string are needed to be coherent with a given string. It is also discussed in the CodeXGlue [49] benchmark as a metric for evaluating code-completion models.

It can be recursively defined as:

$$d(i,j) = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ d(i-1, j-1), & \text{if } a_i = b_j \\ 1 + \min \begin{cases} d(i-1, j) \\ d(i, j-1) \\ d(i-1, j-1) \end{cases}, & \text{if } a_i \neq b_j \end{cases} \tag{6.4}$$

This metric quantifies dissimilarity between generated and reference code by counting minimal edits needed for an exact match, offering insights into how many changes a developer needs to make.

Exact Match & Token-Level-Accuracy

Token-level accuracy evaluates how correctly each token is predicted given a context. Exact Match (EM) measures the accuracy of generating sequences identical to the reference from start to finish, requiring complete matching. Thus, these two metrics are the most strict ones we employ.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Metric	Min	Max
Rouge	0	1
BLEU	0	1
ChrF	0	100
Levensthein Distance	0	Max #characters
EM	0	1
Token Level Acuraccy	0	1

Table 6.1: Evaluation metrics and min/max values.

To evaluate the performance of the LLMs, we use six distinct accuracy metrics that, while designed for similar goals, each capture different aspects of performance. In addition, CodeBLEU incorporates programming language-specific syntax and semantics into its calculation. Table 6.1 summarizes these metrics along with their respective minimum and maximum values. Note that a lower value is better for Levenshtein distance, while higher values are better for all other metrics.

6.2 Experiment Setup

In the following, we describe the experimental setup used for evaluations, including the software and hardware employed. We also detail the model training process and the design of our experiments.

6.2.1 Hardware and Software Used

We used the GPU server provided by the HPC Group at TU Wien, with the hardware details as seen in Table 6.2.

GPU	CPU	CPU cores	CPU Speed	RAM	VRAM	OS
NVIDIA Quadro RTX 8000	AMD EPYC 7452	32	1.5-2.35 GHz	1.0Ti	49152MiB	Ubuntu 20.04.6 LTS

Table 6.2: Hardware Setup of GPU Server.

Table 6.3 summarizes the most important software used for the experiments. We used the Python programming language for all experiments and implementations since it has the best support for our setup, such as the DL and RL libraries.

Name	Version
Python	3.12
CUDA	12.7
Nvidia-Driver	565.57.01

Table 6.3: Software and Versions used.

In addition to the usual Python libraries—those for handling strings, regex, APIs, and more—we employed the specific libraries and versions detailed in Table 6.4. This table

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

encompasses libraries essential for DL/RL and LLM training and inference, extending to data manipulation, NLP assessment, and energy measurement packages. The column labeled 'purpose' details why each library was utilized. All the models we trained can be found on Huggingface [1].

Name	Version	Purpose
`accelerate`	1.0.0	Utility for accelerating PyTorch models.
`codebleu`	0.7.1	Metric for evaluating code generation.
`contourpy`	1.3.0	Library for contour plotting.
`datasets`	2.20.0	Dataset library for NLP tasks.
`evaluate`	0.4.3	Tool for evaluating NLP model performance.
`gymnasium`	0.29.1	Reinforcement learning environment suite.
`huggingface-hub`	0.23.4	Client library for Hugging Face Hub.
`joblib`	1.4.2	Library for lightweight pipelining in Python.
`Levenshtein`	0.26.1	Library for Levenshtein distance calculations.
`matplotlib`	3.9.0	2D plotting library for Python.
`multiprocess`	0.70.16	Multiprocessing for Python.
`nltk`	3.9.1	Natural Language Toolkit for Python.
`numpy`	1.26.4	Fundamental package for numerical computations.
`nvidia-cuda-*-cu12`	12.1.105	CUDA Tools.
`pandas`	2.2.2	Data analysis and manipulation library.
`rouge_score`	0.1.2	ROUGE score calculations.
`sacrebleu`	2.4.3	BLEU score calculations.
`safetensors`	0.4.5	Safe and fast tensor format.
`scikit-learn`	1.5.2	Machine learning library for Python.
`scipy`	1.14.1	Scientific and technical computing library.
`stable_baselines3`	2.3.2	Reinforcement learning algorithms suite.
`tensorboard`	2.18.0	TensorFlow's visualization toolkit.
`tokenizers`	0.20.1	Fast implementation of tokenizers.
`torch`	2.4.1	PyTorch machine/deep learning library.
`transformers`	4.45.2	Library for natural language processing with Huggingface.
`tree-sitter`	0.23.1	Incremental parsing system.
`zeus-ml`	0.10.1	Energy & Time measurement with GPU support.

Table 6.4: Most important Python packages used.

6.2.2 Hyperparameter Settings

For fine-tuning our models, as introduced in Section 4.4 and analyzed in Section 4.4.1, we used standard hyperparameters selected to meet memory and computational constraints. These settings, summarized in Table 6.5, include many defaults of the Hugging Face training API [75].

Due to significant differences in dataset sizes, we adjusted the training process: one epoch for PY150 (100k samples) and five epochs (OPT) or 10 epochs (Llama) for JavaCorpus

[1] `https://huggingface.co/briemelchen` (currently listed private), last accessed 25-11-24

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

(15k samples). To handle variability in training sample sizes, we split samples by maximum sequence length and used packing when necessary. We performed none to minimal hyperparameter tuning and recognized the significant potential for optimizing fine-tuning, particularly for intermediate layer decoding, as discussed in Section 4.4. However, this was excluded due to the resource-intensive nature of the task, which required at least two days of computation per iteration.

Hyperparameter	**Value**	**Description**
`epochs`	1 (PY150), 5/10 (JavaCorpus)	Number of training epochs.
`batch_size`	4	Batch size for training.
`bf16`	False	Use bfloat16 precision during training.
`fp16`	True	Use float16 precision during training.
`gradient_accumulation_steps`	32	Number of gradient accumulation steps before updating weights.
`context_length`	256	Maximum length of input context for training.
`learning_rate`	1e-5	Learning rate for the optimizer.
`Optimizer`	AdamW	Adam with weight decay regularization
`AdamW` β_1	0.9	Exponential decay rate for the first-moment estimate (mean of gradients)
`AdamW` β_2	0.99	Exponential decay rate for the second-moment estimate (variance of gradients)
`max_gradient_norm`	1	maximum allowed value for gradient norm, helps to prevent exploding gradients
`weight_decay`	0.01	Weight decay for regularization during training.
`lr_scheduler_type`	`constant`	Learning rate scheduler type (constant).
`max_seq_length`	256	Maximum sequence length for tokenized inputs.
`packing`	True	Whether to enable packing of sequences.

Table 6.5: Hyperparameters for LLM fine-tuning.

We present the hyperparameters used for training the RL agents in Table 6.6. The primary algorithm was PPO, with most settings based on standard parameters from the RL library and best practices [2]. Due to time constraints, we did not extensively tune hyperparameters, opting for configurations that achieved reasonable convergence within a few hundred thousand steps.

We limited the network hidden size to 32 or 64 and used 1 or 2 layers to (i) minimize inference overhead, (ii) reduce the risk of overfitting, and (iii) achieve faster learning.

Deep RL algorithms are highly sensitive to hyperparameters, and the results are often challenging to reproduce [32, 22]. More optimal configurations might significantly improve performance. The convergence results and the performance analysis of the RL agent are provided in Section 6.3.

When evaluating, we use the softmax output of the policy network to decide which action to use, therefore following a Boltzmann distribution [74]. By adjusting the temperature parameter and softmax thresholds , we can control the balance between exploration/exploitation during inference. Softmax thresholds and temperature adjustments are standard techniques in modern neural networks, as discussed by Guo et al. [31]. In other words, these thresholds can control how "sure" the agent must be before deciding to exit (e.g., a threshold of 0.9 is much stricter than a threshold of 0.5). We continuously report results with different thresholds.

[2]`https://github.com/EmbersArc/PPO/blob/master/best-practices-ppo.md`, accessed 15-11-24

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Hyperparameter	Value	Description
`steps`	500000	Number of training steps.
`batch_size`	512,32	Number of experiences per gradient update.
`Buffer Size`	4096, 256	Number of experiences collected before gradient update
`Epochs`	6,2	Number of epochs through the buffer when updating gradients.
`learning_rate`	5e-5, 1e-4	Learning rate.
`lr_scheduler_type`	`linear`	Linear learning rate scheduled over the steps.
γ	0.99	Discount factor
`Number of hidden layers`	2	How many hidden layers to use
`Number of hidden units`	32, 64	How many hidden units in each hidden layer to use

Table 6.6: Hyperparameters for PPO training.

6.2.3 Experiments Design

For evaluation in the following experiments, we primarily focused on line-completion tasks. This means that, given a code context, the objective is to complete the next LOC, similar to how tools like Copilot suggest code completions. According to the CodeXGlue Github repository [3], the average output length for line-level completion is approximately seven tokens. In addition, related work on code completion in early exiting uses 10 tokens [73]. Therefore, in most experiments, we set the maximum number of tokens generated by the model to 15, ensuring that a complete LOC is completed. This choice also enables more comprehensive comparisons, as complex metrics such as ROUGE-L or BLEU can result in zero when only few tokens are generated. We refer to the maximum number of tokens as `max_new` in the experiments.

For experiments, we always split the code file into two parts given a specified context length, keeping in mind the observations discussed in Section 4.2. The split is made uniformly across the samples, with, for example, a fraction of 0.2 indicating that the first 20% of the tokens of a sample is used as context. This results in variable context lengths for different samples due to the variety of file lengths in the datasets. However, we limited the maximum context length to 512 tokens to speed up evaluations and avoid memory constraints. This choice is justified because the average number of tokens used in CodeXGlue benchmarks is 488 for PY150 and 365 for JavaCorpus; however, they used random context splits. In subsequent experiments, we will report results on different context fractions.

For evaluating results, we give the model the first n tokens of a sample as input and use the tokens from the dataset in the range $[n+1, n+max\_new]$ as ground truth labels. Then, we calculate corpus-level metrics by comparing the ground-truth tokens against the tokens generated by the model. We always evaluate 1000 samples of the test sets from the datasets. We always use *greedy decoding* when predicting with the LLMs to have comparable and reproducible results; that is, we do not use any inference optimizations such as beams or sampling.

[3]`https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/CodeCompletion-line`, accessed 17-11-24

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

To illustrate our experiment setup, Listing 6 shows a context and prediction of EE-enhanced Llama on PY150, given 20% context and asking the model to generate 15 tokens.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
from south.db import db
from django.db import models
from django_lean.experiments.models import *

class Migration:
    def forwards(self, orm):
        db.create_table(
            '',
            (
                ('id', orm['']),
                ('name', orm['']),
                ('state', orm['']),
                ('start_date', orm['']),
                ('end_date', orm['']),
            )
        )
        db.send_create_signal('experiments', ['Experiment'])

        db.create_table(
            '',
            (
                ('id', orm['']),
                ('user', orm['']),
                ('experiment', orm['']),
                ('', orm['']),
                ('group', orm['']),
            )
        )
        db.send_create_signal('experiments', ['Participant'])

        db.create_table(
            '',
            (
                ('id', orm['']),
                ('date', orm['']),
                ('experiment', orm['']),
                ('test_score', orm['']),
                ('control_score' # context end
            # prediction of EE-Llama: , orm [ '' ] ), <EOL> ) ) <EOL> db.
```

Listing 6: Context & prediction example on PY150 with EE enhanced Llama (special characters in context removed for readability).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

6.3 RL Agent Evaluation

Figure 6.2 shows the mean reward per step over episodes for PPO agents trained over 500,000 steps. These curves represent the policies used in subsequent evaluations. The x-axis (episodes) varies across models and training runs due to differences in episode lengths, influenced by model and dataset specifics (e.g., OPT has more exit points than Llama). For JavaCorpus, training used coefficients β and γ set to 1, while for PY150, they were adjusted to 0.5 to account for the higher token-level accuracy of lower layers on this dataset (see Section 4.4).

The plots demonstrate typical reinforcement learning training dynamics: agents initially perform poorly, but their performance improves rapidly after periods of exploration. Around 2,000 to 2,500 episodes, they reach convergence, with only small variations in rewards observed thereafter.

Notably, agents trained on Llama hidden states appear more stable with fewer fluctuations than those trained on OPT models. The exact cause is unclear, but the stochastic nature of the environment, driven by random samples of data sets and variability in optimal exit layers, likely contributes to reward inconsistencies. Despite this, clear convergence behavior is observed within the 500,000-step training limit.

Another noteworthy observation is that the maximum rewards achieved are consistently higher when training on Llama models. One possible explanation is the distribution of optimal exit layers during training. Figure 6.3 illustrates the optimal exits for all samples encountered during the RL agent's training. An optimal exit is defined as the first layer that aligns with the last layer of the model.

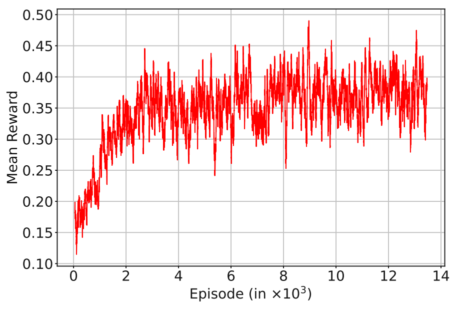
The figure reveals that, with Llama, earlier layers (that is, 59% for optimal exits for the first five layers) are more frequently optimal compared to OPT (50%). This difference may influence learning dynamics. A potential reason for this is the disparity in the number of layers: OPT has 32 layers, while Llama has 28. As a result, during fine-tuning, OPT may distribute weight across more layers, leading to less emphasis on earlier ones, and therefore, there is a shift when training the RL agents; furthermore, the Llama model is more capable overall.

We now evaluate the trained agent's performance by examining how well its exits align with successful token predictions. A successful exit occurs when the predicted token matches the final layer's output. For this, we conducted an experiment in which the context length was randomly set between 0.2 and 0.6, and the model generated 15 tokens. Using the hidden states, the RL model predicted the exit points.

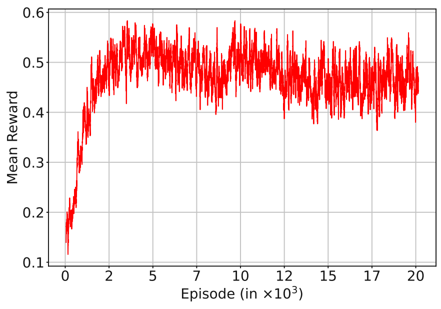
Figure 6.4 shows the results for two Llama settings: one in JavaCorpus ($T = 0.6$, Figure 6.4a) and one with an extreme setting on PY150 ($T = 0.94$, Figure 6.4b). The lighter bar colors indicate layers chosen more frequently as exit points.

Even with low thresholds, the agent achieves about 80% accuracy in selecting good exits. For example, at layer 3, 8,000 exits occur with 82% accuracy. Exits are rare between

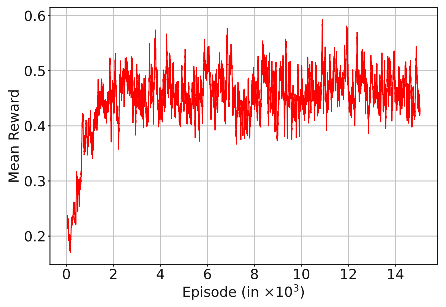
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



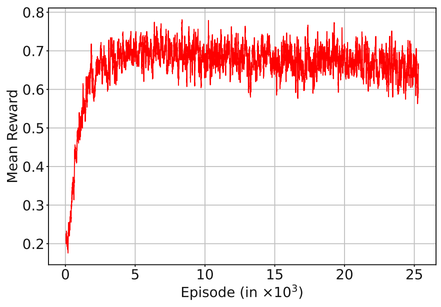
(a) OPT, JavaCorpus.



(b) Llama, JavaCorpus.

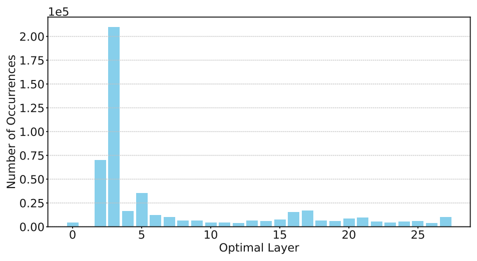


(c) OPT, PY150.

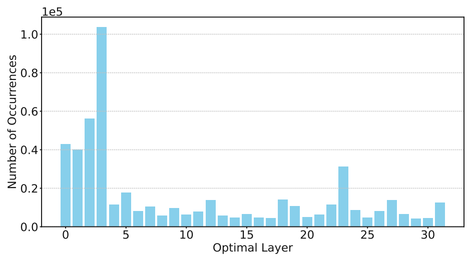


(d) Llama, PY150.

Figure 6.2: Mean step reward per episode of PPO training with Llama and OPT. (moving average over 50 episodes).



(a) LLama.



(b) OPT.

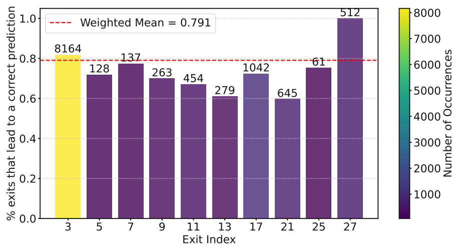
Figure 6.3: Occurrences of optimal exits during training of RL agent (JavaCorpus).

layers 5 and 13 but increase at layer 17, with 1,000 exits, and fall of again at layers 21 and 25. This distribution aligns with the agent's training data, where layers 16 and 17 show higher optimal exit distributions (Figure 6.3a), indicating effective learning and generalization. However, there are discrepancies, such as only 128 exits in layer 5, despite more optimal exits during training.

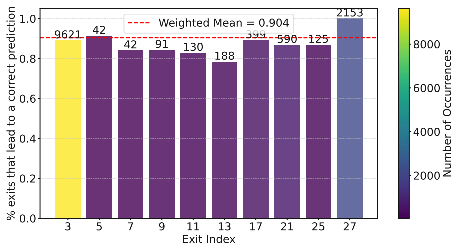
In Figure 6.4b, the PY150 dataset uses a stricter threshold (0.94), which results in an

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

overall accuracy of 90%. Most tokens exit at layer three or the final layer, with few intermediate exits. This setup favors early exits, as optimal exits are more common in earlier layers compared to JavaCorpus. With per-layer accuracies between 78% and 91%, the agent demonstrates overall strong performance.



(a) JavaCorpus, $T = 0.6$.



(b) PY150, $T = 0.94$.

Figure 6.4: Percentage of successful exits of the RL Agent on Llama (1000 samples, random context, maximum 15 tokens generated).

In summary, this section discussed how agents can learn within our environment and reward function and show convergence behavior during training. Additionally, we briefly showed how the agent learns data patterns and how its behavior can be influenced by adjusting the softmax threshold, which can be used to make trade-offs between accuracy and energy consumption.

6.4 Analysis of Code Generation Task

In this section, we present evaluations of our method, which integrates the RL agent into the LLM inference pipeline to decide when to exit early dynamically. We begin by analyzing the performance on both datasets, followed by a discussion of the computational overhead introduced. Lastly, we discuss some results from experiments that incorporate KV cache propagation, while the other results are presented without KV caching.

6.4.1 Performance Evaluation on JavaCorpus Dataset

We begin with Figure 6.5, which presents metrics related to model performance (Figure 6.5a) alongside metrics related to resource consumption (Figure 6.5b), in 1000 JavaCorpus samples and context 20% as input to the model. The model's output is at most 15 tokens long. "syntax" and "dataflow" refers to the syntax and dataflow sub-metrics of CodeBLEU, which we include here, as it is reasonable to explore them further during our assessment of the code completion task. These results are from an experiment using the fine-tuned Llama model enhanced with RL agent that determines when to exit. The figure includes results for various agent thresholds (T) and baseline comparisons: the base model (i.e., the non fine-tuned version) and the fine-tuned model evaluated using all layers, i.e., without early exits.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Looking at the scores achieved by the two baselines, we can observe that the base model performs slightly better than the fine-tuned one in terms of accuracy for all metrics except CodeBLEU, where the fine-tuned model performs marginally better—the better syntax and dataflow scores influence this. Furthermore, the energy/time of these two models differ only marginally, as expected, since they use the same number of layers.

If we look closer at the scores achieved using the LLM with early exiting, we can observe that it performs quite well with low exit thresholds. For example, with $T = 0.6$, the model achieves RougeL of about 0.29, while the full model achieves about 0.42 using less than half of the energy or time. Furthermore, we can see that accuracies can be increased by trading off some of the computational resources. For example, in the most aggressive setting ($T = 0.92$), the EE model achieves a RougeL of about 0.41, while the full model achieves around 0.43. It does this while still saving significant resources, as visible in the plots; it requires approximately 23% less energy and time. Moreover, the throughput of the base models is roughly 13 tokens/second, compared to 18 tokens/second in this setting. Additionally, with a more "balanced" threshold of $T = 0.9$, we can still achieve good performance (e.g., RougeL of 0.39) while saving even more resources ($\sim$ 32%). Therefore, different configurations allow adjustments according to user demands.

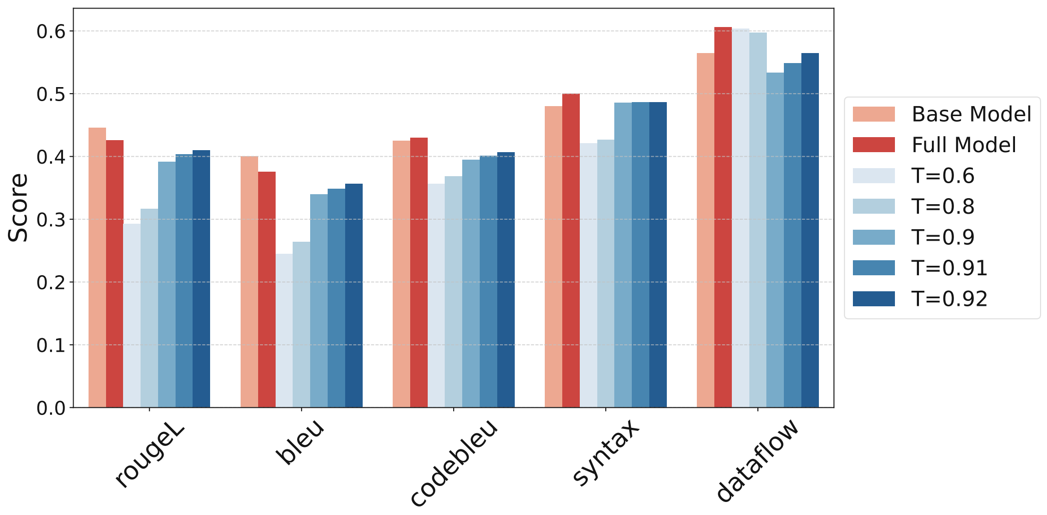
An interesting observation from these plots is the behavior of syntax and dataflow scores. While standard NLP metrics like ROUGE-L and BLEU show gradual improvements with increasing exit thresholds, the syntax-match score tends to plateau, showing limited improvement compared to other metrics. For the dataflow score, resource-efficient settings achieve higher scores initially (consistent with the full model), but the score drops before rising again.

In summary, it can be seen that under the most aggressive configurations (higher T values), the EE-enhanced Llama conserves $\geq$ 23% of energy, with only a 1-5 point decrease in accuracy metrics. When using lower thresholds, the accuracy deficits become more pronounced (up to a reduction of 0.12), which allows energy savings of up to 55%.

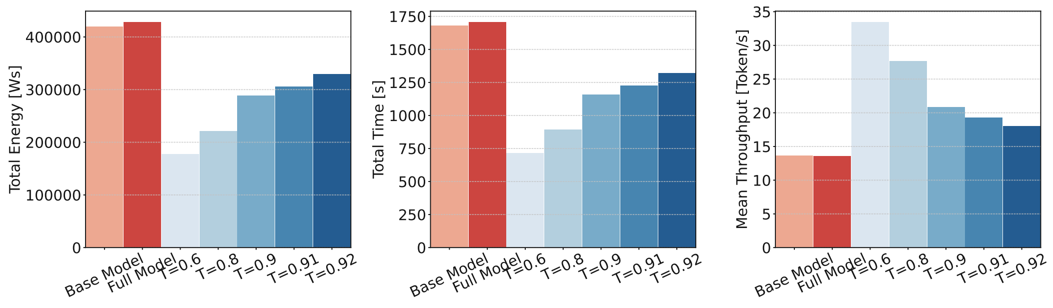
In larger contexts, as shown in Figure 6.6, where 60% of the code file is provided as input, syntax and dataflow scores align more closely with other metrics. This highlights the importance of experimenting across different sections of a code file: 20% corresponds to the beginning (near imports), while 60% represents the middle. Although overall patterns are similar, the fine-tuned full model performs worse in larger contexts than in smaller ones. For instance, in the "most accurate" setting, the full model achieves a ROUGE-L score of 0.447, while the early exit model scores 0.413, with similar trends in other metrics. The savings are similar overall, with roughly 25% less energy and time in the most expensive setting and about 56% on the lowest thresholds.

The results and findings for OPT are generally comparable to those discussed for Llama. Figure 6.7 presents the results for OPT in a context of 20%, while Figure 6.8 shows the results with 60% tokens as input to the model: it can achieve performance similar to the full-layer model by "trading off" enough computational resources. Furthermore, we see that the early exit model performs particularly well in smaller contexts, where it

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Score metrics (RougeL, CodeBleu, Bleu)



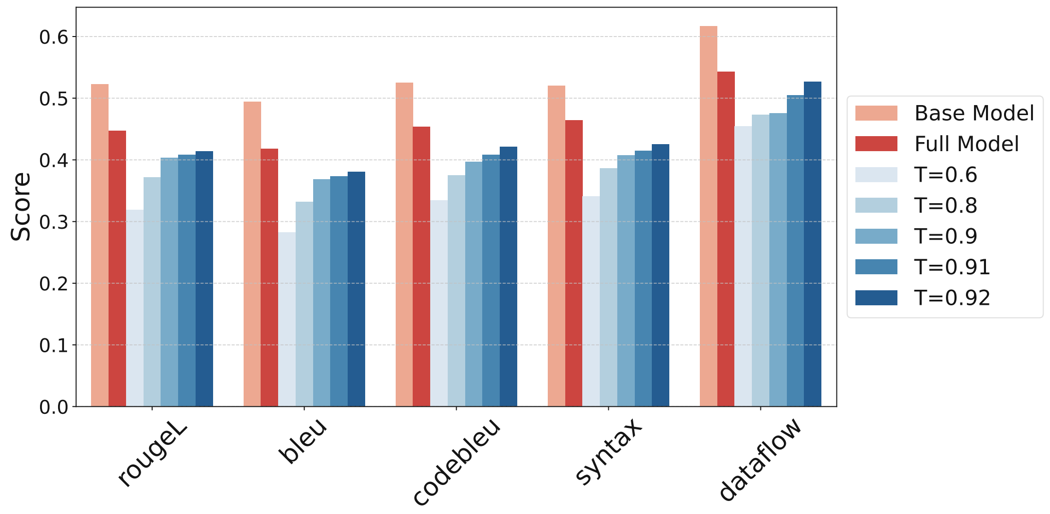
(b) Resource Consumption metrics (Energy, Time, Throughput)

Figure 6.5: Results of Llama on JavaCorpus, generating maximum 15 tokens with different RL agent thresholds T and baselines. (1000 samples, 20% context).

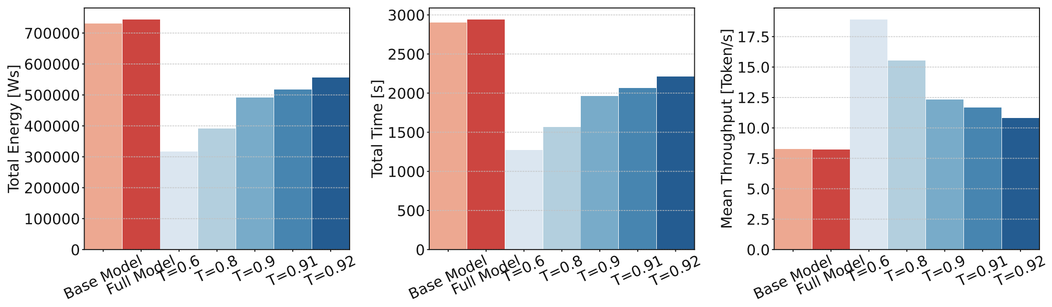
even matches the full-layer model in terms of RougeL and BLEU scores while still saving 27% of energy consumption. However, in the 60% setting, the early exit model performs worse compared to the lower contexts, as score metrics are off up to 0.1.

Finally, we present the ChrF score and the Levenshtein distance in Figure 6.9, evaluated under the same settings as previously discussed. The baseline performance of the full model is presented in these plots in the same color as the early exit model as a horizontal line. The trends in these plots align with our earlier observations. Specifically, OPT at the 20% context setting performs surprisingly well, achieving results that are at least as good as the full model- indeed, the early-exit model is even a little bit better in the Levensthein distance setting than the full model. In contrast, Llama approaches the performance of the full model but does not match the full performance, even at the highest threshold setting. In particular, in the context setting 60%, the differences for Llama are less pronounced in relative terms, showing behavior similar to the setting 20%.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Score metrics (RougeL, CodeBleu, Bleu)



(b) Resource Consumption metrics (Energy, Time, Throughput)

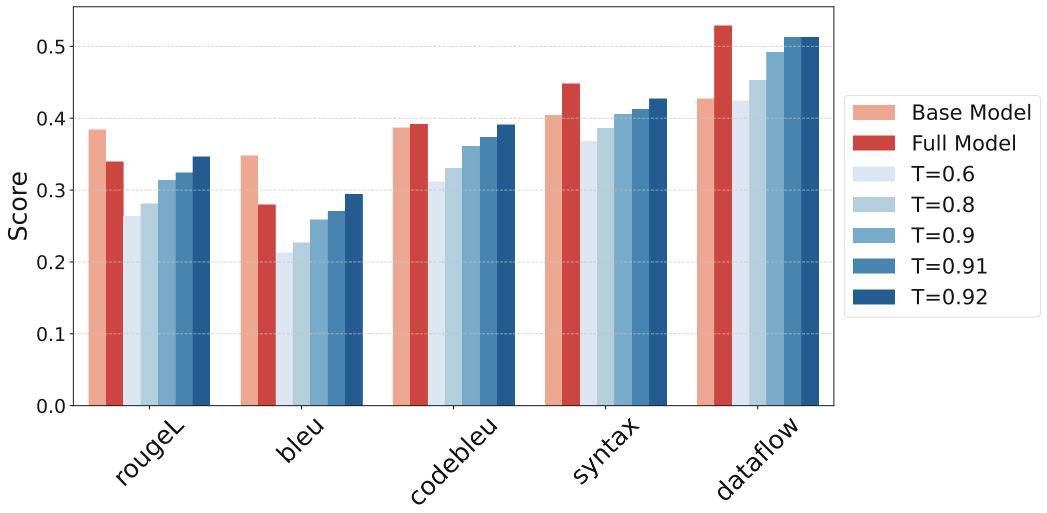
Figure 6.6: Results of Llama on JavaCorpus, generating maximum 15 tokens with different RL agent thresholds T and baselines. (1000 samples, 60% context).

However, in this case, OPT experiences a more significant performance drop compared to the context setting of 20%, indicating reduced effectiveness at higher context levels.

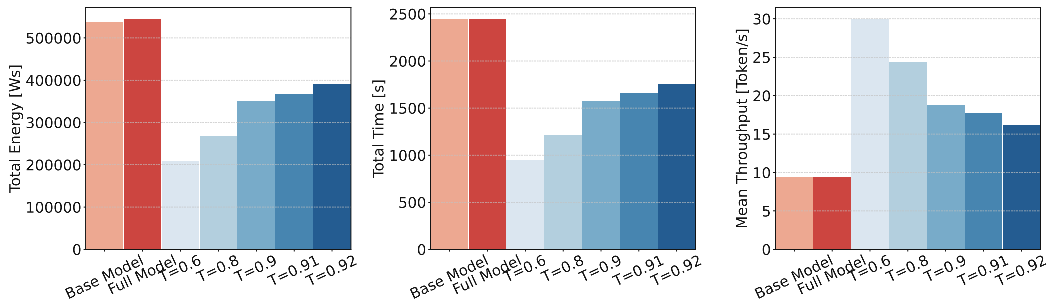
In summary, our method demonstrated the ability to reduce resource consumption (in energy and time) by 20%-55% with minimal loss to moderate in model accuracy, depending on the chosen RL-agent thresholds. The models sacrificed up to 0.2 in accuracy-related metrics at most and no accuracy losses in the most aggressive setting (OPT, 20% context). By specifying RL-thresholds, we are able to trade-off those accuracy losses by using more resources. We presented results across various context settings and observed better performance in smaller context configurations. Additionally, the results were generally consistent across both models; however, OPT performed slightly better than Llama, as it matched the full model's accuracy in some settings, whereas Llama showed deviations of up to 0.02 points in accuracy-related metrics.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Score metrics (RougeL, CodeBleu, Bleu)



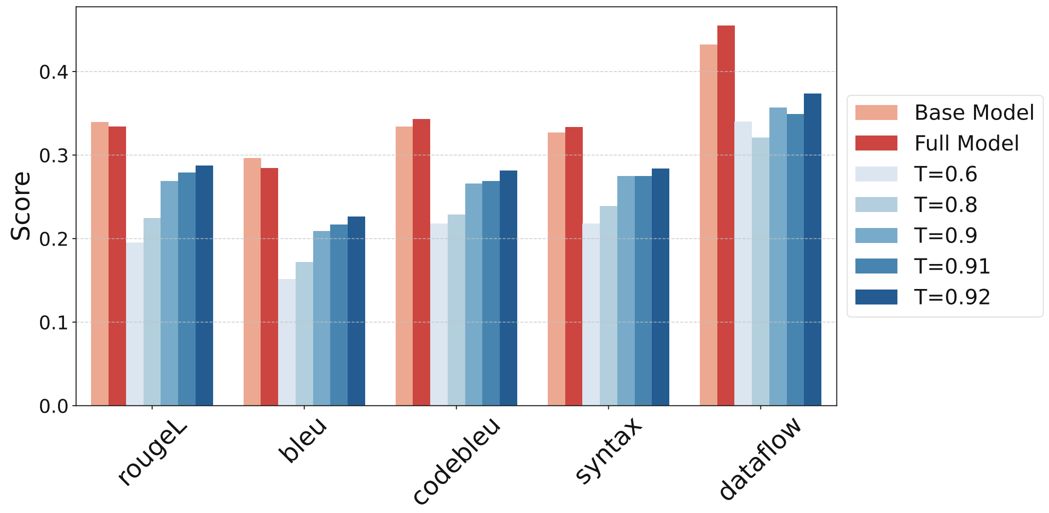
(b) Resource Consumption metrics (Energy, Time, Throughput)

Figure 6.7: Results of OPT on JavaCorpus, generating maximum 15 tokens with different RL agent thresholds T and baselines. (1000 samples, 20% context).

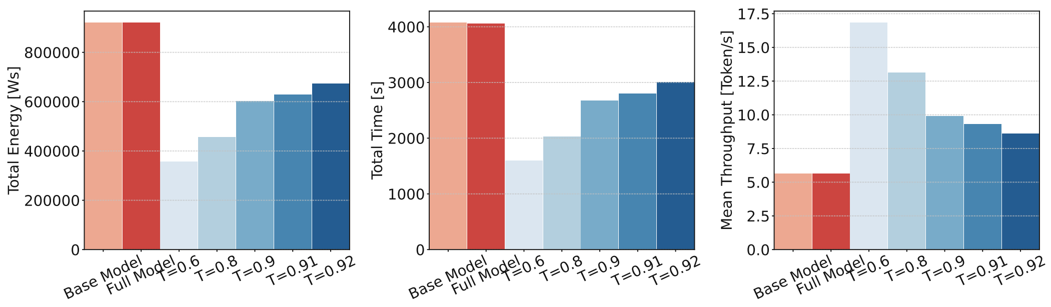
6.4.2 Performance Evaluation on PY150 Dataset

Figure 6.10 presents the results for Llama, while Figure 6.11 shows the results for OPT on 20% setting. Overall, the findings are aligned with those observed for the JavaCorpus dataset. For example, with Llama (Figure 6.10), we observe pretty reasonable performance across all metrics, achieving scores such as 0.46 vs. 0.44 for RougeL and 0.403 vs. 0.356 for CodeBLEU in the "most aggressive" setting ($T = 0.92$), while saving significant energy of approximately 29%.

In contrast to the results observed with the JavaCorpus dataset, we observe that variations among higher thresholds for the agent are less pronounced. This indicates that performance has reached a saturation point in this context, and additional threshold increases are unlikely to significantly enhance accuracy, as decisions are already made with a high level of confidence.



(a) Score metrics (RougeL, CodeBleu, Bleu)



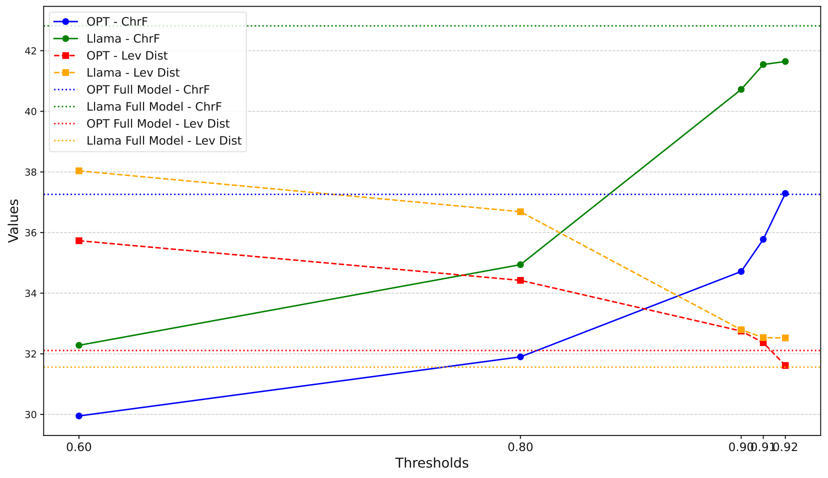
(b) Resource Consumption metrics (Energy, Time, Throughput)

Figure 6.8: Results of OPT on JavaCorpus, generating maximum 15 tokens with different RL agent thresholds T and baselines. (1000 samples, 60% context).

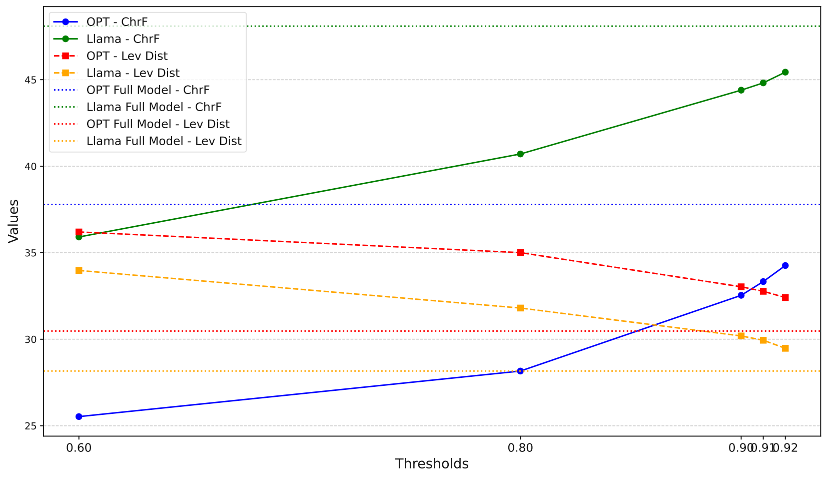
This effect is less pronounced in the OPT experiment, as shown in Figure 6.11. Although the EE-enhanced model achieves reasonable overall results, its performance is slightly worse than Llama's. For instance, the full model achieves a RougeL score of 0.37, while the EE model reaches a maximum of 0.33. Additionally, the resource-related metrics reveal that, on the Python dataset, the OPT-enhanced EE model relies more heavily on early exits compared to its performance on Java or Llama's performance on the datasets. This is particularly evident in energy and time savings ($\sim$ 40% savings on OPT, $\sim$ 30% on Llama).

Another notable observation is that the CodeBLEU metric is generally lower on the PY150 dataset compared to other metrics and relative to the evaluation on the JavaCorpus dataset. One potential reason for this discrepancy is Python's reduced syntax complexity and the way we process Python code, specifically by including a special token for newlines. This approach may cause a mismatch between the abstract syntax tree (AST) structures

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub



(a) 20% context



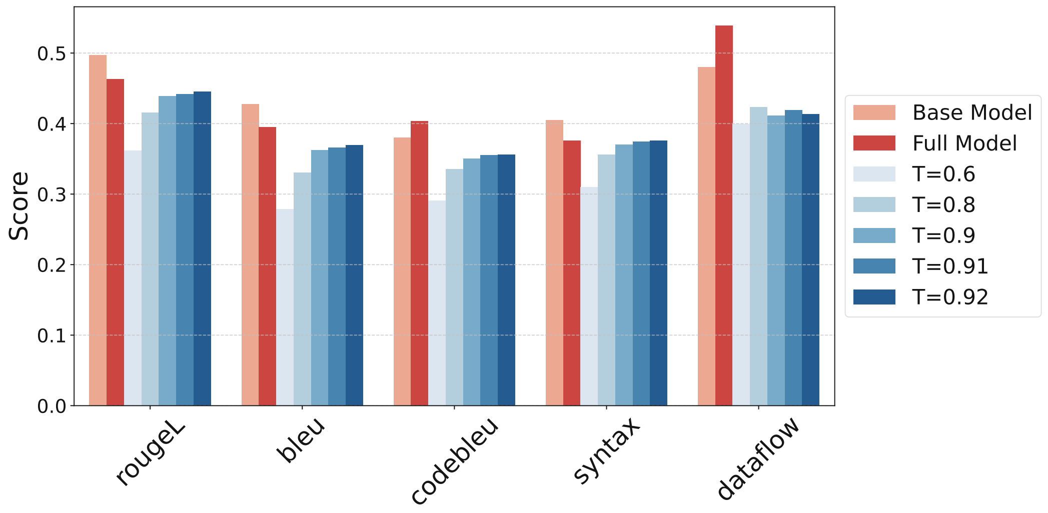
(b) 60% context.

Figure 6.9: ChrF and mean Levensthein Distance on JavaCorpus, generating maximum 15 tokens with different RL agent thresholds T and baselines (over 1000 samples).

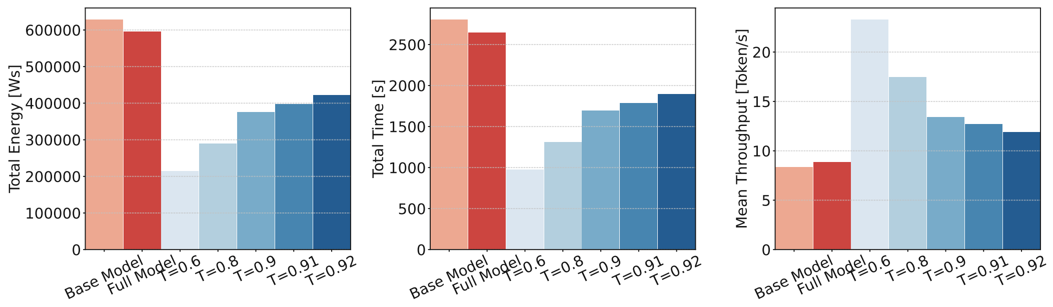
of Python programs and the expectations of the CodeBLEU implementation.

In summary, the results on the PY150 dataset are comparable to those on the JavaCorpus. Energy savings ranged from approximately 20% to 50%, with a loss of 0.02 to 0.2 in absolute accuracy-related metrics. On the PY150 dataset, Llama slightly outperformed OPT in the most accurate settings. Additionally, we observed that OPT relied more heavily on earlier exits than Llama, resulting in greater resource savings (around 10% more in the most accurate settings). This could explain its slightly lower performance compared to Llama. Moreover, higher thresholds had a less pronounced impact on performance on the PY150 dataset compared to what was observed with the JavaCorpus.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Score metrics (RougeL, CodeBleu, Bleu)

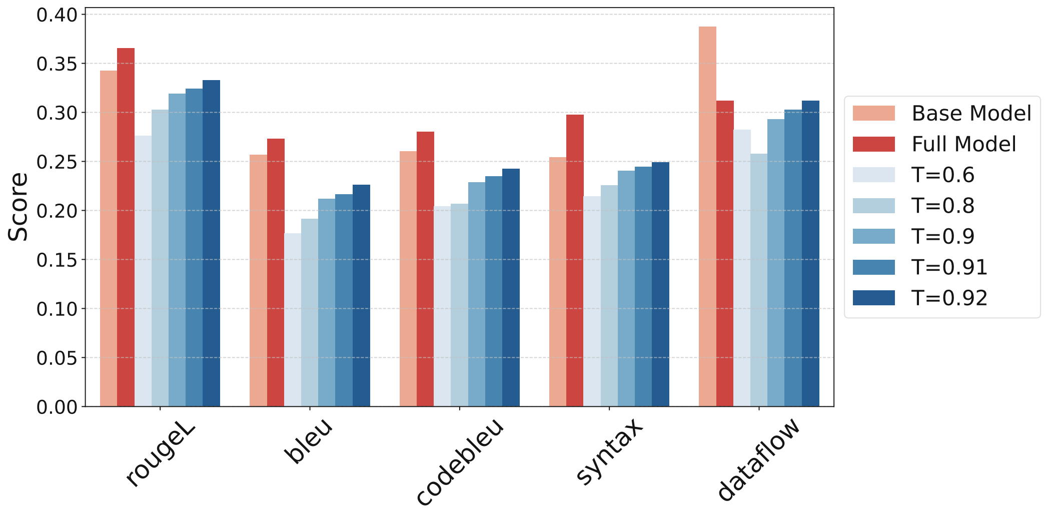


(b) Resource Consumption metrics (Energy, Time, Throughput)

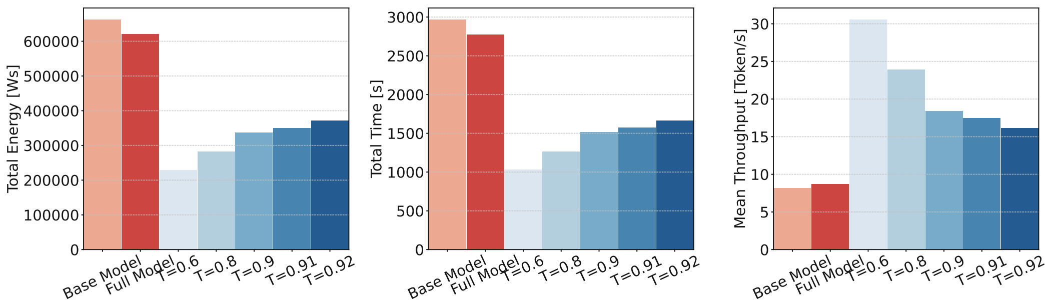
Figure 6.10: Results of Llama on PY150, generating maximum 15 tokens with different RL agent thresholds T and baselines. (1000 samples, 20% context).

Given that other Figures display comparable behavior to those previously shown, we have not included them here to maintain brevity. In summary, we provide tables presenting detailed numerical scores across various metrics for all four experimental settings. The JavaCorpus results are shown in Table 6.8 for Llama 3.2 and Table 6.7 for OPT, while the results for PY150 are shown in Table 6.9 for Llama 3.2 and Table 6.10 for OPT. An observation we can make here is that, on average, the exits occur at lower layers for the Python dataset. This is likely due to what we have observed: The lower layers exhibit higher accuracies in the Python dataset, and the agents adapt to take advantage of this. Furthermore, increasing the thresholds does not yield substantial benefits in some scenarios. For example, on the LLaMA model with the PY150 dataset under the 0.2 context setting, the performance scores remain nearly unchanged for thresholds greater than or equal to 0.9. However, this comes at the cost of using approximately three additional layers, with no notable improvement in the results. We also observe

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Score metrics (RougeL, CodeBleu, Bleu)



(b) Resource Consumption metrics (Energy, Time, Throughput)

Figure 6.11: Results of OPT on PY150, generating maximum 15 tokens with different RL agent thresholds T and baselines. (1000 samples, 20% context).

that the exact match score is generally quite low (maximum 17%) overall. To better understand this, we manually analyzed several predictions. Many examples are inherently challenging to predict accurately, especially in the code completion setting. For instance, consider Listing 7, which illustrates a prediction within the constructor (`__init__` method) of a Python class. In this example, `_proxy` is a parameter to the method, and the ground truth sets `self.proxy = _proxy`. However, the prediction uses `self._proxy` instead. Accurately predicting this subtle difference is nearly impossible, as all other variable definitions before the prediction reuse the parameter name without an underscore. Consequently, predicting the exact ground truth in such cases is not realistically feasible.

This section analyzes experimental results, highlighting how our proposed method compares to the full model. Key observations from these experiments include the following:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Performance Trade-offs with Aggregated Loss Fine-tuning : We observed that aggregated loss fine-tuning, while necessary for enabling dynamic early exits, slightly degrades the full model's performance. This effect becomes more pronounced in settings with higher context, where the model struggles to maintain the same accuracy as the full model. Furthermore, we did not compare it to a "standard" fine-tuned model, where results could be even more significant.

Energy and Time Efficiency with Minimal Metric Loss: In the most optimal settings, our dynamic early exit model achieves significant resource savings, reducing energy and time consumption by approximately 30%. Importantly, this efficiency gain comes at a relatively small cost regarding accuracy. For example, on some metrics such as RougeL, the early exit model achieves a score of 0.46 compared to 0.50 in the full model—a modest reduction given the substantial resource savings. Further, we have seen that even more resource-friendly settings can be used by trading in some more of the accuracy (up to $\sim 56\%$ energy savings).

Trade-offs through RL Agent : We demonstrated how the RL agent, used during inference, enables flexible optimization by adapting thresholds when selecting actions. This mechanism allows us to balance resource consumption and prediction accuracy. By defining and using these thresholds, users and developers can "trade in" computational resources to achieve higher accuracy or conserve resources while accepting performance trade-offs. We have provided a set of working thresholds, which can bring even better results when engineering more such thresholds.

Comparison across models and datasets: Overall, we have seen consistent results in the four settings we used (2 models, two datasets). However, slightly different results can be seen in relative terms between these experiments: for example, OPT on JavaCorpus achieves the same results as the full model in a setting (while still saving energy), which we did not achieve with Llama. Further, we see some different behavior between the datasets; for example, CodeBleu is always lower than other metrics, which is the opposite on Java. Since the way CodeBleu is designed and implemented, this could be due to how we use Python files (e.g., with special tokens marking newlines).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Table 6.7: Summary of resource and accuracy-related metrics of OPT on JavaCorpus. 1000 Samples, generating 15 tokens, different contexts.

Context	Thresh	CodeBLEU	RougeL	ChrF	EM	Levenshtein Dist.	Total Energy [Wh]	Total Time [s]	Mean Layers
Full Model									
0.2	/	0.39	0.34	37.26	0.03	32.11	151.18	2443.74	32.00
0.3	/	0.35	0.31	35.77	0.03	33.50	187.46	3006.33	32.00
0.6	/	0.34	0.33	37.79	0.05	30.47	255.56	4057.71	32.00
Early Exit Model									
	0.6	0.31	0.26	29.95	0.00	35.73	57.89	947.84	10.79
	0.8	0.33	0.28	31.90	0.00	34.42	74.49	1213.73	14.29
0.2	0.9	0.36	0.31	34.72	0.01	32.75	97.16	1576.32	18.95
	0.91	0.37	0.32	35.78	0.02	32.37	102.19	1657.37	20.09
	0.92	0.39	0.35	37.29	0.03	31.61	108.55	1759.19	21.52
	0.6	0.26	0.21	26.40	0.01	37.98	71.04	1150.87	10.85
	0.8	0.27	0.22	28.44	0.01	36.88	91.75	1479.67	14.50
0.3	0.9	0.30	0.26	32.23	0.01	35.18	120.31	1935.59	19.47
	0.91	0.32	0.27	33.40	0.02	34.66	126.90	2040.17	20.68
	0.92	0.33	0.29	34.35	0.02	34.13	134.96	2168.40	22.19
	0.6	0.22	0.20	25.53	0.00	36.20	99.22	1592.24	11.14
	0.8	0.23	0.22	28.16	0.01	35.01	126.89	2030.24	14.85
0.6	0.9	0.27	0.27	32.54	0.02	33.03	167.20	2669.47	20.08
	0.91	0.27	0.28	33.33	0.03	32.77	174.68	2803.23	21.24
	0.92	0.28	0.29	34.26	0.03	32.41	187.05	3009.06	22.94

Table 6.8: Summary of resource and accuracy-related metrics of LLama3.2 on JavaCorpus. 1000 Samples, generating 15 tokens, different contexts.

Context	Thresh	CodeBLEU	RougeL	ChrF	EM	Levenshtein Dist.	Total Energy [Wh]	Total Time [s]	Mean Layers
Full Model									
0.2	/	0.43	0.43	42.82	0.05	31.56	118.78	1704.59	28.00
0.3	/	0.40	0.39	41.15	0.07	33.38	147.36	2108.40	28.00
0.5	/	0.44	0.41	45.50	0.10	30.73	191.07	2722.95	28.00
Early Exit Model									
	0.6	0.36	0.29	32.28	0.01	38.04	49.17	715.92	7.44
	0.8	0.37	0.32	34.94	0.03	36.69	61.37	892.03	10.22
0.2	0.9	0.39	0.39	40.72	0.04	32.79	80.13	1158.65	14.74
	0.91	0.40	0.40	41.54	0.04	32.53	84.83	1227.10	16.02
	0.92	0.41	0.41	41.64	0.04	32.52	91.44	1319.94	17.59
	0.6	0.30	0.26	30.26	0.02	39.25	62.88	913.68	7.90
	0.8	0.33	0.29	33.32	0.03	37.34	78.85	1139.41	10.89
0.3	0.9	0.37	0.35	38.27	0.05	34.95	100.14	1444.25	15.14
	0.91	0.38	0.36	38.83	0.05	34.88	105.09	1516.33	16.24
	0.92	0.38	0.37	39.40	0.06	34.59	113.08	1630.33	17.76
	0.6	0.32	0.29	34.32	0.02	36.19	81.04	1172.77	8.17
	0.8	0.36	0.34	38.67	0.04	33.96	100.25	1445.37	11.02
0.5	0.9	0.40	0.38	42.58	0.07	32.29	126.61	1820.94	15.05
	0.91	0.40	0.39	43.14	0.07	31.97	133.50	1917.16	16.13
	0.92	0.40	0.40	43.53	0.08	31.43	143.77	2063.59	17.60

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
# imports omited for clearity
class HTTPConnectionPool(ConnectionPool, RequestMethods):
scheme = 'http'
ConnectionCls = HTTPConnection

def __init__(
    self,
    host,
    port=None,
    strict=False,
    timeout=Timeout.DEFAULT_TIMEOUT,
    maxsize=1,
    block=False,
    headers=None,
    _proxy=None,
    _proxy_headers=None
):
    ConnectionPool.__init__(self, host, port)
    RequestMethods.__init__(self, headers)
    self.strict = strict
    if not isinstance(timeout, Timeout):
        timeout = Timeout.from_float(timeout)
    self.timeout = timeout
    self.pool = self.QueueCls(maxsize)
    self.block = block

    # ground truth prediction:
    self. proxy = _proxy
    self. proxy_headers = _proxy

    # prediction of EE-Llama:
    self. _proxy = _proxy
    self. _proxy
```

Listing 7: Ground truth vs prediction example (EE-enhanced LLama).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 6.9: Summary of resource and accuracy-related metrics of Llama on PY150. 1000 Samples, generating 15 tokens, different contexts.

Context	Thresh	CodeBLEU	RougeL	ChrF	EM	Levenshtein Dist.	Total Energy [Wh]	Total Time [s]	Mean Layers
Full Model									
0.2	/	0.40	0.46	46.20	0.13	23.21	165.49	2644.96	28.00
0.3	/	0.39	0.49	48.94	0.14	21.84	198.88	3168.89	28.00
0.5	/	0.42	0.51	51.83	0.17	20.57	250.61	3976.39	28.00
Early Exit Model									
0.2	0.6	0.29	0.36	36.20	0.04	27.13	59.45	977.29	5.86
	0.8	0.34	0.42	41.00	0.06	25.01	80.30	1309.80	9.56
	0.9	0.35	0.44	43.21	0.09	24.30	104.31	1694.02	13.84
	0.91	0.36	0.44	43.58	0.10	24.21	110.21	1786.98	14.88
	0.92	0.36	0.45	44.14	0.10	24.01	117.08	1896.69	16.12
0.3	0.6	0.26	0.21	26.40	0.01	37.98	71.04	1150.87	10.85
	0.8	0.32	0.42	41.72	0.07	24.64	96.03	1555.91	9.35
	0.9	0.35	0.46	45.00	0.11	23.51	125.04	2016.12	13.63
	0.91	0.35	0.46	45.08	0.11	23.39	132.16	2128.56	14.66
	0.92	0.35	0.46	45.12	0.11	23.51	139.92	2251.12	15.86
0.6	0.6	0.29	0.38	38.54	0.05	25.51	93.72	1517.07	5.80
	0.8	0.34	0.44	44.70	0.09	23.28	126.36	2032.11	9.23
	0.9	0.37	0.47	47.53	0.12	22.29	165.40	2648.97	13.45
	0.91	0.38	0.47	47.96	0.12	22.11	174.50	2792.78	14.43
	0.92	0.38	0.48	48.32	0.12	22.01	184.58	2952.56	15.56

Table 6.10: Summary of resource and accuracy-related metrics of OPT on PY150. 1000 Samples, generating 15 tokens, different contexts.

Context	Thresh	CodeBLEU	RougeL	ChrF	EM	Levenshtein Dist.	Total Energy [Wh]	Total Time [s]	Mean Layers
Full Model									
0.2	/	0.28	0.37	36.60	0.06	24.60	172.46	2772.57	32.00
0.3	/	0.29	0.37	36.81	0.06	24.19	210.01	3357.09	32.00
0.5	/	0.28	0.35	35.98	0.05	24.22	265.67	4230.69	32.00
Early Exit Model									
0.2	0.6	0.20	0.28	27.84	0.01	28.04	63.45	1031.00	9.75
	0.8	0.21	0.30	29.47	0.01	27.16	78.24	1266.34	12.59
	0.9	0.23	0.32	31.12	0.02	26.79	93.50	1512.11	15.92
	0.91	0.23	0.32	31.86	0.02	26.48	97.07	1570.24	16.68
	0.92	0.24	0.33	32.71	0.03	26.26	103.09	1666.35	17.93
0.3	0.6	0.19	0.26	27.24	0.01	28.06	79.98	1289.13	10.27
	0.8	0.21	0.29	29.19	0.02	27.19	97.53	1568.36	13.06
	0.9	0.35	0.46	45.00	0.11	23.51	125.04	2016.12	13.63
	0.91	0.24	0.31	31.80	0.03	26.23	120.98	1942.39	17.12
	0.92	0.25	0.32	32.46	0.03	25.90	128.32	2060.19	18.28
0.5	0.6	0.19	0.25	26.64	0.01	27.68	104.59	1676.47	10.77
	0.8	0.21	0.28	28.65	0.01	26.99	126.85	2029.06	13.68
	0.9	0.23	0.30	30.82	0.02	26.43	149.56	2390.01	16.78
	0.91	0.23	0.31	31.04	0.02	26.45	155.04	2476.23	17.54
	0.92	0.24	0.31	31.68	0.03	26.20	164.21	2624.44	18.73

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

6.4.3 Overhead Analysis

After evaluating the overall performance of our method, we now briefly discuss the overhead introduced by early exit decision-making. This overhead mainly consists of the forward pass through the RL network and a softmax computation to determine the action based on a threshold. If the action is *continue*, the overhead includes both the RL decision and the LLM layer computation, making the RL decision redundant. However, subsequent layers (and RL decisions) are skipped when the agent exits.

Measuring this overhead posed challenges due to errors caused by the NVML energy counter's update period being shorter than the measurement window, leading to misleading zeros. To address this, we used ZeusMonitor, which approximates energy by multiplying instantaneous power consumption by execution time [4]. Although still an approximation, it is more accurate than native `pynvml`, which often records zeros.

To measure overhead, as shown in Listing 8, we recorded energy and time whenever the layer was a designated exit point. Measurements were aggregated over a complete generation (summed for N tokens) and compared against the total energy for generation.

We experimented with isolated energy measurements on the PY150 dataset for both models to eliminate inaccuracies caused by nested measurements in the previous experiments.

The results indicate that the RL agent's additional energy consumption is small compared to the overall energy usage. The overhead remains nearly constant across most threshold settings, with a slight increase at higher thresholds due to more frequent "continue" actions, which introduce additional overhead.

Figure 6.12 illustrates the relative overheads of Llama in context settings 20% and 50%. In particular, the overhead decreases as the size of the context increases: 2-6% on 50% context and 3 to 8% on 20% against full model and 7-18% and 5.5-12.5% to the EE-model. This is expected since LLM computational costs grow with more input tokens. At the same time, the RL policy network overhead remains constant, as it involves a simple forward pass independent of input size.

Interestingly, the energy overhead is slightly higher than the time overhead, possibly due to factors such as energy-intensive forward passes or minor inaccuracies in the energy approximations. At larger context sizes, the early exit model overhead reaches about 7% of the full model's energy consumption. Although this may seem significant, it is manageable and allows reasonable performance, as discussed earlier.

Finally, we conducted a brief study to compare the overhead introduced by the RL-based approach with related methods. For this, we measured 100,000 forward passes of random tensors using three methods: (i) our method followed by a softmax operation, (ii) a single fully connected classifier layer followed by softmax, and (iii) passing through a layer normalization and LM-head followed by softmax (i.e., confidence-based exits).

[4]https://ml.energy/zeus/reference/monitor/energy/, accessed 25-11-24

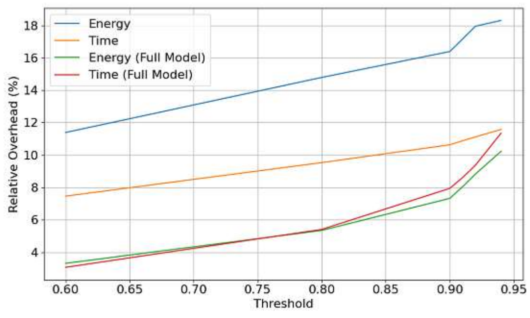
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
# during evaluation
def evaluate(...):
    ...
    self.monitor.begin_window("full_measurement",...)
    model.generate(...)
    measurement = self.monitor.end_window("full_measurement",...)
    rl_energy = np.sum(model.get_rl_energies())
    model.reset_rl_energies()

# in LLM decoder
def forward(...):
    ...

    for layer in layers:
        # layer forward calculations

        if layer in exit_indices:
            self.monitor.begin_window("overhead",...)


            action_logits = self.rl_model(hidden_state)
            action_probabilities = F.softmax(action_logits / temperature,
            ↪   dim=0)
            action = torch.argmax(action_probabilities).item()

            measurement = self.monitor.end_window("overhead",...)

            if action_probabilities[1] > threshold:
                #exit
                ...
            else:
                continue
    ...
```

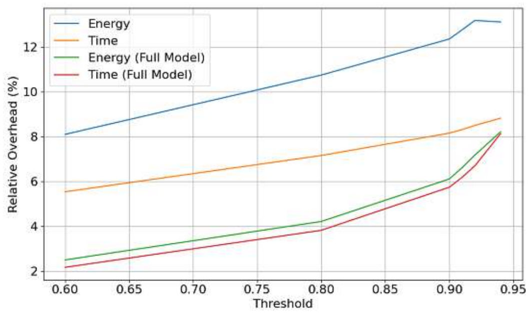
Listing 8: Overhead measurement example.

The results, shown in Figure 6.13, demonstrate that our method performs similarly, but slightly worse, to the classifier-based exit approach and significantly outperforms the confidence-based exiting approach in terms of overhead. This indicates that our method might be practical as an exit strategy.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

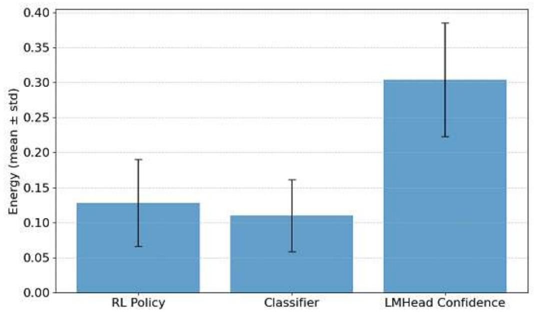


(a) 20% context as input.

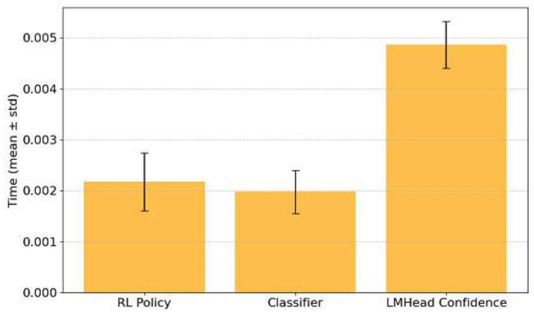


(b) 50% context as input.

Figure 6.12: Relative Overhead on Llama. (1000 samples of PY150, max 15 new tokens.



(a) Mean energy per forward pass (± std).



(b) Mean time per forward pass (± std).

Figure 6.13: Overhead comparison to related exiting strategies (100k random hidden states as input, single forward pass).

6.4.4 Experiments with KV Cache Propagation

For completeness, we present results using KV cache propagation as described in Chapter 4. Since no optimizations, such as parallelization, were applied to compute caches, we report efficiency regarding the number of layers. This analysis demonstrates that KV cache propagation is compatible with our method; however, optimizing the associated overhead is beyond the scope of this thesis.

Incorporating KV caching introduces two potential error sources that can affect performance metrics. First, approximations in KV cache calculations during early exits may result in cached values deviating from their exact representations, potentially introducing errors. Second, during training, the RL agent learns using fully calculated hidden states, which may differ from the approximated states used during inference with KV caching. This discrepancy could also contribute to performance deviations.

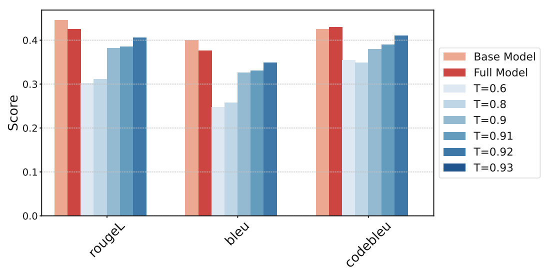
Figure 6.14 presents results on the JavaCorpus dataset, showcasing various settings with KV cache propagation. Performance remains stable overall when compared to the baseline without KV caching. For example, in the context setting 20%, the CodeBLEU

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

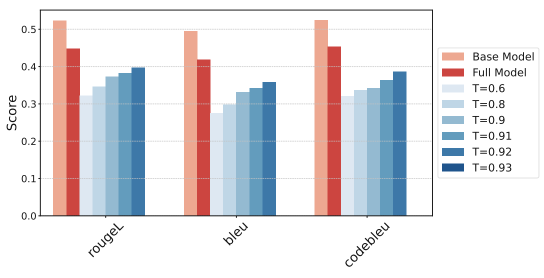
score remains unchanged. However, minor declines occur in some cases, such as a drop in BLEU for OPT from 0.29 in the baseline to 0.25 with KV caching, while no such drop is observed in the 60% context setting. Depending on the configuration, these performance losses are generally small, ranging from negligible to modest.

Additionally, Figure 6.15 highlights the fraction of layers skipped with KV caching. The observed patterns align closely with those from configurations without caching, reinforcing the compatibility of KV cache propagation with our early exiting method.

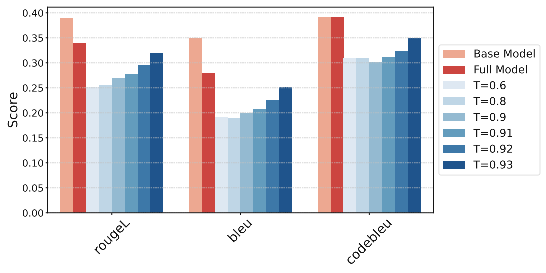
In summary, while KV caching introduces slight performance trade-offs in some scenarios, these losses are typically minor and do not compromise the method's robustness. The stability observed in both performance metrics and skipped layers demonstrates that KV cache propagation integrates well with our early exiting approach, supporting its applicability across different configurations.



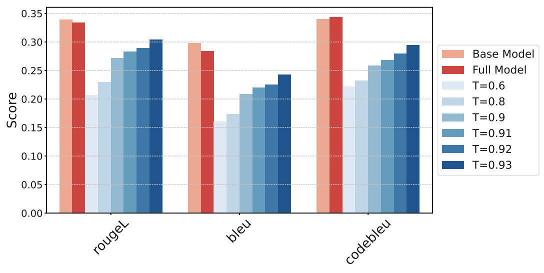
(a) 20% context, Llama.



(b) 60% context, Llama.



(c) 20% context, OPT.



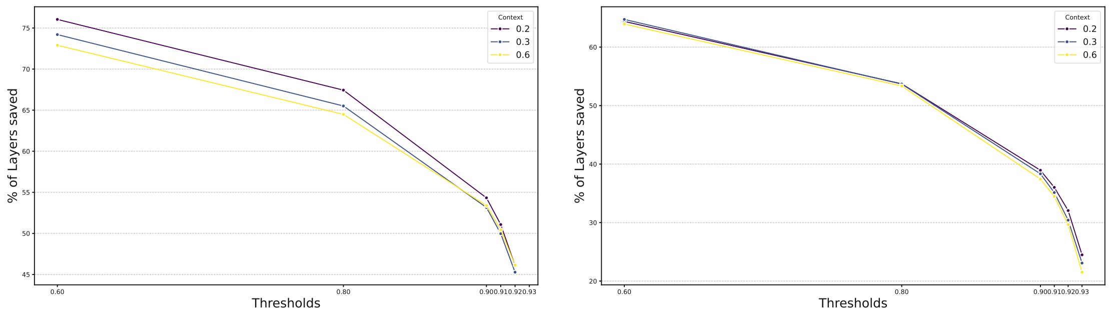
(d) 60% context, OPT.

Figure 6.14: Metrics with KV Cache Propagation. (1000 samples of JavaCorpus, max 15 new tokens).

6.5 Discussions and Limitations

In the previous sections, we observed that the RL-enhanced early exiting method can substantially improve the efficiency of LLMs. Although it results in a low to moderate reduction in accuracy metrics, it can lead to an average decrease in resource consumption by 20-50%. Despite these promising outcomes, there are several limitations in this work, which are discussed below:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Percentage of skipped layers on Llama.

(b) Percentage of skipped layers on OPT.

Figure 6.15: Mean % of layers skipped with KV Cache Propagation. (1000 samples of JavaCorpus, max 15 new tokens).

Model Choice: The approach was evaluated on two open-source models with 2.7B and 3B parameters. However, we did not extend our experiments to larger models, which typically have significantly more layers. More extensive models may require different reward designs, additional fine-tuning strategies, or entirely new settings to accommodate their increased complexity. Furthermore, the hidden size of larger models is often bigger than that of smaller models; thus, shallow layers might have more capabilities. Exploring such models could uncover scalability challenges and opportunities for improvement.

Fine-Tuning for Early Exiting: The fine-tuning process for enabling early exiting was not extensively optimized. In most evaluated settings, there was a sacrifice in model performance compared to the base models. This was partly due to the limited scope of our fine-tuning experiments, which were constrained by time and resources since fine-tuning LLMs can be time-consuming. We only used one fine-tuning strategy, which paid much attention to shallow layers, based on the hypothesis that it would enhance decoding from early layers. However, this approach might have led to overfitting in scenarios involving very early exits and could introduce unnecessary performance degradation in deeper layers. Future work could investigate alternative fine-tuning strategies and perform comprehensive hyperparameter tuning to mitigate these issues. Further comparisons of different fine-tuning settings would certainly be interesting.

Hyperparameter Tuning for the RL Agent: Due to the computationally intensive nature of RL training, we did not extensively explore hyperparameter tuning for the RL agent. Instead, we prioritized configurations that exhibited relatively fast convergence. While this approach provided usable results, more thorough hyperparameter optimization, and multiple training runs could yield improved performance and robustness. Future studies might consider a systematic search for optimal RL settings, such as adjustments to learning rates, rewards, model size, and exploration parameters.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Complexity of RL: Compared to other methods, such as confidence-based or classifier-based exiting, RL is a more fragile and complex technique to implement, train, and evaluate. The entire process—from reward design to training and evaluation—is more cumbersome, which can be seen as a limitation of our approach. Exploring different environments and rewards and trying different algorithm designs may improve results, making this an intriguing possibility for future work.

Baseline comparisons: Based on the fact that most baseline EE strategies must be implemented and trained and often require engineering of parameters (such as confidence thresholds), we did not conduct comparisons against other early exiting methods. We compared our method to the full model, as most related work does. However, comparisons across multiple baselines would undoubtedly be interesting for future work. Further, comparison against standard fine-tuned models (i.e., without aggregated loss) would bring additional insights, especially to get more profound knowledge on how much performance is lost in the full early-exit model compared to a standard finetuned one.

Evaluations: The proposed approach was evaluated using fixed metrics primarily adopted from NLP settings. While these metrics are informative and widely used, we did not perform experiments in real-world deployment settings, such as using our framework for actual code completion tasks during software engineering workflows. Evaluating the method in such practical scenarios could provide valuable insights into its usability and potential limitations. Furthermore, more optimized KV propagation implementations and comparisons would be great to analyze in future work to ensure the real-world applicability of our approach.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 7

Conclusions and Future Work

7.1 Summary

Large language models (LLMs) have become state-of-the-art for NLP, NLU, and code completion tasks, powering tools like GitHub Copilot and Amazon Q. However, these models have a substantial resource footprint, requiring expensive, energy-intensive hardware for deployment and inference. This creates significant sustainability challenges, making resource-efficient methods critical. Code completion, in particular, is resource-intensive because developers rely on these tools in IDEs for real-time suggestions with every file change, resulting in high computational demands. Various approaches have been developed for efficient LLM inference, each with unique advantages. Early exiting is a promising method that dynamically skips deeper layer computations during runtime to save resources. However, its application to code-related tasks remains underexplored. Additionally, most existing methods rely on metric-based exits and require extra language model heads at each exit layer, adding complexity.

In this thesis, we introduced an early-exiting approach specifically designed for the code completion task. Unlike many existing methods that require additional language model heads at each exit layer, we reused the original language model head. We fine-tuned the models using an aggregated loss formulation for early exiting. This approach is based on recent advances in the field [84], enabling early exiting without adding additional LM heads. Using these fine-tuned models, we trained a dynamic RL agent capable of learning the trade-off between efficiency and accuracy. The RL agent dynamically decides during runtime whether and when to exit early. Finally, we developed a deployment endpoint for integrating the early-exit-enhanced LLM as a practical prototype for code completion in IDEs.

We evaluated our proposed approach on two SOTA decoder-only LLMs, OPT-2.7B, and Llama3.2-3B, using two datasets of widely-used programming languages: JavaCorpus

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

for Java and PY150 for Python. The evaluations were conducted across various settings, demonstrating that our method can reduce resource consumption by 20% to 50% with small to moderate trade-offs in accuracy compared to the full model. Our analysis revealed that the approach performs particularly well in smaller context settings, achieving significant savings without compromising accuracy. However, it faces greater challenges in higher context scenarios, highlighting an area for further optimization. Additionally, we evaluated the computational overhead introduced by our method and found it to be within a reasonable range of 5% to 20%.

7.2 Future Work

Future work could include experiments that compare the proposed approach with thoroughly fine-tuned models to assess its effectiveness better. To ensure real-world applicability, integrating post-processing for LLM generations would be essential, enabling on-the-fly completions for developers. Additionally, while our experiments did not extensively explore KV-caching, this is a critical area for real-world scenarios, as it can significantly accelerate inference, and therefore, integrating more optimized KV-caching strategies in the future is essential. Additionally, evaluations beyond standard metrics could provide valuable insights, particularly regarding developer experience, such as measuring acceptance rates of the generated completions. Similarly, conducting case studies to measure energy consumption in real-world deployed settings would be valuable, showcasing the potential sustainability benefits achieved during typical coding or software engineering sessions. In addition, while we employed conventional training objectives for the code completion task using an autoregressive approach, code completion in practical situations often involves making predictions that fill in the middle (FIM). Consequently, utilizing FIM transformations [7] to fine-tune the model might yield superior and more realistic outcomes.

Further improvements could involve exploring more advanced reward structures or algorithms to enhance the performance of the RL agent and better address the efficiency-accuracy trade-off. Finally, comparing with other early exit methodologies would provide valuable insight into the broader applicability and competitiveness of the approach presented in this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Overview of Generative AI Tools Used

The Overleaf Writefull [1] Extension and ChatGPT (Version 4o) [2] were used to improve the textual quality of the work, in terms of grammar and language improvements. However, it is expressly pointed out that these tools have not contributed to the content in another way than language improvement and that their output has always been checked for consistency with the input. That is, all suggestions are completely revised and changed to match desired behavior.

[1] `https://www.writefull.com/`, accessed 25-11-24
[2] `https://chatgpt.com/`, accessed 25-11-24

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, et al. *SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills*. 2023. arXiv: `2308.16369 [cs.LG]`.

[2] Llama AI. *Llama 3: Model Cards and Prompt Formats*. `https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_2/`. Accessed: 2024-11-08. 2024.

[3] Meta AI. *Introducing Meta Llama 3: The most capable openly available LLM to date*. `https://ai.meta.com/blog/meta-llama-3/`. Accessed: 2024-11-08. 2024.

[4] Miltiadis Allamanis and Charles Sutton. "Mining source code repositories at massive scale using language modeling". In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. 2013, pp. 207–216.

[5] Fadi AlMahamid and Katarina Grolinger. "Reinforcement Learning Algorithms: An Overview and Classification". In: *2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, Sept. 2021, pp. 1–7. URL: `http://dx.doi.org/10.1109/CCECE53047.2021.9569056`.

[6] Amazon. *Amazon Q Developer, AI for Software Development*. URL: `https://aws.amazon.com/de/q/developer/`.

[7] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, et al. *Efficient Training of Language Models to Fill in the Middle*. 2022. arXiv: `2207.14255 [cs.CL]`. URL: `https://arxiv.org/abs/2207.14255`.

[8] Seda Bayat and Gultekin Isik. "Assessing the Efficacy of LSTM, Transformer, and RNN Architectures in Text Summarization". In: *International Conference on Applied Engineering and Natural Sciences* 1.1 (July 2023), pp. 813–820. URL: `https://as-proceeding.com/index.php/icaens/article/view/1099`.

[9] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, et al. *OpenAI Gym*. 2016. arXiv: `1606.01540 [cs.LG]`. URL: `https://arxiv.org/abs/1606.01540`.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[10] Cristian Bucila, Rich Caruana, and Alexandru Niculescu-Mizil. “Model compression”. In: *Knowledge Discovery and Data Mining*. 2006. URL: https://api.semanticscholar.org/CorpusID:11253972.

[11] David F. Carr. *ChatGPT Topped 1 Billion Visits in February*. Accessed: 2024-06-05. Mar. 2023. URL: https://www.similarweb.com/blog/insights/ai-news/chatgpt-1-billion/.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].

[13] Xiaodong Chen, Yuxuan Hu, and Jing Zhang. *Compressing Large Language Models by Streamlining the Unimportant Layer*. 2024. arXiv: 2403.19135 [cs.CL].

[14] Li Chenglin, Qianglong Chen, Liangyue Li, Caiyu Wang, et al. “Mixed Distillation Helps Smaller Language Models Reason Better”. In: *Findings of the Association for Computational Linguistics: EMNLP 2024*. Ed. by Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 1673–1690. URL: https://aclanthology.org/2024.findings-emnlp.91/.

[15] Luciano Del Corro, Allie Del Giorno, Sahaj Agarwal, Bin Yu, et al. *SkipDecode: Autoregressive Skip Decoding with Batching and Caching for Efficient LLM Inference*. 2023. arXiv: 2307.02628 [cs.CL].

[16] Alex de Vries. “The growing energy footprint of artificial intelligence”. In: *Joule* 7.10 (2023), pp. 2191–2194. ISSN: 2542-4351. URL: https://www.sciencedirect.com/science/article/pii/S2542435123003653.

[17] Radosvet Desislavov, Fernando Martínez-Plumed, and José Hernández-Orallo. “Trends in AI inference energy consumption: Beyond the performance-vs-parameter laws of deep learning”. In: *Sustainable Computing: Informatics and Systems* 38 (2023), p. 100857. ISSN: 2210-5379.

[18] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. “QLORA: efficient finetuning of quantized LLMs”. In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. NIPS ’23. New Orleans, LA, USA: Curran Associates Inc., 2024.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[20] Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. “Depth-Adaptive Transformer”. In: *International Conference on Learning Representations*. 2020.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[21] Mostafa Elhoushi, Akshat Shrivastava, Diana Liskovich, Basil Hosmer, et al. “LayerSkip: Enabling Early Exit Inference and Self-Speculative Decoding”. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 12622–12642. URL: `https://aclanthology.org/2024.acl-long.681/`.

[22] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, et al. *Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO*. 2020. arXiv: `2005.12729 [cs.LG]`. URL: `https://arxiv.org/abs/2005.12729`.

[23] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. “Out of the BLEU: How should we assess quality of the Code Generation models?” In: *Journal of Systems and Software* 203 (Sept. 2023), p. 111741. ISSN: 0164-1212. URL: `http://dx.doi.org/10.1016/j.jss.2023.111741`.

[24] Hugging Face. *LLM-VSCode: LLM powered development for VSCode*. `https://github.com/huggingface/llm-vscode`. Accessed: 2024-11-22. 2024.

[25] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, et al. “Large Language Models for Software Engineering: Survey and Open Problems”. In: *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. 2023, pp. 31–53.

[26] Chun Fan, Jiwei Li, Tianwei Zhang, Xiang Ao, et al. “Layer-wise Model Pruning based on Mutual Information”. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Ed. by Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 3079–3090. URL: `https://aclanthology.org/2021.emnlp-main.246`.

[27] Ya Gao and GitHub Customer Research. *Research: Quantifying GitHub Copilot’s Impact in the Enterprise with Accenture*. Accessed: 2024-11-12. 2024. URL: `https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/?utm_source=chatgpt.com`.

[28] *Github Copilot*. Accessed: October 4, 2024. URL: `https://github.com/features/copilot`.

[29] Andrey Gromov, Kushal Tirumala, Hassan Shapourian, Paolo Glorioso, and Daniel A. Roberts. *The Unreasonable Ineffectiveness of the Deeper Layers*. 2024. arXiv: `2403.17887 [cs.CL]`.

[30] Yue Guan, Zhengyi Li, Zhouhan Lin, Yuhao Zhu, et al. “Block-Skim: Efficient Question Answering for Transformer”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 36.10 (June 2022), pp. 10710–10719. URL: `https://ojs.aaai.org/index.php/AAAI/article/view/21316`.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[31] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. *On Calibration of Modern Neural Networks*. 2017. arXiv: `1706.04599 [cs.LG]`. URL: `https://arxiv.org/abs/1706.04599`.

[32] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, et al. *Deep Reinforcement Learning that Matters*. 2019. arXiv: `1709.06560 [cs.LG]`. URL: `https://arxiv.org/abs/1709.06560`.

[33] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. 2015. arXiv: `1503.02531 [stat.ML]`. URL: `https://arxiv.org/abs/1503.02531`.

[34] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, et al. "Large Language Models for Software Engineering: A Systematic Literature Review". In: *ACM Trans. Softw. Eng. Methodol.* (Sept. 2024). ISSN: 1049-331X. URL: `https://doi.org/10.1145/3695988`.

[35] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, et al. "LoRA: Low-Rank Adaptation of Large Language Models". In: *International Conference on Learning Representations*. 2022.

[36] Yuan Huang, Yinan Chen, Xiangping Chen, Junqi Chen, et al. *Generative Software Engineering*. 2024. arXiv: `2403.02583 [cs.SE]`.

[37] Chris P. Hughes. *Understanding PPO: A Game-Changer in AI Decision-Making Explained for RL Newcomers*. `https://medium.com/@chris.p.hughes10/understanding-ppo-a-game-changer-in-ai-decision-making-explained-for-rl-newcomers-913a0bc98d2b`. Accessed: 2024-11-12. 2023.

[38] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. 2020. arXiv: `1909.09436 [cs.LG]`.

[39] *Introducing Meta Llama 3: The most capable openly available LLM to date*. Accessed: 2024-06-05. Apr. 2024. URL: `https://ai.meta.com/blog/meta-llama-3/`.

[40] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, et al. "TinyBERT: Distilling BERT for Natural Language Understanding". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Ed. by Trevor Cohn, Yulan He, and Yang Liu. Online: Association for Computational Linguistics, Nov. 2020, pp. 4163–4174. URL: `https://aclanthology.org/2020.findings-emnlp.372/`.

[41] Andy L. Jones. *Debugging RL, Without the Agonizing Pain*. `https://andyljones.com/posts/rl-debugging.html`. Accessed: 2024-11-10. 2024.

[42] Eirini Kalliamvakou. *Github Research: Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness*. Accessed: October 4, 2024. 2023. URL: `https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/`.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[43] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, et al. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG].

[44] Bo-Kyeong Kim, Geonmin Kim, Tae-Ho Kim, Thibault Castells, et al. "Shortened LLaMA: A Simple Depth Pruning for Large Language Models". In: *ICLR 2024 Workshop on Mathematical and Empirical Understanding of Foundation Models*. 2024.

[45] Surafel Melaku Lakew, Mauro Cettolo, and Marcello Federico. "A Comparison of Transformer and Recurrent Neural Networks on Multilingual Neural Machine Translation". In: *Proceedings of the 27th International Conference on Computational Linguistics*. Ed. by Emily M. Bender, Leon Derczynski, and Pierre Isabelle. Santa Fe, New Mexico, USA: Association for Computational Linguistics, Aug. 2018, pp. 641–652. URL: https://aclanthology.org/C18-1054.

[46] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://aclanthology.org/W04-1013.

[47] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, et al. "AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration". In: *Proceedings of Machine Learning and Systems*. Ed. by P. Gibbons, G. Pekhimenko, and C. De Sa. Vol. 6. 2024, pp. 87–100. URL: https://proceedings.mlsys.org/paper_files/paper/2024/file/42a452cbafa9dd64e9ba4aa95cc1ef21-Paper-Conference.pdf.

[48] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, et al. *QServe: W4A8KV4 Quantization and System Co-design for Efficient LLM Serving*. 2024. arXiv: 2405.04532 [cs.CL]. URL: https://arxiv.org/abs/2405.04532.

[49] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, et al. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. 2021. arXiv: 2102.04664 [cs.SE]. URL: https://arxiv.org/abs/2102.04664.

[50] Yiren Lu. *How much VRAM do I need for LLM model fine-tuning?* Accessed 06-11-24. 2024. URL: https://modal.com/blog/how-much-vram-need-fine-tuning.

[51] Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. *Estimating the Carbon Footprint of BLOOM, a 176B Parameter Language Model*. 2022. arXiv: 2211.02001 [cs.LG]. URL: https://arxiv.org/abs/2211.02001.

[52] Xinyin Ma, Gongfan Fang, and Xinchao Wang. "LLM-pruner: on the structural pruning of large language models". In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. NIPS '23. New Orleans, LA, USA: Curran Associates Inc., 2024.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[53] Xin Men, Mingyu Xu, Qingyu Zhang, Bingning Wang, et al. "ShortGPT: Layers in Large Language Models are More Redundant Than You Expect". In: *ArXiv* abs/2403.03853 (2024). URL: `https://api.semanticscholar.org/CorpusID:268253513`.

[54] *Meta Llama 3.1 Model Card*. Accessed on 4-10-24. URL: `https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/MODEL_CARD.md`.

[55] Satya Nadella and Amy Hood. *Microsoft Fiscal Year 2024 Second Quarter Earnings Conference Call Transcript*. Accessed: October 4, 2024. 2024. URL: `https://www.microsoft.com/en-us/investor/events/fy-2024/earnings-fy-2024-q2`.

[56] OpenAI. *ChatGPT: Optimizing Language Models for Dialogue*. `https://openai.com/blog/chatgpt`. Accessed: 2024-11-27. 2022.

[57] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, et al. *GPT-4 Technical Report*. 2024. arXiv: `2303.08774 [cs.CL]`.

[58] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. "BLEU: a method for automatic evaluation of machine translation". In: ACL '02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318. URL: `https://doi.org/10.3115/1073083.1073135`.

[59] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: `1912.01703 [cs.LG]`. URL: `https://arxiv.org/abs/1912.01703`.

[60] David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Le, et al. *The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink*. 2022. arXiv: `2204.05149 [cs.LG]`.

[61] Maja Popović. "chrF: character n-gram F-score for automatic MT evaluation". In: *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Ed. by Ondřej Bojar, Rajan Chatterjee, Christian Federmann, Barry Haddow, et al. Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 392–395. URL: `https://aclanthology.org/W15-3049`.

[62] Alec Radford, Jeff Wu, Rewon Child, David Luan, et al. "Language Models are Unsupervised Multitask Learners". In: (2019).

[63] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, et al. "Stable-baselines3: reliable reinforcement learning implementations". In: *J. Mach. Learn. Res.* 22.1 (Jan. 2021). ISSN: 1532-4435.

[64] Sebastián Ramírez. *FastAPI*. `https://github.com/tiangolo/fastapi`. 2018.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[65] Veselin Raychev, Pavol Bielik, and Martin Vechev. “Probabilistic model for code with decision trees”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 731–747. ISBN: 9781450344449. URL: `https://doi.org/10.1145/2983990.2984041`.

[66] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, et al. *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis*. 2020. arXiv: `2009.10297 [cs.SE]`. URL: `https://arxiv.org/abs/2009.10297`.

[67] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: `1707.06347 [cs.LG]`. URL: `https://arxiv.org/abs/1707.06347`.

[68] Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, et al. “Confident Adaptive Language Modeling”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, et al. Vol. 35. Curran Associates, Inc., 2022, pp. 17456–17472. URL: `https://proceedings.neurips.cc/paper_files/paper/2022/file/6fac9e316a4ae75ea244ddcef1982c71-Paper-Conference.pdf`.

[69] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. “Green AI”. In: *Commun. ACM* 63.12 (Nov. 2020), pp. 54–63. ISSN: 0001-0782. URL: `https://doi.org/10.1145/3381831`.

[70] Emma Strubell, Ananya Ganesh, and Andrew McCallum. “Energy and Policy Considerations for Modern Deep Learning Research”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.09 (Apr. 2020), pp. 13693–13696. URL: `https://ojs.aaai.org/index.php/AAAI/article/view/7123`.

[71] Tianxiang Sun, Xiangyang Liu, Wei Zhu, Zhichao Geng, et al. “A Simple Hash-Based Early Exiting Approach For Language Understanding and Generation”. In: *Findings of the Association for Computational Linguistics: ACL 2022*. Ed. by Smaranda Muresan, Preslav Nakov, and Aline Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 2409–2421. URL: `https://aclanthology.org/2022.findings-acl.189/`.

[72] Tianxiang Sun, Yunhua Zhou, Xiangyang Liu, Xinyu Zhang, et al. “Early Exiting with Ensemble Internal Classifiers”. In: *ArXiv* abs/2105.13792 (2021). URL: `https://api.semanticscholar.org/CorpusID:235248158`.

[73] Zhensu Sun, Xiaoning Du, Fu Song, Shangwen Wang, and Li Li. “When Neural Code Completion Models Size up the Situation: Attaining Cheaper and Faster Completion through Dynamic Model Inference”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24. , Lisbon, Portugal, Association for Computing Machinery, 2024. URL: `https://doi.org/10.1145/3597503.3639120`.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[74] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[75] Hugging Face Team. *Transformers Trainer: Main Classes Documentation*. https://huggingface.co/docs/transformers/main_classes/trainer. Accessed: 2024-11-28. 2024.

[76] ML.Energy Team. *Zeus: Deep Learning Energy Measurement and Optimization*. https://ml.energy/zeus/. Accessed: 2024-11-12. 2024.

[77] PyTorch Team. *PyTorch Working with gym*. https://pytorch.org/rl/main/reference/generated/knowledge_base/GYM.html. Accessed: 2024-11-08. 2024.

[78] Ray Team. *RLlib: Environments*. https://docs.ray.io/en/latest/rllib/rllib-env.html. Accessed: 2024-11-08. 2024.

[79] Yijun Tian, Yikun Han, Xiusi Chen, Wei Wang, and Nitesh V. Chawla. *TinyLLM: Learning a Small Student from Multiple Large Language Models*. 2024. arXiv: 2402.04616 [cs.CL]. URL: https://arxiv.org/abs/2402.04616.

[80] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[81] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, et al. *Gymnasium: A Standard Interface for Reinforcement Learning Environments*. 2024. arXiv: 2407.17032 [cs.LG]. URL: https://arxiv.org/abs/2407.17032.

[82] L. Tunstall, L. von Werra, and T. Wolf. *Natural Language Processing with Transformers: Building Language Applications with Hugging Face*. O'Reilly Media, 2022. ISBN: 9781098103248. URL: https://books.google.at/books?id=pNBpzwEACAAJ.

[83] Umweltbundesamt. *Wie hoch sind die Treibhausgasemissionen pro Person in Deutschland durchschnittlich?* https://www.umweltbundesamt.de/service/uba-fragen/wie-hoch-sind-die-treibhausgasemissionen-pro-person. Accessed: 2024-10-08. Apr. 2023.

[84] Neeraj Varshney, Agneet Chatterjee, Mihir Parmar, and Chitta Baral. "Investigating Acceleration of LLaMA Inference by Enabling Intermediate Layer Decoding via Instruction Tuning with 'LITE'". In: *Findings of the Association for Computational Linguistics: NAACL 2024*. Ed. by Kevin Duh, Helena Gomez, and Steven Bethard. Mexico City, Mexico: Association for Computational Linguistics, June 2024, pp. 3656–3677. URL: https://aclanthology.org/2024.findings-naacl.232/.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[85] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[86] Hanrui Wang, Zhekai Zhang, and Song Han. “SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning”. In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Feb. 2021. URL: http://dx.doi.org/10.1109/HPCA51647.2021.00018.

[87] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. “LightSeq: A High Performance Inference Library for Transformers”. In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*. Ed. by Young-bum Kim, Yunyao Li, and Owen Rambow. Online: Association for Computational Linguistics, June 2021, pp. 113–120. URL: https://aclanthology.org/2021.naacl-industry.15.

[88] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, et al. *Sustainable AI: Environmental Implications, Challenges and Opportunities*. 2022. arXiv: 2111.00364 [cs.LG].

[89] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. “DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Ed. by Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault. Online: Association for Computational Linguistics, July 2020, pp. 2246–2251. URL: https://aclanthology.org/2020.acl-main.204.

[90] Ji Xin, Raphael Tang, Yaoliang Yu, and Jimmy Lin. “BERxiT: Early Exiting for BERT with Better Fine-Tuning and Extension to Regression”. In: *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. Ed. by Paola Merlo, Jorg Tiedemann, and Reut Tsarfaty. Online: Association for Computational Linguistics, Apr. 2021, pp. 91–104. URL: https://aclanthology.org/2021.eacl-main.8.

[91] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, et al. “WizardLM: Empowering Large Pre-Trained Language Models to Follow Complex Instructions”. In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=CfXh93NDgH.

[92] Xiaohan Xu, Ming Li, Chongyang Tao, Tao Shen, et al. *A Survey on Knowledge Distillation of Large Language Models*. 2024. arXiv: 2402.13116 [cs.CL].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[93] Yifei Yang, Zouying Cao, and Hai Zhao. “LaCo: Large Language Model Pruning via Layer Collapse”. In: *Findings of the Association for Computational Linguistics: EMNLP 2024*. Ed. by Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 6401–6417. URL: https://aclanthology.org/2024.findings-emnlp.372/.

[94] Zeyu Yang, Karel Adamek, and Wesley Armour. *Part-time Power Measurements: nvidia-smi's Lack of Attention*. 2024. arXiv: 2312.02741 [cs.DC]. URL: https://arxiv.org/abs/2312.02741.

[95] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*. 2023. arXiv: 2304.10778 [cs.SE]. URL: https://arxiv.org/abs/2304.10778.

[96] Alexander Yom Din, Taelin Karidi, Leshem Choshen, and Mor Geva. “Jump to Conclusions: Short-Cutting Transformers With Linear Transformations”. In: *arXiv preprint arXiv:2303.09435* (2023).

[97] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. “Orca: A Distributed Serving System for Transformer-Based Generative Models”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 521–538. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/yu.

[98] Shuzhou Yuan, Ercong Nie, Bolei Ma, and Michael Färber. *Why Lift so Heavy? Slimming Large Language Models by Cutting Off the Layers*. 2024. arXiv: 2402.11700 [cs.CL].

[99] Zhihang Yuan, Yuzhang Shang, and Zhen Dong. “PB-LLM: Partially Binarized Large Language Models”. In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=BifeBRhikU.

[100] Ziqian Zeng, Yihuai Hong, Hongliang Dai, Huiping Zhuang, and Cen Chen. “ConsistentEE: A Consistent and Hardness-Guided Early Exiting Method for Accelerating Language Models Inference”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.17 (Mar. 2024), pp. 19506–19514. URL: https://ojs.aaai.org/index.php/AAAI/article/view/29922.

[101] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, et al. *OPT: Open Pre-trained Transformer Language Models*. 2022. arXiv: 2205.01068 [cs.CL]. URL: https://arxiv.org/abs/2205.01068.

[102] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, et al. “H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models”. In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh, T. Naumann, A. Globerson, K. Saenko, et al. Vol. 36. Curran Associates, Inc., 2023, pp. 34661–34710.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/6ceefa7b15572587b78ecfcebb2827f8-Paper-Conference.pdf.

[103] Meizhi Zhong, Chen Zhang, Yikun Lei, Xikai Liu, et al. *Understanding the RoPE Extensions of Long-Context LLMs: An Attention Perspective*. 2024. arXiv: 2406.13282 [cs.CL]. URL: https://arxiv.org/abs/2406.13282.

[104] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, et al. *A Survey on Efficient Inference for Large Language Models*. 2024. arXiv: 2404.14294 [cs.CL]. URL: https://arxiv.org/abs/2404.14294.

[105] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. *A Survey on Model Compression for Large Language Models*. 2024. arXiv: 2308.07633 [cs.CL]. URL: https://arxiv.org/abs/2308.07633.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub