h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt - University of Applied Sciences

– Fachbereich Mathematik und Naturwissenschaften–

Optimization Techniques for LLM Inference: Quantitative Analysis of Knowledge Distillation, Batching and Quantization

Abschlussarbeit zur Erlangung des akademischen Grades

B. Sc.

vorgelegt von

Max Uhl

Matrikelnummer: 772773

Referent	:	Prof. Dr. Thomas März
Korreferent	:	Dr. Timo Schürg

DECLARATION

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, July 2025

Max Uhl

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my academic supervisor, Prof. Dr. März, for his invaluable guidance, insightful feedback and consistent encouragement throughout the course of this thesis. I am also thankful to Dr. Schürg for his support and helpful input during this work.

We gratefully acknowledge the support of the hessian.AI Innovation Lab, funded by the Hessian Ministry for Digital Strategy and Innovation (grant no. S-DIW04/0013/003), as well as the hessian.AI Service Center, funded by the Federal Ministry of Education and Research (BMBF, grant no. 01IS22091).

I would also like to sincerely thank Dianovi and my industry supervisor, Mr. Prisching, for their generous support and for providing me with the opportunity to work on the highly relevant and exciting topic of large language model inference optimization.

Last but not least, I wish to thank my family, especially my mother and sister for their unwavering support, patience and encouragement throughout this journey.

ABSTRACT

In the fast-paced and high-stakes environment of medical emergency rooms, both time efficiency and procedural accuracy are critical. Dianovi GmbH is committed to supporting medical professionals in these intense situations by assisting them in clinical decision-making, particularly in the quality assurance of doctors letters. As part of this effort, we focus on optimizing systems that incorporate text generation from Large Language Models (LLMs), which, while powerful, are often computationally intensive and time-consuming. In this work, we explore three complementary strategies to accelerate LLM-based text generation without significantly compromising output quality: *distillation*, *batching* and *quantization*. *Distillation* involves training a smaller, more efficient model to replicate the behavior of a larger, more capable one, thereby reducing inference cost. *Batching* enables the simultaneous processing of multiple inputs, greatly improving throughput. *Quantization* compresses the model by reducing numerical precision, which lowers memory usage and accelerates execution, typically with minimal loss in output quality. We evaluate these techniques using a set of custom benchmarks tailored to realistic medical text generation scenarios. Our experiments demonstrate that batching, when combined with a well-optimized inference framework, can yield up to a $40\times$ speedup in generation time. Additionally, we show that instruction tuning a foundation model on a curated mix of domain-specific data and instruction-following examples in the target language significantly enhances the zero shot classification capabilities of the model for in domain tasks of the language. Quantized models were evaluated for both latency and output fidelity, showing substantial reductions in inference time while maintaining high agreement with uncompressed model outputs. Furthermore, our distilled model with only 1.7 billion parameters achieved a similarity score exceeding 82% compared to outputs from the much larger Qwen3-32B model (32 billion parameters), indicating strong alignment. This study demonstrates that an LLM-based inference system can be significantly accelerated through the combined use of batching, quantization and distillation, with negligible degradation in quality. It also underscores the importance of domain-aware instruction tuning in improving the utility and reliability of generative models for critical applications such as emergency medicine.

ZUSAMMENFASSUNG

Im hektischen und anspruchsvollen Umfeld medizinischer Notaufnahmen sind sowohl Zeiteffizienz als auch prozedurale Genauigkeit von entscheidender Bedeutung. Die Dianovi GmbH setzt sich dafür ein, medizinisches Fachpersonal in diesen intensiven Situationen zu unterstützen, wobei sich diese Arbeit insbesondere auf das System zur Qualitätssicherung von Arztbriefen durch Hilfestellung bei klinischen Entscheidungsprozessen fokussiert. Da der Großteil der effektiven Laufzeit dieses Systems aus der Textgeneration von Large-Language Modellen (LLMs) besteht werden wir uns darauf fokussieren diese zu beschleunigen. Diese Modelle sind zwar leistungsfähig, jedoch oft rechenintensiv und zeitaufwendig. In dieser Arbeit untersuchen wir drei komplementäre Strategien zur Beschleunigung der LLM basierten Textgenerierung, ohne die Ausgabequalität wesentlich zu beeinträchtigen: *Destillierung*, *Batching* und *Quantisierung*. Bei der *Destillierung* wird ein kleineres, effizienteres Modell darauf trainiert, das Verhalten eines größeren, leistungsfähigeren Modells nachzuahmen, wodurch die Inferenzkosten reduziert werden. *Batching* ermöglicht die gleichzeitige Verarbeitung mehrerer Eingaben, was den Durchsatz erheblich verbessert. *Quantisierung* komprimiert das Modell durch Reduktion der numerischen Genauigkeit, was den Speicherbedarf senkt und die Ausführung beschleunigt, in der Regel mit nur geringem Qualitätsverlust. Wir evaluieren diese Techniken anhand eines Sets maßgeschneiderter Benchmarks, die realitätsnahe Szenarien medizinischer Textgenerierung und Klassifikation abbilden. Unsere Experimente zeigen, dass Batching in Kombination mit einem gut optimierten Inferenzframework eine bis zu $40\times$ schnellere Generierung ermöglichen kann. Darüber hinaus belegen wir, dass das Instruction Tuning eines Foundation Models auf einer kuratierten Mischung aus domänenspezifischen Daten und Anweisungsbeispielen in der Zielsprache die Zero Shot Klassifikationsfähigkeiten des Modells für domänenspezifische Aufgaben erheblich verbessert. Quantisierte Modelle wurden hinsichtlich Latenz und Ausgabequalität bewertet und zeigen dabei eine deutliche Reduktion der Inferenzzeit bei gleichzeitig hoher Übereinstimmung mit den Ausgaben unkomprimierter Modelle. Unser destilliertes Modell mit nur 1,7 Milliarden Parametern erreichte zudem eine Ähnlichkeitsbewertung von über 82 % im Vergleich zu den Ausgaben des deutlich größeren Qwen3 32B Modells (32 Milliarden Parameter). Dies zeigt, dass ein spezialisiertes und kompaktes Modell ähnliche Resultate wie ein größeres, teureres und langsameres Modell liefern kann. Diese Studie zeigt, dass ein LLM basiertes Inferenzsystem durch die kombinierte Anwendung von Batching, Quantisierung und Destillierung erheblich beschleunigt werden kann, mit geringem bis keinem Qualitätsverlust. Zudem unterstreicht sie die Bedeutung domänenbewussten Instruction Tunings zur

Verbesserung der Nutzbarkeit und Zuverlässigkeit generativer Modelle in kritischen Anwendungsbereichen wie der Notfallmedizin.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

AGI Artificial General Intelligence
AI Artificial Intelligence
API Application Programming Interface
AWQ Activation-Aware Weight Quantization
B Billion
BN Batch Normalization
CPU Central Processing Unit
FFN Feed Forward Neural Network
FLOP Floating Point Operations
FLOPS Floating Point Operation per Second
GB Gigabyte
GLU Gated Linear Unit
GQA Grouped Query Attention
GPT Generative Pretrained Transformer
GPTQ Generative Pre-trained Transformer Quantization
GPU Graphics Processing Unit
ID identity
It Instruct
KL Kullback-Leibler Divergence
KV Key-Value
LLM Large Language Model
LLMs Large Language Models
MHA Multi Head Attention
MQA Multi Query Attention
MSE Mean Squared Error
NaN Not a Number (also used for undefined)
NLP Natural Language Processing
OOV out-of-vocabulary
OOM out-of-memory
PTQ Post Training Quantization
QA Question Answering
QAT Quantization Aware Training
QT Quantized Training
ReLU Rectified Linear Unit
ResNet Residual Neural Network
RMSNorm Root Mean Square Root Normalization

RoPE Rotary Positional Embeddings

SFT Supervised Fine-Tuning

SOTA State of the Art

STE Straight Through Estimate

TFLOPS Tera (10^{12}) Floating Point Operations per Second

VRAM Video Random Access memory

Table 1: Mathematical operations

Notation	Operation	Description
x	Vector	Our default vectors are always of shape (1,n)
xW	Vector-Matrix Multiplication	Multiplication of vector x with matrix W. If W is a tensor of order > 2, this refers to the vector-matrix multiplication of x with the last two dimensions of W.
QK	Matrix Multiplication	Standard matrix multiplication between matrices Q and K.
$\frac{Q}{c}$	Element wise division	We divide each element in any tensor by the value c.
$Q \times K$	Element-wise Tensor Multiplication	Element-wise multiplication of tensors Q and K.
$Q + K$	Element-wise Tensor Addition	Element-wise addition of tensors Q and K.
$x \cdot y$	Dot Product	Dot product between vectors x and y.
xy	Vector Multiplication	Depending on the used elements
X^T	Transpose	Transposes the matrix or vector
$diag(x)$	Diagonalize	Creates a matrix with all values from x on the main diagonal and o elsewhere

1

INTRODUCTION

In modern emergency medical settings, healthcare professionals operate under intense time pressure, making rapid decisions while managing high patient volumes. Amid these challenges, the creation of accurate and timely documentation, particularly doctors letters is essential for effective clinical communication, continuity of care and legal compliance. This thesis was developed in collaboration with Dianovi GmbH, a company focused on supporting emergency room personnel through artificial intelligence. Their system assists with the quality management of medical documentation by enhancing the clarity, consistency, completeness and guideline adherence of doctors letters using natural language processing (NLP) tools.

To be viable in a clinical environment, the system is subject to strict runtime requirements. Specifically, the entire processing pipeline must execute in under 15 seconds, with the first generation completed in under 5 seconds to avoid disrupting physicians workflows. At the start of this project, the systems end-to-end latency in most cases exceeded 30 seconds, necessitating a thorough investigation into methods for optimizing inference speed. This real-world constraint directly motivated the technical focus of the thesis.

At the core of the system are large language models (LLMs), which have demonstrated impressive capabilities in natural language generation. However, their deployment in time-sensitive environments like emergency medicine presents major challenges. LLMs are computationally intensive, requiring substantial memory and processing resources, which can lead to high inference latency and limited scalability.

To address these limitations, this thesis investigates optimization techniques aimed at improving the efficiency of LLM inference without significantly compromising model performance. The work focuses on three complementary strategies that have gained traction in both academia and industry: *knowledge distillation*, *batching* and *quantization*. Knowledge distillation compresses large models by training smaller ones to replicate their behavior, thereby reducing computational demands. Batching increases throughput by processing multiple inputs simultaneously, improving hardware utilization. Quantization reduces the numerical precision of model weights and activations, leading to lower memory consumption and faster execution.

Through a quantitative evaluation of these techniques, this thesis explores their individual and combined impacts on inference latency, memory usage and model accuracy. By grounding the analysis in a concrete and high-stakes clinical use case, this work contributes practical insights into how LLMs can be optimized for deployment in resource-constrained, time-critical environments such as emergency rooms.

2

BACKGROUND ON LARGE LANGUAGE MODELS (LLMS) AND INFERENCE

To accurately estimate the inference cost of Large Language Models (LLMs), it is essential to first understand the mechanisms underlying their operation. This chapter provides an overview of the core concepts and the mathematical principles relevant for this work. To support readers who may be less familiar with the conceptual ideas of deep learning, the following **prequel section** offers a concise explanation of how deep learning models learn from data. This section may be skipped by readers who already have a basic understanding of the training process.

2.1 PREQUEL: HOW DEEP LEARNING MODELS LEARN

To understand the learning process of Deep Learning Models, such as LLMs we will explain briefly how a Deep Learning Models is able to learn.

2.1.1 *Universal Approximation Theorem*

The Universal Approximation Theorem [4] states, that any continuous real valued function can be approximated by a combination of neural networks. Mathematically, we define a neural network as in [4]:

Definition 2.1.1 (Neural Network)**.** *Let $d, L \in \mathbb{N}$ and $N = (N_0, N_1, \ldots, N_L) \in \mathbb{N}^{L+1}$ and let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous nonlinear* activation function*. We define:*

- *L as the number of layers,*
- *N_0 as the number of input neurons,*
- *N_L as the number of output neurons,*
- *N_ℓ for $\ell \in [L-1]$ as the number of neu rons in the ℓ-th hidden layer.*

Let the parameters of the network be denoted by

$$\theta = (\theta_\ell)_{\ell=1}^{L}$$

with

$$\theta_\ell = (W_\ell, b_\ell)\,,$$

where $W_\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ *is the* weight matrix *and* $b_\ell \in \mathbb{R}^{N_\ell}$ *is the* bias vector *of layer* ℓ*.The total number of parameters is given by:*

$$P_N = \sum_{\ell=1}^{L} \left(N_\ell N_{\ell-1} + N_\ell\right).$$

. We define the realization function $\Phi_\alpha : \mathbb{R}^{N_0} \times \mathbb{R}^{P_N} \to \mathbb{R}^{N_L}$ *where* $\alpha = (N, \sigma)$*, such that for any input* $x \in \mathbb{R}^{N_0}$ *and parameters* θ*, we recursively compute:*

$$\Phi_1(x, \theta) = xW_1 + b_1 \tag{2.1}$$
$$\bar{\Phi}_\ell(x, \theta) = \sigma\left(\Phi_\ell(x, \theta)\right), \quad \text{for } \ell \in [L-1] \tag{2.2}$$
$$\Phi_{\ell+1}(x, \theta) = \bar{\Phi}_\ell(x, \theta)W_{\ell+1} + b_{\ell+1} \tag{2.3}$$

where the activation function σ *is applied componentwise.*
Thus, a neural network can be viewed as a successive composition of affine transformations followed by nonlinear activations:

$$x \mapsto xW_\ell + b_\ell.$$

An example of a nonlinear activation function is the Rectified Linear Unit(ReLU) [55], which is defined as

$$\mathrm{ReLU}(x) = \max(0, x)$$

and displayed in Figure 2.1.

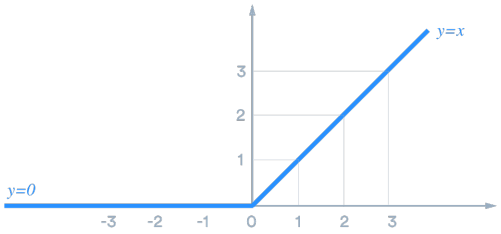

Figure 2.1: ReLU Activation Function [55]

Definition 2.1.2. *For the remainder of this work we will refer to a weighted linear activation* $\Phi_1(x, \theta)$ *as described in* (2.1) *as linear projection.*

2.1.2 *Learning process of neural networks*

Since the Universal Approximation Theorem only guarantees the *existence* of a neural network capable of approximating any continuous real-valued function, there is still the practical challenge of *finding* the appropriate weights for the network's neurons to approximate a specific target function.
In the context of LLMs, this target function represents the conditional probability distribution over the vocabulary for the next sentence piece(e.g. token, we will discuss this later in 2.3.1), given the previous context. This means,

that the model learns to estimate $P(\text{token}_t \mid \text{tokens}_{<t})$.
To adjust the network's parameters so that it produces accurate predictions, most data scientists employ a training algorithm known as *backpropagation*, which efficiently computes gradients used to update the model's weight matrices.
However, to understand how backpropagation guides these updates, it is essential to introduce the concept of *loss functions*, which quantify the error between the model's predictions and the true target values.

2.1.3 *Loss Functions*

In supervised learning, a *loss function* quantifies the discrepancy between the predicted output of a model and the corresponding true target values. Formally, consider a training dataset defined as:

$$\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n},$$

where each input $x^{(i)} \in \mathbb{R}^d$ represents the feature vector and the associated target $y^{(i)} \in \mathbb{R}$ (or $\mathbb{R}^k$ for multi-dimensional outputs) denotes the ground truth. Let $f_\theta : \mathbb{R}^d \to \mathbb{R}$ be a model parameterized by $\theta \in \mathbb{R}^p$.
A **loss function** $\mathcal{L} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_{\geq 0}$ assigns a non-negative scalar cost to the prediction $f_\theta(x)$ given the true label y, thereby measuring the prediction error. The **empirical risk**, or average loss over the dataset, is expressed as:

$$\mathcal{R}_{\text{emp}}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\big(f_\theta(x^{(i)}), y^{(i)}\big).$$

Training a model corresponds to finding the optimal parameters θ^* that minimize this empirical risk:

$$\theta^* = \arg\min_{\theta \in \mathbb{R}^p} \mathcal{R}_{\text{emp}}(\theta).$$

2.1.4 *Relevant Loss Functions*

This subsection introduces the loss functions particularly relevant to the methods discussed in this work.

MEAN SQUARED ERROR

The Mean-Squared Error (Mean Squared Error (MSE)), as defined by Hastie et al. [31] is a measure of how far two distributions are apart, which punishes far deviations exponentially harder than small deviations. Hastie et al. [31]

define it as following:

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y_n})^2 \tag{2.4}$$

Where N is the total amount of predictions, y_n is the ground truth value for the n-th prediction and $\hat{y_n}$ is the n-th prediction.

KULLBACK-LEIBLER DIVERGENCE

The Kullback-Leibler (KL) divergence [46] is a measure of how one probability distribution diverges from a reference distribution. For discrete probability distributions P and Q defined over the same domain X, it is given by:

$$KL(P\|Q) = \sum_{x\in X} P(x)\log\left(\frac{P(x)}{Q(x)}\right). \tag{2.5}$$

In this work, Kullback-Leibler Divergence (KL) divergence is primarily used to compare the output probability distributions generated by different LLMs, capturing how one models predictions differ from the predictions made by another model.

CROSS ENTROPY

Cross entropy is one of the most widely used loss functions for training LLMs, as it quantifies the difference between the predicted probability distribution over classes and the true distribution (typically a one-hot encoding of the correct class). It is defined by Bishop [16] as:

$$L = \{l_1, \ldots, l_N\}, \quad l_n = -w_{y_n}\log\left(\frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C}\exp(x_{n,c})}\right), \tag{2.6}$$

where x denotes the input logits, y_n is the correct class index for the n-th example, C is the total number of classes with $c \in [1, C]$ and w_{y_n} is a class-specific weight that can adjust the contribution of each class to the loss. The term inside the logarithm corresponds to the softmax probability of the correct class.
Since typically one scalar loss value per training example is computed, the vector L is often aggregated by taking a weighted mean. According to the PyTorch documentation [25], this reduction is calculated as:

$$l = \frac{\sum_{n=1}^{N} l_n}{\sum_{n=1}^{N} w_{y_n}}. \tag{2.7}$$

This weighted averaging can be used to account for class imbalance or other importance weighting schemes during training, the most common approach is to set the weight of each class to one.

2.1.5 *Backpropagation*

To find possible approximations of the weights W_ℓ most Data Scientists first initialize a set of weights using an initialization method like He Initialization [32]. Then they Iterate over the Dataset and use the Backpropagation Algorithm [21], which uses the gradient of the loss in respect to the models weights to gradually update the model weights. For further details we refer to Damadi et al. [21].

2.1.6 *Metrics*

While loss functions (see Section 2.1.4) quantify the difference between a model's predictions and the true values during training, metrics in data science are primarily used to assess the overall performance and applicability of a trained model. Unlike loss functions, metrics offer interpretable insights into how well a model performs in a given task, especially during evaluation and testing phases.

ACCURACY

Accuracy is one of the most commonly used metrics for classification problems. It measures the proportion of correctly predicted instances out of the total number of predictions made [20]:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{2.8}$$

Accuracy provides an intuitive measure of correctness. However, it can be misleading in cases of class imbalance. For example, in a dataset where 95% of the samples belong to class A and only 5% to class B, a model that always predicts class A will achieve 95% accuracy,despite failing completely to identify class B.

BALANCED ACCURACY

Balanced accuracy addresses the shortcomings of standard accuracy in imbalanced classification problems. It accounts for the model's performance on each class equally by averaging the true positive rate (sensitivity) and the true negative rate (specificity). For binary classification, it is defined as [84]:

$$\text{Balanced Accuracy} = \frac{\text{Sensitivity} + \text{Specificity}}{2} \tag{2.9}$$

where

$$\text{Sensitivity} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{2.10}$$

and

$$\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}} \tag{2.11}$$

Balanced accuracy provides a more reliable performance measure in scenarios where classes are unequally represented. For instance, in medical diagnostics where the number of healthy patients far outweighs those with a disease, balanced accuracy ensures that both detection of the disease (sensitivity) and correct classification of healthy individuals (specificity) are fairly evaluated.

2.1.7 *Cohens Kappa*

Cohens Kappa is a metric originally designed to measure the agreement between two rating instances. It is calculated as following [19]:

$$\kappa = \frac{p_0 - p_c}{1 - p_c} \tag{2.12}$$

where p_0 is the measured agreement rate between two rating instances and p_c is the expected random agreement. For a more in-depth look we refer to the original paper written by Cohen [19] and a follow up work by Artstein et al. [9] which includes the calculation of p_c for classification problems.

2.1.8 *Cosine Similarity*

Cosine similarity measures how similar two vectors, which usually represent documents or texts are by computing [57]:

$$sim(\vec{V_1}, \vec{V_2}) = \frac{\vec{V_1} \cdot \vec{V_2}}{\|\vec{V_1}\| \|\vec{V_2}\|} \tag{2.13}$$

Where $\vec{V_1}, \vec{V_2}$ are the vector representations of the compared documents or texts.

2.2 BRIEF OVERVIEW OF LLM ARCHITECTURE

Large Language Model (LLM)s represent a specialized class of deep learning architectures designed to process, understand and generate human language. Fundamentally, a LLM functions as a next-token predictor [81], where a **token** typically corresponds to a subword unit or sentence fragment. Given an input sequence, the model iteratively predicts the most likely subsequent token, enabling it to generate coherent and contextually relevant text.
A high-level abstraction of the text generation process employed by LLMs can be described as follows:

Algorithm 1 Simplified LLM Text Generation

```
context ← model context + initial prompt
generated_text ← [ ]
while token ≠ Stop_Token do
    token ← LLM(context)
    append token to generated_text
    context ← context + token
end while
return generated_text
```

We provide a concrete text generation example in Figure 2.2.

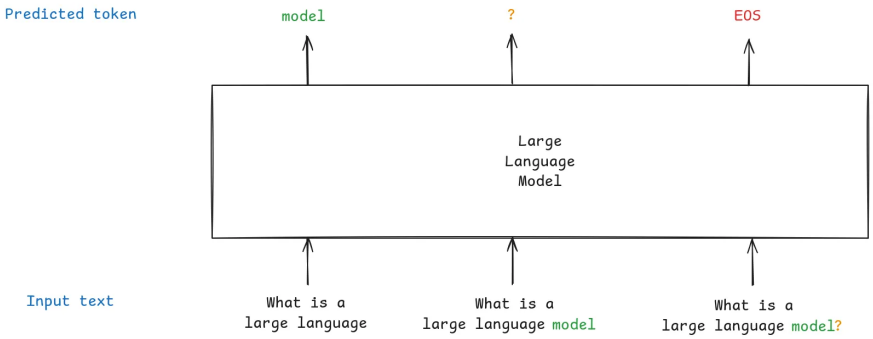

Figure 2.2: Example of LLM text-generation [14]

The later sections delve into the internal architecture of LLMs, examining the mechanisms that allow them to model linguistic patterns and predict successive tokens with remarkable fluency and coherence.

2.3 TOKENIZATION AND EMBEDDINGS

2.3.1 *Tokenization*

Since machine learning models require numerical input to approximate functions, there is a need to transform textual data into numerical representations. For LLMs, the standard method for this transformation is called **tokenization**. The core idea of tokenization is to split a given sequence of text characters into multiple sub-sequences, called *tokens*, which are then mapped to unique numerical identifiers. Although the way tokens are split may seem arbitrary at first glance, in practice, the choice of tokenization method greatly impacts the model's performance and efficiency. Different tokenization approaches produce vastly different vocabulary sizes and token distributions [7]. For example, consider the sentence:

> *Transformers revolutionize natural-language processing.*

If tokenized at the word level, the sentence would be split into the tokens:

["Transformers", "revolutionize", "natural-language", "processing", "."]

This approach treats each word (or punctuation) as a separate token, which can result in a huge vocabulary and problems when encountering words not seen during training, known as the *out-of-vocabulary* (OOV) problem.
On the other hand, tokenization methods that break text into smaller units, such as subwords or characters are better at handling rare or new words by combining known tokens [75]. For instance, subword tokenization might split the word "Transformers" into "Transform" and "ers," reducing vocabulary size and improving generalization.
Rigid tokenization approaches, such as strict word-level tokenizers, often struggle when no suitable token exists for a given sub-sequence. This limitation typically results in the use of unknown token placeholders, thereby reducing the model's expressiveness. Moreover, such methods face additional challenges with languages like Chinese, which do not use spaces to separate words[45]. SentencePiece tokenization addresses both of these issues by segmenting text into subsentence units, or *sentence pieces* without relying on spaces, punctuation, or any other special characters to split text sequences[45].
Because inference cost and memory usage scale with the number of input and output tokens, LLMs need a tokenization method that minimizes token count while minimizing the risk of encountering out-of-vocabulary (OOV) issues. SentencePiece [45] has become the preferred tokenization method in many modern LLMs due to its effective balance between minimizing token count and managing vocabulary size, as well as its strong support for multilingual text. We provide an example of sentencepiece tokenization in Figure 2.3.

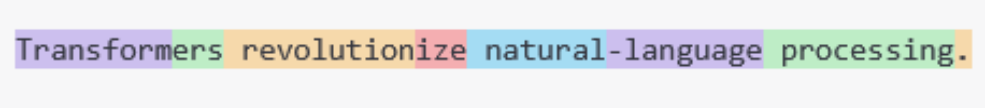

Figure 2.3: Sentencepiece Tokenization Example [66]

SPECIAL TOKENS

While normal tokens numerically represent a given text sequence for a large language model, **special tokens** convey commands or additional information to the model or user. One example of this are the `<|user|>` and `<|assistant|>` tokens, which signal to a Llama 3 chatbot who sent which text in a multi-turn conversation [59]. While these tokens may differ in looks between models, their functionality is effectively the same.

Figure 2.4: User view of example Conversation [65]

For example, the chat shown in Figure 2.4 will get transformed into something like:

```
<|user|>
Hi
<|assistant|>
Hey! What's up?
<|user|>
Transformers are great models
```

before it gets tokenized and sent to the model.
Special tokens like `<|user|>` and `<|assistant|>` are treated as atomic units with their own distinct token identity (ID)s and are not split into sentence-piece tokens.
Other common use-cases of special tokens include:

- **Padding tokens**, which fill sequences to a fixed length and are typically ignored by the model via attention masks [47].
- **End-of-Sequence tokens**, which signal the model to stop generating further output [12].
- **Beginning-of-Sequence tokens**, which signal the model the starting position of the current sequence.
- **End-of-turn tokens**, which signal the model the end of a message in a multi-turn conversation
- **Delimiter tokens**, which mark the beginning and end of metadata fields used by the model

2.3.2 *Embeddings*

Now that we can transform text into a numerical representation using tokenization, the next step is to encode both the *semantic meaning* of each token and its *position* within the sequence. This is achieved through a process called **embedding**.
Each token ID is mapped to a high-dimensional vector (depending on the model the dimension is usually in the range between 256 and 8192) using a learnable embedding matrix [60]. These token embeddings capture semantic relationships between words, allowing the model to understand that similar words (e.g., "dog" and "puppy") have similar representations. Additionally the model learns to understand semantic context mathematically, for example in [60] the embedding model understands that $\vec{Paris} - \vec{France} + \vec{Italy} = \vec{Rome}$.
In addition to semantic meaning, transformers must account for the order of tokens, since they lack inherent recurrence or convolution [93]. To do this, **positional encodings** or **positional embeddings** are added to the token embeddings, providing the model with information about the position of each token in the input sequence.
Together, these embeddings are passed into the model as the input representations for further processing through the attention layers, which we will explore in the next section.

2.4 ATTENTION MECHANISMS

The attention mechanism represents the core innovation of the Transformer architecture, which serves as the backbone structure of all modern LLMs. In this thesis, we focus on two prominent attention mechanisms: Multi-Head Attention (MHA) [93] and Grouped Query Attention (GQA) [6].

2.4.1 *Softmax*

To understand multi-head attention, we first introduce the *softmax* function. Widely used in machine learning, softmax transforms a vector of real numbers (e.g., logits) into a probability distribution. The i-th probability is computed as:

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} \quad \text{for } i = 1, \ldots, n \tag{2.14}$$

A key property of the softmax function is that the output values always sum to 1 for any real-valued input vector $\mathbf{z}$. This is shown by:

$$\sum_{i=1}^{n} \text{softmax}(\mathbf{z})_i = \sum_{i=1}^{n} \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} = \frac{\sum_{i=1}^{n} \exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} = 1 \tag{2.15}$$

Moreover, since $\exp(x) > 0$ for all real numbers x, every softmax output is strictly positive and less than or equal to one. Therefore, the softmax function always produces a valid probability distribution, regardless of the specific real-number values in $\mathbf{z}$.

2.4.2 *Attention*

The central idea behind Attention is to model dependencies between elements of sequences, allowing a model to dynamically focus on relevant parts of the input when producing an output. Given two sequences, the Attention mechanism computes pairwise interactions between all tokens, producing a weighted representation that reflects their contextual relevance and interactions[93].
To formalize this, we begin with the *Scaled Dot-Product Attention*, which operates on differently embedded representations of the inputs: Queries ($\mathbf{Q}$), Keys ($\mathbf{K}$) and Values ($\mathbf{V}$), which all get calculated by a different linear projection 2.1.2 of our input vector x. Each query vector interacts with all key vectors to produce attention weights, which are then used to compute a weighted sum over the value vectors [93].
In practice, rather than processing individual query, key and value vectors, we pack them into matrices $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ for efficient parallel computation. The attention output is then computed as following [93]:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_{head}}}\right)\mathbf{V} \tag{2.16}$$

Here, d_{head} is the dimensionality of the key vectors and softmax denotes the row-wise softmax function. The softmax ensures that the attention weights are positive and sum to one, effectively forming a probability distribution over the keys for each query.

INTERPRETATION OF ATTENTION

Intuitively, attention captures relationships such as coreference (e.g., resolving pronouns to the correct antecedent), syntactic dependencies (e.g., subject-verb agreement) and semantic associations (e.g., linking related entities across a sentence). The weights produced by the softmax operation indicate which parts of the input the model is "attending" to when generating representations for a specific position [93]. These weights are interpretable: higher values correspond to greater influence from a particular token and visualizing them as in Figure 2.5 can reveal how the model reasons about input sequences.

From a geometric perspective, the dot product in Equation 2.16 measures the similarity between queries and keys. Scaling by $\sqrt{d_{head}}$ prevents the dot products from growing too large in high dimensions, stabilizing gradients during

	Hello	I	love	you
Hello	0.8	0.1	0.05	0.05
I	0.1	0.6	0.2	0.1
love	0.05	0.2	0.65	0.1
you	0.2	0.1	0.1	0.6

Figure 2.5: Exemplary Attention Matrix [1]

training. The softmax then transforms these similarities into a distribution over the input positions, allowing the model to focus more on relevant tokens while still maintaining differentiability for learning.
Ultimately, Attention enables flexible and data-driven modeling of dependencies without assuming fixed spatial or temporal relationships, making it particularly powerful in handling long-range interactions in sequences, as seen in tasks like translation, summarization and text generation.

2.4.3 *Multi-Head Attention*

Multi Head Attention (MHA) extends the attention mechanism by allowing the model to jointly attend to information from different representation subspaces at multiple positions. Instead of performing a single attention function with queries, keys and values, MHA computes several attention heads in parallel, each with its own set of learned linear projections [93].
Formally, the Multi-Head Attention mechanism (omitting bias terms) is defined as [93]:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O \tag{2.17}$$

where each attention head is given by:

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i) \tag{2.18}$$

Where for each head an unique Q_i, K_i and V_i are calculated by linearly projecting the input with an unique set of weights.
These matrices are learned during training and enable each attention head to focus on different parts of the input representation space. The outputs of all heads are concatenated and linearly transformed via $\mathbf{W}^O$ to produce

the final result. Experiments in [93] show that each attention head learns to represent different interactions in the input sequence.
While MHA is highly effective during training and has consistently demonstrated state-of-the-art (SOTA) performance, it can become computationally expensive during inference, particularly as model size increases.

2.4.4 *Multi-Query Attention*

To address inefficiencies in inference, Shazeer [79] proposed Multi-Query Attention (MQA). While MHA benefits from parallelization during training,inference remains computationally intensive because new key, query and value embeddings must be generated for each token and each attention head. Multi Query Attention (MQA) alleviates this burden by sharing the same key and value projections across all attention heads. This modification substantially reduces inference time by a factor of nine in Shazeer's experiments while only causing a minor decrease in benchmark performance (approximately 1%).
Although MQA significantly speeds up inference, the slight reduction in answer quality can still be noticeable. Additionally, MQA can lead to training instability, which is particularly concerning given the high cost of training LLMs since Li estimates the training cost for Generative Pretrained Transformer (GPT)-3 at approximately $4.6 million using cloud compute in 2020 [50]. To address these challenges, Ainslie et al. proposed Grouped Query Attention (GQA) [6], which serves as an interpolation between MHA and MQA.

2.4.5 *Grouped Query Attention*

GQA strikes a balance between the computational efficiency of MQA and the representational flexibility of MHA. In GQA, attention heads are divided into groups and each group shares a single key and value projection [6]. This approach reduces the number of key and value computations compared to MHA, but retains more head-specific variability than MQA. By allowing a tunable number of key/value groups, typically more than one (as in MQA), but fewer than the number of attention heads (as in MHA) GQA offers a configurable trade-off between inference speed and model performance. Empirical evaluations have shown that GQA can achieve near MHA accuracy while maintaining much of MQA's inference efficiency [6]. Consequently, GQA has been adopted in several large-scale transformer architectures, including Llama 3 [29], where it contributes to improved inference throughput without significant degradation in quality. It is important to note that Grouped Query Attention (GQA) with a group size of 1 is equivalent to MQA, while GQA with a group size equal to the number of attention heads corresponds to MHA.
While the attention mechanism is the hallmark of the Transformer architec-

ture, which is the backbone of most State of the Art (SOTA) LLMs, there are several other essential deep learning components that also play a critical role within each Transformer block.

2.5 ADDITIONAL COMMON LANGUAGE MODEL BUILDING BLOCKS

Beyond the attention mechanism, Transformer models include several other key components, that are consistently used across architectures. These include feed-forward networks, normalization layers, activation functions, positional encodings and sampling mechanisms. While often treated as standard building blocks, each contributes significantly to both the functionality and computational cost of language model inference. In the following, we briefly examine each of these components with a focus on their roles and efficiency implications.

2.5.1 *Rotary Positional Embeddings*

Rotary positional Embeddings (RoPE) are a SOTA method to efficiently incorporate absolute token position and relative token position into the self attention mechanism, while adding additional desirable properties like flexible sequence length and token inter-dependencies which decay with increasing distance between two tokens. This is achieved by rotating each embedded key and query token vector with the even embedding dimension d at position m denoted by x_m by a multiple of the predefined constant angle θ by calculating

$$Rot(x_m, m, \theta) = R^d_{\Theta,m} W_{(Q,K)} x_m \tag{2.19}$$

where

$$R^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} \end{pmatrix} \tag{2.20}$$

is the corresponding rotation matrix and W_Q, W_K are the learnable Query and Key Matrices. [85, Su et al.] take advantage of the sparsity in 2.20 and

simplify the rotation computation to

$$R^{d}_{\Theta,m}\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix} \tag{2.21}$$

Here m is the absolute position of a token vector representation in a given sequence. Keeping the simplification (2.21) in mind, we formulate the element wise rotation in a singular attention head as following:

$$\text{head}_i^{\text{RoPE}}(Q, K, V, \theta) = \text{Attention}(R^{d}_{\Theta}\mathbf{Q}\mathbf{W}_i^{Q}, R^{d}_{\Theta}\mathbf{K}\mathbf{W}_i^{K}, \mathbf{V}\mathbf{W}_i^{V}) \tag{2.22}$$

Having discussed Rotary Positional Embeddings (RoPE) and their integration into the attention mechanism, we now shift focus to another crucial part of neural network architectures: activation functions.

2.5.2 *Activation Functions*

Activation functions are non-linear functions which are usually applied in neural networks after the weighted linear function. According to multiple studies, mainly [26, 70, 71, 80] the choice of activation function can significantly impact model performance. For this thesis we focus on the commonly used SwiGLU [80]activation function. In order to properly define the SwiGLU function we first have to introduce the sigmoid and Swish activation functions.

SIGMOID ACTIVATION FUNCTION

The (logistic) sigmoid activation function [26] is defined as:

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.23}$$

Using the help of Figure 2.6, we can see, that the sigmoid function takes any numerical input and squeezes it into a value between 0 and 1. This behavior was inspired by biological neurons, which have a probability of firing depending on their inputs. Despite its early success, the sigmoid function is rarely used as an activation function in modern neural networks. The main reason is that for large positive or negative inputs, the sigmoid saturates towards 0 or 1, causing its derivative to approach zero [26]. As a result,

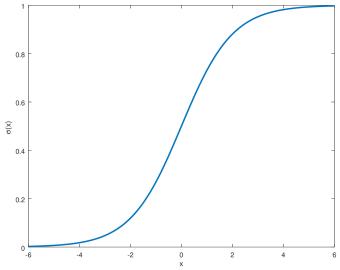

Figure 2.6: Sigmoid Activation Function and its Graph [38]

during backpropagation, the gradients flowing through these saturated neurons become very small. This effectively prevents the affected layers from significantly updating their weights, limiting their ability to learn meaningful representations of the input data and potentially rendering them ineffective. To address these limitations, several alternative activation functions have been proposed that alleviate the saturation problem while preserving desirable nonlinearity. One such function is the Swish activation [71], which has demonstrated improved performance over sigmoid and Rectified Linear Unit (ReLU) 2.1.

SWISH ACTIVATION FUNCTION

The Swish (sometimes also called SiLU) [71] activation function was introduced in 2017 in an attempt to replace the SOTA ReLU [55] function at the time. The Swish paper swish:2017 shows, that on deeper neural networks (40+ layers) Swish outperforms ReLU. This may be because Swish has a one sided zero-boundedness like ReLU, while opposed to ReLU being smooth and non-monotonous [71]. Also a paper [28] shows, that functions like Swish or ReLU, which are saturated towards zero actually improve the supervised learning process as long as they leave a path for the gradient to flow (e.g. at least one nonzero neuron per layer).

$$f(x) = x \times sigmoid(x) \tag{2.24}$$

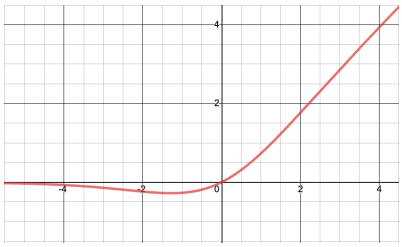

Figure 2.7: Swish Activation Function and its Graph [82]

We also note, that there is a variant of the Switch function which is called

$Swish - \beta$ (often also denoted as $Swish_\beta$). $Swish - \beta$ just adds a learnable parameter β to the Swish function as following:

$$Swish(x; \beta) = x \times sigmoid(\beta x) \tag{2.25}$$

Despite outperforming the long time SOTA ReLU function modern LLMs since 2020 use variations of the Gated Linear Unit (GLU) activation function since they outperform Swish in Natural Language Processing (NLP) tasks [80]. For this thesis we will focus on the SwiGLU function in particular, since it is the activation function used in the Llama 3.x models [29], which are mostly used in our real-world use-case.

SWIGLU ACTIVATION FUNCTION

SwiGLU is a variant of the Gated Linear Unit (Gated Linear Unit (GLU)) [24] and differs notably from the activation functions introduced earlier. Unlike standard activations such as ReLU or Swish, which apply a single weighted linear transformation before the nonlinearity, SwiGLU requires two separately learned linear transformations of the input. Formally, given an input vector x, two learnable parameter matrices W and V and two learnable bias vectors b and c, the SwiGLU activation is computed as follows [80]:

$$\text{SwiGLU}(x, W, V, b, c, \beta) = \text{Swish}_\beta(xW + b) \otimes (xV + c) \tag{2.26}$$

Here $\text{Swish}_\beta(\cdot)$ denotes the $Swish - \beta$ activation function 2.25. Empirical results reported in [80] demonstrate that GLU variants like SwiGLU can improve transformer model performance over conventional activations like ReLU or Swish. However, despite these practical successes, their theoretical underpinnings remain unclear, even Shazeer himself succinctly attributes their effectiveness to "divine benevolence" [80].
Given this empirical motivation for adopting SwiGLU and similar gated activations, we now turn to the broader structure in which they are typically deployed: the feed-forward neural network layers that form a crucial part of transformer architectures.

2.5.3 *Feed-Forward Neural Networks*

Feed-forward neural networks (Feed Forward Neural Network (FFN)s) are a core component of each Transformer block. Positioned after the attention mechanism, they consist of two linear transformations with a non-linear activation function in between. The standard formulation is as follows:

$$\text{FFN}(x) = \text{Activation}(xW_1 + b_1)W_2 + b_2 \tag{2.27}$$

where W_1, W_2 are learnable weight matrices and b_1, b_2 are bias vectors. Typ-

ically, the intermediate dimensionality is larger than the model dimension, increasing expressivity but also computational cost. Despite their simplicity, FFNs significantly contribute to the model's capacity and overall inference time.
In our exemplary case, with Llama 3.1 8B Instruct a SwiGLU FFN we formally introduce its computation similar to [80]:

$$FFN_{SwiGLU}(x, W, V, W_2, b, c, d, \beta) = SwiGLU(x, W, V, b, c, \beta) \times W_2 + d \quad (2.28)$$

Where W, V, W_2 are learnable weight matrices, b, c, d are learnable bias vectors and x is the input tensor.
Since data is usually Normalized before and after traversing the Feed Forward neural Net (FFN) we will now introduce normalization and the different methods that have proven their effectiveness over time.

2.6 NORMALIZATION

In the context of this work, normalization refers to transforming tensors to enforce specific properties. This process is not confined to tensors representing input data, it can equally be applied to the hidden units of model layers. Beyond improving optimization efficiency and enhancing a model's generalization capabilities, normalization plays a critical role in addressing fundamental limitations of certain architectures. In particular, it helps mitigate issues such as exploding and vanishing gradients [11, 13, 40, 72, 97]as well as numerical instabilities like overflows and underflows. Since the introduction and success of Batch Normalization [40], normalization techniques have become an indispensable component of modern deep learning architectures. In the following subsections, we will examine several important normalization methods and their impact on LLMs. We begin with Batch Normalization, one of the earliest and most influential normalizations.

2.6.1 *Batch Normalization*

Batch Normalization (BN) [40] was the first normalization technique to gain widespread adoption in deep learning. The central idea is to transform the inputs x_k of a given layer k so that they have zero mean, $\bar{x}_k = 0$ and unit variance, $\sigma(x_k) = 1$. This is achieved by applying the transformation

$$y_k = \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\sigma[x_k] + \epsilon}} \cdot \gamma + \beta, \quad (2.29)$$

where ϵ is a small constant for numerical stability and γ, β are learnable parameters that allow the model to adapt the normalization if needed. Experiments reported in [40] demonstrated that Batch Normalization (BN) speeds

up training by up to 14× and improves classification accuracy on ImageNet by 2.6% compared to the same at that time SOTA model architecture without BN.
Initially, these improvements were attributed to the reduction of *Internal Covariate Shift*, defined as the change in the distribution of layer activations as model parameters change during training. However, this explanation was later challenged by Danturkar et al. [72], who argued that BN primarily benefits optimization by smoothing the loss landscape rather than mitigating covariate shift.
Despite its effectiveness in many domains, BN is rarely used in language modeling tasks. Its performance depends heavily on large batch sizes and it requires a batch size greater than one to function properly. Furthermore, applying BN to Recurrent Neural Networks (RNNs) has proven challenging [11], which limited its applicability in sequence models like LLMs.
Given these limitations, alternative normalization methods better suited to sequential and small-batch settings have been proposed. One such approach is Layer Normalization, which we will discuss next.

2.6.2 *Layer Normalization*

While BN normalizes the features across the batch dimension, *layer normalization* normalizes the values of hidden units within a given layer.
Ba et al. [11] propose that this form of normalization addresses the *covariate shift* problem while being applicable to sequence models and independent of the batch size.
The general formula for layer normalization is almost identical to Equation 2.29, with the main difference being the dimension over which the mean and standard deviation are computed:

$$y_l = \frac{x_l - \mathbb{E}[x_l]}{\sqrt{\sigma^2[x_l] + \epsilon}} \cdot \gamma + \beta, \tag{2.30}$$

where the expectation $\mathbb{E}$ and variance σ^2 are computed over the hidden units in layer l, as opposed to over the batch.
The mean is computed as follows:

$$\mathbb{E}[x_l] = \frac{1}{H} \sum_{i=1}^{H} a_{l,i}, \tag{2.31}$$

where H denotes the number of hidden units in layer l and $a_{l,i}$ is the i-th activations output within the current layer.
The variance is computed as:

$$\sigma^2[x_l] = \frac{1}{H} \sum_{i=1}^{H} (a_{l,i} - \mathbb{E}[x_l])^2. \tag{2.32}$$

Ba et al. [11] demonstrate experimentally that layer normalization yields improvements similar to batch normalization while remaining applicable to sequential models and invariant to batch size.
Despite its success, especially in tasks such as text generation and machine translation, layer normalization is computationally expensive, slowing down the networks in which it is employed [97].
Root Mean Square Layer Normalization [97] attempts to reduce this computational burden while maintaining the desirable properties of layer normalization, including improved training speed, better generalization and mitigation of training errors.

2.6.3 *Root Mean Square Layer Normalization*

Root Mean Square Layer Normalization (RMSNorm) [97] is a more efficient yet similarly performant layer-wise normalization technique compared to standard Layer Normalization. Zhang et al. [97] argue that enforcing mean invariance does not substantially contribute to the benefits typically attributed to Layer Normalization.
Consequently, they simplify the standard formulation (2.30) to:

$$y_l = \frac{x_l}{\mathrm{RMS}(x)} \times \gamma_l, \tag{2.33}$$

where

$$\mathrm{RMS}(x) = \sqrt{\epsilon + \frac{1}{H}\sum_{i=1}^{H} x_i^2}. \tag{2.34}$$

Root Mean Square Root Normalization (RMSNorm) has been evaluated across a variety of tasks, including machine translation, reading comprehension, image-caption retrieval and image classification.
In all experiments, RMSNorm achieved comparable or better performance on standard test metrics while significantly reducing runtime, yielding speed improvements ranging from 7% to 64%, depending on the model architecture [97].

Overall, RMSNorm demonstrates that simpler normalization techniques can retain or even improve model performance while reducing computational overhead. Building on such architectural refinements, modern neural networks also rely on other structural innovations to facilitate effective training and deeper representations. One of the most influential of these innova-

tions is the introduction of skip connections, which will be discussed in the following section.

2.7 SKIP CONNECTIONS

Skip connections, often also called *Residual Connections*, are additional connections between nodes in different layers of a neural network that bypass one or more layers of nonlinear processing [68]. He et al. [33] demonstrated that skip connections significantly improve model performance, particularly in deeper networks. In the following subsection, we explain how skip connections work and discuss hypotheses about why they are effective.

2.7.1 *Explanation*

He et al. [33] observed that neural network architectures of that time became saturated in accuracy as they grew deeper and eventually even yielded worse results when their depth increased further. Surprisingly, this eventual degradation in accuracy was not due to overfitting. Instead, according to He et al. [33], it was attributed to the increased difficulty of training deeper architectures, especially considering the vanishing and exploding gradient problems [33]. Additionally, He et al. [33] argued that deeper model architectures should at least be capable of matching the performance of their shallower counterparts. This deduction stems from the fact that layers in a deeper model could, in principle, adopt the exact same weights as the shallow counterpart, while the extra layers could simply implement an identity mapping. Based on this insight, He et al. [33] introduced a new building block that facilitates learning an identity mapping: the *Residual Block*.

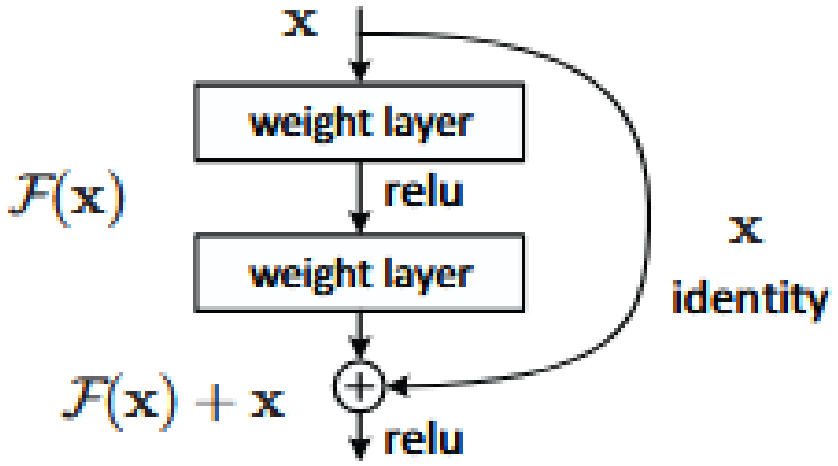

Figure 2.8: Residual Block[33]

He et al. [33] hypothesized that it is easier to learn that a residual is zero

than to learn that a layer implements the identity function. This intuition motivated the introduction of the Residual Block, shown in Figure 2.8. It is important to note that, while in Figure 2.8 $F(x)$ denotes the composition of two weight layers with a ReLU activation in between, this block is applicable for any $F(x)$ as long as the output shape of $F(x)$ matches the shape of input x. In their experiments, He et al. [33] demonstrated that neural networks employing residual blocks, which we from now on we will call ResNets consistently outperformed equivalent architectures without skip connections. Furthermore, they showed that this performance gap increases with model depth [33].
Despite demonstrating superior empirical performance, He et al. [33] did not provide a definitive explanation for why residual blocks facilitate the training of deeper models.
To address this gap, Orhan et al. [68] explored ResNet architectures and proposed that their superior performance arises because residual connections eliminate several types of singularities during optimization:

1. overlap singularities caused by the permutation symmetry of nodes within a layer
2. elimination singularities corresponding to the consistent deactivation of nodes
3. singularities generated by linear dependence among nodes

Since all three forms of singularities impair the training process, Orhan et al. [68] argue, that the improvement from skip connections results, at least partially from mitigating these singularities.
While Orhan et al. [68] demonstrated that the elimination of singularities at least partially contributes to the performance improvements of Residual Neural Network (ResNet)s, they acknowledged that this property alone does not fully explain the observed performance gains.
Building on the understanding of skip connections and their impact on training deep neural networks, we can appreciate how architectural innovations simplify optimization and improve model capacity. One of the most influential advancements that incorporate and extend these principles is the transformer architecture. Transformers leverage all the methods discussed in this chapter and, as of 2025, represent the SOTA for most language modeling tasks. In the following section, we provide an overview of a representative transformer block as it is used in modern high-performance LLMs from the Llama 3.x series developed by Meta.

2.8 TRANSFORMER BLOCK

The transformer block, originally introduced by Vaswani et al. [93], is the backbone component of modern language model architectures. Owing to their remarkable success, transformer-based models have been widely adopted

for a variety of tasks, including machine translation, text classification, semantic retrieval and image classification. In this thesis, however, we focus specifically on the task of text generation. In addition to formally presenting the architectural components and techniques employed in contemporary text generation models, we aim to offer a high-level conceptual explanation of how LLMs predict the next token in a given input sequence, as illustrated in Algorithm 1. As a representative example, we focus on the Llama 3 model family, which is particularly relevant due to the deployment of one of its variants in a production system at dianovi GmbH.

2.8.1 *Hyperparameters*

In the table below we display the Llama 3.1 Model Hyperparameters:

Table 2.1: Llama 3.1 Model Hyperparameters

	8B	**70B**	**405B**
Layers	32	80	126
Model Dimension	4,096	8,192	16,384
FFN Dimension	14,336	28,672	53,248
Attention Heads	32	64	128
Key/Value Heads	8	8	8
Peak Learning Rate	3×10^{-4}	1.5×10^{-4}	8×10^{-5}
Activation Function	SwiGLU		
Vocabulary Size	128,000		
Positional Embeddings	RoPE ($\theta = 500{,}000$)		

The *Layers* row denotes the number of transformer blocks in the model (i.e., N in Figure 2.9). The *Model Dimension* refers to the dimensionality of the hidden representations, resulting in tensors of shape $(b, s_{\text{len}}, d_{\text{model}})$, where b is the batch size, s_{len} is the maximum sequence length within the batch and d_{model} is the model's hidden dimension.
The *FFN Dimension* indicates the size of the feedforward projection layer within each transformer block (see Section 2.5.3). The *Attention Heads* parameter specifies the number of parallel attention mechanisms computed in each self-attention layer (cf. Equation 2.18). The *Key/Value Heads* column corresponds to the number of shared key-value groups used in GQA, as discussed in Section 2.4.5.
The *Peak Learning Rate* represents the maximum learning rate reached during training, prior to any decay schedule. The *Activation Function* used in the feedforward networks is SwiGLU (see Section 2.5.2). The *Vocabulary Size* is 128,000 tokens. The model also employs RoPE [85] with a scaling factor

$\theta = 500{,}000$, as indicated in Equation 2.22.

2.8.2 *Conceptual Explanation of Llama 3.x Models*

Below we display an image displaying the architecture of any Llama 3.x model:

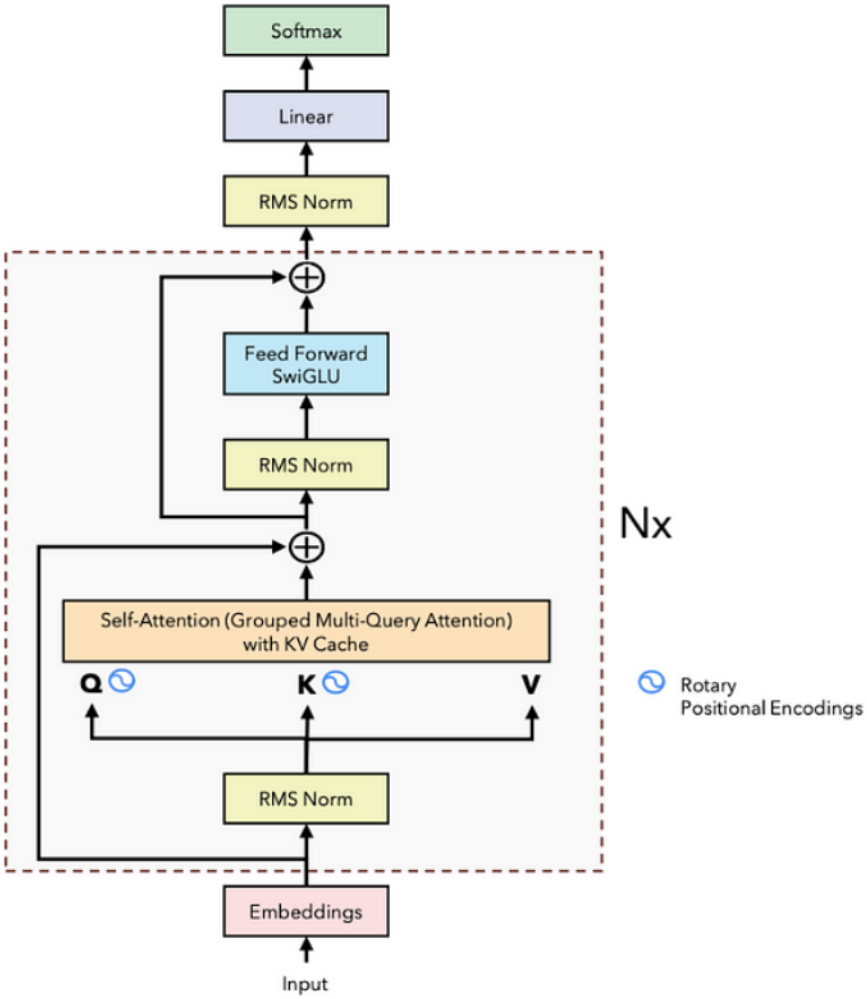

Figure 2.9: General architecture of the Llama 3 model series. Nx indicates the number of stacked transformer blocks[96]

Figure 2.9 shows a simplified view of how a Llama 3.x language model works [29].

SIMPLIFIED CONCEPTUAL EXPLANATION

In this paragraph we will explain the key concepts of a LLM in s simplified manner, readers who have a solid understanding and intuition of LLMs inner workings may skip this section. This model is an extremely advanced

text autocompletion: when you give it some words, it tries to guess what word comes next. The process begins with **tokenization**. This means the input text (like a sentence) is broken down into smaller units called *tokens*. These tokens are not always full words, they can be parts of words, especially in languages with many word forms. For example, the word "unbelievable" might be split into "un", "believ" and "able". Each token is then assigned a unique number (an *ID*) from the model's vocabulary. Next, these token IDs are passed through an **embedding layer**. This layer turns each token ID into a list of numbers, known as a *vector*. These vectors (called *embeddings*) capture some basic information about each token, such as meaning or usage. While token IDs are just simple lookup codes, embeddings are richer and more useful, since they can represent more nuanced meanings then just a single number. Once the tokens are embedded into vectors, the model applies a technique called **RMS Normalization** (explained in Section 2.6.3) to adjust their values. This helps keep things stable during training and makes learning easier. The next step is called **self-attention** (see Section 2.4.5). This is where the model looks at all the tokens at once and figures out which ones are important in relation to each other. For example, in the sentence "The cat sat on the mat," it learns that "cat" and "sat" are related. To help the model understand the order of words, it adds extra information about position using something called **Rotary Positional Embeddings** (see Equation 2.22). To make this attention step faster and more efficient, the model uses a trick called **Grouped Multi-Query Attention** (Section 2.4.3). This lets multiple attention components share information, saving memory with only negligible accuracy loss. After attention, the output is combined with the input using a **residual connection** (Section 2.7),basically a shortcut that helps the model remember the original input, which also helps large models to learn better and more efficiently. Then the result is normalized again and passed through a small neural network called a **Feedforward Network**, which uses a special mathematical function called **SwiGLU** 2.5.2 to process the data further, enabling the model to learn better. All of these steps, attention, shortcuts, normalization and feedforward are repeated many times (specifically, N_x times, where N_x is the number of layers in the model). Each repetition helps the model build a deeper understanding of the text. Finally, the model takes its final output and transforms it into scores (called *logits*) for every possible word in its vocabulary. These scores are turned into probabilities using something called the **Softmax** function. The word with the highest probability is the model's best guess for what comes next. During training, it compares this guess to the actual next word and learns from its mistakes.

2.8.3 *Comments on Common Large Language Model Misconceptions*

Despite the impressive capabilities of transformer-based architectures like Llama 3, there are widespread misconceptions, which are sometimes amplified by marketing or popular media, mostly about what such models can or

will do. We address the following misconceptions:

- **"We are close to achieving Artificial General Intelligence (AGI)."**
  Current LLMs (LLMs), as explored in this chapter, are powerful statistical pattern matchers trained on massive text corpora [61]. There is no proof that they possess understanding, consciousness, or general reasoning abilities beyond their training data and learned heuristics [61]. While they can generate convincing text, this should not be confused with true general intelligence or autonomy [61]. Statements implying that AGI is imminent often overstate current progress and underestimate the complexity of general cognition [61]. Some experts even claim that LLMs will never be able to achieve Artificial General Intelligence (AGI) [94]. Additionally, there is the ARC Competition [17], which offers a total prize pool of 1 million US dollars. In this competition, a score of 85% or higher would be considered indicative of AGI by the organizers. As of July 8, 2025, the best score achieved by competitors is 15.42%, clearly indicating that reaching AGI is extremely challenging and that we are currently far from achieving it.

- **"AI will replace programmers."**
  LLMs can help with code completion, refactoring and generating boilerplate, but they are not yet capable of reliably designing robust systems, performing deep debugging, or interpreting nuanced requirements without significant human oversight. Rather than serving as a wholesale replacement, these models are better understood as productivity tools that augment human programmers by automating repetitive tasks and providing contextual suggestions.
  However, critical thinking, domain expertise and thorough validation remain indispensable. For example, in closed-source benchmark tasks such as those in the Konwinski Prize challenge [44],domain experts using state-of-the-art LLMs and advanced NLP techniques achieved only a score of 0.0985 on the following evaluation metric:

$$\text{score} = \frac{n_{\text{correct}} - n_{\text{incorrect}} - \frac{n_{\text{skipped}}}{10000}}{n_{\text{correct}} + n_{\text{incorrect}} + n_{\text{skipped}}} \tag{2.35}$$

  This result underscores the significant performance gap between LLM-assisted automated approaches and professional programmers, who have been able to solve all of these problems, since otherwise they could not have been benchmarked.

- **"Bigger models are always better."**
  While scaling up model parameters generally improves raw benchmarks within the same training data and architecture (e.g., perplexity), larger models can also lead to diminishing returns, higher infer-

ence costs and increased brittleness in unexpected contexts. Moreover, a study conducted by Meta has demonstrated that smaller models with improved architectures can outperform larger ones [56] and that smaller models trained on less, but better curated data can also surpass larger models trained on massive, unfiltered text corpora [56].
For instance, on the MedQA benchmark, GPT-3.5 with its 175 billion parameters achieves 4.2% lower accuracy than Llama 3.1 Instruct Turbo, which has only 8 billion parameters [5].

- **"Language models understand language the way humans do."**
  Although models produce fluent text, they lack grounded semantic understanding and instead rely on correlational patterns [61]. This limits their reliability in domains requiring reasoning about the world or verifying factual correctness [61].

In sum, while LLMs like Llama 3 represent a significant engineering achievement, it is important to maintain realistic expectations about their capabilities and limitations. Because of this in our opinion it is essential to critically reflect upon social media or company claims and check their factual correctness, especially in the complex topic of Artificial Intelligence (AI) / LLMs.

2.9 FLASHATTENTION

With the success and widespread adoption of the beforementioned Transformer architecture, the improvement of their training and inference efficiency has become a central topic of research. While our work up to now centers on explaining architectural improvements, we acknowledge that comparatively less attention has been paid in this thesis to explain the optimization of the characteristics of the underlying hardware during inference. For LLM inference, the most commonly used hardware is the Graphics Processing Unit (GPU), which is specifically designed for massively parallel computation. This makes it particularly suitable for workloads dominated by tensor operations.
Despite the parallelism afforded by Graphics Processing Unit (GPU)s, Dao et al. [23] have identified, that the attention mechanism in Transformers involves a large number of memory access (I/O) operations, which can become a bottleneck in both training and inference of attention base models like LLMs.
To mitigate this issue, they introduce *FlashAttention*, an I/O-aware attention algorithm that reduces memory overhead through optimized memory access patterns. Their implementation achieves up to a $3\times$ speedup in training and inference, while also improving accuracy on standard benchmarks [23].
We emphasize that FlashAttention does not modify the mathematical formulation of the attention mechanism itself. Instead, it restructures the computa-

tional workload to minimize redundant data movement and maximize GPU floating-point operations per second (FLOP) utilization. As such, we regard these developments as problems of information science and hardware-aware algorithm engineering, rather than as mathematical contributions.
Accordingly, we treat FlashAttention as an external and mature line of work and will refer to the relevant publications when necessary. Due to its demonstrated impact, this area remains under active development,with later iterations, namely FlashAttention 2 [22] and FlashAttention 3 [78]continuing to push the boundaries of GPU utilization and numerical precision.

2.10 PRECISION IN MODERN COMPUTERS

To estimate inference costs for LLM operations it is necessary to understand how modern computers represent numbers and perform calculations. At a fundamental level computers calculate with bits, which can be either zero or one (which in hardware terms meaning voltage passing through transistors for a one or zero if not). Then multiple bits are combined to represent a number, for example unsigned integers in a precision of eight bits are represented by a sequence of eight binary operators, which encode the following formula [83]:

$$b_1 \times 2^7 + b_2 \times 2^6 + b_3 \times 2^5 + b_4 \times 2^4 + b_5 \times 2^3 + b_6 \times 2^2 + b_7 \times 2^1 + b_8 \times 2^0 \tag{2.36}$$

For example, the number 135 can be represented in binary as $10000111 = 2^7 + 2^2 + 2^1 + 2^0 = 1 + 2 + 4 + 128 = 135$. Representing real numbers, however, is more complex, as it requires encoding both a sign and decimal values. This is typically achieved using the following formula [83]:

$$x = \text{sign} \times \text{mantissa} \times 2^{\text{exponent}} \tag{2.37}$$

In this representation, one bit is reserved for the sign, while the remaining bits are divided between the mantissa and the exponent [83]. It's important to note that, for a given level of precision, the number of unique values that can be represented is always exactly $2^{\text{precision}}$. Whereas the range of representable values depends on how the bits are allocated among the different components.

3 INFERENCE COST

Since the goal of this thesis is to optimize the runtime of LLM-based systems, we must first examine two key aspects:

1. whether a given setup can perform inference within its hardware constraints, which we investigate in Chapter 4 and
2. how long a specific model, in a given setup, will approximately take to generate a sequence of length L.

In this chapter, we estimate the number of floating-point operations (FLOP) required by a model to generate a sequence of length L.
For consistency and real-world relevance, we use the Llama 3.1 8Billion (B) Instruct model as a representative example. We analyze the inference cost of all relevant steps outlined in Chapter 2.

3.1 COMMON OPERATION COSTS

Most LLMs, including Llama 3.1 8B Instruct primarily consist of matrix multiplications, element-wise operations and softmax functions, we introduce the Floating Point Operations (FLOP) costs of these common operations to simplify later calculations.

3.1.1 *Cost of Matrix Multiplication*

According to [91], the number of FLOP required for a matrix multiplication between matrix A of shape (M, N) and matrix B of shape (N, L) is:

$$\text{FLOP}_{\text{MatMul}} = 2MNL - ML \tag{3.1}$$

3.1.2 *Cost of Element-Wise Matrix Multiplication or Addition*

For element-wise multiplication or addition between two matrices of shape (M, N), each element requires one FLOP. Therefore, the total FLOP cost is:

$$\text{FLOP}_{\text{mul, add}} = MN \tag{3.2}$$

3.1.3 *Cost of the Softmax Operation*

According to the TensorFlow Profiler [90], the softmax operation over N elements (logits) takes approximately:

$$\text{FLOP}_{\text{softmax}} = 5N \tag{3.3}$$

FLOP to calculate.

3.1.4 *Cost of a weighted Linear Layer*

When a matrix addition follows a matrix multiplication, as it is done in a learnable weight layer the FLOP cost simplifies to:

$$\begin{aligned} \text{FLOP}_{\text{Layer}} &= \text{FLOP}_{\text{MatMul}} + \text{FLOP}_{\text{add}} && (3.4)\\ &= (2MNL - ML) + ML && (3.5)\\ &= 2MNL && (3.6) \end{aligned}$$

Here, the addition cost is ML, corresponding to the shape of the matrix resulting from multiplying shapes (M, N) and (N, L), which yields a matrix of shape (M, L).

3.2 TOKENIZATION COST

The Llama 3.1 8B Instruct model employs a Unigram SentencePiece [45] tokenizer, whose inference cost scales linearly with the input sequence length L [45].
However, since Kudo et al. [45] do not specify the constant of proportionality in this linear relationship, accounting for $FLOP_T = cs_L$.

3.3 EMBEDDING COST

As discussed in Section 2.3.2, embeddings are implemented as a learned lookup table that maps each unique token ID to a vectorized representation. This operation is at inference time merely a look up and thus has constant-time complexity, making it computationally insignificant in practice. Therefore, its FLOP contribution during inference is so insignificant, that we regard it as zero.

3.4 TRANSFORMER BLOCK COST

The transformer block forms the computational backbone of modern LLMs, performing the majority of work in the model architecture. Many LLMs, including our example model, Llama 3.1 8B Instruct stack multiple trans-

former blocks sequentially. In this section, we compute the FLOP cost of the mathematical operations within a single transformer block, parameterized by relevant hyperparameters. In the subsequent section, we generalize this to estimate the total compute required for a complete text generation scenario.

3.4.1 *Skip Connection Cost*

As illustrated in Figure 2.9, each transformer block begins with the first part of a skip connection, which involves copying the input tensor. From a FLOP perspective, this operation is constant-time and thus negligible.
The second part of the skip connection involves an element-wise addition between two tensors of shape (b, s_L, d_m), where:

- b: batch size,
- s_L: length of the longest sequence in the batch,
- d_m: the model's hidden dimension.

Since the tensors contain $n_{elem} = bs_Ld_m$ elements and the addition is element-wise, the FLOP count is:

$$FLOP_{skip} = bs_Ld_m \tag{3.7}$$

3.4.2 *RMSNorm Cost*

RMSNorm is typically applied to each token vector within every sequence in a batch. The FLOP required per batch are:

$$FLOP^{Batch}_{RMSNorm} = bs_L(FLOP^{Vector}_{RMS} + 2d_m) \tag{3.8}$$

Where the addition in the end accounts for one multiplication and one division per element (see Equation (2.33)). Using the RMS equation from (2.34), the FLOP for normalizing a single token vector of dimension H are:

- H element-wise squares (1 FLOP each),
- $H + 1$ additions,
- 1 square root,
- 1 division,

This yields:

$$FLOP^{Vector}_{RMS} = H + 1 + 1 + H + 1 = 2H + 3 \tag{3.9}$$

Substituting (3.9) into Equation (3.8), we obtain:

$$FLOP_{RMS}^{Batch} = bs_L(2H + 3 + 2d_m) \tag{3.10}$$

Since H is equal to d_m in our case, we can simplify further:

$$FLOP_{RMS}^{Batch} = bs_L(4d_m + 3) \tag{3.11}$$

3.4.3 *RoPE Cost*

Since RoPE in its simplified form (2.21) is a token-wise applied vector rotation consisting of two element-wise vector multiplications, one element wise vector addition and one vector permutation, which according to [91] can be regarded as taking zero FLOP. We assume the cost of calculating $\cos(m\theta_i), \sin(m\theta_i)$ for the elements inside the two rotation vectors as constant, since these values can be precomputed and looked up at inference time. So in total, we can calculate the cost of applying RoPE to an input batch as:

$$FLOP_{RoPE} = b(3s_L d_m) \tag{3.12}$$

3.4.4 *Attention Block Cost*

Due to the fact, that GQA calculates n_g projections of Q, K, V we have to calculate $3n_g$ tensor-matrix multiplications of shape $(b, s_L, d_m)(d_m, d_h)$ where $d_h = \lfloor \frac{d_m}{n_h} \rfloor$. This tensor-matrix product essentially just computes the matrix multiplication with the matrices of shapes $(s_L, d_m)(d_m, d_h)$ b-times. Additionally we have to factor in the addition of the bias terms, which lets us use formula (3.4) to calculate the FLOP for each projection. So in total we have

$$FLOP_{P,i} = b(2s_L d_m d_h n_g) \tag{3.13}$$

FLOP per projection which for our three different projections Q, K, V totals a projection cost of

$$FLOP_P = 6bs_L d_m d_h n_g \tag{3.14}$$

per GQA-block.
Then we have to compute the head-wise attention scores as described in (2.17) applying the formula (2.16) to all n_g Q, K, V projections of shape (b, s_L, d_h).

First we compute QK^T, which is a batch-element-wise matrix multiplication with matrices of shapes $(s_L, d_h), (d_h, s_L)$ which take a total amount of $2d_h s_L^2 - s_L^2$ FLOP per batch element. Then we have to element-wise divide the output matrices of shapes $(s_L, s_L)$by $\sqrt{d_h}$, which takes a total amount of s_L^2 FLOP per batch element. After that we take the row-wise softmax of the resulting matrices,which costs us $5s_L$ operations per token in s_L. Using this information we can calculate that computing the softmax for all attention heads takes $5s_L^2$ operations per batch element. Finally, we batch-wise multiply the Attention-Matrix of shape (s_L, s_L) with the value matrices of shape (s_L, d_h), costing us another $2s_L^2 d_h - s_L d_h$ FLOP and resulting in matrices of shape (s_L, d_h). Using these insights we can formulate the amount of FLOP taken per attention head as

$$FLOP_{head_i} = s_L(d_h[4s_L - 1] + 5s_L) \tag{3.15}$$

Now that we know $FLOP_{head_i}$ we can calculate the amount of FLOP per Attention Block as in (2.17). Since the concat operation does not perform any calculations we assume its FLOP amount to be zero. Now we can finalize the FLOP count per attention block by computing the amount of FLOP needed for our final matrix multiplication between our concatenated multi head attention matrices of shape (s_L, d_m) and the final weight matrix W_O of shape (d_m, d_m), which including the addition of learned bias terms takes another $2s_L d_m^2$ FLOP per batch element. This results in a total amount of

$$FLOP_h = b(n_h FLOP_{head_i} + 2s_L d_m^2) \tag{3.16}$$

which fully written out and simplified, while regarding $d_M = n_h d_h$ equalizes to

$$FLOP_h = b(n_h s_L(d_h[4s_L - 1] + 5s_L + 2d_m^2) \tag{3.17}$$

Having established the computational cost of the attention mechanism in detail, we now turn to the final core component of the transformer block: the feed-forward neural network (FFN). This module is applied independently to each token in the sequence and is responsible for the majority of the model's parameter count. In the following subsection, we derive the exact FLOP requirements of the FFN and incorporate them into our overall cost estimation.

3.4.5 *Cost of Feed-Forward Network (FFN)*

Since we use Llama 3.1 8B Instruct as a representative model, we compute only the FLOP required for a SwiGLU FFN as described in Equation 2.28. According to (2.26), the SwiGLU activation involves two linear activations, one element wise matrix multiplication and one $Swish_\beta$ activation. The $Swish_\beta$

activation consists of an element-wise matrix multiplication, a scalar-matrix multiplication and an element-wise sigmoid function. According to Serhiienko et al., a single sigmoid operation requires 8 FLOP.
Using this, we can calculate and simplify the FLOP per $Swish_\beta$ activation for a sequence of length s_L as follows:

$$FLOP_{Swish_\beta} = 10 s_L d_{in} \tag{3.18}$$

Where d_{in} stands for the inner dimension of the Feed-Forward Network. Using Equation (3.18), the total FLOP required for one SwiGLU FFN is:

$$FLOP_{FFN\,SwiGLU} = b(FLOP_{Swish_\beta} + 2FLOP_{linear} + FLOP_{MatMul}) \tag{3.19}$$

where:

$$FLOP_{linear} = 2 s_L d_m d_{in} \tag{3.20}$$
$$FLOP_{MatMul} = s_L d_{in} d_m \tag{3.21}$$

Substituting these terms into Equation (3.19) and simplifying yields:

$$FLOP_{FFN} = b s_L d_{in}(10 + 5 d_m) \tag{3.22}$$

3.4.6 *Total Cost*

To improve readability and reduce clutter in lengthy equations, we adopt a shortened notation for frequently used variables. Specifically, we use d_m for d_m, d_i for d_{inner}, d_h for d_h, n_h for n_h, d_F for d_{FFN}, n_g for n_g and F for $FLOP$. These abbreviations are used consistently throughout the remaining equations where appropriate.
Now that we know the formulas for each operation conducted inside a transformer-block we can calculate the cost for a transformer block depicted in 2.9 as following:

$$F_{TB,i} = 2F_{skip} + 2F_{RMS}^{Batch} + F_{RoPE} + F_P + F_h + F_{FFN} \tag{3.23}$$

which fully written out and simplified equates to

$$F_{TB,i} = b s_L (6 + 10 d_i + d_m[15 + 2 s_L + d_m + 6 d_h n_g + 3 d_i]) \tag{3.24}$$

FLOP.

3.5 COST OF FINAL OPERATIONS

The final linear layer, as shown in Figure 2.9, transforms the input of shape (b, s_L, d_m) into (b, s_L, L_v), where L_v is the number of unique vocabulary tokens in the tokenizer.
This transformation is performed using a batched matrix multiplication of shape $(s_L, d_m)(d_m, L_v)$ followed by a bias addition, requiring:

$$F_{Linear} = b(2s_L d_m L_v) \tag{3.25}$$

Finally, a softmax operation is applied batch-wise to the last token of each sequence, yielding:

$$F_{softmax}^{end} = 5bL_v \tag{3.26}$$

Having quantified the computational cost of the final linear projection and the following softmax operation, we now know the FLOP of all model operations depicted in 2.9 which we will use in the following sections to calculate the FLOP of one forward pass.

3.5.1 *Total Model Cost*

Now that we know the costs of each unique operation performed inside Llama 3 2.9 we can calculate the total FLOP count for one forward pass as following:

$$F_{forward} = bF_T + n_B F_{TB,i} + F_{RMS}^{Batch} + F_{Linear} + F_{softmax}^{end} \tag{3.27}$$

4

BATCHING FOR EFFICIENT MULTI-INPUT INFERENCE

As discussed in 3, modern large language models are extremely computationally demanding, with model sizes ranging from 1 billion to 685 billion parameters. Efficiency is therefore paramount.
In the real-world use-case inference must be fast, since emergency doctors have little time to spare. One simple but effective method to improve throughput in systems that process multiple inputs simultaneously is batching, which we will discuss in this chapter.

4.1 PRINCIPLES OF BATCHING

The core idea of batching in the context of LLMs is to process multiple inputs simultaneously, leveraging the parallelization capabilities of modern GPUs. We note, that in this chapter we only refer to batching at inference time, not as hyperparameter during model training.
In this context, *parallelization* means that many computations can be carried out concurrently, in contrast to the largely sequential processing of central processing units (CPUs). This parallelism enables GPUs to perform many operations in nearly the same time it takes to compute a single operation [62], provided that the operations can be vectorized, sufficient GPU memory is available and the workload does not exceed the GPU's core limits. For reference, the current state-of-the-art (SOTA) server-grade GPU (NVIDIA A100) offers 432 tensor cores [63], while the leading server-grade CPU (Intel® Xeon® 6780E Processor) has 144 cores [39].
For example, whereas a Central Processing Unit (CPU) executing sequentially would take roughly twice as long to compute the operations $a \times b$ and $c \times d$, a GPU can perform the vectorized operation

$$[a, c] \circ [b, d] = [\, a \times b,\ c \times d \,]$$

in almost the same time it takes to compute a single multiplication, producing the results as a single array instead of separate sequential outputs.
In LLMs, we extend the principles discussed in 2 by introducing a new dimension: the batch size b. This allows the model to process multiple inputs in parallel, as long as memory resources permit. Despite its advantages, batching introduces some challenges, particularly shape mismatches arising from variable input sequence lengths.
This issue is addressed through a technique called *padding*, which we discuss in the next section.

4.1.1 *Padding*

While vectorization is an effective strategy for speeding up inference, we cannot simply combine arbitrary-length sequences into a tensor without additional considerations. For example, consider the tokenized sequences with IDs $[1, 2]$ and $[3, 4, 5, 6, 7]$. These cannot be placed into a rectangular matrix without either introducing inefficient dynamic shapes or incurring inconsistent memory allocation, which degrades performance by preventing optimal workload distribution across GPU cores. To address this, transformer architectures employ a special *padding token*, which signals to the model that the corresponding positions should be ignored. In causal language models (which LLMs are), padding tokens are appended to the right of each sequence to avoid issues with causality and caching when generating new tokens. For example, using a padding token with ID 0, the padded input for our earlier example would be:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 2 \\ 3 & 4 & 5 & 6 & 7 \end{bmatrix}.$$

To ensure the padding tokens do not affect the model outputs, most architectures use an *attention mask*, which assigns a large negative value to the positions of the padding tokens. This effectively zeroes them out before the softmax operation in the attention mechanism. The attention mask for this example would be:

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

where in practice, $-\infty$ is sometimes implemented as a huge negative constant. With this foundation for understanding batching and the potential speedups it enables, next we will examine a more practical constraint: GPU Video Random Access memory (VRAM). We will derive a general formula to approximate the maximum possible batch size for a given causal language model and VRAM limitation.

4.2 THEORETICAL DERIVATION OF MAXIMUM BATCH SIZE

To estimate the maximum batch size b_{max}, we first need to calculate the peak VRAM usage of a given model at inference time, assuming a fixed sequence

length. Our approximation begins with the following definitions:

$$W_m = \text{model weight size (in bytes)} \tag{4.1}$$
$$C_{K,V} = \text{Key-Value (KV) cache size (in bytes)} \tag{4.2}$$
$$A = \text{activation memory during inference (in bytes)} \tag{4.3}$$
$$\text{VRAM}_{\text{used}} = W_m + C_{K,V} + A \tag{4.4}$$

Here, the memory units are consistently expressed in bytes.

4.2.1 *Model Weight Size*

The total memory occupied by model weights can be computed as:

$$p = \text{precision per parameter (in bytes)} \tag{4.5}$$
$$n = \text{total number of learnable parameters} \tag{4.6}$$
$$W_m = p \times n \tag{4.7}$$

Substituting this into equation (4.1) yields:

$$VRAM_{used} = p \times n + C_{K,V} + A \tag{4.8}$$

4.2.2 *Key-Value Cache Size*

The Key-Value (KV) cache memory depends on the attention mechanism used. We base our formulation on GQA form subsection 2.4.5, since MHA from section 2.4.3 and MQA from section 2.4.4 are special cases of GQA.
The KV cache consists of cached key and value tensors, so its size is the sum of the key cache size C_K and value cache size C_V:

$$C_{K,V} = C_K + C_V \tag{4.9}$$

Each cache tensor C_i, for $i \in \{K, V\}$, stores values for b sequences of length s_l, for n_L distinct transformer layers, with a group dimension dim_g and byte precision p_i. Thus, the tensor shape for one batch element is (n_L, s_l, dim_g) and the corresponding memory is:

$$C_i = b \times n_L \times s_l \times dim_g \times p_i \tag{4.10}$$

Substituting (4.10) into (4.9) gives:

$$C_{K,V} = b \times n_L \times s_l \times dim_g \times (p_K + p_V) \tag{4.11}$$

Replacing $C_{K,V}$ in (4.8) results in:

$$VRAM_{used} = p \times n + b \times n_L \times s_l \times dim_g \times (p_K + p_V) + A \tag{4.12}$$

4.2.3 *Activation Memory Size*

We approximate the activation memory for one batch as memory cost of applying the Swish Activation 2.24, which due to copying accounts for 4 tensors of shape (b, s_L, d_{FFN}) in precision p

$$A = 4 \times b \times s_L \times d_{FFN} \times p \tag{4.13}$$

Incorporating this into our VRAM usage formula (4.12), we get the final general expression:

$$VRAM_{used} = p \times n + b \times n_L \times s_l \times dim_g \times (p_K + p_V) + 4 \times b \times s_L \times d_{FFN} \tag{4.14}$$

Having established this formula for VRAM consumption, we can now use it to predict the theoretical maximum batch size supported by a given model configuration, hardware setup and sequence length at inference time.

4.3 DERIVATION OF MAXIMUM BATCH SIZE

Given a constant GPU memory capacity $VRAM_{max}$ we have established the following relation:

$$VRAM_{max} = p \times n + b \times n_L \times s_l \times dim_g \times (p_K + p_V) + 4 \times b \times s_L \times d_{FFN} \times p \tag{4.15}$$

Since all parameters except the batch size b are known constants in our case, this expression can be simplified as follows:

$$VRAM_{max} = c_{Model} + b \times c_{K,V} + b \times c_A \tag{4.16}$$

Where $c_{model} = p \times n$, $c_{K,V} = n_L \times s_l \times dim_g \times (p_K + p_V)$ and $c_A = 4 \times s_L \times$

$d_{FFN} \times p$. Solving Equation (4.16) for b yields:

$$b = \frac{VRAM_{max} - c_{Model}}{c_{K,V} + c_A} \tag{4.17}$$

Since batch size must be an integer, we finalize the formula by rounding down to the nearest whole number:

$$b = \left\lfloor \frac{VRAM_{max} - c_{Model}}{c_{K,V} + c_A} \right\rfloor \tag{4.18}$$

After deriving a formula to approximate the maximum batch size that can fit within the available GPU memory ($VRAM_{max}$) for a model with specified parameters and settings, we will validate its accuracy in the experimental section 7.4.

5 QUANTIZATION

As established in the Inference Cost Chapter 3, LLMs demand significant processing power and memory resources. Furthermore, we have demonstrated that numerical precision significantly affects the memory footprint of these models 4. Beyond memory, precision also has a pronounced impact on runtime performance, as shown in [41].
Quantization [41] is an umbrella term for techniques that convert mathematical operations in machine learning models to lower-precision representations. This approach speeds up computations and reduces memory consumption, albeit at the cost of some reduction in model accuracy.
In this chapter, we first introduce the principles of quantization and explain how various techniques are applied in practice. Next, we derive a theoretical formulation for the impact of quantization on a model's output probabilities. Finally, we conclude by experimentally comparing different quantization strategies to highlight their respective trade-offs and to test how well our formula holds up in practice.
We now turn to a detailed discussion of the most popular quantization techniques.

5.1 POST TRAINING QUANTIZATION (PTQ)

PTQ is the process of converting a model that was originally trained using a higher-precision data type, which in LLMs is typically 32-bit floating point (float32), into a lower-precision format such as float16, bfloat16 or int8.

5.1.1 *16-Bit Quantization*

While converting data from float32 to float16 or bfloat16 might seem trivial, it can harbor hidden issues like introducing undefined values (we call an undefined value Not a Number (also used for undefined) (NaN) from here on out) [37]. Mostly, this happens when using normalization layers, since they require a small value ϵ to be added to the denominator to avoid division by zero errors. Since most model engineers want to minimize the influence of that ϵ, it is usually a tiny number. Due to the lower precision of float16 or bfloat16, this number cannot always be represented accurately and is often cast to a NaN or zero, which causes NaN values in all operations it is used in, rendering the models predictions unusable [37].
Additionally, lower precision can cause NaN values if one of the numbers

in the calculation becomes too large, since the maximum values for bfloat16 and float16 are approximately

$$2^{2^7} = 2^{128} \approx 3.4 \times 10^{38}$$

and

$$2^{2^4} = 2^{16} \approx 6.5 \times 10^4,$$

, respectively [73]. This is especially a problem for float16 quantization, since float32, like bfloat16, has 7 bits reserved for the exponent, resulting in the same value range as 5.1.1. So we can clearly see, that there is a large range of numbers that are defined in float32 that bfloat16 can represent, but float16 is unable to represent.
Since in current architectures large numbers are common and decimal precision is less important and bfloat16 can be directly used as a float32 replacement in training and inference. This explains why bfloat16 is currently the preferred lower precision data type [73].
While 16-bit formats such as float16 and bfloat16 offer a reasonable compromise between memory savings and numerical stability, there are scenarios where even more aggressive compression is desirable. In particular, reducing model size and improving inference latency on resource-constrained hardware often requires moving beyond floating-point representations altogether. This motivates the use of integer quantization, most commonly to 8 bits, which further decreases memory footprint and computational cost, but introduces additional challenges related to dynamic range and accuracy.

5.1.2 *8-Bit Quantization*

While 16-bit quantization is relatively straightforward, since it can be performed within the same data type, 8-bit quantization is more complicated because it typically involves using a different data type, namely `int8` [37]. The reason for this data type switch is that there is currently no standardized `float8` data type. The current practical solution to this issue is to map the output range $[\alpha, \beta]$ into the `int8` space,which is precisely what is done in 8-Bit Quantization [41].
The formula for the quantized value of $x[\alpha, \beta]$, which we from now on call $x_q[\alpha_q, \beta_q]$ is derived from the so-called *Affine Quantization Scheme* [41].

$$x = S \times (x_q - Z) \tag{5.1}$$

Where $S = \frac{\beta - \alpha}{\beta_q - \alpha_q}$ is a float32 scaling factor and $Z = \lfloor \frac{\beta \times \alpha_q - \alpha \times \beta_q}{\beta - \alpha} \rceil$ is the int8 representation of zero, which we have to have for padding and masking purposes and the occasional ReLU function or its variants. Using 5.1 we can

infer that

$$x_q = \lfloor \frac{x}{S} + Z \rceil \tag{5.2}$$

We have to round since `int8` only supports whole numbers. A special case arises when $\alpha = -\beta$, which leads to $\alpha_q = -\beta_q$. In this situation, the zero-point offset $Z = 0$ and there are exactly 127 positive and 127 negative values. The two remaining bits are reserved for representing zero and the mathematical sign [37]. This scheme is called *symmetric quantization mapping* [37]. We note that affine quantization is invariant to precision, meaning this scheme is applicable for any arbitrary bit precision [37].
Despite understanding how to calculate the quantized value x_q for any given float32 x we still have to find a way to determine the range of values $x[\alpha, \beta]$. Finding this range is a process called *Calibration*. Currently, there are two commonly used post-training calibration approaches:
Dynamic Quantization, which computes the activations ranges at runtime. While this is convenient and produces great results, it introduces overhead and is not supported by all hardware devices.
Static Quantization precomputes activation ranges.
This is done by passing exemplary data through the model and recording its activation values. Once all examples are passed through the model, usually one of the following calibration techniques is used to compute the static values $[\alpha, \beta]$ for each activation:

- **Min-max:** The computed range is

  $$[\text{min observed value}, \text{max observed value}]$$

  This works well with weights.

- **Moving average min-max:** The computed range is

  $$[\text{moving average min observed value}, \text{moving average max observed value}]$$

  This works well with activations.

- **Histogram:** Records a histogram of values along with min and max values, then chooses the range according to some criterion:
  - **Entropy:** The range is computed as the one minimizing the error between the full-precision and the quantized data.
  - **Mean Square Error:** The range is computed as the one minimizing the mean square error between the full-precision and the quantized data.
  - **Percentile:** The range is computed using a given percentile value p on the observed values. The idea is to try to have p% of the observed values in the computed range. While this is possible

when doing affine quantization, it is not always possible to exactly match that when doing symmetric quantization.

5.1.3 *Generative Pre-trained Transformer Quantization (GPTQ)*

Generative Pre-trained Transformer Quantization (GPTQ) is a post-training quantization method designed to efficiently quantize large-scale transformer models, with a particular focus on minimizing the memory bandwidth bottleneck of GPUs.
This is accomplished by applying the Cholesky decomposition [27] to the inverse Hessian matrix of the model's weights, allowing for an efficient adjustment of the columns not affected by the current update in order to preserve numerical stability. Simultaneously, the current batch of columns is updated to minimize the MSE as defined in Equation 2.4. For a comprehensive explanation, we refer the reader to the GPTQ paper [27] by Frantar et al.

5.1.4 *Activation-Aware Weight Quantization (AWQ)*

Activation-Aware Weight Quantization (AWQ) [54] is a recent post-training quantization technique that achieves high compression rates with minimal accuracy degradation, particularly effective on LLMs. It extends traditional weight-only quantization by taking into account the activation statistics of each input channel during calibration.
The core idea of Activation-Aware Weight Quantization (AWQ) [54] is to scale weights on a per-channel basis, informed by the relative importance of each input channel. This importance is estimated using activation values collected during a short calibration run, which makes AWQ a static quantization technique.
These scaling weights are determined by using a calibration dataset to solve the following equation [54]:

$$s^* = argmin_s L(s) \tag{5.3}$$

with

$$L(s) = \|Q(W \cdot diag(s))(diag(s)^{-1} \cdot X) - WX\| \tag{5.4}$$

Here Q is the Quantization function, W are the original weights, $diag(s)$ makes a diagonal matrix out of the input vector s and X are the cached input features form the calibration dataset.
To solve Equation (5.3) in a stable manner, Lin et al. modify the search space as following[54]:

$$\alpha^* = argmin L(s_X^\alpha) \tag{5.5}$$

Where s_X is the channel wise average magnitude of activation and α is found

using a grid search [18] in the range of $[0, 1]$. Importantly, this method does not require any retraining, gradient computation, or mixed-precision arithmetic. It only relies on statistics collected from a few calibration samples and works only post training.
AWQ strikes an effective balance between quantization efficiency and model quality. Its advantages include:

- No need for fine-tuning or retraining.
- Fully static and hardware-friendly (no mixed precision).
- Supports extremely low-bit quantization (e.g., 4-bit) while preserving model performance.

Because of these properties, AWQ is particularly well-suited for deploying LLMs on edge devices or latency-critical environments where both memory and compute resources are limited.

5.1.5 *Quantization Aware Training (QAT)*

While PTQ is convenient and capable of preserving a model's quality to some extent, Figure 5.1 demonstrates that Quantization Aware Training (QAT) can significantly improve the performance of quantized models in comparison. Quantization Aware Training (QAT) is a training technique that incorporates quantization effects directly into the training process, allowing the model to adapt to the quantization noise. As shown in the results from the HellaSwag benchmark [51], which evaluates LLM performance on common sense reasoning tasks, QAT models experience only a minor degradation of 0.7% in accuracy relative to the unquantized baseline. In contrast, PTQ models show a 6.2% drop in accuracy when both quantization techniques use INT8 dynamic activation quantization and 4-bit per-channel weight quantization. Furthermore, on the WikiText dataset [58], the PTQ model exhibits a perplexity that is 2.026 points higher than that of the QAT model. Since perplexity is a measure of next-token prediction quality over a given text corpus, where higher values indicate worse performance, this result further supports the hypothesis that quantization-aware trained models can outperform post-training quantized models on similar tasks.

These results motivate a closer look at how QAT works in practice. Because quantization typically involves non-differentiable operations such as rounding, QAT employs *Straight-Through Estimators* (STE) during the backward pass. Straight Through Estimate (STE)s approximate the gradients of these non-smooth functions, allowing effective gradient flow to the original full-precision weights [67]. By simulating quantization during training, QAT enables the model to adapt its parameters to the quantization-induced noise. This leads to increased robustness and better accuracy in the final quantized model [67]. While QAT supports arbitrary bit-widths and has shown strong

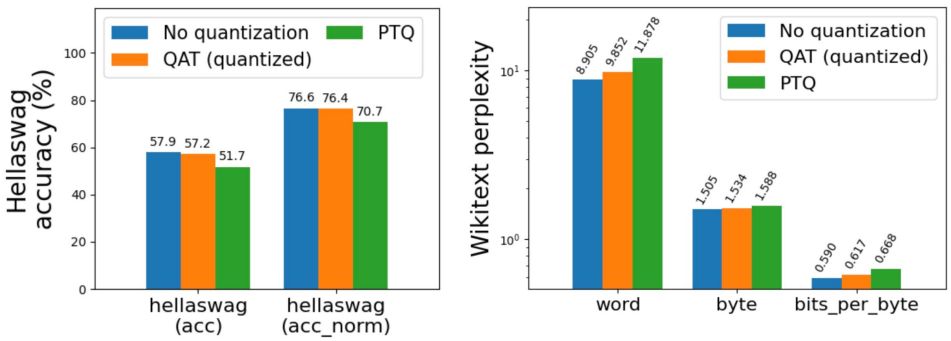

Figure 5.1: QAT vs. PTQ performance comparison [67].

performance across various tasks, an alternative approach is to train the model directly at the desired precision. This method, known as **Quantized Training** (Quantized Training (QT)), has so far demonstrated reliable results only at 8-bit precision [67].

6

KNOWLEDGE DISTILLATION

As discussed in Chapters 3, 4 and 5, LLMs are both compute- and memory-intensive. To address these limitations, one promising approach is *Knowledge Distillation*, which aims to reduce inference time and memory usage without significantly sacrificing performance.
The core idea of knowledge distillation is straightforward: a large, high-capacity model (the "teacher") is used to train a smaller, more efficient model (the "student") by transferring knowledge through its output predictions, either as text or as probability distributions over tokens.
This approach is particularly well-suited for our real-world application, where a compact yet capable model is required. We hypothesize that a student model, distilled from a larger LLM, can achieve near-identical results. In our case this may be especially plausible given that many of the teacher's parameters are likely optimized for out-of-domain tasks, such as content in different languages, which are irrelevant for our specific use case.
In the following sections, we explore several promising distillation techniques. We conclude the chapter by evaluating the effectiveness of a self-distilled model applied to our real-world scenario.

6.1 SUPERVISED FINE-TUNING (SFT)

Supervised Fine-Tuning (SFT) or *Sequence-Level Knowledge Distillation (SeqKD)* is a straightforward yet effective distillation method, which also works for black-box models like Open AIs GPT-4 [95], whose architecture and weights are not publicly available.
SFT fine-tunes a student model using either the output sequences of the teacher model. When using output sequences, SFT maximizes the likelihood of sequences generated by the teacher model, aligning the student's predictions with those of the teacher. This process can be mathematically formulated as minimizing the following objective function as stated in [95]:

$$\mathcal{L}_{\text{SFT}} = \mathbb{E}_{x\sim\mathcal{X}, y\sim p_T(y|x)} \left[-\log p_S(y|x) \right], \tag{6.1}$$

where x is the input used to produce the output sequence y by the corresponding model, $p_T(y|x)$ is the teachers output distribution and $p_S(y|x)$ is the students output distribution. This simple yet powerful technique has been widely adopted in many recent works to train student models using data generated by teacher LLMs. Despite the prevalence and ease of SFT there are more efficient approach for white box model knowledge distilla-

tion, mainly divergence and similarity based knowledge distillation.

6.1.1 *Divergence-Based Distillation*

Divergence-based methods aim to minimize the difference between the output distributions of the teacher and student models [95]. This is formalized using a general divergence objective:

$$\mathcal{L}_{\text{Div}} = \mathbb{E}_{x\sim\mathcal{X}, y\sim\mathcal{Y}} \left[D(p_T(y|x), p_S(y|x)) \right], \tag{6.2}$$

where D represents a divergence measure between the teacher distribution $p_T(y|x)$ and the student distribution $p_S(y|x)$, $x \in X$ is the input, $y \in Y$ is the output generated using the input x and $\mathbb{E}$ is the expected value (e.g. mean). The specific choice of divergence significantly influences the behavior of the student model, which [95] describes as following:

- **Forward KL divergence** ($\text{KL}(p_T \| p_S)$), commonly used in traditional knowledge distillation, encourages the student to approximate all modes of the teacher's distribution. This *mode-covering* behavior may lead to overestimation of low-probability tokens, increasing the risk of hallucinations or degraded output quality.
- **Reverse KL divergence** ($\text{KL}(p_S \| p_T)$), in contrast, emphasizes high-probability regions in the teacher's output. This *mode-seeking* approach tends to yield more accurate predictions but may reduce output diversity.

Recent studies [3, 30, 74] have explored various divergence measures for distilling LLMs, demonstrating that the most effective choice is often task-dependent. For instance, forward KL divegrence is generally better suited to tasks with low output variability such as machine translation, while reverse KL is preferable for more diverse tasks like dialogue generation or instruction tuning.
Some methods, such as [30], leverage reinforcement learning techniques, like policy gradients to optimize reverse KL divergence objectives, particularly in complex generation tasks. However, this approach assumes the teacher and student models use identical tokenizers, which may limit its applicability.
To address limitations in output-level alignment, more recent work has investigated alternatives such as similarity-based feature distillation, which will be discussed next.

6.1.2 *Similarity-Based Distillation*

Similarity-based methods shift the focus from matching output distributions to aligning internal representations, such as hidden states or feature maps between teacher and student models. These approaches aim to ensure that the student processes inputs in a manner similar to the teacher, providing a complementary signal to output-based goals like those in Section 6.1.1. The objective function typically takes the form:

$$\mathcal{L}_{\text{Sim}} = \mathbb{E}_{x\sim\mathcal{X}, y\sim\mathcal{Y}} \left[\mathcal{L}_F \left(\Phi_T(f_T(x,y)), \Phi_S(f_S(x,y))\right)\right], \tag{6.3}$$

where $f_T(x,y)$ and $f_S(x,y)$ represent the feature representations extracted from the teacher and student models, respectively. The transformation functions Φ_T and Φ_S project these features into a shared representation space and $\mathcal{L}_F$ denotes a similarity metric, commonly Mean Squared Error (MSE), cosine similarity, or other distance-based metrics. While similarity-based methods are widely used in encoder-style language models [36, 43, 86, 87],their application to autoregressive LLMs remains relatively limited. A notable exception is Task-Aware Layer-Wise Distillation (TED) [53], which introduces task-specific filters to identify and align the most relevant feature layers between teacher and student. Despite being underexplored in the LLM context, similarity-based distillation shows considerable promise. By promoting alignment at the representation level, these methods can improve learning efficiency and enhance model generalization. It is important to note that this approach requires matching internal dimensions between the teacher and student model architectures, which can impose constraints on model design.

While more advanced techniques such as *Distilled Reward Model Training* or *Skill Distillation* offer promising future directions, they remain outside the scope of this work. Due to practical considerations, including time and resource constraints, we focus primarily on supervised fine-tuning in the experimental sections.

7

EXPERIMENTS

This chapter presents all experiments related to inference costs, batching, quantization, Dianovi's use case, and model distillation. All experiments that do not involve sensitive company data have been made publicly available on GitHub [92]. For all experimental results where appropriate metrics are available, the best-performing result is highlighted in red, while the second-best is underlined.

7.1 INTRODUCTION

Having thoroughly explored the internal mechanisms of Large Language Models LLMs and inference optimization techniques, we are now prepared to investigate the practical optimization of inference time in a real-world deployment.
To support this investigation, we introduce a novel Question Answering (Question Answering (QA)) benchmark derived from a 2011 German medical examination. This benchmark dataset offers a publicly shareable means to assess medical model performance. Additionally, we incorporate internal benchmarks from *dianovi GmbH* to provide a comprehensive evaluation of model quality.

7.2 GERMEXAM

The *GerMExam* benchmark was constructed based on data from an open sourced multiple choice doctors exam that includes answers [69]. The original dataset comprises multiple-choice questions, each offering five answer options, with exactly one correct choice. To adapt these questions for our benchmark, we reformulated them into a boolean format. Specifically, for each original answer option, we created a corresponding binary question asking whether the answer is correct. Each instance is then labeled with *Ja* (German for "Yes") if the answer is correct, or *Nein* (German for "No") otherwise. Following this procedure, the resulting GerMExam benchmark consists of 680 binary question-answer pairs. The total token count across all questions is approximately 151,000, though this may vary slightly depending on the tokenizer employed. It is important to note that the dataset is highly imbalanced: only about 20% of the instances are labeled with *Ja*. Therefore, we recommend evaluation metrics that account for this imbalance, such as the balanced accuracy or Cohen's kappa score.

7.3 INFERENCE COST EXPERIMENTS

7.3.1 *Description*

We aim to validate our theoretical inference cost model through empirical experimentation.
To do this, we combine GPU utilization efficiency estimates reported in the FlashAttention series of papers [22, 23, 78] with observed forward pass durations across different input sequence lengths.
We model the expected inference time as a function of the total number of FLOP and the effective execution throughput (Floating Point Operation per Second (FLOPS)) as follows:

$$t_{estimate} = \frac{FLOP_{forward}}{\gamma_{effective} \times FLOPS} \tag{7.1}$$

Here, $t_{estimate}$ represents the estimated time required for a single forward pass, $FLOP_{forward}$ is the approximated number of FLOP per forward pass, $\gamma_{effective}$ is the effective utilization rate of the GPU (i.e., the fraction of theoretical peak FLOPS actually achieved by the FlashAttention implementation) and $FLOPS$ is the theoretical peak throughput of the GPU at the relevant numerical precision.

7.3.2 *Experimental Setup*

To ensure high inference throughput, we use the *SGLang* module [77], adapting server settings and the model to suit our experimental goals. Our hardware setup consists of an NVIDIA GeForce RTX 4090 GPU with 24 GB of VRAM. According to [64], this GPU delivers a peak performance of 82.6 TFLOPS, or 82.6×10^{12} FLOPS. The system is powered by an AMD Ryzen 9 5950X 16-core processor running at 3.4 GHz per core and is equipped with 96 GB of RAM. Due to VRAM constraints, we conduct our experiments using the *Llama 3.1 8B Instruct* model. We use the following FLOPS fraction, as reported in [78]:

- Flash Attention 3: 75%

Using this FLOPS fraction and the computed FLOPs required for a forward pass, we perform inference for sequence lengths {128, 256, 512, 1024, 2048, 4096, 8192}, as well as randomly generated lengths {1366, 5264, 4677, 7151, 5875}. We then compare the estimated inference times, calculated using (7.1), with the actual measured times.
As the results in Table 7.1 show, the theoretical FLOP count does not correlate linearly with actual inference time. For example, the sequence length of 128 tokens results in approximately 1.03e12 FLOPs, while 7151 tokens result in roughly 70 times more operations. Yet, the measured inference times are nearly identical.

Seq. Len.	FLOPs	Estimated Time	Inference Time	Time Delta	Delta %
128	1.03e12	0.016674	0.169113	0.152439	90.14%
256	2.07e12	0.033487	0.176942	0.143455	81.07%
512	4.18e12	0.067529	0.158545	0.091016	57.41%
1024	8.50e12	0.137277	0.158224	0.020947	13.24%
2048	1.76e13	0.283428	0.158125	-0.125303	79.24%
4096	3.73e13	0.602353	0.184976	-0.417376	225.64%
1366	1.15e13	0.185102	0.194239	0.009137	4.70%
5264	4.96e13	0.800134	0.181893	-0.618241	339.89%
4677	4.33e13	0.699292	0.188178	-0.511115	271.61%
7151	7.09e13	1.144061	0.167521	-0.976540	582.94%
5875	5.63e13	0.908197	0.173350	-0.734847	423.91%

Table 7.1: Inference Time and Estimation Errors for Various Sequence Lengths

This discrepancy suggests that FLOP count alone is a poor predictor of inference latency. Instead we suspect other factors like memory bandwidth, kernel launch overhead and GPU parallelism to be dominant factors, especially for longer sequences. Thus, even though FLOP provide a useful approximation of computational workload, they are insufficient for accurately modeling real-world inference performance in GPU inference pipelines.

7.4 BATCHING EXPERIMENTS

In this section, we evaluate the hypothesis stated in Chapter 4 that batching improves the throughput of LLM inference. Furthermore, we empirically validate the maximum batch size formula presented in Equation (4.18).

7.4.1 *Throughput Experiments*

Batching enables parallel inference by allowing the model to process b independent inputs simultaneously. This is expected to increase overall throughput, making it particularly beneficial for applications that require high-volume, parallelizable LLM queries (e.g., grading, retrieval augmentation, or chat-based support). To test this assumption, we conducted a series of experiments using the GerMExam Benchmark 7.2, running inference for all batch sizes $b \in [1, 50]$. For each batch size, we recorded both the total batch inference time and the average time taken per individual input within the batch.

EXPERIMENTAL SETUP. We used an SGLang inference server [77] running the Llama 3.1 8B Instruct model in 'bfloat16' precision, with FlashAttention-3 [78] as the attention backend. The hardware configuration included an NVIDIA GeForce RTX 4090 GPU with 24GB VRAM and a peak compute

capacity of 165.2 TFLOPS [64], an AMD Ryzen 9 5950X CPU with 16 cores running at 3.4 GHz and 96 GB of system RAM. The results are summarized in Figures 7.1 and 7.2.

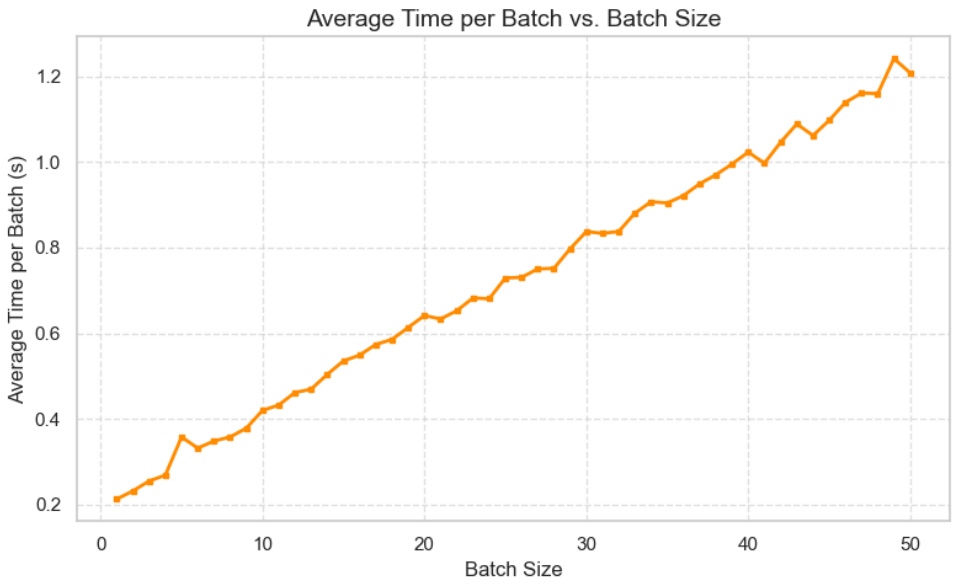

Figure 7.1: Average time taken to process a full batch, grouped by batch size.

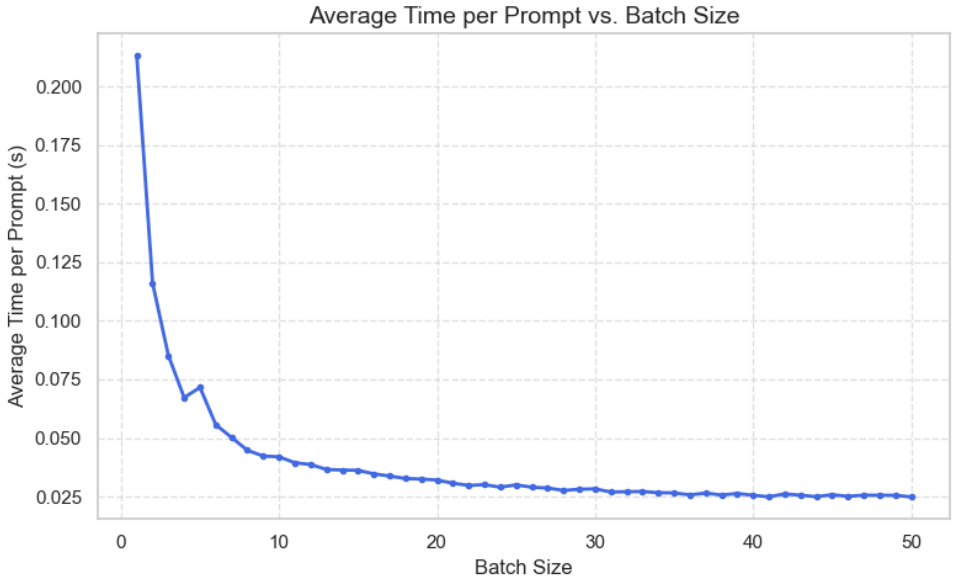

Figure 7.2: Average time taken to process a single prompt as a function of batch size.

The results confirm our expectations. As we can observe in Figure 7.1 the total batch inference time increases approximately linearly with batch size, while Figure 7.2 shows, that average time per input decreases logarithmically as batch size increases. This inverse relationship implies higher throughput

and efficiency for larger batch sizes, provided the batch fits within the GPU's memory.
These findings indicate that batching is an effective optimization strategy for workloads involving parallel LLM inferences, offering significant improvements in resource utilization and latency per request.

7.4.2 *Maximum Batch Size Experiments*

As demonstrated in Section 7.4.1, batching is only effective if the hardware can process the full batch without memory overflows. One critical question is therefore: How large can a batch be before encountering out-of-memory (OOM) errors?
To estimate this, we developed a formula to approximate the *maximum feasible batch size* for a given model and hardware setup. We compare our formula's (4.18) predictions against two open-source tools: the ApX VRAM calculator [48] (referred to as *apxml*) and Alex Asmirnov's VRAM estimator [10] (referred to as *asmirnov*).

EXPERIMENTAL DESIGN. We used the same hardware setup as described previously. We performed 80 inference runs using the Huggingface 'generate()' Application Programming Interface (API) with the Llama 3.1 8B Instruct model. Each run was parameterized with a fixed input token length and output token limit and batch sizes were incrementally increased until an OOM error occurred.
The largest successful batch size before OOM was recorded as the ground truth for each setting. We then computed the Mean Absolute Error (MAE) between the predicted and actual maximum batch size, as well as the Mean Absolute Percentage Error (MAPE) relative to the true maximum batch size.

RESULTS. The aggregated results are shown in Table 7.2.

Estimation Method	Mean Absolute Error	Mean Absolute Percentage Error
Thesis Formula	1.75	13.28%
apxml	8.225	65.54%
asmirnov	5.9875	64.74%

Table 7.2: Comparison of maximum batch size prediction accuracy across methods

Our formula achieved the best performance in both absolute and percentage error compared to the true maximum batch sizes. Although more accurate than the alternatives, the mean absolute pergentage error of approximately 13.28% suggests that further refinement is necessary.
The complete list of individual batch size experiments and predictions can be found in Appendix 9.1.

7.5 QUANTIZED MODEL COMPARISON

To evaluate the performance of quantized versions of the Llama 3.1 8B Instruct model, we compared several quantized variants against the original model using the *GerMExam* benchmark (Section 7.2).
All experiments were conducted on a system equipped with an NVIDIA GeForce RTX 4090 GPU with 24 Gigabyte (GB) of VRAM (peak performance: 165.2 Tera (10^{12}) Floating Point Operations per Second (TFLOPS) or 165.2×10^{12} FLOPS [64]), powered by an AMD Ryzen 9 5950X 16-core processor at 3.4 GHz and 96 GB of RAM.
As a baseline, we also include a trivial classifier that always predicts the majority class ("Nein").
We evaluated the following models:

- **Llama-3.1-8B-It**: The original instruction-tuned model.
- **Llama-3.1-8B-It-bnb-4bit**: A 4-bit post-training quantized version using the BitsAndBytes library.
- **Llama-3.1-8B-It-GPTQ-INT4**: A 4-bit quantized version using GPTQ with a group size of 128.

The evaluation results are shown in Table 7.3:

Model	Accuracy	Balanced Accuracy	Cohen's Kappa
Llama-3.1-8B-It	0.65	0.6397	0.2049
Llama-3.1-8B-It-bnb-4bit	0.68	0.6075	0.1754
Llama-3.1-8B-It-GPTQ-INT4	0.78	0.5763	0.1822
Always guessing "Nein"	0.80	0.5000	0.0000

Table 7.3: Performance comparison of original and quantized models on the GerMExam benchmark.

Although both quantized models achieve higher raw accuracy scores than the original model, this is likely due to over-prediction of the majority class. This is evident from the significantly lower **Balanced Accuracy** and **Cohen's Kappa** scores, which account for class imbalance and chance agreement, respectively.
The trivial model that always guesses "Nein" achieves the highest raw accuracy (80%), yet performs worst on Balanced Accuracy and Kappa. This underscores the importance of using metrics beyond accuracy for imbalanced datasets.
Based on Cohen's Kappa, which adjusts for random chance, we conclude that the original model (`Llama-3.1-8B-It`) is likely the most reliable across classes, while the quantized models trade off some generalization for simpler predictions.

Beyond benchmark scores, we also evaluated how closely the output probability distributions of the quantized models align with the original model. We use two metrics:

- **KL Divergence**: An unbounded measure of the difference between two probability distributions.
- **Hellinger Distance**: A bounded metric (range [0,1]) that quantifies the similarity between distributions; lower values indicate higher similarity.

Results are presented in Table 7.4:

Model	KL Divergence	Hellinger Distance
Llama-3.1-8B-It-bnb-4bit	0.0724	0.0995
Llama-3.1-8B-It-GPTQ-INT4	0.3089	0.2424

Table 7.4: Comparison of next-token probability distributions between quantized models and the original.

The *bnb-4bit* quantized model more closely replicates the output distribution of the original model than the GPTQ variant. This suggests that, if distributional fidelity is important for downstream tasks (e.g., uncertainty estimation or generative quality), *bnb-4bit* is the preferable quantized alternative.
In summary, while quantized models may achieve higher accuracy due to majority-class bias, the original model offers better performance when evaluated with class-sensitive metrics. Among the quantized models, *Llama-3.1-8B-It-bnb-4bit* more faithfully mirrors the original model's output distribution and is thus the recommended option from our candidates in our use-case.

7.6 COMPANY USE CASE EXPERIMENTS

For Dianovi's emergency room quality assurance system, we aim to respond to up to 151 prompts, which can be grouped into batches of sizes $[1, 50, 50, 50]$. As the batching experiments in Figure 7.2 demonstrate a substantial increase in throughput, measured as the time per input prompt, batching is employed whenever feasible in this experiment.

7.6.1 *Experiments*

As a representative example, we use an extended medical report consisting of 3308 characters (equivalent to 1063 tokens). Our objective is to generate a response within 5 seconds, aligned with the time critical nature of emergency room workflows.
Using the default configuration of SGLang [77], we obtained the performance metrics summarized in Table 7.5:

Table 7.5: System runtime comparison (in seconds) across different LLM models

Metric	AWQ INT4	Default	Llama 3.2 3B It	Speculative Decoding
First LLM Call	0.947	1.704	0.989	1.171
Second LLM Call	0.446	0.620	0.433	0.446
Third LLM Call	1.320	1.344	1.128	1.648
Final LLM Call	0.243	0.368	0.244	0.303
Total Time Taken	2.957	4.035	2.794	3.568

Speculative decoding [49] is yet another LLM inference optimization technique, which uses multiple predictions made by a smaller approximative model (e.g. drafts) which enables the model to potentially produce multiple tokens at once, which according to Leviathan, Kalman, and Matias [49] can speed up inference times by up to 3×. The speculative decoding algorithm we use in our epxeriemnts is EAGLE 3 [52]. For further explanations we refer to the papers [49, 52]. All models in Table 7.5, except for the Llama 3.2 3B It baseline, are based on the Llama 3.1 8B Instruct model with various system-level optimizations (e.g., quantization or speculative decoding), without modifying the model weights. The internal goal was to complete the entire processing pipeline, including additional computational overhead, within a five-second window. This overhead could potentially be mitigated by executing such computations in parallel with the LLM inference. Using the default Llama 3.1 8B Instruct model, we successfully met this time constraint. The results further indicate that smaller models, such as the 3B Llama 3.2 model used in our case, even without enhancements like quantization or speculative decoding, can outperform larger and optimized models in terms of latency. This suggests that model distillation may represent the most efficient approach for accelerating inference while maintaining comparable output quality. The company's initial request was to reduce inference latency, as the previously used server setup exhibited significantly slower performance due to unexplained factors when compared to SGLang. Table 7.6 presents a comparison of inference times across frameworks:

Recognizing that the Llama 3.1 8B Instruct (Instruct (It)) model is capable of achieving the desired latency under proper parallelization and infrastructure, we implemented further optimizations. These included batched tokenization and integration of FlashAttention 3 [78]. The results of these enhancements are reported in Table 7.7:

As with Table 7.5, all models in Table 7.7 except the Llama 3.2 3B It baseline utilize the Llama 3.1 8B Instruct model with different configurations and op-

Table 7.6: Inference time across different frameworks for 512 input tokens per batch element

	SGLang Server	Triton Server
Batch Size 16	2.19 s	41.54 s
Batch Size 32	2.61 s	83.45 s

Table 7.7: Runtime comparison including FlashAttention 3 and batched tokenization

Metric	AWQ INT4	Default	Llama 3.2 3B It	Speculative Decoding	FlashAttention 3 + Batched Tokenization
First LLM Call	0.947	1.704	0.989	1.171	1.618
Second LLM Call	0.446	0.620	0.433	0.446	0.446
Third LLM Call	1.320	1.344	1.128	1.648	1.426
Final LLM Call	0.243	0.368	0.244	0.303	0.266
Total Time Taken	2.957	4.035	2.794	3.568	3.755

timizations. By leveraging SGLang, batching and targeted system optimizations including FlashAttention 3 and parallel execution of computations, we successfully reduced inference latency below the 5 second target. Notably, this was achieved without modifying the underlying model, thereby avoiding the time consuming process of validating behavioral consistency after model changes. Although the current solution meets the project's performance requirements, further improvements in both inference speed and response quality may be attainable through model distillation, which we explore in the following section.

7.7 DISTILLATION EXPERIMENTS

To further reduce inference time and computational overhead for the system described in Section 7.6.1, we apply knowledge distillation by transferring capabilities from the high-performing `Qwen3-32B` model which was selected via internal benchmarks as the best fit for our specific task into its smaller counterpart, `Qwen3-1.7B`. Due to time and resource constraints (only two A100 GPUs provided by Hessian.AI), we initially generated approximately 180,000 distillation samples using `mistralai/Mixtral-8x7B-Instruct-v0.1`. After later identifying `Qwen3-32B` as a better-suited teacher model, we sup-

plemented the dataset with an additional 20,000 high-quality samples generated directly using `Qwen3-32B`.

7.7.1 *Pretraining*

We hypothesize that targeted pretraining on curated, domain-specific, instruction-tuned German datasets can significantly enhance the downstream performance of `Qwen3-1.7B` on German medical tasks. To test this hypothesis, we performed additional pretraining using the following datasets:

- *MMLU_de* [89] – a German translation of the MMLU benchmark [34, 35], covering diverse question-answering domains, including medical (e.g., clinical knowledge, virology) and non-medical fields (e.g., abstract algebra, philosophy).
- *BioInstructQA* [15] – biomedical question-answering tasks translated into German.
- *German-RAG-ORPO-Alpaca-HESSIAN-AI* [2] – a set of instruction-tuning datasets optimized for German-language alignment.
- *GPT-4-Self-Instruct-German* [88] – synthetic prompt-response pairs in German, generated using GPT-4.
- *Open-R1-Multilingual-SFT* [8] – multilingual instruction-tuned data, from which we exclusively used the German subset.

The combined dataset consists of 282, 982, 909 tokens across 141, 324 samples. We limited individual samples to a maximum of 8192 tokens and performed instruction tuning, where loss is computed only on the assistant's most recent output i.e., the model's response in a chat format as in the example 2.3.1. We trained the standard `Qwen3-1.7B` model using the `SFTTrainer` from the `trl` library with a fixed random seed of 1 and the following non-default hyperparameters:

Table 7.8: German Pretraining Hyperparameters

Neftune Noise Alpha	Model Precision	Learning Rate	Batch Size	Epochs
5	bfloat16	10^{-5}	1	3

Neftune Noise Alpha (α) corresponds to the regularization strength proposed in the NEFTune method [42], which has shown improvements across a variety of fine-tuning settings. **Model Precision** refers to the numerical representation of model weights. **Learning Rate** controls the step size during gradient updates. **Batch Size** denotes the number of samples per training step and **Epochs** represent how many times the model iterates over the full dataset. To evaluate the impact of our pretraining, we compared the

pretrained `Qwen3-1.7B` against its baseline using the GERMEXAM benchmark (see Section 7.2).

Table 7.9: Performance on the GERMEXAM Benchmark

Model	Accuracy	Balanced Accuracy	Cohen's Kappa
Qwen/Qwen3-32B	0.7132	0.7491	0.3573
Llama-3.1-8B-It	0.6500	0.6397	0.2049
Qwen/Qwen3-1.7B (baseline)	0.2426	0.5156	0.0130
Pretrained Epoch 1	0.8088	0.5276	0.0845
Pretrained Epoch 2	0.8029	0.5404	0.1161
Pretrained Epoch 3	0.8000	0.5386	0.1099
Always-False Baseline	0.8000	0.5000	0.0000

These results validate our pretraining approach. Accuracy increased from 24.26% to over 80%, a relative gain of +56.62 percentage points. While balanced accuracy and Cohen's kappa showed smaller but consistent improvements, the overall gains suggest that instruction-tuned, domain-specific pretraining significantly boosts task performance. Although `Qwen3-1.7B` still lags behind larger models like `Qwen3-32B`, these results highlight the value of efficient, low-resource transfer learning on language- and domain-aligned datasets.

7.7.2 *Distillation Results*

Following the pretraining phase, we fine-tuned the model using supervised learning on outputs originally generated by the company system using `Mixtral 8x7B Instruct v0.1`. We then further fine-tuned the model on a balanced subset of samples generated by running the corresponding step with its exemplary inputs while employing `Qwen3-32B`, where all samples of the underrepresented class were included and an equal number of samples were randomly drawn from the overrepresented class. Since the target task, a single LLM call in our production system, is a hybrid of classification and generation, we evaluated model performance using a mixed metric approach. From a random test set of 500 samples, we assessed:

- **Classification similarity:** standard classification metrics compared model predictions to those of `Qwen3-32B`.
- **Generation similarity:** cosine similarity of sentence embeddings generated via `Qwen3 Embedding 8B`.

Note: the test dataset was randomly sampled and is inherently imbalanced, with 85.2% of examples belonging to the majority class.
The unmodified `Qwen3-1.7B` model performs similarly to the Always-True baseline, suggesting its predictions largely mirror guessing the majority class.

Table 7.10: Distillation Classification Metrics

Model	Accuracy	Balanced Accuracy	Cohen's Kappa
Qwen/Qwen3-1.7B	0.850	0.5010	0.0325
Pretrained Model	0.626	0.6130	0.1323
Distilled Mixtral	0.828	0.5752	0.1808
Distilled Qwen3	0.882	0.6963	0.4562
Always-True Baseline	0.852	0.5000	0.0000

Pretraining improves this and distillation particularly from `Qwen3-32B` yields further gains. The distilled model achieves the highest accuracy, balanced accuracy and kappa score, confirming the effectiveness of knowledge distillation in transferring complex behavior from large models to smaller ones. Since the Qwen3 distillation dataset was due to the reasons mentioned in 7.7 and the rebalancing only consisting of 2164 training samples, we hypothesize, that we can get better scores by creating more samples in the minority class and thus beein able to create a larger balanced dataset. Despite the

Model	Mean Cosine Similarity	Num. Generations
Qwen3-1.7B	0.7143	2
Distillation Qwen3 32B	0.8279	25

Table 7.11: Cosine similarity statistics comparison

limited number of training samples used for distillation from `Qwen3-32B`, the resulting model exhibits a substantial increase in generation similarity (+11.35%). Notably, since the original `Qwen3-1.7B` model almost exclusively predicts the majority class, it produced only two generation outputs that could be meaningfully compared. This further underscores the advantage of distillation, which not only improves classification metrics but also enhances generative alignment with the teacher model.

7.8 CONCLUSION

This chapter has presented a comprehensive suite of experiments aimed at understanding and optimizing LLM inference for practical deployment, particularly in our real-world medical use case. We began by validating the theoretical FLOP-based inference cost model and showed that actual latency diverges significantly from FLOP estimates, especially for longer sequences, which suggests that other factors like memory throughput or parallelization capabilities play a more important role in throughput compared to FLOP.
We then demonstrated the substantial benefits of batching, both in terms of throughput and latency per input and proposed a conservative yet comparatively accurate formula for predicting maximum feasible batch sizes.
Our quantization experiments revealed trade-offs between model fidelity

and performance, with BitsAndBytes 4-bit quantization preserving distributional similarity better than GPTQ. However, we also showed that naive accuracy can be misleading on imbalanced datasets and emphasized the importance of balanced metrics like Cohen's Kappa or Balanced Accuracy.
In the company-specific setting, we evaluated multiple system-level optimizations, which include batching, FlashAttention 3, speculative decoding and a smaller model as placeholder for distillation. These efforts successfully reduced end-to-end latency below the critical five-second threshold without modifying the model weights, ensuring stability and maintainability of the production pipeline.
Finally, we investigated task-specific pretraining and knowledge distillation to compress high-performing models into smaller, more efficient variants. Our results showed that even with limited resources, distillation can yield notable gains in both classification and generation quality, potentially enabling deployment of lighter models without sacrificing essential performance.
Together, these experiments validate a layered optimization strategy that spans hardware utilization, software infrastructure and model-level adaptations. These results support the efficient deployment of LLMs in time-critical and resource-constrained settings and serve as a blueprint for future work in LLM inference optimization.
Additionally, our insights into instruction tuning and distillation can also be used to improve the quality of outputs from already deployed models.

7.9 FUTURE DIRECTIONS

The experiments presented in this thesis establish a strong foundation for the practical deployment and optimization of LLMs. Nonetheless, several promising research directions remain that could further enhance the performance, scalability and maintainability of LLMs in real-world applications.

7.9.1 *Inference Time Estimation*

As shown in our inference cost experiments (Section 7.3), theoretical FLOP counts alone are insufficient to accurately estimate inference latency, especially for long input sequences. The observed discrepancies indicate that system-level factors such as memory bandwidth, kernel launch overhead and GPU-level parallelism play a substantial role in determining actual runtime. Future work should aim to develop more comprehensive latency models that incorporate aspects such as memory access patterns, bandwidth saturation, GPU SM occupancy, kernel scheduling behavior and framework-level optimizations like token streaming or fused kernel execution. Such models would enable latency-aware scheduling, more efficient batch management and stronger guarantees for real-time performance.

7.9.2 *Maximum Batch Size Estimation*

Our batch size estimation approach (4.18) outperformed several existing online tools in terms of Mean Absolute Percentage Error, yet still incurred an average error of 47.6%. This highlights a clear opportunity for improvement. Future research could focus on more granular modeling of VRAM usage by profiling model-specific memory allocation patterns, considering runtime fragmentation effects and accounting for dynamic buffer allocations during attention and activation computations. A more detailed understanding of memory behavior would enable safer and more aggressive batching strategies, especially in scenarios with strict latency constraints.

7.9.3 *Quantization*

Our quantization experiments demonstrated that PTQ can produce competitive results, particularly in memory-constrained environments. However, more advanced methods may offer better trade-offs between accuracy and efficiency. One direction involves quantization-aware training (QAT), which fine-tunes the model to anticipate quantization-induced errors, thereby maintaining performance. Another promising avenue is the use of mixed-precision or float8 training formats such as FP8, which allow models to be trained from scratch using compact representations. Additionally, the generation of synthetic training data using larger teacher models (e.g., `Qwen3-32B`) may offer an effective way to fine-tune quantized models for specific tasks. These strategies could help produce lightweight models that preserve alignment and accuracy with minimal computational overhead.

7.9.4 *Distillation*

The distillation experiments described in Section 7.7 showed that it is feasible to compress large teacher models into smaller student models while retaining both classification and generative capabilities. However, the distilled dataset used was relatively limited in size and class balance. To further improve alignment, future efforts could focus on scaling up the number of distilled samples, particularly for underrepresented classes while also leveraging active learning techniques to identify and select the most informative or uncertain examples for teacher generation. Moreover, employing a multi-teacher setup, where knowledge is aggregated from multiple high-performing models, could help improve both generalization and task-specific robustness. These improvements would make distilled models more effective for real-world deployment, especially in clinical or time-sensitive applications.

8

CONCLUSION

This thesis explored practical optimization techniques for accelerating Large Language Model (LLM) inference in real-world settings, with a particular focus on emergency medicine. In collaboration with Dianovi GmbH, a company supporting medical professionals through the automated quality assurance of doctors' letters, we investigated how to reduce latency and computational overhead without compromising the quality of model outputs.

We began by establishing a theoretical foundation for understanding LLMs as next-token predictors, where each token typically corresponds to a subword unit. Based on this understanding, we estimated the floating-point operations (FLOPs) required to generate predictions and evaluated the limitations of FLOP-based inference cost models. Our experiments showed that actual inference latency often diverges from FLOP estimates, especially for longer sequences indicating that other factors such as memory bandwidth and parallelism play a more critical role in practical throughput.

To mitigate latency in production scenarios, we first analyzed batching, which enables the parallel processing of multiple prompts. Our results showed that a combination of batching and an efficient inference backend can yield up to a 40$\times$ speedup in generation throughput. We proposed a conservative yet practically useful formula for estimating the maximum feasible batch size based on GPU memory constraints, enabling efficient utilization without overcommitting system resources.

Next, we investigated quantization, a technique that reduces the numerical precision of model weights and activations to compress memory usage and accelerate inference. Among the methods tested, BitsAndBytes 4-bit quantization preserved distributional output similarity more effectively than GPTQ. However, we highlighted the limitations of naïve accuracy metrics especially on imbalanced classification tasks and emphasized the importance of balanced evaluation metrics such as Cohen's Kappa and Balanced Accuracy to assess fidelity more reliably.

In addition to infrastructure-level optimizations, we explored knowledge distillation and task-specific instruction tuning to compress larger foundation models into smaller, more efficient variants. A distilled model with only 1.7 billion parameters achieved over 82% output similarity to a much larger 32-billion-parameter model, demonstrating that lightweight models can maintain strong alignment with expert-level predictions. Furthermore, we showed that instruction tuning on a mix of domain-specific and instruction-following data significantly improved zero-shot classification capabilities for medical question answering in the target language.

Finally, in a system-level evaluation tailored to the partner company's production environment, we combined batching, quantization, FlashAttention

3, speculative decoding and task-specific model adaptation. This composite optimization pipeline successfully reduced end-to-end latency below the critical five-second threshold without requiring modifications to model weights, thus preserving stability in deployment.

In conclusion, this thesis demonstrates that LLM inference can be made substantially more efficient through a layered optimization approach. By combining batching, quantization and distillation alongside careful instruction tuning and infrastructure-aware deployment strategies it is possible to build LLM systems that meet the latency and reliability demands of high-stakes applications like emergency medicine. These results offer a robust foundation for deploying LLMs in time-critical, resource-constrained environments and provide a blueprint for future work on inference optimization in domain-specific contexts.

9 APPENDIX

9.1 MAXIMUM BATCH SIZE EXPERIMENTS TABLE

Each row in the results table contains the following information: **len in** refers to the number of input tokens provided to the model per sample, while **len out** indicates the maximum number of tokens the model is allowed to generate. The **max batch size** column records the largest batch size that was successfully executed without triggering an OOM error and thus serves as the ground truth. The columns labeled **formula**, **apxml** and **asmirnov** represent the predicted maximum batch sizes according to our formula 4.18, the ApX VRAM calculator [48] and Asmirnov's estimator [10], respectively. Finally, the columns δ_{formula}, δ_{apxml} and δ_{asmirnov} denote the absolute error between each method's prediction and the actual maximum batch size.

Table 9.1: Batch size data with missing values replaced by NaN

len in	**len out**	**max batch size**	**formula**	**apxml**	**asmirnov**	δ_{formula}	δ_{apxml}	δ_{asmirnov}
1024	128	31	34	12.0	25	-3	19.0	6
1024	256	31	30	12.0	25	1	19.0	6
1024	512	31	25	12.0	25	6	19.0	6
1024	1024	25	19	12.0	25	6	13.0	0
1024	2048	16	12	12.0	25	4	4.0	-9
1024	158	31	33	12.0	25	-2	19.0	6
1024	289	31	30	12.0	25	1	19.0	6
1024	548	31	25	12.0	25	6	19.0	6
1024	1132	23	18	12.0	25	5	11.0	-2
1024	2079	16	12	12.0	25	4	4.0	-9
2048	128	16	18	6.0	8	-2	10.0	8
2048	256	16	17	6.0	8	-1	10.0	8
2048	512	16	15	6.0	8	1	10.0	8
2048	1024	16	12	6.0	8	4	10.0	8
2048	2048	13	9	6.0	8	4	7.0	5
2048	158	16	17	6.0	8	-1	10.0	8
2048	289	16	16	6.0	8	0	10.0	8
2048	548	16	15	6.0	8	1	10.0	8
2048	1132	16	12	6.0	8	4	10.0	8
2048	2079	12	9	6.0	8	3	6.0	4

len in	len out	max batch size	formula	apxml	asmirnov	δ_{formula}	δ_{apxml}	δ_{asmirnov}
4096	128	8	9	2.0	2	-1	6.0	
4096	256	8	9	2.0	2	-1	6.0	
4096	512	8	8	2.0	2	0	6.0	
4096	1024	8	7	2.0	2	1	6.0	
4096	2048	8	6	2.0	2	2	6.0	
4096	158	8	9	2.0	2	-1	6.0	
4096	289	8	8	2.0	2	0	6.0	
4096	548	8	8	2.0	2	0	6.0	
4096	1132	8	7	2.0	2	1	6.0	
4096	2079	8	6	2.0	2	2	6.0	
8192	128	4	4	1.0	0	0	3.0	
8192	256	4	4	1.0	0	0	3.0	
8192	512	4	4	1.0	0	0	3.0	
8192	1024	4	4	1.0	0	0	3.0	
8192	2048	4	3	1.0	0	1	3.0	
8192	158	4	4	1.0	0	0	3.0	
8192	289	4	4	1.0	0	0	3.0	
8192	548	4	4	1.0	0	0	3.0	
8192	1132	4	4	1.0	0	0	3.0	
8192	2079	4	3	1.0	0	1	3.0	
1537	128	21	23	NaN	13	-2	NaN	
1537	256	21	22	NaN	13	-1	NaN	
1537	512	21	19	NaN	13	2	NaN	
1537	1024	21	15	NaN	13	6	NaN	
1537	2048	13	11	NaN	13	2	NaN	
1537	158	21	23	NaN	13	-2	NaN	
1537	289	21	21	NaN	13	0	NaN	
1537	548	21	18	NaN	13	3	NaN	
1537	1132	19	14	NaN	13	5	NaN	
1537	2079	13	10	NaN	13	3	NaN	
2347	128	14	15	NaN	6	-1	NaN	
2347	256	14	15	NaN	6	-1	NaN	
2347	512	14	13	NaN	6	1	NaN	
2347	1024	14	11	NaN	6	3	NaN	
2347	2048	13	8	NaN	6	5	NaN	
2347	158	14	15	NaN	6	-1	NaN	
2347	289	14	14	NaN	6	0	NaN	

len in	len out	max batch size	formula	apxml	asmirnov	δ_{formula}	δ_{apxml}	δ_{asmirnov}
2347	548	14	13	NaN	6	1	NaN	8
2347	1132	14	11	NaN	6	3	NaN	8
2347	2079	13	8	NaN	6	5	NaN	7
5139	128	7	7	NaN	1	0	NaN	6
5139	256	7	7	NaN	1	0	NaN	6
5139	512	7	6	NaN	1	1	NaN	6
5139	1024	7	6	NaN	1	1	NaN	6
5139	2048	7	5	NaN	1	2	NaN	6
5139	158	7	7	NaN	1	0	NaN	6
5139	289	7	7	NaN	1	0	NaN	6
5139	548	7	6	NaN	1	1	NaN	6
5139	1132	7	6	NaN	1	1	NaN	6
5139	2079	7	5	NaN	1	2	NaN	6
7133	128	5	5	NaN	0	0	NaN	5
7133	256	5	5	NaN	0	0	NaN	5
7133	512	5	5	NaN	0	0	NaN	5
7133	1024	5	4	NaN	0	1	NaN	5
7133	2048	5	4	NaN	0	1	NaN	5
7133	158	5	5	NaN	0	0	NaN	5
7133	289	5	5	NaN	0	0	NaN	5
7133	548	5	5	NaN	0	0	NaN	5
7133	1132	5	4	NaN	0	1	NaN	5
7133	2079	5	4	NaN	0	1	NaN	5

BIBLIOGRAPHY

[1] Nikolas Adaloglou and Sergios Karagiannakos. *How Attention Works in Deep Learning*. https://theaisummer.com/attention/. Accessed: 2025-06-14. 2020.

[2] Avemio AG, Hessian AI, and VAGO Solutions. *German-RAG-ORPO Alpaca Dataset*. https://huggingface.co/datasets/avemio/German-RAG-ORPO-Alpaca-Hessian-AI/. 2024.

[3] Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. "On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes." In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=3zKtaqxLhW.

[4] Ahmet and Zehra. *Universal Approximation Theorem*. https://numerics.ovgu.de/teaching/psnn/2122/handout_ahmet.pdf. Otto von Guericke University Magdeburg and Institute for Numerical Mathematics. 2022.

[5] Vals AI. *MedQA Benchmark*. https://www.vals.ai/benchmarks/medqa-02-25-2025. Accessed: 2025-07-08. Vals AI, Feb. 2025.

[6] Joshua Ainslie, James Lee-Thorp, Michiel de Jong Yury Zemlyanskiy, Federico Lebrón, and Sumit Sangha. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints." In: (2023).

[7] Mehdi Ali et al. "Tokenizer Choice For LLM Training: Negligible or Crucial?" In: (2024).

[8] amphora. *Open-R1-Multilingual-SFT: Curated Multilingual Supervised Fine-Tuning Dataset*. https://huggingface.co/datasets/amphora/Open-R1-Mulitlingual-SFT. Accessed: 2025-07-23. 2025.

[9] Ron Artstein and Massimo Poesio. "Inter-Coder Agreement for Computational Linguistics." In: *Computational Linguistics* 34.4 (2008), pp. 555–596.

[10] Alex Asmirnov. *VRAM Estimator*. https://vram.asmirnov.xyz/. Web-based VRAM usage estimator for transformer-based LLMs (inference training), open-source simulation tool; accessed July 2025. Dec. 2023.

[11] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML]. URL: https://arxiv.org/abs/1607.06450.

[12] Bai et al. "Semantics of the Unwritten: The Effect of End of Paragraph and Sequence Tokens on Text Generation with GPT2." In: *arXiv* (2020).

[13] Bengio, Y., Simard, P., Frasconi, and P. "Learning long-term dependencies with gradient descent is difficult." In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181.

[14] Mikhail Berkov. "A Guide to Large Language Models For Non-Techies." In: (2025).

[15] BioMistral. *BioInstructQA: Biomedical Instruction-Finetuning Dataset*. https://huggingface.co/datasets/BioMistral/BioInstructQA. Accessed: 2025-07-23. 2024.

[16] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. New York: Springer, 2006. ISBN: 9780387310732.

[17] Francois Chollet, Mike Knoop, Greg Kamradt, Walter Reade, and Addison Howard. *ARC Prize 2025*. https://kaggle.com/competitions/arc-prize-2025. Kaggle. 2025.

[18] Marc Claesen and Bart De Moor. "Hyperparameter Search in Machine Learning." In: *arXiv preprint arXiv:1502.02127*. 2015.

[19] Jacob Cohen. "A Coefficient of Agreement for Nominal Scales." In: *Educational and Psychological Measurement* 20.1 (1960), pp. 37–46.

[20] IBM Corporation. *Accuracy Evaluation Metric*. IBM Cloud Pak for Data / watsonx.governance documentation. Last updated: July 4, 2025; accessed on July 28, 2025. 2025. URL: https://www.ibm.com/docs/en/ws-and-kc?topic=metrics-accuracy.

[21] Saeed Damadi, Golnaz Moharrer, and Mostafa Cham. *The Backpropagation algorithm for a math student*. 2023. arXiv: 2301.09977 [cs.LG]. URL: https://arxiv.org/abs/2301.09977.

[22] Tri Dao. *FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning*. 2023. arXiv: 2307.08691 [cs.LG]. URL: https://arxiv.org/abs/2307.08691.

[23] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022. arXiv: 2205.14135 [cs.LG]. URL: https://arxiv.org/abs/2205.14135.

[24] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. *Language Modeling with Gated Convolutional Networks*. 2017. arXiv: 1612.08083 [cs.CL]. URL: https://arxiv.org/abs/1612.08083.

[25] PyTorch Developers. *torch.nn.CrossEntropyLoss*. https://docs.pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html. Accessed: 2025-07-25. 2025.

[26] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*. 2022. arXiv: 2109.14545 [cs.LG]. URL: https://arxiv.org/abs/2109.14545.

[27] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*. 2023. arXiv: 2210.17323 [cs.LG]. URL: https://arxiv.org/abs/2210.17323.

[28] Glorot, Xavier, Bordes, Antoine, Bengio, and Yoshua. "Deep Sparse Rectifier Neural Networks." In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Gordon, Geoffrey, Dunson, David, Dudík, and Miroslav. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, and USA: PMLR, Nov. 2011, pp. 315–323. URL: https://proceedings.mlr.press/v15/glorot11a.html.

[29] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[30] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. "MiniLLM: Knowledge Distillation of Large Language Models." In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=5h0qf7IBZZ.

[31] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd ed. New York: Springer, 2009. ISBN: 9780387848570.

[32] He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, Sun, and Jian. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In: *arXiv preprint arXiv:1502.01852v1* (2015). URL: https://arxiv.org/pdf/1502.01852v1.pdf.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[34] Dan Hendrycks, Collin Burns, Steven Basart, Andrew Critch, Jerry Li, Dawn Song, and Jacob Steinhardt. "Aligning AI With Shared Human Values." In: *Proceedings of the International Conference on Learning Representations (ICLR)* (2021).

[35] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. "Measuring Massive Multitask Language Understanding." In: *Proceedings of the International Conference on Learning Representations (ICLR)* (2021).

[36] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. *DynaBERT: Dynamic BERT with Adaptive Width and Depth*. 2020. arXiv: 2004.04037 [cs.CL]. URL: https://arxiv.org/abs/2004.04037.

[37] Hugging Face. *Quantization*. https://huggingface.co/docs/optimum/concept_guides/quantization. Accessed: 2025-07-03. 2025. URL: https://huggingface.co/docs/optimum/concept_guides/quantization.

[38] Hvidberrrg. *The sigmoid function (a.k.a. the logistic function) and its derivative*. Online blog post on GitHub Pages. Accessed: July 28, 2025; contains definitions and derivative of the logistic activation function. 2025. URL: https://hvidberrrg.github.io/deep_learning/activation_functions/sigmoid_function_and_derivative.html.

[39] Intel Corporation. *Intel Xeon 6780E Processor (108M Cache and 2.20GHz): Specifications*. https://www.intel.com/content/www/us/en/products/sku/240362/intel-xeon-6780e-processor-108m-cache-2-20-ghz/specifications.html. Accessed: 2025-07-01. 2024.

[40] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG]. URL: https://arxiv.org/abs/1502.03167.

[41] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. 2017. arXiv: 1712.05877 [cs.LG]. URL: https://arxiv.org/abs/1712.05877.

[42] Neel Jain et al. *NEFTune: Noisy Embeddings Improve Instruction Finetuning*. 2023. arXiv: 2310.05914 [cs.CL]. URL: https://arxiv.org/abs/2310.05914.

[43] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. *TinyBERT: Distilling BERT for Natural Language Understanding*. 2020. arXiv: 1909.10351 [cs.CL]. URL: https://arxiv.org/abs/1909.10351.

[44] Andy Konwinski, Christopher Rytting, Justin Fiedlerand Alex Shaw, Sohier Dane, Walter Reade, and Maggie Demkin. *Konwinski Prize*. https://kaggle.com/competitions/konwinski-prize. Kaggle. 2024.

[45] Taku Kudo and John Richardson. "SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing." In: (2018).

[46] S. Kullback and R. A. Leibler. "On Information and Sufficiency." In: *The Annals of Mathematical Statistics* 22.1 (Mar. 1951), pp. 79–86. DOI: 10.1214/aoms/1177729694. URL: https://doi.org/10.1214/aoms/1177729694.

[47] Kundu, Achintya, Lee, Rhui Dih, Wynter, Laura, Ganti, Raghu Kiran, Mishra, and Mayank. "Enhancing Training Efficiency Using Packing with Flash Attention." In: *arXiv*. 2024.

[48] ApX Machine Learning. *LLM Inference: VRAM & Performance Calculator*. https://apxml.com/tools/vram-calculator. Web tool accessed July 2025 includes VRAM + throughput simulation for inference and fine-tuning, with CPU/NVMe offloading options. June 2025.

[49] Yaniv Leviathan, Matan Kalman, and Yossi Matias. *Fast Inference from Transformers via Speculative Decoding*. 2023. arXiv: 2211.17192 [cs.LG]. URL: https://arxiv.org/abs/2211.17192.

[50] Chuan Li. "OpenAI's GPT-3 Language Model: A Technical Overview." In: *Lambda Labs Blog* (2020). Accessed 14.06.2025. URL: https://lambda.ai/blog/demystifying-gpt-3.

[51] Xiaoyuan Li, Moxin Li, Rui Men, Yichang Zhang, Keqin Bao, Wenjie Wang, Fuli Feng, Dayiheng Liu, and Junyang Lin. *HellaSwag-Pro: A Large-Scale Bilingual Benchmark for Evaluating the Robustness of LLMs in Commonsense Reasoning*. 2025. arXiv: 2502.11393 [cs.CL]. URL: https://arxiv.org/abs/2502.11393.

[52] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. *EAGLE-3: Scaling up Inference Acceleration of Large Language Models via Training-Time Test*. 2025. arXiv: 2503.01840 [cs.CL]. URL: https://arxiv.org/abs/2503.01840.

[53] Chen Liang, Simiao Zuo, Qingru Zhang, Pengcheng He, Weizhu Chen, and Tuo Zhao. *Less is More: Task-aware Layer-wise Distillation for Language Model Compression*. 2023. arXiv: 2210.01351 [cs.CL]. URL: https://arxiv.org/abs/2210.01351.

[54] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*. 2024. arXiv: 2306.00978 [cs.CL]. URL: https://arxiv.org/abs/2306.00978.

[55] Danqing Liu. *A Practical Guide to ReLU*. https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7. Accessed: 2025-06-12. 2017.

[56] Emmy Liu et al. *Not-Just-Scaling Laws: Towards a Better Understanding of the Downstream Impact of Language Model Design Decisions*. 2025. arXiv: 2503.03862 [cs.CL]. URL: https://arxiv.org/abs/2503.03862.

[57] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge, UK: Cambridge University Press, 2008.

[58] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. *Pointer Sentinel Mixture Models*. 2016. arXiv: 1609.07843 [cs.CL].

[59] Meta. *Meta LLaMA 3: Model Cards and Prompt Formats*. https://www.llama.com/docs/model-cards-and-prompt-formats/meta-llama-3/. Accessed: 2025-06-12. 2025.

[60] Mikolov, Tomas, Chen, Kai, Corrado, Greg, Dean, and Jeffrey. "Efficient Estimation of Word Representations in Vector Space." In: *International Conference on Learning Representations (ICLR)*. 2013.

[61] Alhassan Mumuni and Fuseini Mumuni. *Large language models for artificial general intelligence (AGI): A survey of foundational principles and approaches*. 2025. arXiv: 2501.03151 [cs.AI]. URL: https://arxiv.org/abs/2501.03151.

[62] Nickolls, John, Buck, Ian, Garland, Michael, Skadron, and Kevin. "Scalable Parallel Programming with CUDA." In: *ACM Queue* 6.2 (2008), pp. 40–53.

[63] NVIDIA Corporation. *NVIDIA Ampere Architecture Whitepaper*. Tech. rep. Accessed: 2025-07-01. NVIDIA, 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[64] NVIDIA Corporation. *NVIDIA Ada GPU Architecture*. https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf. Accessed: July 18 and 2025. May 2025.

[65] OpenAI. *Screenshot of ChatGPT Interface*. Personal screenshot from https://chat.openai.com. Captured on July 28, 2025. 2025.

[66] OpenAI. *Screenshot of OpenAI Tokenizer Repository (tiktoken)*. Personal screenshot from https://github.com/openai/tiktoken. Captured on July 28, 2025. 2025.

[67] Andrew Or, Jerry Zhang, Evan Smothers, Kartikay Khandelwal, and Supriya Rao. *Quantization-Aware Training for Large Language Models with PyTorch*. Accessed: 2025-07-03. July 30, 2024. URL: https://pytorch.org/blog/quantization-aware-training/.

[68] A. Emin Orhan and Xaq Pitkow. *Skip Connections Eliminate Singularities*. 2018. arXiv: 1701.09175 [cs.NE]. URL: https://arxiv.org/abs/1701.09175.

[69] praktischArzt Redaktion. *Physikum Medizin und Hammerexamen 2025 / 26*. https://www.praktischarzt.de/medizinstudium/physikum-und-hammerexamen/. Zuletzt aktualisiert: 03.02.2025. Feb. 2025. (Visited on 07/21/2025).

[70] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. *Searching for Activation Functions*. 2017. arXiv: 1710.05941 [cs.NE]. URL: https://arxiv.org/abs/1710.05941.

[71] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Swish: A Self-Gated Activation Function." In: *arXiv preprint arXiv:1710.05941v1* (2017). URL: https://arxiv.org/pdf/1710.05941v1.pdf.

[72] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. *How Does Batch Normalization Help Optimization?* 2019. arXiv: 1805.11604 [stat.ML]. URL: https://arxiv.org/abs/1805.11604.

[73] Grigory Sapunov. "FP64 and FP32 and FP16 and BFLOAT16 and TF32 and and other members of the ZOO." In: *Medium* (2020). Accessed: 2025-07-03. URL: https://moocaholic.medium.com/fp64-fp32-fp16-bfloat16-tf32-and-other-members-of-the-zoo-a1ca7897d407.

[74] Sason, Igal, Verdu, and Sergio. "f -Divergence Inequalities." In: *IEEE Transactions on Information Theory* 62.11 (Nov. 2016), pp. 5973–6006. ISSN: 1557-9654. DOI: 10.1109/tit.2016.2603151. URL: http://dx.doi.org/10.1109/TIT.2016.2603151.

[75] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural Machine Translation of Rare Words with Subword Units." In: (2015).

[76] Serhiienko, Pavlo, Sergiyenko, Anatoliy, Telenyk, Sergii, Nowakowski, and Grzegorz. "Calculation of the Sigmoid Activation Function in FPGA Using Rational Fractions." In: *Computational Science – ICCS 2024: 24th International Conference and Malaga and Spain and July 2–4 and 2024 and Proceedings and Part VI*. Malaga and Spain: Springer-Verlag, 2024, pp. 146–157. ISBN: 978-3-031-63777-3. DOI: 10.1007/978-3-031-63778-0_11. URL: https://doi.org/10.1007/978-3-031-63778-0_11.

[77] SGL Project Contributors. *SGLang: A Programming Language for Building LLM Applications*. https://github.com/sgl-project/sglang. Accessed: July 18 and 2025. 2023.

[78] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. *FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision*. 2024. arXiv: 2407.08608 [cs.LG]. URL: https://arxiv.org/abs/2407.08608.

[79] Noam Shazeer. "Fast Transformer Decoding: One Write-Head is All You Need." In: (2019).

[80] Noam Shazeer. *GLU Variants Improve Transformer*. 2020. arXiv: 2002.05202 [cs.LG]. URL: https://arxiv.org/abs/2002.05202.

[81] Buck Shlegeris, Fabien Roger, Lawrence Chan, and Euan McLean. "Language Models Are Better Than Humans at Next-token Prediction." In: *Transactions on Machine Learning Research* (2024).

[82] Saurabh Singh. *Swish as an Activation Function in Neural Network*. Online article on Deep Learning University. Accessed: July 28, 2025. 2025. URL: https://deeplearninguniversity.com/swish-as-an-activation-function-in-neural-network/.

[83] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2019 (Revision of IEEE 754-2008). https://doi.org/10.1109/IEEESTD.2019.8766229. July 2019.

[84] Statology. *Balanced Accuracy*. Online tutorial. Accessed: July 28, 2025. 2025. URL: https://www.statology.org/balanced-accuracy/.

[85] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: 2104.09864 [cs.CL]. URL: https://arxiv.org/abs/2104.09864.

[86] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. *Patient Knowledge Distillation for BERT Model Compression*. 2019. arXiv: 1908.09355 [cs.CL]. URL: https://arxiv.org/abs/1908.09355.

[87] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. *MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices*. 2020. arXiv: 2004.02984 [cs.CL]. URL: https://arxiv.org/abs/2004.02984.

[88] CausalLM team. *GPT-4-Self-Instruct-German*. https://huggingface.co/datasets/CausalLM/GPT-4-Self-Instruct-German. Accessed: 2025-07-22. 2024.

[89] LeoLM Team. *MMLU_de: A German Translation of the Massive Multitask Language Understanding Benchmark*. https://huggingface.co/datasets/LeoLM/MMLU_de. Accessed: 2025-07-22. 2024.

[90] TensorFlow Authors. *flops_registry.py*. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/profiler/internal/flops_registry.py. Accessed: 2025-07-12. n.d.

[91] Ryan Tibshirani, Bohan LI, Donghan Yu, and Ge Huang. *Convex Optimization: Lecture 19*. https://www.stat.cmu.edu/~ryantibs/convexopt-F18/scribes/Lecture_19.pdf. Carnegie Mellon University and Convex Optimization course (Fall 2015). 2015.

[92] Max Uhl. *Thesis*. https://github.com/MaxUhl98/Thesis. Accessed: 2025-07-28. 2025.

[93] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need." In: *CoRR* abs/1706.03762 (2017).

[94] Weiss and Eva-Marie. "Metas KI-Chef: Yann LeCun glaubt nicht an die Zukunft generativer KI." In: *Heise Online* (Feb. 10, 2025). In German.

[95] Xiaohan Xu, Ming Li, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao, and Tianyi Zhou. *A Survey on Knowledge Distillation of Large Language Models*. 2024. arXiv: 2402.13116 [cs.CL]. URL: https://arxiv.org/abs/2402.13116.

[96] Vignesh Yaadav. *Exploring and Building the LLaMA 3 Architecture: A Deep Dive into Components, Coding, and Inference Techniques*. https://example.com/llama3-deep-dive. Accessed: 2025-07-27. 2025.

[97] Biao Zhang and Rico Sennrich. *Root Mean Square Layer Normalization*. 2019. arXiv: 1910.07467 [cs.LG]. URL: https://arxiv.org/abs/1910.07467.