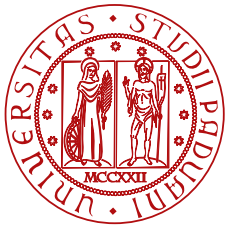

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Bachelor Thesis in Computer Engineering

The cost of backpropagation in transformer networks

Candidate

Barbato Alberto

Student ID 2073961

Supervisor

Prof. Bilardi Gianfranco

University of Padova

Academic Year
2024/2025

To my grandparents,
for their unwavering support and love.

Abstract

This thesis studies the training cost of transformer networks by analyzing the number of arithmetic operations required during backpropagation. Focusing on the original transformer block in a simplified, single-headed version, we break down the time complexity of each major component: attention, feedforward layers, and normalization. We also include a comparison with the multi-head version. The aim is to understand the contribution of each part to the total training cost, providing a hint of where future optimization might be most effective.

Contents

List of Figures

1 Introduction

1.1 Motivation

Lately, transformer networks are beginning to be widely used in modern machine learning across many fields. Since their introduction, they have become the main building block for many new AI systems. These models work well for tasks like language processing, computer vision, and features extraction. However, training these models is very expensive. The first transformer model only cost about $670 to train in 2017, but training GPT-4 cost over $78 million[8]. Training costs have been growing 2-3 times every year for the past eight years, and by 2027, the largest models could cost over a billion dollars just to train. This huge increase in cost is not just about money, but also about the environment: the massive, continuous power consumption has the indirect consequence of dumping massive amounts of CO2 into the environment. Thus, understanding where and why these expensive calculations happen, and how to optimize them, is an ongoing problem that needs to be addressed.

1.2 Objective

The objective of this thesis is to provide an analysis of the arithmetic operations required during the backpropagation phase of the transformer block. We will be focusing on the original architecture introduced by Vaswani et al. with the paper "Attention is All You Need" [9]. Ultimately, this thesis aims at giving a better

understanding of the underlying computational costs and their implications for the design and training of transformer models.

1.3 Structure of the Thesis

This thesis is organized as follows:

Chapter 2: Background Concepts Reviews the mathematical tools and transformer architecture needed for the analysis.

Chapter 3: Training Cost Analysis Derivation of the close form of the arithmetic operations needed in transformer training, with a focus on each component.

Chapter 4: Discussion of Results Highlights which components dominate training cost and discusses practical implications.

Chapter 5: Conclusions Summarizes findings and suggestions.

2

Background Concepts

2.1 Useful Mathematical Concepts

This section introduces the mathematical concepts necessary to understand the following analysis.

2.1.1 Hadamard Product and Kronecker Delta

The Hadamard product (element-wise multiplication) between two matrices or vectors of the same sizes is denoted by $\odot$. For matrices $\mathbf{a}, \mathbf{b} \in \mathbb{R}^{n \times m}$, the Hadamard product is defined as:

$$(\mathbf{a} \odot \mathbf{b})_{i,j} = a_{i,j} \cdot b_{i,j}$$

The Kronecker delta function δ_{ij} is defined as:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

This function is particularly useful in gradient computations and will appear frequently in our backpropagation derivations.

2.1.2 Softmax Function

The softmax function is an interesting component of the attention block. For a vector $\mathbf{z} \in \mathbb{R}^n$, the softmax function on the vector $\mathbf{z}$ is defined as:

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

The softmax converts a vector of real numbers into a probability distribution vector of the same size, using the values as weights, therefore making all outputs positive and sum to 1. The gradient of the softmax function *w.r.t.* (with respect to) its input has a simple close form:

$$\frac{\partial \text{softmax}(\mathbf{z})_i}{\partial z_j} = \text{softmax}(\mathbf{z})_i(\delta_{ij} - \text{softmax}(\mathbf{z})_j)$$

2.1.3 Chain Rule

The chain rule is a math concept very important to backpropagation, as it reduces significantly the computation necessary for training.

Single-variable reminder. If

$$y = f(g(x)), \quad g : \mathbb{R} \to \mathbb{R}, \quad f : \mathbb{R} \to \mathbb{R},$$

then the chain rule is

$$\frac{dy}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}.$$

Multivariable version. Suppose now that

$$y = f(g(\underline{x})), \quad g : \mathbb{R}^n \to \mathbb{R}^m, \quad f : \mathbb{R}^m \to \mathbb{R}.$$

Then, the derivative of y with respect to $\underline{x}$ for each coordinate x_j is:

$$\frac{\partial y}{\partial x_j} = \sum_{i=1}^{m} \frac{\partial f}{\partial u_i}\Big|_{u=g(\underline{x})} \cdot \frac{\partial g_i}{\partial x_j}(\underline{x}).$$

This makes the backpropagation algorithm much more efficient in comparison with naive approaches.

2.1.4 Loss Function & Gradient Descent

The loss function $\mathcal{L}(\theta)$ measures the correlation, or lack thereof, between model predictions and target values, where $\underline{\theta}$ is a vector of model parameters. In transformer training, especially for language modeling, the cross entropy loss function is primarily used.

Cross-entropy loss:

$$\mathcal{L} = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij})$$

This function defines the cross-entropy between the true distribution $\underline{\mathbf{y}}$ of the output and the predicted distribution $\underline{\hat{\mathbf{y}}}$. The aim of the training is to reduce this loss by changing the model parameters.This change is accomplished using gradient descent.

Gradient descent: it is a method to update parameters using some function, typically

$$\underline{\theta}_{t+1} = \underline{\theta}_t - \eta\nabla_{\underline{\theta}}\mathcal{L}(\underline{\theta}_t),$$

where $\nabla_{\underline{\theta}}\mathcal{L}(\underline{\theta}_t)$ is the gradient of the loss function w.r.t. the parameters of the model. An efficient way to compute this gradient is necessary for completing training in a reasonable amount of time.

Notation: During the analysis we will often need to refer to a specific partial derivative of the loss function w.r.t. an intermediate variable or a matrix of parameters $\mathbf{X}$. To simplify the notation, we will use this shorthand:

$$\overline{X} = \frac{\partial\mathcal{L}}{\partial X}.$$

2.2 The Transformer Architecture

2.2.1 Brief History

The Problem of Sequence Modeling. Many tasks in natural language processing, such as translation, summarization, or dialogue, require models that can process and generate sequences of tokens (words, subwords, or characters) in a coherent way. A token is mapped to a numerical vector representation, called an embedding, that captures its meaning in a continuous space. The central difficulty lies in handling the dependencies that exist between tokens: the interpretation of a word

or phrase often depends on other words that may occur either nearby or far apart in the sequence. Models must therefore be able to represent both local relationships (between adjacent tokens) and long-range dependencies (across distant positions). Another challenge is that sequences can vary in length, requiring flexible architectures that generalize to inputs of different sizes. Efficient computation also becomes critical, since real-world datasets contain millions or billions of tokens. Designing architectures that can capture dependencies of varying ranges, handle variable-length sequences, and scale to large datasets is the core problem that motivated the development of recurrent networks and, later, Transformers.

Before Transformers. Early natural language models often used recurrent neural networks (RNNs). They processed sequences iteratively, which made training slow and hard to parallelize for large datasets. They also had some difficulties with long sequences, as it was hard for the model to retain information from the beginning to the end of the data. The attention mechanisms in RNNs helped by making the model able to find and focus on important parts of the input, but since the core was still sequential by design, scaling up remained difficult.

The Transformer. The breakthrough came with the Transformer, an architecture introduced in 2017 by the paper "Attention is All You Need" [9]. Instead of relying on recurrence, the Transformer is built on self-attention, a mechanism that allows every token in a sequence to directly consider all other tokens at once. This design makes training very parallelizable, since all positions can be processed at the same time on modern hardware. Furthermore, it handles long-range dependencies better than RNNs, because distant tokens can be directly connected without passing information throughout the sequence. Finally, the architecture scales well with data, allowing increasingly large models to be trained efficiently, provided the necessary parallel hardware is available. These advantages are the reason why Transformer based approaches quickly replaced RNN based ones and is today the main architecture used in natural language processing.

2.2.2 The Big Picture

The transformer follows an encoder-decoder architecture. The core principle is self-attention: each token in a sequence can directly affect every other token, allowing the model to capture long-range dependencies without the sequential

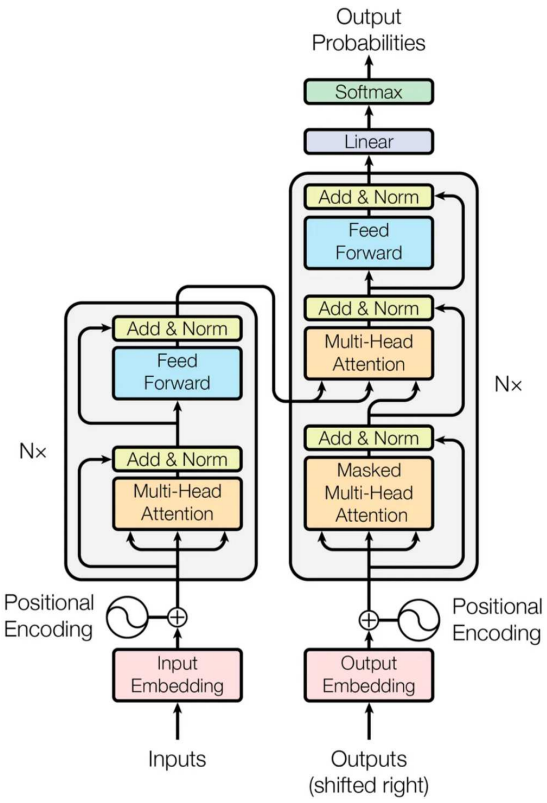

Figure 2.1: Architecture of the Transformer model.[9]

bottleneck of RNNs.

The transformer block is repeated N times to form deeper networks. Each block takes the output of the previous block as input. We will focus on the encoder part of the architecture with single head attention, as the analysis of the decoder is similar. Input sequences are first converted to embeddings, vectors of a chosen dimension d, and then added to positional encodings. These representations then flow through multiple transformer layers where they are updated. Each layer contains:

- A Self-attention mechanism
- A Position-wise feed-forward network
- Residual connections and layer normalizations

Note that even though the transformer block is repeated multiple times with the same structure, the parameters are not shared between layers, so each layer has its own set of parameters. In the following analysis we will focus on a single transformer block, as the computations and costs will be the same for each layer.

2.2.3 Overview of Components and Operations

A single transformer block performs the following key computations:

Self-Attention: For input sequence $\mathbf{X} \in \mathbb{R}^{n \times d}$ (where n is the sequence length and d is the model dimension):

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}^K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}^V$$

$$Y = \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V}$$

Here,

$$\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d \times d}$$

are learnable weight matrices, parameters of the transformer, that project the input $\mathbf{X}$ into queries ($\mathbf{Q}$), keys ($\mathbf{K}$), and values ($\mathbf{V}$) matrices.

The attention mechanism enables the model to find relevant parts of the input sequence when processing each token. Self-attention allows direct connections between any pair of positions. The queries, keys, and values provide a way to compute representations of features based on learned patterns.

Feed-Forward Network: A two-layer MLP is applied to each row of the attention output individually:

$$\text{FFN}(\underline{\mathbf{x}}) = \max(0, \underline{\mathbf{x}}\mathbf{W}_1 + \underline{\mathbf{b}}_1)\mathbf{W}_2 + \underline{\mathbf{b}}_2$$

Note that $max(0, .)$ is the ReLU activation function and is applied element-wise. Here,

$$\mathbf{W}_1 \in \mathbb{R}^{d \times d_{ff}}, \quad \mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d}$$

are weight matrices, and $\underline{\mathbf{b}}_1 \in \mathbb{R}^{d_{ff}}$, $\underline{\mathbf{b}}_2 \in \mathbb{R}^d$ are bias vectors. Both weights and biases are parameters of the transformer. The dimension d_{ff} is typically larger than d, often set to $4d$.

The feed-forward network adds a non-linear transformation to the model. While self-attention provides a way to mix information across positions, the FFN processes each position independently. A recent study demonstrated the importance of this step for the model's performance[4].

Layer Normalization: This trasformation is applied for each row vector before each sub-layer:

$$\text{LayerNorm}(\underline{\mathbf{x}}) = \underline{\gamma} \odot \frac{\underline{\mathbf{x}} - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \underline{\beta}$$

where μ and σ^2 are the mean and variance of the elements in $\underline{\mathbf{x}}$. $\underline{\gamma}$ and $\underline{\beta}$ are learnable parameters, and ε is a small constant for numerical stability. In this analysis we will assume $\varepsilon = 0$ for simplicity. Layer normalization stabilizes training by normalizing the inputs to the next component.

2.3 Training

2.3.1 How Similar Models are Trained

Transformer models are trained using supervised learning on large text data. The training process also usually considers:

Data Preparation: Text is tokenized into discrete units and converted to numerical representations, usually vectors of numbers. For language modeling, the training objective is to predict the next token given the previous context.

Optimization: Modern transformers use variants of gradient descent, such as Adam [6], with variable learning rate scheduling to ensure stable training.

Scale: Large models like GPT-3 [2] require distributed training across hundreds or thousands of GPUs, with advanced techniques to handle memory and time constraints.

2.3.2 Three Training Steps

Each training iteration consists of three consecutive phases:

Forward Pass: Input data flows through the network layers sequentially. The final output produces predictions $\hat{\mathbf{y}}$ for the next token, and then the loss $\mathcal{L}(\theta)$ is computed.

Backward Pass (Backpropagation): Gradients are computed by applying the chain rule from the loss value back throughout each layer, with the goal of finding $\nabla_{\underline{\theta}}\mathcal{L}(\underline{\theta})$.

Weight Update: Parameters are updated using the computed gradients and an optimization algorithm, such as gradient descent or Adam[6]. Modern optimizers may maintain additional state, making this step more computationally complex but still negligible compared to the gradient computation.

2.3.3 Why Backpropagation is Interesting

Even though the chain rule simplifies the calculation for backpropagation, it remains a computationally significant part of training. Studies show that backpropagation typically accounts for 60-70% of total training time in large models. Furthermore, by design, all intermediate activations must be stored during the forward pass to compute gradients, leading to high memory usage that scales with sequence length and batch size. For large transformers, this often becomes the limiting factor in training. Understanding these costs is important for optimizing model architectures, developing more efficient training algorithms and being able to evaluate budgets for training. This analysis becomes particularly important as models scale to billions of parameters, where even small improvements in backpropagation efficiency can lead to big reductions in training time and energy consumption.

3

Training Cost Analysis

3.1 Structure to be Analyzed

The structure that we will analyze is the encoder part of the original transformer architecture[9]. The block is composed of an attention layer, followed by a feed-forward layer, with normalization layers and residual connections applied at various points. The figure below illustrates the architecture.

We have choosen this architecture because is it the basis of many newer and

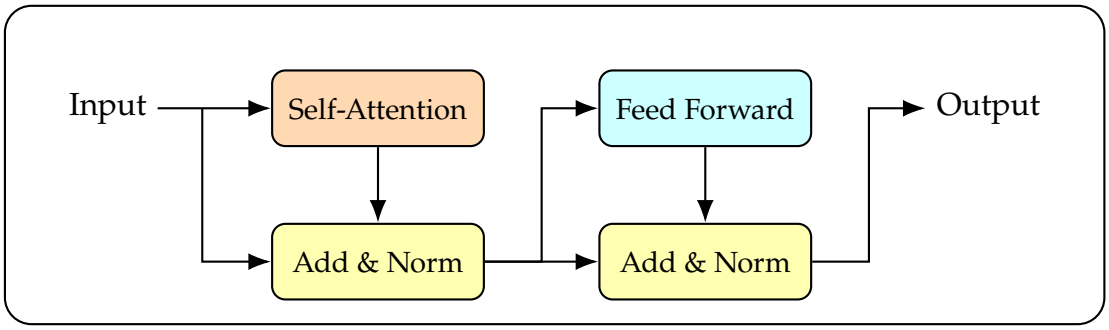

Figure 3.1: Transformer network

performing transformer networks. We will break down the network by analysing each component individually and then sum up the results to determine the overall training cost.

3.1.1 Single-head attention block

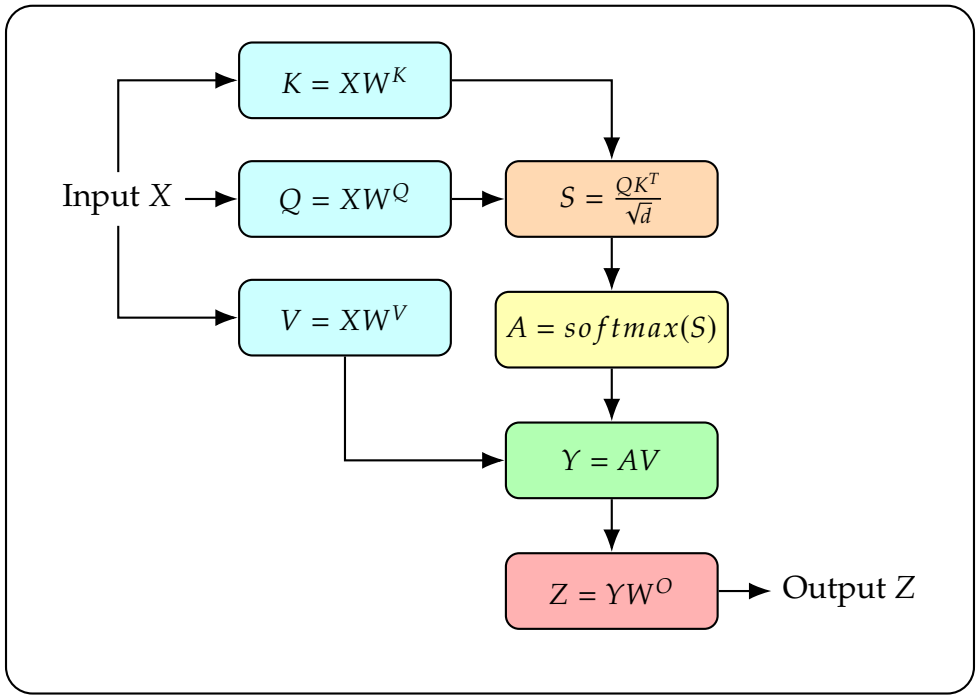

Figure 3.2: Summary of the operations in the attention block

Consider a sequence input $X \in \mathbb{R}^{n \times d}$ (n tokens). Learnable parameters are projection matrices $W^Q, W^K, W^V, W^O \in \mathbb{R}^{d \times d}$. We are using single-head attention of the original transformer. We want to emphasize that n is the sequence length (number of tokens to be processed in a pass) and d is the embedding dimension.

Forward pass

We will now recall some definitions used in the attention mechanism.
The definitions of queries, keys, values:

$$Q = XW^Q \in \mathbb{R}^{n \times d},$$
$$K = XW^K \in \mathbb{R}^{n \times d},$$
$$V = XW^V \in \mathbb{R}^{n \times d},$$

scaled dot-product and attention weights:

$$S = \frac{1}{\sqrt{d}} QK^\top \in \mathbb{R}^{n\times n},$$
$$A_{i,:} = \text{softmax}(S_{i,:}) \quad \text{(softmax applied rowwise)}, \quad A \in \mathbb{R}^{n\times n},$$

attention output and final projection:

$$Y = A\,V \in \mathbb{R}^{n\times d},$$
$$Z = YW^O \in \mathbb{R}^{n\times d}.$$

Derivative of a Matrix Product

Let $Z = XW$ with

$$X \in \mathbb{R}^{m\times n}, \quad W \in \mathbb{R}^{n\times p}, \quad Z \in \mathbb{R}^{m\times p}.$$

Suppose a scalar loss $\mathcal{L} = f(Z)$.
Derivative w.r.t. X: From $Z_{ij} = \sum_{k=1}^{n} X_{ik}W_{kj}$,

$$\frac{\partial Z_{ij}}{\partial X_{ab}} = \delta_{ia}W_{bj}.$$

Hence, by the chain rule:

$$\overline{X}_{ab} = \sum_{i,j} \overline{Z}_{ij}\,\delta_{ia}W_{bj} = \sum_{j=1}^{p} \overline{Z}_{aj}W_{bj} = (\overline{Z}W^\top)_{ab}.$$

Therefore

$$\boxed{\overline{X} = \overline{Z}W^\top}$$

Derivative w.r.t. W: Similarly,

$$\boxed{\overline{W} = X^\top\overline{Z}}$$

Parameter gradients & intermediate gradients

We compute gradients in a reverse order using the chain rule and the matrix product derivative above.

Output projection [$Z = YW^O$] In this step we can use directly the matrix product derivative. The number of aritmetic operations needed to mutliply 2 matrices of size $m \times n$ and $n \times p$ is $2mnp$.

$$\begin{aligned} \overline{W^O} &= Y^\top \overline{Z} \quad \in \mathbb{R}^{d \times d}, \\ \overline{Y} &= \overline{Z} W^{O^\top} \quad \in \mathbb{R}^{n \times d}. \end{aligned}$$

Therefore the number of operations needed in this step is $\mathbf{2nd^2} + \mathbf{2nd^2} = \mathbf{4nd^2}$.

Attention outputs [$Y = AV$] Similarly,

$$\begin{aligned} \overline{A} &= \overline{Y} V^\top \in \mathbb{R}^{n \times n}, \\ \overline{V} &= A^\top \overline{Y} \in \mathbb{R}^{n \times d}. \end{aligned}$$

Therefore the number of operations needed in this step is $\mathbf{2n^2d} + \mathbf{2n^2d} = \mathbf{4n^2d}$.

Attention weights [$A = \text{softmax}(S)$] Let us fix a row i. Then

$$S_{i,:} \in \mathbb{R}^n, \qquad A_{i,:} \in \mathbb{R}^n, \qquad \overline{A}_{i,:} \in \mathbb{R}^n,$$

and recall that $A_{i,k}$ is the k-th element of the softmax of $S_{i,:}$.

To compute the gradient with respect to the input row $S_{i,:}$, we apply the chain rule componentwise. Differentiating $A_{i,k}$ with respect to $S_{i,j}$, we obtain

$$\frac{\partial A_{i,k}}{\partial S_{i,j}} = A_{i,k}\big(\delta_{kj} - A_{i,j}\big).$$

Plugging this into the standard chain rule formula and using the gradient $\overline{A}_{i,k}$:

$$\overline{S}_{i,j} = \sum_{k=1}^{n} \overline{A}_{i,k} \frac{\partial A_{i,k}}{\partial S_{i,j}} = \sum_{k=1}^{n} \overline{A}_{i,k} A_{i,k} (\delta_{kj} - A_{i,j}).$$

The sum can be simplified by separating the $k = j$ term:

$$\overline{S}_{i,j} = \overline{A}_{i,j} A_{i,j} - A_{i,j} \sum_{k=1}^{n} \overline{A}_{i,k} A_{i,k} = A_{i,j} \left(\overline{A}_{i,j} - \sum_{k=1}^{n} \overline{A}_{i,k} A_{i,k} \right).$$

Collecting all coordinates into a vector expression, the gradient with respect to

row i is

$$\boxed{\overline{S}_{i,:} = A_{i,:} \odot \overline{A}_{i,:} - A_{i,:} \left(A_{i,:}^\top \overline{A}_{i,:}\right) = A_{i,:} \odot \left(\overline{A}_{i,:} - \left(A_{i,:}^\top \overline{A}_{i,:}\right) \mathbf{1}\right)}$$

where $\mathbf{1}$ is the all-ones vector of size n.

Applying this derivation row by row gives the full matrix gradient $\overline{S} \in \mathbb{R}^{n \times n}$ with rows $\overline{S}_{i,:}$.

The number of operations needed in this step is $(\mathbf{n} + \mathbf{2n} + \mathbf{n}) = \mathbf{4n}$ per row (n for the elementwise product, $2n$ for the dot product, n for the subtraction) . Therefore the total number of operations for all rows is $\mathbf{4n^2}$.

Scaled dot-product $[S = \frac{1}{\sqrt{d}} Q K^\top]$ We have

$$\overline{Q} = \frac{1}{\sqrt{d}} \overline{S}\, K \quad \in \mathbb{R}^{n \times d},$$
$$\overline{K} = \left(\frac{1}{\sqrt{d}} Q^\top \overline{S}\right)^\top = \frac{1}{\sqrt{d}} \overline{S}^\top Q \quad \in \mathbb{R}^{n \times d}.$$

This step is similar to the previous ones, except for the scaling by $1/\sqrt{d}$. The number of operations needed in this step is $\mathbf{2n^2d} + \mathbf{2n^2d} + \mathbf{2nd} = \mathbf{4n^2d} + \mathbf{2nd}$.

Projection matrices $[Q = XW^Q,\ K = XW^K,\ V = XW^V]$

$$\overline{W^Q} = X^\top \overline{Q} \in \mathbb{R}^{d \times d},$$
$$\overline{W^K} = X^\top \overline{K} \in \mathbb{R}^{d \times d},$$
$$\overline{W^V} = X^\top \overline{V} \in \mathbb{R}^{d \times d}.$$

The gradient w.r.t. the input X accumulates contributions from all three projection paths:

$$\overline{X} = \underbrace{\overline{Q} W^{Q^\top}}_{\text{from } Q} + \underbrace{\overline{K} W^{K^\top}}_{\text{from } K} + \underbrace{\overline{V} W^{V^\top}}_{\text{from } V} \in \mathbb{R}^{n \times d}.$$

Again, the number of operations needed in this step is:
$\mathbf{3} \cdot (\mathbf{2nd^2}) + \mathbf{3} \cdot (\mathbf{2nd^2}) + \mathbf{2} \cdot (\mathbf{nd}) = \mathbf{12nd^2} + \mathbf{2nd}$.

Summary

The numbers of operations required for the backpropagation through the attention block is summarized as follows:

- **Output projection:** $4nd^2$
- **Attention outputs:** $4n^2d$
- **Attention weights (softmax):** $4n^2$
- **Scaled dot-product:** $4n^2d + 2nd$
- **Projection matrices and input gradient:** $12nd^2 + 2nd$

Total: $\mathbf{16nd^2 + 8n^2d + 4n^2 + 4nd}$ operations per attention block.

3.1.2 Feed-Forward Layer (FFL)

Let $X_i \in \mathbb{R}^{1\times d}$ denote the i-th row of the input $X \in \mathbb{R}^{n\times d}$. The forward pass for this row is

$$\begin{aligned} H_i &= X_i W_1 + \underline{b_1} \quad \in \mathbb{R}^{1\times d_{\text{ff}}}, \\ U_i &= max(0, H_i) \quad \in \mathbb{R}^{1\times d_{\text{ff}}}, \\ Z_i &= U_i W_2 + \underline{b_2} \quad \in \mathbb{R}^{1\times d}. \end{aligned}$$

max is applied elementwise (ReLU nonlinearity).

Backward pass

Output projection $[Z_i = U_i W_2 + \underline{b_2}]$

$$\begin{aligned} \overline{W_2} &= U_i^\top \overline{Z_i} \quad \in \mathbb{R}^{d_{\text{ff}}\times d}, \\ \overline{U_i} &= \overline{Z_i} W_2^\top \quad \in \mathbb{R}^{1\times d_{\text{ff}}}, \\ \overline{\underline{b_2}} &= \overline{Z_i} \quad \in \mathbb{R}^{1\times d}. \end{aligned}$$

Operations needed: $2d_{\text{ff}}d + 2d_{\text{ff}}d = \mathbf{4d_{\text{ff}}d}$.

Nonlinearity [$U_i = max(0, H_i)$] For ReLU,

$$\overline{H_{ij}} = \mathbf{1}_{(H_i)_j > 0} \, \overline{U_{ij}}, \quad j = 1, \ldots, d_{\text{ff}}.$$

No aritmetic operations are needed in this step, only comparisons.

First projection [$H_i = X_i W_1 + \underline{b_1}$]

$$\begin{aligned} \overline{W_1} &= X_i^\top \, \overline{H_i} \quad \in \mathbb{R}^{d \times d_{\text{ff}}}, \\ \overline{X_i} &= \overline{H_i} \, W_1^\top \quad \in \mathbb{R}^{1 \times d}, \\ \overline{\underline{b_1}} &= \overline{H_i} \quad \in \mathbb{R}^{1 \times d_{\text{ff}}}. \end{aligned}$$

Operations: $2dd_{\text{ff}} + 2dd_{\text{ff}} = \mathbf{4dd_{\text{ff}}}$.

Summary

The total number of operations for backpropagation through the FFL for a single input row X_i is composed of:

- **Output projection:** $4dd_{\text{ff}}$
- **Activation:** 0
- **First projection:** $4dd_{\text{ff}}$

Total: $8dd_{\text{ff}}$ operations per input row. So for the whole input $X \in \mathbb{R}^{n \times d}$, the total number of operations is $\mathbf{8ndd_{\text{ff}}}$.
Typically $d_{\text{ff}} = 4d$, so the total is $\mathbf{32nd^2}$ operations for the FFL.

3.1.3 Normalization layer

Consider a single input row $X_i \in \mathbb{R}^{1 \times d}$. Learnable scale and shift parameters are $\underline{\gamma}, \underline{\beta} \in \mathbb{R}^d$. Normalization is applied across the d features of X_i.

Forward pass

For each input row X_i, we compute

$$\begin{aligned}
\mu_i &= \frac{1}{d}\sum_{j=1}^{d} X_{ij}, \\
\sigma_i^2 &= \frac{1}{d}\sum_{j=1}^{d}(X_{ij}-\mu_i)^2, \\
\sigma_i &= \sqrt{\sigma_i^2}, \\
Y_{ij} &= \frac{X_{ij}-\mu_i}{\sigma_i}, \\
Z_i &= \underline{\gamma} \odot Y_i + \underline{\beta}.
\end{aligned}$$

Parameter gradients & normalized input

Using the usual methods:

$$\begin{aligned}
\overline{\gamma_j} &= \sum_i \overline{Z}_{ij}\, Y_{ij}, \\
\overline{\beta_j} &= \sum_i \overline{Z}_{ij}, \\
\overline{Y_{ij}} &= \overline{Z}_{ij}\, \gamma_j.
\end{aligned}$$

For a single row X_i, the number of arithmetic operations needed in this step is **3d**.

Derivative of normalized input

We want to find $\frac{\partial Y_{ij}}{\partial X_{ik}}$. We will derivate the standard formula step by step.

$$\frac{\partial Y_{ij}}{\partial X_{ik}} = \frac{1}{\sigma_i}\frac{\partial(X_{ij}-\mu_i)}{\partial X_{ik}} - \frac{X_{ij}-\mu_i}{\sigma_i^2}\frac{\partial \sigma_i}{\partial X_{ik}}.$$

$$\frac{\partial \mu_i}{\partial X_{ik}} = \frac{1}{d} \quad \Longrightarrow \quad \frac{\partial(X_{ij}-\mu_i)}{\partial X_{ik}} = \delta_{jk} - \frac{1}{d}$$

$$\frac{\partial \sigma_i}{\partial X_{ik}} = \frac{1}{2\sqrt{\sigma_i^2}} \cdot \frac{\partial \sigma_i^2}{\partial x_{ik}} = \frac{1}{2\sigma_i} \cdot \frac{2}{d}(x_{ik} - \mu_i) = \frac{x_{ik} - \mu_i}{d\,\sigma_i}.$$

Now we just need to plug these into the original formula, and make some simplifications:

$$\frac{\partial Y_{ij}}{\partial X_{ik}} = \frac{1}{\sigma_i}\left(\delta_{jk} - \frac{1}{d} - \frac{(X_{ij} - \mu_i)(X_{ik} - \mu_i)}{d\,\sigma_i^2}\right) = \frac{1}{\sigma_i}\left(\delta_{jk} - \frac{1}{d} - \frac{1}{d}Y_{ij}Y_{ik}\right).$$

Therefore the gradient with respect to X_{ik} is

$$\begin{aligned}\overline{X}_{ik} &= \sum_{j=1}^{d} \overline{Y}_{ij} \frac{\partial Y_{ij}}{\partial X_{ik}} \\ &= \frac{1}{\sigma_i}\left(\overline{Y}_{ik} - \text{mean}(\overline{Y}_i) - Y_{ik}\ \text{mean}(\overline{Y}_i \odot Y_i)\right),\end{aligned}$$

where $\overline{Y}_i, Y_i \in \mathbb{R}^d$ are the vectors for row i.

The number of operations needed in this step is **7d** per row, assuming the mean is computed once and reused.

Summary

The number of operations required for backpropagation through the normalization layer for each token is summarized as follows:

- **Gradients w.r.t. parameters & normalized input:** $2d$
- **Gradient w.r.t. original input:** $7d$

Total: $9d$ operations per token, thus for the whole input $X \in \mathbb{R}^{n \times d}$, the total number of operations is **9nd**.

3.2 Total Cost of Training

The transformer block analyzed consists of:

- 1 Attention layer
- 1 Feed-Forward Layer (FFL)

- 2 Normalization layers

The total number of operations for backpropagation through the entire transformer block is the sum of the individual components:

$$\begin{aligned} \textbf{Attention block:} \quad & 16nd^2 + 8n^2d + 4n^2 + 4nd \\ \textbf{FFL:} \quad & 32nd^2 \\ \textbf{Normalization layers:} \quad & 2 \cdot 9nd = 18nd \end{aligned}$$

Total:

$$\mathbf{48nd^2 + 8n^2d + 4n^2 + 22nd}$$

In Big O notation, the dominant terms are:

$$\mathbf{O(nd^2 + n^2d)}$$

4 Discussion of the Results

4.1 Result Recap

The analysis in Chapter 3 yielded a closed-form expression for the number of arithmetic operations required during backpropagation through a single Transformer block:

$$\textbf{Total:} \quad \mathbf{48nd^2 + 8n^2d + 4n^2 + 22nd},$$

where n is the sequence length and d the embedding dimension. This form already reflects the key scaling challenges of Transformers: quadratic cost in d for a feed-forward layer, and quadratic cost in n for a self-attention block.

4.2 Practical Implications

4.2.1 Heavy Components vs. Light Components

The results allow us to compare the computational weight of each component:

The Feed-Forward Layer (FFL) is the heaviest contributor in terms of d^2, requiring $32nd^2$ operations. As noted in the literature, FFLs are often the dominant part of Transformer training when embedding sizes grow.

The Attention block is more sensitive to sequence length, because of terms $16nd^2 + 8n^2d$. The n^2d term becomes the main bottleneck for long contexts, which is consistent with the widely discussed quadratic complexity of self-attention.

The Normalization layers require only $20nd$ operations, which is negligible compared to the quadratic terms. This explains why normalization is not a target for optimization in large scale systems.

4.2.2 Embedding Dimension vs. Context Size

The two dominant terms highlight the trade-off between widening the embedding dimension and extending the sequence length. Increasing d improves the representational capacity but results in a quadratic penalty through nd^2. This makes the FFL particularly heavy when d is large. Increasing n expands the usable context, the number of tokens that can be processed in one forward pass, but the quadratic n^2d cost from attention grows much faster. In practice, this is the limiting factor when training on very long sequences (thousands of tokens). This trade-off has motivated the research into more efficient attention mechanisms that reduce the scaling cost. It is important to emphasize that having a large context size, without a sufficiently big embedding dimension, can lead to underfitting, as the model may lack the capacity to capture complex patterns in the data. On the contrary, a very high embedding dimension with a small context size may lead to overfitting, as the model might memorize training examples rather than generalizing well. Therefore, balancing n, d and training cost is one of the most difficult tasks in designing Transformer architectures.

4.2.3 Optimization Focus

This breakdown highlights where and why optimizations found in the literature are most impactful:

Attention optimization : Because of the n^2d term, many methods focus on reducing its cost. These approaches aim to achieve sub-quadratic complexity while retaining high accuracy on the benchmark tasks. Although the practical effectiveness of these methods is empirically proven, they often come with trade-offs in terms of model expressivity and training stability. The quadratical nature

of attention is still a fundamental limitation.

FFL optimization : Feed-forward layers (FFLs) are a major computational bottleneck in transformers, scaling as $O(nd^2)$. To reduce this cost, several strategies are used: (i) *low-rank decompositions*, which factorize weight matrices and reduce complexity to $O(ndr)$ with $r \ll d$ [5]; (ii) *bottleneck structures*, which project through a smaller intermediate dimension d_{int} [1]; and (iii) *Mixture-of-Experts (MoE)*, where only a small subset of expert networks is activated per token. These methods lower effective computation while maintaining or even expanding model capacity, enabling the construction of larger and more efficient transformers[3].

It is also worth noting that, in real training systems, memory bandwidth and data movement can become as important as arithmetic complexity, and thus a system specific analysis is often required to identify the true bottlenecks.

4.2.4 Single-Head vs. Multi-Head Attention

Our analysis focused on single-head attention, but in practice, most models use multi-head attention. Both share the same asymptotic cost $O(n^2d)$, but multi-head attention divides the embedding dimension across multiple smaller heads. This parallelism increases training stability and improves the model's ability to capture diverse relationships.

Recent research shows that while deep single-head Transformers can achieve comparable expressivity, they are harder to train and less stable without careful initialization[7]. Multi-head attention tends to perform better in in-context learning tasks and is generally preferred in modern architectures. Therefore, although the complexity remains quadratic in n, the practical benefits justify the use of multiple heads.

5

Conclusions

In this thesis we analyzed in detail the training cost of a Transformer block, breaking down the contribution of each component. We derived a closed form expression for the number of arithmetic operations required during backpropagation through a Transformer network. The analysis showed that the **feed-forward layer** is expensive in terms of embedding dimension d (scaling as nd^2), while the **attention mechanism** is expensive in terms of sequence length n (scaling as n^2d). Normalization layers contribute negligibly in comparison.

The final closed-form expression,

$$\mathbf{48nd^2 + 8n^2d + 4n^2 + 22nd \in O(nd^2 + n^2d)},$$

highlights the scaling limitations of Transformers.

Looking forward, the analysis clarifies where and why research and engineering efforts are most impactful. The quadratic dependence on sequence length in attention remains the most restrictive factor, motivating the need for the development of efficient attention variants and sparse approximations. Future Transformer models will likely improve on this issue, combining more efficient mathematical formulations with hardware driven optimizations.

In conclusion, while the Transformer has set an important foundation, its training cost is still a central challenge. Understanding these costs precisely, as done in this work, provides the ground on which new architectures and optimizations can be evaluated and developed.

References

[1] Srinadh Bhojanapalli et al. *Low-Rank Bottleneck in Multi-Head Attention Models*. arXiv preprint. arXiv:2002.07028. Feb. 2020. URL: https://arxiv.org/abs/2002.07028.

[2] Tom Brown et al. "Language Models Are Few-Shot Learners". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020. URL: https://arxiv.org/abs/2005.14165.

[3] Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. *SwitchHead: Accelerating Transformers with Mixture-of-Experts Attention*. arXiv preprint. arXiv:2312.07987. Dec. 2023. URL: https://arxiv.org/abs/2312.07987.

[4] Isaac Gerber. *Attention Is Not All You Need: The Importance of Feedforward Networks in Transformer Models*. arXiv preprint. arXiv:2505.06633. May 2025. URL: https://arxiv.org/abs/2505.06633.

[5] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv preprint. arXiv:2106.09685. June 2021. URL: https://arxiv.org/abs/2106.09685.

[6] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations (ICLR)*. Originally published on arXiv in Dec 2014. 2017. URL: https://arxiv.org/abs/1412.6980.

[7] Liyuan Liu, Jialu Liu, and Jiawei Han. *Multi-head or Single-head? An Empirical Comparison for Transformer Training*. arXiv preprint. arXiv:2106.09650. June 2021. URL: https://arxiv.org/abs/2106.09650.

[8] Nestor Maslej et al. *The AI Index 2025 Annual Report*. Tech. rep. Stanford, CA: AI Index Steering Committee, Institute for Human-Centered AI, Stanford University, Apr. 2025. URL: https://arxiv.org/abs/2504.07139.

[9] Ashish Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017. URL: https://arxiv.org/abs/1706.03762.