# Thresholding Binarized Neural Networks to improve accuracy in Large Language Models Training

A thesis submitted in partial fulfillment of the requirements for the award of the degree of

**B.Tech.**

**in**

**Computer Science and Engineering**

By

**Arpon Kapuria (106120014)**

**Brintha M (106120025)**

**Udipta Pathak (106120135)**


**DEPARTMENT OF**
**COMPUTER SCIENCE AND ENGINEERING**
**NATIONAL INSTITUTE OF TECHNOLOGY**
**TIRUCHIRAPPALLI-620015**

**MAY 2024**

**BONAFIDE CERTIFICATE**

This is to certify that the project titled **Thresholding Binarized Neural Networks to improve accuracy in Large Language Models Training** is a bonafide record of the work done by

**Arpon Kapuria (106120014)**

**Brintha M (106120025)**

**Udipta Pathak (106120135)**

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** of the **NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI**, during the year 2023-2024.

**Dr. N Ramasubramnaian**
Project Guide

**Dr. S. Mary Saira Bhanu**
Head of the Department

Project Viva-voce held on _____

**Internal Examiner**

**External Examiner**

# ABSTRACT

Large Language Models are powerful models that are trained on large amount of data to perform human level task and Transformers work as their backbone architecture. Transformers have become SOTA. It can accommodate large scale data. However the disadvantage is it's model size and even larger runtime usages. Therefore it is important to compress the model to lower it's hardware cost and accelerate the model inference. We aim to solve the issue by compacting the data to train and infer on transformer using techniques like quantization and binarization. First we explore different quantization techniques then we apply them on transformers to check sensitivity of different parts of transformer. We tried to achieve a trade off between quantization which compresses data to some extent, and binarization that compress data to extreme. We propose a method that is more compressed than quantization and more accurate than binarized method. Instead of mapping weights to -1 and 1 like we do in binarized neural network, we use mean of weights from Xavier initialisation to get more accurate representation of weights without losing much information. Although we couldn't surpass the baseline accuracy but this resulted in a better reduction ratio of the model size. We got a better compression than quantization and better accuracy than binarization.

*Keywords*: Large Language Models, Transformers, Compression, Quantization, Binarization, Binarized Neural Networks

## ACKNOWLEDGEMENT

We are grateful to the following individuals whose support and invaluable guidance have been instrumental in the successful completion of this project.

First and foremost, we extend our sincere appreciation to our project mentor, **Dr. N. Ramasubramanian**, for his guidance, constructive feedback and timely advice throughout the course of this project.

Secondly, we would like to thank **Dr. S. Mary Saira Bhanu**, the Head of the Department, Computer Science and Enginnering.

We would also like to express our heartfelt gratitude to our internal reviewers, **Dr. M. Brindha**, **Dr. B. Shameedha Begum**, and **Dr. Jakkepalli Pavan Kumar**, for their expert evaluation and insightful comments. Their valuable suggestions and recommendations have played a crucial role in refining and improving our work.

Special thanks to **Dipanjan Banarjee**, for his guidance, constructive feedback and timely advice throughout the course of this project.

Moreover, we are deeply thankful to our loved ones, including our parents and close friends for their constant support and motivation. Finally, we wholeheartedly acknowledge our immense gratitude to all those who have contributed to the successful culmination of this project.

# TABLE OF CONTENTS

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Large Language Models

Large language models (LLMs) have emerged as a transformative force in natural language processing (NLP). These robust neural networks, trained on massive datasets of text and code, can perform a wide range of tasks with remarkable fluency and accuracy. At their core, LLMs are complex neural networks trained on vast amounts of text and code data. This training process allows them to learn the intricate relationships between words, understand the nuances of language, and generate grammatically correct and semantically meaningful text. The architecture of LLMs often relies heavily on the transformer model, a groundbreaking advancement in NLP introduced in the paper "Attention is All You Need" by Vaswani et al. (2017) [1]. Unlike traditional recurrent neural networks (RNNs), transformers leverage self-attention mechanisms to capture long-range dependencies within sequences [1]. This allows the model to understand complex relationships between words, even if they are far apart in the sequence, leading to superior performance in various NLP tasks [2]. LLMs generally follow a three-step process to perform their tasks:

1. **Training:** This initial phase involves training the model on a massive dataset of text and code. During this process, the LLM learns to identify relationships between words, understand the nuances of language, and generate grammatically correct and semantically meaningful text.

2. **Encoding:** When presented with an input, such as a prompt or a question, the

LLM encodes it into a numerical representation. This encoding captures the meaning and context of the input, allowing the model to process it effectively.

3. **Decoding:** Based on the encoded input, the LLM generates an output sequence. This could be a translation, a summary, a response to a question, or any other form of text that aligns with the task at hand.

Despite their impressive capabilities, LLMs face several challenges that need to be addressed [3]. Training and running LLMs require significant computational resources, including high-performance GPUs and large amounts of memory. This limits their deployment on resource-constrained devices, such as smartphones or laptops, hindering widespread application adoption. LLMs trained on biased data can perpetuate societal biases, leading to discriminatory outputs or unfair treatment of certain groups [3]. Careful attention to data selection and model training processes is crucial to mitigate these issues and ensure responsible development. Understanding the internal workings of large and complex LLMs can be challenging. This lack of transparency makes it difficult to interpret their decision-making processes, raising concerns about potential biases or errors that may go unnoticed.

The high computational demands of LLMs necessitate optimization techniques to enable their wider deployment on various hardware platforms and facilitate real-world applications [4]. Explore various optimization strategies to accelerate LLMs and make them more efficient, paving the way for their broader adoption and impact across diverse domains is necessary.

## 1.2 Transformer Architecture - The Backbone of LLMs

The transformer architecture plays a pivotal role in the remarkable capabilities of modern large language models (LLMs). It provides a powerful and efficient way to process and understand natural language, overcoming limitations faced by previous architectures like recurrent neural networks (RNNs). RNNs, while effective in language processing tasks, suffer from inherent drawbacks. Their sequential processing nature

makes them computationally expensive, and they need help to capture long-range dependencies within sequences effectively. With their self-attention mechanism, transformers address these limitations, leading to significant advancements in NLP performance [1].

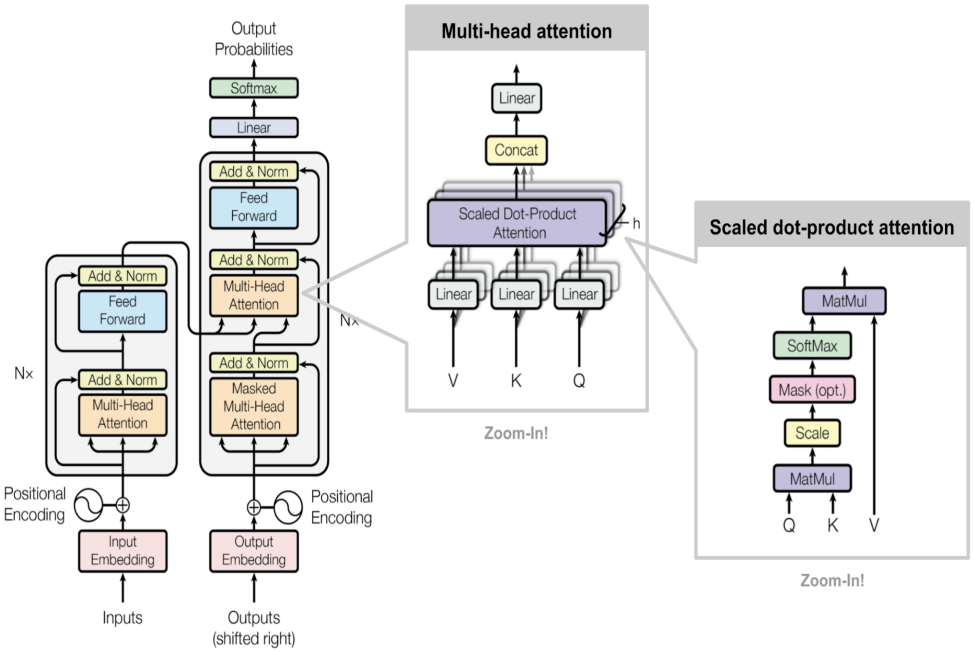


Figure 1.1: Transformer Architecture with Self-Attention Mechanism

### 1.2.1 Encoder: Processing the Input Sequence

The encoder is responsible for processing the input sequence and generating a representation that captures its meaning and context. It typically consists of multiple layers, each containing a self-attention mechanism and a feed-forward network.
Here's what happens within an encoder layer:

- **Self-Attention:** The self-attention mechanism within each layer allows the model to attend to different parts of the input sequence and capture relationships between words. This enriched understanding is crucial for tasks like machine translation or question answering.

- **Add & Norm:** The outputs of the self-attention mechanism are added to the original input and then normalized.

- **Feed-Forward Network:** The normalized sum is passed through a feed-forward network, which adds non-linearity to the model and allows it to learn more complex relationships between words.

- **Add & Norm (Again):** The output of the feed-forward network is added to the previous output and then normalized again.

The encoder stacks multiple layers, each building upon the previous one to create a progressively richer representation of the input sequence.

### 1.2.2 Self Attention: Unveiling the Core Mechanism

At the heart of the transformer lies the self-attention mechanism, a revolutionary concept that allows the model to understand the intricate relationships within a sequence. Here is a deeper look at how it works:

- **Query, Key, and Value Vectors:** Each word in the input sequence is represented by three vectors: query, key, and value. These vectors are created by passing the word embedding through separate linear layers.

- **Attention Scores:** The model calculates attention scores for each pair of words in the sequence. This is done by computing the dot product between the query vector of one word and the key vector of every other word. The higher the attention score, the more relevant the other word is to the current word.

- **Weighted Sum:** The attention scores are then used to weight the value vectors of all words in the sequence. This creates a context vector that summarizes the information relevant to the current word, considering its relationships with other words in the sequence.

### 1.2.3 Decoder: Generating the Output Sequence

The decoder takes the encoded representation from the encoder and uses it to generate the output sequence, such as a translation, a summary, or a response to a question. It

also utilizes self-attention mechanisms with additional complexities to handle the output generation process.

Here is what happens within a decoder layer:

- **Masked Self-Attention:** The decoder employs masked self-attention, which prevents the model from attending to future words in the output sequence. This ensures the model generates the output sequentially, one word at a time.
- **Encoder-Decoder Attention:** The decoder also attends to the encoded representation from the encoder using another self-attention mechanism. This allows the decoder to incorporate the context of the input sequence into the generated output.
- **Add & Norm:** Similar to the encoder, the outputs of the attention mechanisms are added to the original input and then normalized.
- **Feed-Forward Network:** The normalized sum is passed through a feed-forward network.
- **Add & Norm (Again):** The output of the feed-forward network is added to the previous output and then normalized again.

The decoder also stacks multiple layers, with each layer using the information from the previous layer to refine the generated output sequence.

By combining the power of self-attention with encoders and decoders, the transformer architecture has revolutionized natural language processing, achieving remarkable results in various tasks.

## 1.3 Accelerating Large Language Models

Despite their remarkable capabilities, LLMs based on transformer architectures face significant computational bottlenecks [3]. Training and running large transformer models require substantial computational resources, including high-performance GPUs and

vast amounts of memory. This limits their deployment on resource-constrained devices like laptops or smartphones, hindering widespread adoption in various applications. Processing long sequences and maintaining attention scores within transformers requires significant memory resources [6]. This can be a significant hurdle for deploying these models on devices with limited memory. To address these challenges and pave the way for broader LLM adoption, researchers have explored various optimization techniques [5] aimed at accelerating transformers and making them more efficient:

### 1.3.1 Model Size Reduction:

- **Parameter Pruning:** Pruning techniques identify and remove redundant or unimportant parameters within the model. This reduces the overall model size, leading to lower memory requirements and faster inference times [7].

- **Knowledge Distillation:** This technique involves training a smaller student model by mimicking the outputs of a larger teacher model. This allows the student model to capture the knowledge of the larger model while being significantly smaller and more efficient [8].

### 1.3.2 Numerical Precision Reduction:

- **Quantization:** Quantization techniques reduce the numerical precision of model weights and activations from 32-bit floating-point numbers (FP32) to lower precision formats like 16-bit (FP16) or even 8-bit integers (INT8) [9]. This leads to significant memory savings and faster computations, often with minimal impact on model accuracy.

### 1.3.3 Hardware and Software Optimizations:

Utilizing specialized hardware like TPUs (Tensor Processing Units) or AI accelerators designed for efficient deep learning computations can significantly accelerate LLM inference. Optimizing software libraries and frameworks to leverage the parallel processing capabilities of GPUs and other hardware can further enhance the speed and efficiency of transformer models.

## 1.4 Objectives

Transformers have become SOTA. It can accommodate large-scale data. However, the disadvantage is that it is a model size with even more extensive runtime usage. Therefore, it is important to compress the model to lower its hardware cost and accelerate the model inference. Exploring efficient and effective quantization and binarization methods for transformers is the main focus of our project so that we can accelerate the training and reach a converging point faster.

- Implement and modify diverse quantization and binarization methods to see how they perform on the transformer.
- Explore the internal feature of the transformer structure and design a special quantization/binarization pattern to improve the optimization of the transformer.
- Reach a trade-off between model size and bearable classification error as an optimal compression strategy.
- We aim to get a compromise between quantization and binarization, achieving the best of both.

# Chapter 2

# Literature Review

## 2.1 Background

### 2.1.1 Quantization

One of the key optimization techniques for accelerating transformers is quantization [9]. This technique involves reducing the numerical precision of model weights and activations from the standard 32-bit floating-point format (FP32) to lower precision formats like 16-bit (FP16) or even 8-bit integers (INT8). This leads to significant benefits in terms of:

- **Memory Savings:** Lower precision formats require less memory to store, leading to a smaller memory footprint for the entire model[5]. This is particularly beneficial for deploying LLMs on resource-constrained devices.
- **Faster Computations:** Lower precision arithmetic operations are generally faster to perform on hardware, leading to faster inference times and improved overall efficiency.

While reducing precision can potentially lead to a slight decrease in model accuracy, careful implementation often results in minimal impact on performance while offering significant efficiency gains [11]. This makes quantization a valuable tool for making LLMs more accessible and deployable on various hardware platforms.
Here are some common quantization techniques used for transformers:

- **Post-Training Quantization:** This involves quantizing a pre-trained FP32 model to a lower precision format. This approach is relatively straightforward but may require fine-tuning the model after quantization to recover some of the lost accuracy [10][16].

- **Quantization-Aware Training:** This technique involves training the model from scratch with lower precision weights and activations [11][16]. This can lead to better compatibility with the lower precision format and potentially higher accuracy compared to post-training quantization.

By leveraging quantization techniques, researchers are continuously pushing the boundaries of what's possible in terms of LLM efficiency, paving the way for their wider adoption in real-world applications.

### 2.1.2 Binarized Neural Networks

The paper by Matthieu Courbariaux et al. [12] proposes a novel type of neural network architecture called Binarized Neural Networks (BNNs). Unlike standard neural networks that use complex numbers for weights and activations, BNNs constrain these values to be either +1 or -1 during runtime.
This extreme simplification leads to several advantages:

- **Reduced memory footprint:** Since weights and activations are binary, they require significantly less memory to store compared to traditional floating-point values.

- **Faster computations:** BNNs replace complex multiplications with bitwise operations like XNOR, which are much faster to perform on hardware. This translates to improved power efficiency.

- **Potential for hardware acceleration:** The binary nature of BNNs aligns well with specialized hardware designs, potentially leading to significant performance gains.

**Threshold Functions in BNNs**

Threshold functions play a crucial role in BNNs, determining how the continuous outputs from the network are converted into the binary values (+1 or -1) required for weights and activations during inference. Here are two commonly used threshold functions in BNNs:

- **Sign Threshold Function:** This is the most widely used threshold function in BNNs. It is a simple function defined as:

$$\text{threshold}(x) = \sigma(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases}$$

  The sign function maps any real-valued input (x) to either +1 or -1 based on its sign. If the input is positive or zero, it becomes +1. Conversely, if the input is negative, it becomes -1. This function effectively binarizes the weights and activations during inference.

- **Hard Tangent Threshold Function:** While less commonly used than the sign function, the hard tangent function can also be employed for binarization. It is defined as:

$$\text{threshold}(x) = \text{hardTanh}(x) = \begin{cases} +1, & \text{if } x > 0 \\ -1, & \text{if } x \leq 0 \end{cases}$$

  Similar to the sign function, the hard tangent function binarizes the input by mapping positive values to +1 and non-positive values to -1. However, unlike the sign function, the hard tangent function introduces a small dead zone around zero. This can potentially lead to slightly different network behavior compared to the sign function.

### Training

However, training BNNs presents a challenge. The standard training algorithms rely on gradients calculated using real-valued weights and activations. The authors address this by proposing a method that uses binary values during training to compute gradients but accumulates them in real-valued variables. This approach leverages the benefits of SGD (Stochastic Gradient Descent) [13] while maintaining the binary constraint for inference. This essentially ignores the non-differentiability of the threshold function and treats it as an identity function (output equals input) for gradient calculation. This allows the gradients from the previous layer to be "passed through" to the weights and activations in the BNN layer.

The authors propose a training method that incorporates the following key steps:

- **Binary Training with Real-Valued Accumulations:** During training, the BNN operates with binary weights and activations. However, for each weight $w_i$, a surrogate variable $w_i'$ (real-valued) is introduced. This surrogate variable is a real-valued number that accumulates the gradients calculated using the binary weights during each training iteration.

- **Gradient Updates:** After each forward and backward pass, the gradients are computed based on the current binary state of the weights and activations. These gradients are used to update the surrogate variables $w_i'$:

$$\Delta w_i' = \eta \times \frac{\delta loss}{\delta a_i}, \text{where } \eta \text{ is the learning rate}$$

- **Binarization after Update:** Following the gradient update using the accumulated values in the surrogate variables, the weights are binarized by applying a threshold function (typically a sign function). This ensures that the weights remain constrained to +1 or -1 during inference.

$$w_i = \sigma(w_i')$$

In essence, the training process in BNNs leverages real-valued calculations during training to compute gradients and then applies a threshold function like the sign function to convert these values to the binary domain (+1 or -1) used for inference. This approach enables BNNs to train using well-established algorithms like SGD while maintaining the desired binary constraint for efficient execution.

### 2.1.3 Straight-Through Estimator

The straight-through estimator provides a way to approximate the gradients of non-differentiable operations during the backward pass of backpropagation. The key idea behind the STE is to treat the non-differentiable operation as if it were the identity function during the backward pass, while still applying the actual non-differentiable operation during the forward pass.

**Methodology**

Mathematically, let's consider a non-differentiable function $f(x)$, where $x$ is the input to the function. During the forward pass, the output $y$ is computed as $y = f(x)$. However, during the backward pass, instead of computing the actual gradient of $f(x)$, which may be undefined or computationally expensive, the STE approximates the gradient as:

$$\frac{\delta y}{\delta x} = 1$$

This approximation means that the gradient is treated as if the non-differentiable function were the identity function, allowing the backpropagation algorithm to propagate the gradients through the non-differentiable operation.

**Use Case**

The straight-through estimator is particularly useful in situations such as:

**Quantization:** Quantization involves rounding or clipping values to a specific number of bits, which is a non-differentiable operation.

**Discretization:** Discrete operations includes sampling from a categorical distribution or selecting the maximum value in a softmax output.

**Attention mechanisms:** In transformer, scaled dot-product attention uses non-differentiable operations like masking or softmax with temperature.

### 2.1.4 Xavier Initialization

The goal of Xavier initialization [14] is to solve the issue of vanishing or exploding gradients, which can happen when deep neural networks are being trained. These problems occur when gradients get too big (exploding gradients) or too tiny (vanishing gradients), which causes instability or inefficiency in the learning process.

#### Initialization

Weights can be initially sampled from either Gaussian or uniform distribution within the open set $(-1, +1)$, and have all the weights scaled down by $\sqrt{d_{in}}$ where $d_{in}$ is the dimension of input vector to a layer.

$$W_{m\times n} = [w_{ij}]_{m\times n} = \frac{1}{\sqrt{d_{in}}} \left[ \underset{x\sim\mathcal{D}}{\text{clip}_{-1,1}}(x) \right]_{m\times n}$$

where,

$$\text{clip}_{\alpha,\beta}(x) = \begin{cases} \alpha & \text{, if } x \leq \alpha \\ \beta & \text{, if } \beta \leq x \\ x & \text{, otherwise} \end{cases}$$

#### Working Principle

Xavier initialization sets the neural network's weights such that the variance of each layer's outputs is roughly equal to the variance of its inputs. This aids in keeping the gradients from inflating or disappearing during the backpropagation phase, which updates the weights in accordance with the gradients.

- If the weights are initialized randomly with mean 0 and variance $\sigma^2$, and there are $n_{in}$ inputs, the variance of the output of a neuron will be $n_{in} \times \sigma^2$.
- To ensure that the variance of the output is approximately equal to the variance of the inputs, we need to scale the weights by $\frac{1}{\sqrt{n_{in}}}$.

Xavier initialization is often used in layers where the activation function can be linearly approximated near zero like tanh, and sigmoid.

## 2.2 Related Works

Weight sharing is based on the idea that large-scale models, like the Transformer [2], are excessively parameterized. By utilising the same parameters for several calculations, this method makes it easier to separate computation from parameters. Pre-trained language model checkpoints are used by [18] to initialise a sequence-to-sequence model. In an effort to use less memory, they test the use of a shared encoder and decoder. In an attempt to find the best usage of parameters from one layer to another layer, [19] carefully explore techniques for weight sharing across levels. "Sandwich-style" parameter sharing is a method [20] that suggests sharing weights for the centre layers while keeping the first and last layers independent.

[21] proposed an idea of pruning, which is a way of eliminating weights from a neural network that are not required to improve storage and memory performances. In certain instances, it can also preserve model performance while improving computational and temporal economy. Redundancy in the multi-head attention of the Transformer was explained in these papers [22]. To remove attention heads from the Transformer, a first-order method was used in both investigations. Later, [23] presented LayerDrop, an organised dropout method, in 2020. During the Transformer training process, this approach applies random dropout to every layer.

W8A8, or INT8 quantization, may now be applied to all linear layers in transformers without reducing accuracy thanks to recent research [24]. In addition, [24] provide an INT8 inference pipeline and show notable end-to-end (E2E) performance gains over FP16 model inference. The state-of-the-art open-source INT8 implementations found in NVIDIA's FasterTransformer [25] explore aggressive quantization techniques: mode-1 quantizes the attention computation beyond linear layers, and mode-2 further quantizes the residual connection while balancing latency and accuracy. A complete W4A4 encoder inference pipeline supporting various quantization techniques was demonstrated

in [26].

[12] presented a way of training neural networks using an approach called Binaryconnects [27] where weights are constraints between -1 to 1. A unique elastic binary activation function with learnt parameters, a two-set binarization scheme, and a technique to quantize a network to its limit by progressively distilling higher precision models into lower precision models are proposed in [28].

# Chapter 3

# Proposed Methodology

We introduce a novel method to improve the performance of binarized nueral networks. Instead of using the signum function, $\sigma(x)$, with range $\{-1, 1\}$, we propose use of better representative for pre-weights that are to be binarized.

## 3.1 Representation Error

Let there be a map $f : X \to Y$. We can restrict $Y$ to make $f$ surjective without loss of generalization. Since $f$ is a function and every element $x \in X$ has an image $f(x) \in Y$, we know that $|X| \geq |Y|$. If $f$ is bijective, then $|X| = |Y|$, and we can construct an inverse map $f^{-1} : Y \to X$. This means that the information is preserved in bijective map.

When we have a non-bijecive map, i.e. there are elements in $X$ that are mapped to same values in $Y$, we have $|X| > |Y|$. In this case, we are losing information and we would not be able to retrieve the information back once the information is mapped.

### 3.1.1 Nearest Point Representation

We define a *nearest point representation* on a set $C \in \wp(\mathbb{R})$, a map $\mathcal{N}_C : \mathbb{R} \to \mathbb{R}$ as

$$\mathcal{N}_C(x) = \underset{C_i \in C}{\text{argmin}} ||x - C_i||$$

### 3.1.2 Measure for Information Loss

If we fix a dataset $X = \{x_i \mid i \in N\}$, we can define *representation error* as the information lost on applying nearest point representation. We can define it as:

$$\epsilon_r(C) = \sum_{x_i \in X} (x_i - \mathcal{N}_C(x_i))^2$$

Since $X$ is fixed, we have to choose $C$ that minimizes the representation error to preserve the maximum information. We will fix the size of $C$ to be $k$, and hence our optimization problem can be written as:

**Objective**

$$\text{minimize } \epsilon_r(C)$$

**Constraint**

$$C \in \mathbb{R}^k$$

### 3.1.3 Minimizing Representation Error

Let us decompose $X$ into $\{X_i \mid i \in k\}$ with following properties:

1. **Exhaustive**
$$\bigcup_{i=1}^{k} X_i = X$$
2. **Mutually Exclusive**
$$X_i \cap X_j = \varnothing \leftrightarrow i \neq j$$
3. **Representative**
$$\forall x \in X_i, \mathcal{N}_C(x) = C_i$$

We can rewrite representation error as

$$\begin{aligned}\epsilon_r(C) &= \sum_{x_i \in X} (x_i - \mathcal{N}_C(x_i))^2 \\ &= \sum_{C_i \in C} \left( \sum_{x_j \in X_i} (x_j - C_i)^2 \right) \\ &= \sum_{C_i \in C} P(C_i)\end{aligned}$$

where $P(C_i) = \sum_{x_j \in X_i} (x_j - C_i)^2$

The way $X$ is decomposed into $X^D = \{X_i \mid i \in k\}$, $P : X^D \to \mathbb{R}$ maps on independent sets, and hence to optimize $\epsilon_r(C)$, we can optimize $P(C_i) \forall i \in k$ individually. Since our objective is to minimize $P(C_i) \in \mathbb{R}$ and our only independent variable is $C_i \in \mathbb{R}$, we can find the value of $C_i$ as:

$$\begin{aligned}\frac{dP_i}{dC_i} &= 0 \\ \frac{d\left(\sum_{x_j \in X_i} (x_j - C_i)^2\right)}{dC_i} &= 0 \\ \sum_{x_j \in X_i} \left( \frac{d(x_j - C_i)^2}{dC_i} \right) &= 0 \\ \sum_{x_j \in X_i} \left( \frac{d(x_j - C_i)^2}{d(x_j - C_i)} \frac{d(x_j - C_i)}{dC_i} \right) &= 0 \\ \sum_{x_j \in X_i} (2(x_j - C_i) \times -1) &= 0 \\ \sum_{x_j \in X_i} (x_j - C_i) &= 0 \\ \sum_{x_j \in X_i} C_i &= \sum_{x_j \in X_i} x_j \\ C_i \sum_{x_j \in X_i} 1 &= \sum_{x_j \in X_i} x_j \\ C_i \times |X_i| &= \sum_{x_j \in X_i} x_j \\ C_i &= \frac{\sum_{x_j \in X_i} x_j}{|X_i|} = \overline{X_i}\end{aligned}$$

## 3.2 Decomposition into Open Sets

As we can see in previous derivations, we achieve the minimum representation error when we pick representative $C_i$ as the mean of $X_i$.

So far we had fixed the decomposition $X_D$, but now that we know, given $X_D$, to pick $C_i = \overline{X_i}$ for minimum $\epsilon_r(C)$, we can now shift our focus to finding the best decomposition $X_D$ to achieve minimum $\epsilon_r(C)$.

Observe:

- Since $X \in \wp(\mathbb{R})$ and $C \in \mathbb{R}^k$, all the points we are interested in exist on $\mathbb{R}$.
- Since we are using nearest point representative, $\exists! C_i \forall x \in \mathbb{R} \ni \mathcal{N}_C(x) = C_i$.
- Since $\lim_{\delta x \to 0} \mathcal{N}_C(x + \delta x) = \mathcal{N}_C(x)$, we can conclude that all small neighborhood of any point on $\mathbb{R}$ will belong to the same decomposed set.

Following the observation, $\mathbb{R}$ being a metric space, we can conclude that given a $C$, we can decompose $\mathbb{R}$ into disjoint open sets $\{S_i \mid i \in k\}$ that is dense in $\mathbb{R}$. Since $X \in \wp(\mathbb{R}), \forall i \in k \exists! S_i \ni X_i \subseteq S_i$.

## 3.3 Thresholding Tree

We can express the thresholding function in the form of tree where each in-node is a threshold deciding which open set a data point falls in, and the leaf node decides the representation from an open set.

The tree we obtain as a result would be full binary tree. If there are $2^k$ open sets, we will have a tree of height $k$ and $2^k - 1$ nodes in total.

We can store the tree as an array $T =< T_i \mid i \in 2^k >$, where $\forall i \ni i < 2^{k-1}, T_i$ is the parent of $T_{2i+1}$ and $T_{2i+2}$.

Since our task is of just binarizing, we will limit ourselves to 2 open set. Hence our thresholding tree will have 3 nodes, of which 1 will be a threshold $T$, and 2 will be representatives, $\overline{X_1}$ and $\overline{X_2}$ respectively.

## 3.4 Bucketing

Given $C$, decomposing $X$ to $X^D$ would require us to sort $X = \{x_i \mid i \in |X|\}$ and this might be expensive using comparision based sorting algorithm that would take $\mathcal{O}(n \log n)$ time, where $n = |X|$ is usually huge.

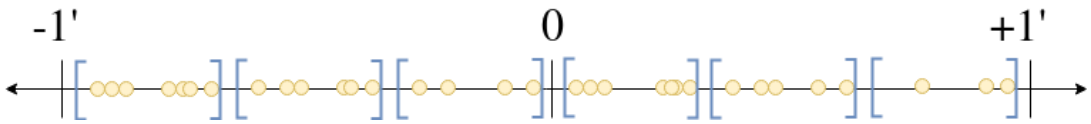


Figure 3.1: Bucketing Data

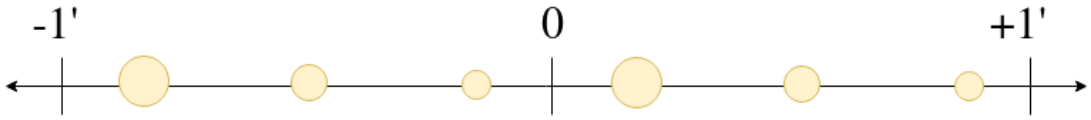


Figure 3.2: Forgetting Actual Value But Tracking Frequency

We will try using linear sorting algorithm like bucket sort, with the only difference is that we will forget the actual value of variables within a bucket. Hence it is a lossy bucket sort. This achieves two things -

1. Sorting $X$ in $\mathcal{O}(n)$ time.

2. Reduce $n$ weights to $k$ data points, where $k \ll n$ and is much faster to work with.

**Algorithm 1** Hash Function that Maps (-lim,lim) to (0,bucket_size) Preserving Order

```
procedure ORDERED_HASH(value,limit,bucket_count)
    return bucket_count × (value + limit) / limit
end procedure
```

**Algorithm 2** Calculates Frequency of Data in Each Bucket

```
procedure BUCKET_DATA(data)
    for i ∈ data do
        bucket[ordered_hash(i)] ← bucket[ordered_hash(i)] +1
    end for
    return bucket
end procedure
```

### 3.4.1 Least Bucket Frequency

Let there be a threshold $T$ that defines two open set on it's left and right. Since these open sets would be dense with data points in $X$ for near representation, $T$ would be in a region with relatively low density.

Hence we can count frequency of data point in each bucket in $\mathcal{O}(n)$ time, and then find the bucket with least frequency in another $\mathcal{O}(n)$ time. The mid-point of this bucket can be chosen as threshold $T$ that defines the two open set.

**Algorithm 3** Finds Threshold Using Least Bucket Frequency

```
procedure LEAST_DENSE_THRESHOLD(data,limit,bucket_size)
    bucket ← bucket_data(data)
    target ← 0
    for f ∈ bucket do
        if bucket[i] < bucket[target] then
            target ← i
        end if
    end for
    return − limit + (1 + target) × bucket_size / 2
end procedure
```

### 3.4.2 Running Sum of Buckets

We can get a more accurate threshold $T$ by taking a running sum of $x_i$ over bucket. The representation error at $i^{th}$ bucket would be:

$$\begin{aligned}\epsilon_r(C) &= P(C_{<i}) + P(C_{i\leq}) \\ &= \sum_{j=0}^{i-1} f_j(x_j - \overline{X}_{<i})^2 + \sum_{j=i}^{n} f_j(x_j - \overline{X}_{i\leq})^2\end{aligned}$$

Since both terms added in last equation are similar, we will simplify only one of them:

$$\begin{aligned}\sum_{j=0}^{i-1} f_j \left(x_j - \overline{X}_{<i}\right)^2 &= \sum_{j=0}^{i-1} f_j \left(x_j^2 - 2x_j\overline{X}_{<i} + \overline{X}_{<i}^2\right) \\ &= \sum_{j=0}^{i-1} f_j x_j^2 - \sum_{j=0}^{i-1} 2f_j x_j \overline{X}_{<i} + \sum_{j=0}^{i-1} f_j \overline{X}_{<i}^2 \\ &= \sum_{j=0}^{i-1} f_j x_j^2 - 2\overline{X}_{<i} \sum_{j=0}^{i-1} f_j x_j + \overline{X}_{<i}^2 \sum_{j=0}^{i-1} f_j\end{aligned}$$

Similarly

$$\begin{aligned}\sum_{j=i}^{n} f_j (x_j - \overline{X}_{i\leq})^2 &= \sum_{j=i}^{n} f_j x_j^2 - 2\overline{X}_{i\leq} \sum_{j=i}^{n} f_j x_j + \overline{X}_{i\leq}^2 \sum_{j=i}^{n} f_j \\ &= \left(\sum_{j=0}^{n} f_j x_j^2 - \sum_{j=0}^{i-1} f_j x_j^2\right) - 2\overline{X}_{i\leq} \left(\sum_{j=0}^{n} f_j x_j - \sum_{j=0}^{i-1} f_j x_j\right) \\ &+ \overline{X}_{i\leq}^2 \left(\sum_{j=0}^{n} f_j - \sum_{j=0}^{i-1} f_j\right)\end{aligned}$$

Let

$$\begin{aligned}T_1 \text{ (Total Sum)} &= \sum_{j=0}^{n} f_j x_j \\ T_2 \text{ (Total Squared Sum)} &= \sum_{j=0}^{n} f_j x_j^2 \\ T_f \text{ (Total Frequency)} &= \sum_{j=0}^{n} f_j \\ S_1 \text{ (Running Sum)} &= \sum_{j=0}^{i-1} f_j x_j \\ S_2 \text{ (Running Squared Sum)} &= \sum_{j=0}^{i-1} f_j x_j^2 \\ S_f \text{ (Running Frequency)} &= \sum_{j=0}^{i-1} f_j\end{aligned}$$

Then we can rewrite both the term as

$$\begin{aligned}\sum_{j=0}^{i-1}\left(x_j-\overline{X}_{<i}\right)^2 &= S_2-2S_1\overline{X}_{<i}+S_f\overline{X}_{<i}^2\\ &= S_2-2S_1\left(\frac{S_1}{S_f}\right)+S_f\left(\frac{S_1}{S_f}\right)^2\\ &= S_2-2\left(\frac{S_1^2}{S_f}\right)+\left(\frac{S_1^2}{S_f}\right)\\ &= S_2-\frac{S_1^2}{S_f}\end{aligned}$$

$$\begin{aligned}\sum_{j=i}^{n}(x_j-\overline{X}_{i\leq}) &= \left(\sum_{j=0}^{n}f_jx_j^2-\sum_{j=0}^{i-1}f_jx_j^2\right)-2\overline{X}_{i\leq}\left(\sum_{j=0}^{n}f_jx_j-\sum_{j=0}^{i-1}f_jx_j\right)\\ &\quad+\overline{X}_{i\leq}^2\left(\sum_{j=0}^{n}f_j-\sum_{j=0}^{i-1}f_j\right)\\ &= (T_2-S_2)-2(T_1-S_1)\overline{X}_{i\leq}+(T_f-S_f)\overline{X}_{i\leq}^2\\ &= T_2-S_2-2(T_1-S_1)\left(\frac{T_1-S_1}{T_f-S_f}\right)+(T_f-S_f)\left(\frac{T_1-S_1}{T_f-S_f}\right)^2\\ &= T_2-S_2-2\frac{(T_1-S_1)^2}{T_f-S_f}+\frac{(T_1-S_1)^2}{T_f-S_f}\\ &= T_2-S_2-\frac{(T_1-S_1)^2}{T_f-S_f}\end{aligned}$$

Hence we can run the representation error at choosing $i^{th}$ bucket for thresholding as

$$\begin{aligned}\epsilon_r(C) &= \sum_{j=0}^{i-1}f_j(x_j-\overline{X}_{<i})^2+\sum_{j=i}^{n}f_j(x_j-\overline{X}_{i\leq})^2\\ &= \left(S_2-\frac{S_1^2}{S_f}\right)+\left(T_2-S_2-\frac{(T_1-S_1)^2}{T_f-S_f}\right)\\ &= T_2-\frac{S_1^2}{S_f}-\frac{(T_1-S_1)^2}{T_f-S_f}\end{aligned}$$

**Algorithm 4** Finding Representation Error At Intermediate Step

**procedure** GET_REPR_ERR($T_1, T_2, T_f, S_1, S_f$)
  **return** $T_2-(S_1^2/S_f)-(T_1-S_1)^2/(T_f-S_f)$
**end procedure**

As we can see, we need to find the total sum and frequency once in $\mathcal{O}(n)$ time while we bucket the data at the same time. Then we go though all the $k$ buckets with running

**Algorithm 5** Find Threshold Using Running Sum and Frequency

```
procedure RUNNING_BUCKET(data,limit,bucket_size)
    bucket ← bucket_data(data)
    target ← 0
    minError ← ∞
    T_1 ← get_total_sum(data)
    T_2 ← get_total_squared_sum(data)
    T_f ← get_total_frequency(data)
    S_1 ← 0
    S_f ← 0
    count ← 0
    for f ∈ bucket do
        bucket_value ← − limit + (1 + count) × bucket_size / 2
        S_1 ← S_1 + f × bucket_value
        S_f ← S_f + f
        curError ← get_repr_err(T_1, T_2, T_f, S_1, S_f)
        if curError < minError then
            target ← i
        end if
        count ← count +1
    end for
    return − limit + (1 + target) × bucket_size / 2
end procedure
```

sum and frequncy to find the threshold that results in minimum representation error in $\mathcal{O}(k)$ time.

Hence we can update the representative in $\mathcal{O}(n + k)$ time, where $n$ is the number of data, and $k$ is the bucket count.

## 3.5 Initialization

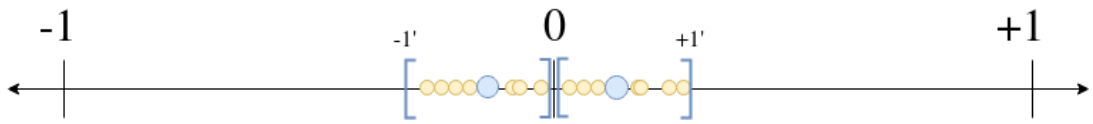


Figure 3.3: Initial State of Threshold Tree

Since at the beginning, the weights are uniformly sampled from the range $(-\frac{1}{\sqrt{d_{in}}}, +\frac{1}{\sqrt{d_{in}}})$, we can pick the initial threshold $T$ as the mean of the weights, which would be 0.

With 0 as our threshold, we will get two open set - $(-\frac{1}{\sqrt{d_{in}}}, 0)$ and $(0, +\frac{1}{\sqrt{d_{in}}})$, whose representative would be their mean, which is $-\frac{1}{2\sqrt{d_{in}}}$ and $+\frac{1}{2\sqrt{d_{in}}}$ respectively.

Hence we start our algorithm with initial thresholding tree as $T =< 0, -\frac{1}{2\sqrt{d_{in}}}, +\frac{1}{2\sqrt{d_{in}}} >$

**Algorithm 6** Constructor for Threshold Tree with Initial State

**procedure** THRESHOLD_TREE_NEW
    repr $\leftarrow \frac{1}{2\sqrt{d_{in}}}$
    **return** $(0, -\text{repr}, \text{repr})$
**end procedure**

# Chapter 4

# Results and Discussion

## 4.1 Experimental Setup

As our baseline, we construct a transformer model with two encoder layers for the text classification task. We use **AG_NEWS** [15] dataset to train the models. Basic quantization (weights only) [16], full quantization (weights and activation) [16], and binarization methods [12] are implemented and tested on the transformer. We also explored the sensitivity of different parts of the transformer. We designed a unique pattern for transformer quantization that only quantizes Query and Key without Value, based on the observation that we only care about the similarity between query and key instead of absolute values.

For all models in all experiments, we maintained the same training setting and trained for ten epochs for comparability. We used **Cross Entropy Loss** as a loss function to calculate the loss. Each model took around 3-4 hours to train. The models were trained using Adam optimizer, with a learning rate of 0.0001 and a batch size of 32. We used MultiStepLR as a learning rate scheduler with a milestone of 10-15 epochs. Hugging face library is used for tokenization using BERT [17]. The code is written in Cuda and Python with the help of the PyTorch framework. Models were trained on RTX3060 GPU with RAM 16G and VRAM 6G.

The evaluation metrics used are:

- **Accuracy**: Measures how often a model predicts a correct outcome. Higher

accuracy indicates better performance.

- **Model Size**: Amount of storage obtained by multiplying the number of parameters with the precision bytes. Lower model size means reduced memory footprint and potentially faster inference times.

- **Reduction Ratio**: The base model size is divided by the compressed model size. The higher the ratio, the better the compression.

$$\text{Reduction Ratio} = \left( \frac{\text{Original size} - \text{Compressed size}}{\text{Original size}} \right) \times 100\%$$

## 4.2 Results Analysis

We designed three experiments to explore quantization patterns for transformers.

**In our first experiment**, we implement and modify diverse quantization methods and basic binarization to see how it performs on the transformer.

**Our second experiment** focuses on the sensitivity of different transformer parts. We leave out the input embedding layer and quantize other parts of the transformer, respectively, to see the sensitivity of each part. Then, based on our observation, we designed two improvements to accommodate the transformer's specialty.

**In our third experiment**, we compared our method with baseline, 4-bit Quantized (weights only), and basic binarized model. We compare the accuracy, model size, and reduction ratio and highlight the best.

### 4.2.1 Experiment 1

In our first experiment, we implement and modify diverse quantization methods to see how they work for transformers. On the left is the training curve plotted every 100 iterations. On the right side is the test set performance for every epoch. On the bottom is the final performance and model size table. First, we plotted for quantizations and then binarization. Here are the results of our first experiment.

**Quantization:**

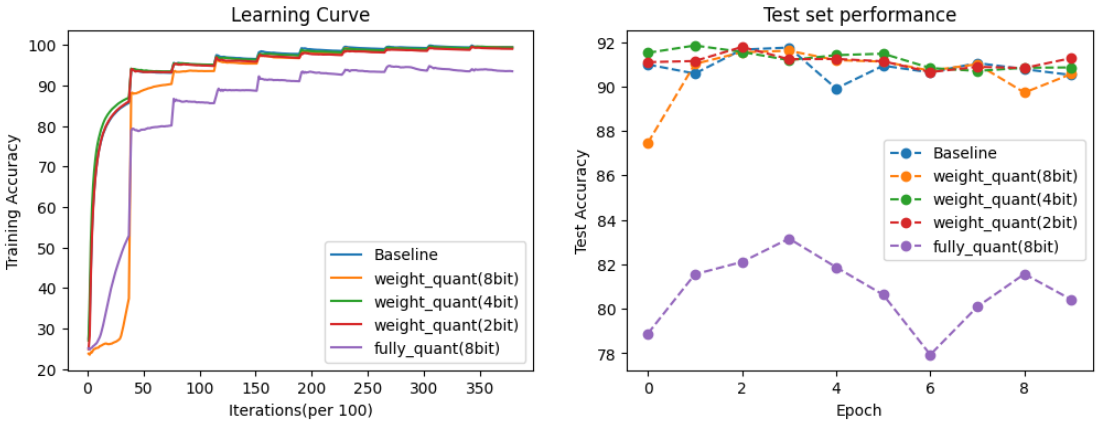


Figure 4.1: Train & and Test set performance (Quantization)

| Model | Accuracy | Model Size | Reduction Ratio |
|---|---|---|---|
| baseline | 91.8 | 78.7M | - |
| 8-bit quant | 91.6 | 38.6M | 50.95 % |
| 4-bit quant | 91.8 | 19.7M | 74.96 % |
| 2-bit quant | 91.8 | 10.2M | 87.03 % |
| fully quant | 83.2 | 38.6M | 50.95 % |

Table 4.1: Quantization methods

**Binarization:**

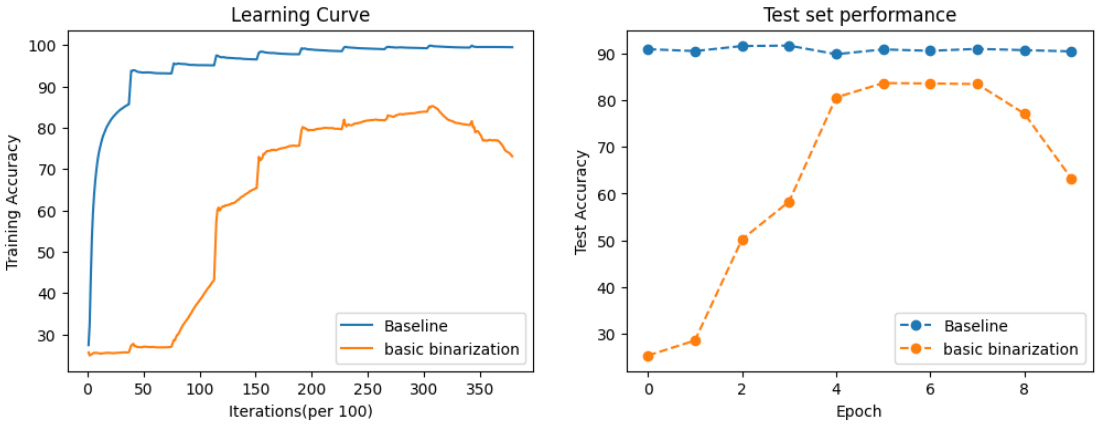


Figure 4.2: Train and Test set performance (Binarization)

| Model | Accuracy | Model Size | Reduction Ratio |
|---|---|---|---|
| baseline | **91.8** | 78.7M | - |
| basic binarization | 83.7 | 4.7M | 94.03 % |

Table 4.2: Binarization method

**Observation:** For quantization, we can see that basic quantization, which quantizes weight only, performs very well; it reduces model size a lot with nearly no sacrifice of accuracy. To continue, we develop a model based on a full quantization method, which quantizes both weight and activation. This model has a large runtime size reduction but suffers from a performance drop, indicating that activation is more critical for the transformer. We will design different methods to improve later.

The model size is significantly reduced for binarization, but a significant drop in performance can also be observed, resulting from minimal model representation ability.

### 4.2.2 Experiment 2

As we saw from experiment 1, Basic Binarization provides a lower accuracy. So, if the whole model is binarized, it cannot learn anything. We will solve this problem with the following two improvements. We explore the sensitivity of different parts of transformers.

**Improvement 1 - Final linear layer is crucial**

After some analysis, we found that the final linear layer in the classifier is crucial. We trained two models, one with all layers binarized and one with all but the final layer binarized. The results are shown below:

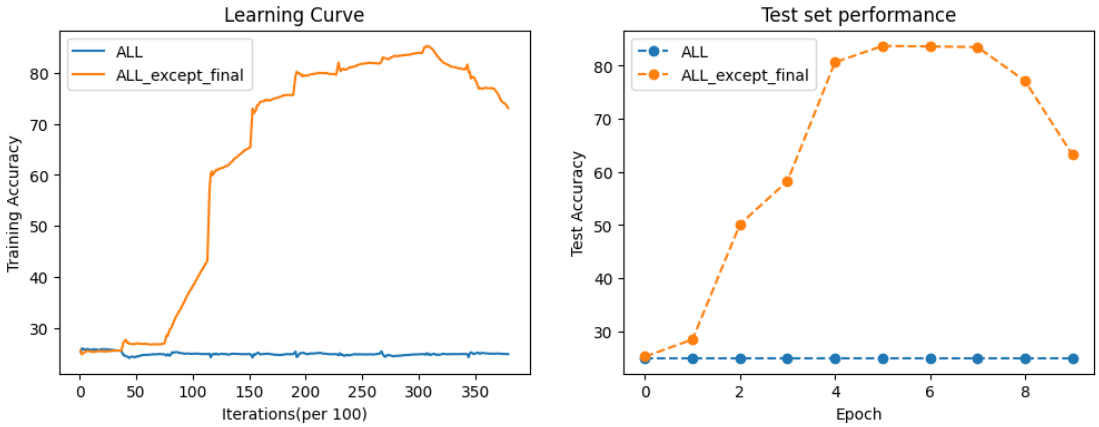


Figure 4.3: Train and Test set performance (W & W/O Final Layer)

| Model | Accuracy | Model Size | Reduction Ratio |
|---|---|---|---|
| all | 25.0 | 4.7M | 94.03 % |
| all except final | **83.7** | 4.7M | 94.03 % |

Table 4.3: Performance W and W/O Final Layer

If the final layer remains complete precision, the whole model performance will increase to normal because the final layer controls the output scores for each class.

**Improvement 2 - Treat Q,K,V in attention layer differently**

Then, as we discussed before, query and key should be more robust to binarization based on the intuition that similar items should still be similar even if binarized. We compare the model with QKV, all binarized, and that with only QK binarized. This ensures that the improvement does not come from incrementing full precision parameters. We also trained the model with only QV and KV binarized. The results show that the QK model only performs significantly better than others, proving our assumption. The results are shown below:

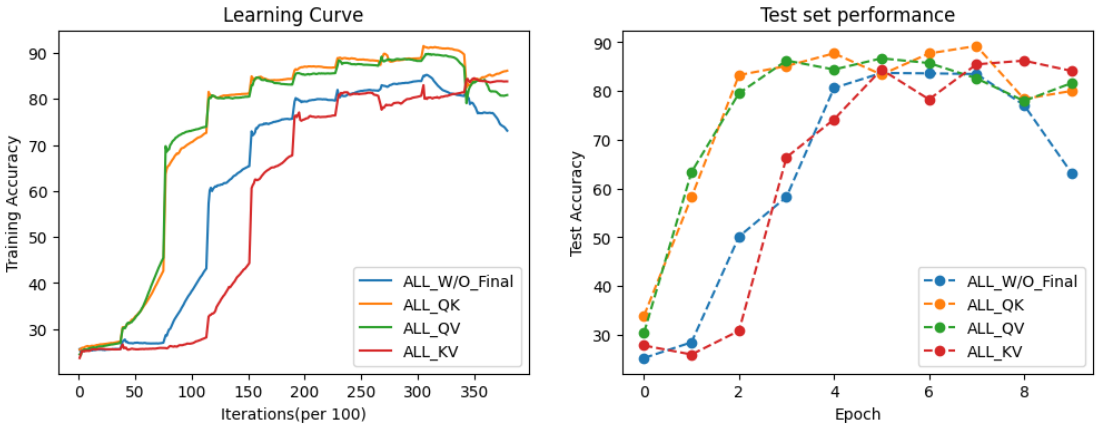


Figure 4.4: Train & Test set performance (Binarized QK vs QKV)

| Model | Accuracy | Model Size | Reduction Ratio |
|---|---|---|---|
| all except final | 83.7 | 4.7M | 94.03 % |
| all (QK) except final | **89.3** | 22.8M | 71.03 % |
| all (QV) except final | 86.7 | 22.8M | 71.03 % |
| all (KV) except final | 86.2 | 22.8M | 71.03 % |

Table 4.4: Effect of Binarized QK vs QKV

### 4.2.3 Experiment 3

Our third experiment compared our method with baseline, 4-bit Quantized (weights only), and basic binarized model. We compare the accuracy, model size, and reduction ratio and highlight the best. The results are shown below:

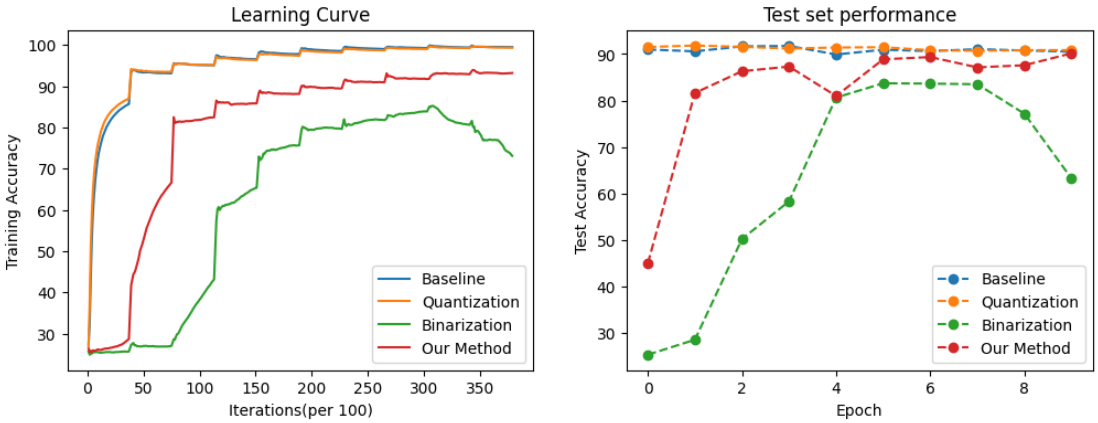


Figure 4.5: Train & Test set performance (Our Method vs Existing)

| Model | Accuracy | Model Size | Reduction Ratio |
|---|---|---|---|
| baseline | **91.8** | 78.7M | - |
| Quantization(4 bit) | **91.8** | 19.7M | 74.96 % |
| Binarization | 83.7 | 4.7M | 94.03 % |
| Our method | **90.1** | **4.7M** | **94.03 %** |

Table 4.5: Our Method vs Existing Optimization Methods

Our method accelerates convergence and has an essential binarized model size. So, we have a better accuracy and reduction ratio than the modified binarized method. Note that an essential advantage of our method is that we no longer need pre-trained weights, which may accelerate the whole training process.

# Chapter 5

# Conclusion and Future work

## 5.1 Conclusion

We have been able to achieve a compromise between baseline, that works with full precision weights achieving highest accuracy, and BNN, that makes computations in LLM fast and efficient on low edge device, but at the cost of losing of accuracy. Our proposed work is as compressed as BNN, where each weight is represented by 1 bit, and yet, it achieves an accuracy close to the baseline.

### 5.1.1 Drawbacks

- It doesn't make computation of matrix multiplication fast as one could do in BNN using bit manipulation.
- It can't make use of tricks like XNORnet that accelerates layer on FPGA .
- It doesn't reach the the peak performance as fast as a full precision transformer.

### 5.1.2 Application

Despite it's drawback, it opens window for many opportunities. Now that we are able to compress data to it's limit without loss of accuracy, it makes data movement easier. This makes even more areas feasible:

**Internet of Things** Each device in an IoT would be able to store the weights, and at worse would need very minimal amount of streaming of data to make an inference on a neural network task.

**Edge Computing** Data needed for inference being compressed makes it possible to move it close to the edge. Data can be cached at different level of proximity of end users using edge devices that makes inference much faster than what a centralized computing system can offer.

**Distributed Network** Compressed data would consume less bandwidth, and with the power of being able to cache them more and making them feasible on low end devices, one can make storing weights and inferring on LLM totally on a distributed system.

## 5.2 Future Work

**Acceleration on FPGA** Once one has found the representation map, one can fix those value of representation on multipliers or have a small look up table close to processing unit for fast access.

**Extending to $n$-bit** One can extend the idea of dividing the weights into two open set and choosing a prepresentative to form $2^n$ open sets and have a hierarchy of open set that can be stored as a tree where each in-nodes would be a threshold and leaf node would be the representative. It can serve as an alternative to $n$-bit quantization.

**Memory Management** One can use custom allocators for efficient use of memory. One can also focus on order of access of elements in matrices to exploit improvement from burst transfer and better caching due to principle of locality.

**Transport Layer Protocol** One can classify the data needed into static, streaming and dynamic class, and have separate protocols best suiting the need and performance for each one of them.

# Bibliography


[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[2] Katikapalli Subramanyam Kalyan, Ajit Rajasekharan, and Sivanesan Sangeetha. Ammus: A survey of transformer-based pretrained models in natural language processing. *arXiv preprint arXiv:2108.05542*, 2021.

[3] Partha Pratim Ray. Chatgpt: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope. *Internet of Things and Cyber-Physical Systems*, 2023.

[4] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. Towards efficient generative large language model serving: A survey from algorithms to systems. *arXiv preprint arXiv:2312.15234*, 2023.

[5] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on machine learning*, pages 5958–5968. PMLR, 2020.

[6] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *AI open*, 3:111–132, 2022.

[7] Suraj Srinivas and R Venkatesh Babu. Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149*, 2015.

[8] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[9] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.

[10] Shiyao Li, Xuefei Ning, Luning Wang, Tengxuan Liu, Xiangsheng Shi, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. Evaluating quantized large language models. *arXiv preprint arXiv:2402.18158*, 2024.

[11] Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. Llm-qat: Data-free quantization aware training for large language models. *arXiv preprint arXiv:2305.17888*, 2023.

[12] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29, 2016.

[13] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, pages 177–186. Springer, 2010.

[14] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[15] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *NIPS*, 2015.

[16] 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. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Sascha Rothe, Shashi Narayan, and Aliaksei Severyn. Leveraging pre-trained checkpoints for sequence generation tasks. *Transactions of the Association for Computational Linguistics*, 8:264–280, 2020.

[19] Sho Takase and Shun Kiyono. Lessons on parameter sharing across layers in transformers. *arXiv preprint arXiv:2104.06022*, 2021.

[20] Machel Reid, Edison Marrese-Taylor, and Yutaka Matsuo. Subformer: Exploring weight sharing for parameter efficiency in generative transformers. *arXiv preprint arXiv:2101.00234*, 2021.

[21] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

[22] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. *arXiv preprint arXiv:1905.09418*, 2019.

[23] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

[24] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[25] NVIDIA. Faster transformer. *https://github.com/NVIDIA/FasterTransformer*, 2023.

[26] Xiaoxia Wu, Cheng Li, Reza Yazdani Aminabadi, Zhewei Yao, and Yuxiong He. Understanding int4 quantization for language models: latency speedup, composability, and failure cases. In *International Conference on Machine Learning*, pages 37524–37539. PMLR, 2023.

[27] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems*, 28, 2015.

[28] Zechun Liu, Barlas Oguz, Aasish Pappu, Lin Xiao, Scott Yih, Meng Li, Raghuraman Krishnamoorthi, and Yashar Mehdad. Bit: Robustly binarized multi-distilled transformer. *Advances in neural information processing systems*, 35:14303–14316, 2022.

# Chapter 6

# Publication Details

Conference Name : 4th International Conference on AI Research(ICAIR)

Venue : Lisbon, Portugal

| | |
|---|---|
| Abstract submission deadline | 15 May 2024 |
| Notification of abstract acceptance | 29 May 2024 |
| Full paper due for review | 04 July 2024 |
| Notification of paper acceptance (with any requested changes) | 12 September 2024 |
| Earlybird registration closes | 26 September 2024 |
| Final paper due (with any changes) | 10 October 2024 |
| Final Author registration date | 31 October 2024 |

Figure 6.1: Timeline

# Chapter 7

# Plagiarism Report

FYP

ORIGINALITY REPORT

| 13% | 10% | 8% | % |
|---|---|---|---|
| SIMILARITY INDEX | INTERNET SOURCES | PUBLICATIONS | STUDENT PAPERS |

PRIMARY SOURCES

| | | |
|---|---|---|
| 1 | arxiv.org<br>Internet Source | 1% |
| 2 | www.coursehero.com<br>Internet Source | 1% |
| 3 | "ECAI 2020", IOS Press, 2020<br>Publication | 1% |
| 4 | "Beyond AI", Springer Science and Business Media LLC, 2023<br>Publication | 1% |
| 5 | Ramisa Alam, Sazan Mahbub, Md. Shamsuzzoha Bayzid. "Pair-EGRET: enhancing the prediction of protein-protein interaction sites through graph attention networks and protein language models", Cold Spring Harbor Laboratory, 2023<br>Publication | <1% |
| 6 | dokumen.pub<br>Internet Source | <1% |
| 7 | ru.overleaf.com<br>Internet Source | <1% |