Quantization, Sparsity, Reliability, and Their Interactions

A THESIS PRESENTED
BY
Joshua H. Park
TO
The Department of Computer Science and Mathematics

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Bachelor of Arts
IN THE SUBJECT OF
Computer Science and Mathematics

Harvard University
Cambridge, Massachusetts
May 2025

Abstract

In this thesis, we will explore and contribute to three seemingly disjoint areas of machine learning: quantization, sparsity, and reliability. While each of these areas are well-understood individually, there is very little existing literature on the interaction of these research areas. They are typically viewed as orthogonal to each other and solved independently. In this thesis, we would like to shed light on the fact that many practical systems employ elements from all three fields. Therefore, it is important to understand how they can influence and interact with each other.

This thesis makes the following three key contributions to the existing literature. First, it presents GoldenEye, a functional simulator for modeling fault injections into machine learning models. In particular, it targets models that use novel number formats to quantize or emulate to different data types. The flexible API design makes it easy to add new number formats as the area evolves. Furthermore, we propose a mathematical framework for using reliability analysis to inform quantization. Second, we present our work on EdgeBERT, an accelerator for accelerating transformer inference. EdgeBERT features both quantization and sparsity optimizations that together achieve strong speedups when profiled on true silicon. We also write compiler code that allows other transformer workloads to be compiled to EdgeBERT. Finally, we attempt to understand the problem of sequential applications of quantization and sparsity. We prove a theoretical result that shows that at the tensor-level sparsity before quantization is preferred over quantization before sparsity. However, we show that at the model level, the order is not too important, so long as the algorithms are "synergetic". Then, we propose a novel quantization-aware sparsity algorithm that consider the problem of sparsity when the weights are already quantized. Together, we believe that these contributions provide valuable insights to understanding the interactions between these distinct problems.

Contents

To Mom and Dad

Acknowledgments

I would be remiss if I did not begin by expressing my gratitude to the many individuals who made this thesis possible. First and foremost, I would like to thank my advisors, David Brooks and Gu-Yeon Wei. Their expertise, research vision, and patience have been invaluable, and I am truly fortunate to have benefited from their guidance over the past four years. Their support and encouragement have profoundly influenced my research interests, and I will always carry their many lessons with me.

I would also like to extend my deepest gratitude to the many mentors and collaborators who have guided me throughout my research journey. First, I am especially indebted to Abdulrahman Mahmoud for his mentorship since my first year at Harvard. His thoughtful guidance, endless wisdom, and genuine care have profoundly shaped me as both a student and a researcher. For all of that, I will be forever grateful. I would also like to thank Thierry Tambe and David Kong for their invaluable collaboration, insightful feedback, and constant encouragement. It is the work of all three that helped form the basis for this thesis.

I would also like to thank Eddie Kohler, Stephen Chong, and Joe Blitzstein. Likely unbeknownst to them, they introduced me to many of the topics that continue to shape my research interests and have left an indelible mark on my life. Without their passion for teaching, I would not have discovered the direction that ultimately guided my work in this thesis.

Throughout my time at Harvard, I have been fortunate to meet some incredible friends. Thank you to Gabriel "Steve" Sun, Simon "Quincy" Sun, and William "Ness" Shi for the uncountable number of late-night Super Smash Bros games. As you know all too well, I never wanted to end on a loss, so thank you for humoring me when I asked for "just one more game". I would also like to thank Brian Ham for the endless stream of memes and Legos. Finally, I wanted to thank Robin Pan for her unwavering support and helping to push me over the finish line.

Finally, I would like to thank my parents and brother for their love and support. My parents instilled in me a lifelong love of learning that formed the backbone of this thesis. Words cannot fully express my gratitude.

Part I

Introduction

0.1 Overview

This thesis will explore three topics related to machine learning and software-hardware co-design: quantization, sparsity, and reliability. The underlying thread between all of these methods is that they are inspired by hardware correctness and efficiency. Quantization attempts to represent machine learning weights and activations in lower precision to reduce the memory and compute footprint of models. Sparsity attempts to "prune" unimportant weights to also reduce the necessary data that needs to be loaded and simplify computation. Finally, reliability attempts to model hardware transient errors that may occur and their effect on downstream applications. While all of these are hardware-inspired, they can also be modeled quite rigorously from a mathematical perspective. Thus, in this thesis, we take a two-pronged approach to these problems. We want to analyze these three topics from both a rigorous mathematical perspective, as well as a more practical empirical perspective. We hope that this thesis can target both types of audiences.

In the current literature, these three topics are treated as orthogonal to each other. However, in this thesis, we will present work that focuses on their interactions. In particular, the central claim of our thesis is that these topics are actually much more intertwined than the current literature states. In particular, we will propose that reliability analysis can be used to inform quantization and sparsity analysis (Chapter 8). Similarly, we will show that sparsity and quantization schemes should be chosen in tandem and not viewed in isolation (Chapter 10). Finally, we will show that we can combine all three of these ideas together to accelerate hardware in practice and show immense speedups in inference routines (Chapter 9).

0.2 Background

The thesis is structured as follows. First, we attempt to provide necessary preliminary background on machine learning models and computer architecture (Chapter 1 and Chapter 2). The focus of this thesis will be on two common machine learning architectures: convolutional neural networks (CNNs) (Chapter 3) and transformers (Chapter 4). In these chapters, we will provide an in-depth mathematical formulation of these models, as well as identify key optimizations used to speed up these routines empirically. Then, the following three chapters provide a detailed literature of review of the three central topics of this thesis: quantization (Chapter 5), sparsity (Chapter 6), and reliability (Chapter 7). For each, we motivate the topic with the hardware implications. Then, we provide a complete survey of the current literature in these fields. In these sections, we discuss both the mathematical

framework that form the foundation for each body of work, as well as implementation and hardware considerations. Although these three chapters are mainly exposition, we provide a very formal presentation of each problem that is not seen in the literature. This allows us to better understand the various algorithms that have been proposed in this domain from a more general perspective. To our knowledge, these problems have not been formalized in such a way previously.

0.3 Our Contributions

The rest of the thesis will discuss novel research that contributes to our understanding of the intersection of these problems. We will both present our joint contributions to the existing body of research done in collaboration with other researchers and individual contributions.

First, we present GoldenEye (Chapter 8), a reliability simulator for various machine learning models that models error injections and different novel number formats. GoldenEye highlights the intersection of quantization and reliability. In particular, it attempts to expand current reliability research by giving an easy-to-use framework for simulating emulation and quantization of a wide array of data types. In this section, we will build a tool that allows us to study how quantization impacts reliability. Furthermore, we will propose a mathematically sound reliability-aware quantization framework that shows reliability can also be used to influence quantization frameworks, drawing on ideas from information theory.

Then, we detail our work on EdgeBERT (Chapter 9), a transformer accelerator that employs both quantization and sparsity optimizations. Furthermore, EdgeBERT uses reliability analysis to inform its decisions on memory design. EdgeBERT is a true practical example that these three topics have strong interactions and can be used to design more efficient hardware. In particular, EdgeBERT uses a novel floating-point quantization strategy, allowing it to reduce its memory footprint, while still maintaining high expressivity. Then, sparsity is compounded to further compress the model and help with efficient data movement. Finally, reliability is used to better understand the effects of choosing different types of memory and their impact on end-to-end analysis. In this chapter, we will present compiler code that allows machine learning models to be efficiently "compiled" to EdgeBERT to be run for fast inference. We will show that EdgeBERT provides strong end-to-end speedups of 64.1× and 2.12×, when compared to CPU and systolic array implementations respectively.

Finally, we discuss the interactions of quantization and sparsity (Chapter 10). Frequently, these algorithms are applied sequentially to achieve even more model compression. However, we claim that these operations should not be applied blindly. In fact, they have strong interactions with each other that can lead to serious accuracy degradations. Thus, we will try to

understand the effect that the choice of quantization algorithm has on the sparsity algorithm and vice versa. Current research in this field is quite limited, and it does not consider very sophisticated quantization algorithms like AWQ [LTT+24] and GPTQ [FAHA23]. Therefore, we provide a novel theoretical framework for understanding the compounding errors from applying these methods sequentially. Furthermore, we prove a very strong theoretical result, that extends on work from [HCK+25] that shows the most effective ordering of operations. Then, with this analysis, we propose a quantization-aware sparsity scheme that performs well in environments where the weights have already been quantized.

1

Machine Learning Preliminaries

In this chapter, we discuss preliminary background on machine learning models that will form the foundations for this thesis. While these topics are quite broad and it would be hard to give a complete overview of each, we will try to include the most pertinent background for this thesis. In recent years, machine learning models have exploded in popularity, particularly in the fields of computer vision and natural language processing. First, we provide a general mathematical framework for machine learning models using the notation from [HABN+21].

1.1 General Formulation

This treatment is of the most general form, and we will later define notation more explicitly in future chapters when discussing specific models. In generality, we are concerned with $f : \mathcal{X} \to \mathcal{Y}$, a model which is parameterized by weights $\mathbf{w} \in \mathbb{R}^d$. This is a general representation of a machine learning model that takes some input $\mathbf{x} \in \mathcal{X}$ and outputs $\mathbf{y} = f(\mathbf{x}; \mathbf{w})$, where $\mathbf{y} \in \mathcal{Y}$. $\mathbf{x}$ is typically called the covariates and $\mathbf{y}$ is called the output or response. Given N

input vectors $\mathbf{x}_i \in \mathbb{R}^p$, we can "batch" them to create a *design matrix* defined by

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\mathsf{T} \\ \mathbf{x}_2^\mathsf{T} \\ \vdots \\ \mathbf{x}_N^\mathsf{T} \end{bmatrix} \in \mathbb{R}^{N \times p}.$$

When a matrix is stored like this in memory, this is often called row-major layout, where each row is a data point.

In this thesis, we focus on the supervised learning scenario, where we are given associated outputs. When training a machine learning model, our goal is to learn the weights $\mathbf{w}$. In an effort to find the optimal $\mathbf{w}$, we first construct a loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, which characterizes the distance between the output of the model and the true output.

The loss function is for one specific observation. Now, we want to generalize this to a full data distribution. Assume that the data $(\mathbf{x}, \mathbf{y})$ comes from some distribution $\mathcal{F}$. Then, our goal is to choose $\mathbf{w}$ such that $\mathbb{E}_{\mathcal{F}}\left[\mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right)\right]$ is minimized. This quantity is the expected loss taken over the entire distribution $\mathcal{F}$, not just one single data point, and it is commonly referred to as the generalization error. Thus, our objective function $L : \mathbb{R}^d \to \mathbb{R}$ is defined as $L(\mathbf{w}) = \mathbb{E}_{\mathcal{F}}\left[\mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right)\right]$. Our goal is to choose $\mathbf{w}$ that minimizes L. The process of finding this optimal $\mathbf{w}$ is referred to as training.

To train our model, we are given some training data $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$, which consists of N ordered pairs of inputs and outputs. For the purposes of this thesis, we will assume that $\mathcal{D} \sim \mathcal{F}$. In other words, the training data is drawn from the same distribution as the true data. Note this is a simplifying assumption that may not always be true. The training data forms its own empirical training data distribution, $\hat{\mathcal{F}}$, and we assume that this empirical distribution closely approximates the true distribution $\mathcal{F}$.

Now, we should note that our current objective function L is not computable, since it is the expectation taken over $\mathcal{F}$. However, $\mathcal{F}$, the true data distribution, is unknown. However, under the assumption that the training data distribution follows the true distribution, we can make an approximate objective function by using the sample average loss. While the sample average loss is merely an estimate of the true average loss, it will be consistent (converges in probability to the true value) under some mild regularity conditions that we will assume holds true about $\mathcal{L}$. This is a consequence of the Central Limit Theorem (CLT). Thus, our final objective function becomes

$$L\left(\mathbf{w}\right) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right),$$

where N is the sample size of the data, $|\mathcal{D}|$.

In this thesis, we are not as interested in training, but more interested in inference. Inference refers to the process where we assume that we have a well-trained model with optimal weights $\hat{\mathbf{w}}$. Now, given some unseen input $\mathbf{x}$, we want to predict its output. To predict the output, we can simply use $f(\mathbf{x}; \hat{\mathbf{w}})$, or the output of our model using the trained weights $\hat{\mathbf{w}}$. Thus, in an inference setting, we are primarily concerned with executing the model quickly, rather than finding some optimum.

When modeling a real-world scenario, we are interested in learning the data generating process (DGP). Broadly, this is how the data that we observe is produced. More specifically, it refers to the structure of $\mathcal{F}$. Now, we discuss two flavors of machine learning models: deterministic and probabilistic formulations. These model varieties allow us to model different DGPs under various assumptions. There does exist a third Bayesian approach to machine learning, which assigns prior distributions to the weights $\mathbf{w}$ but it is beyond the scope of this thesis.

1.2 Deterministic Formulation

As its name suggests, a deterministic model assumes that conditioned on some input, the output is deterministic. In other words, given some input covariates, there is one correct "label" or "answer" outputted by the model. Note that this is a significant assumption, and it typically does not hold for the true data distribution. Even if the real-world process is purely deterministic, one can imagine that there is likely some amount of noise in the data collection process that leads to some variance in the output given the same input. However, this type of model greatly simplifies the necessary model specifications, since we do not need to describe the distribution of outputs conditional on inputs. More specifically, given an input $\mathbf{x}$, we need not describe the distribution of the output $\mathbf{y}$ given $\mathbf{x}$, since there is one single "correct" value.

We typically take our loss function to be $\mathcal{L}(f(\mathbf{x}; \mathbf{w}), \mathbf{y}) = d(f(\mathbf{x}; \mathbf{w}), \mathbf{y})$, where $\mathbf{y}$ is the true output, $\mathbf{x}$ is the associated input, and d is some distance function in $\mathcal{Y}$. Sometimes, we refer to use $\hat{\mathbf{y}}$ to refer to $f(\mathbf{x}; \mathbf{w})$, the predicted output of $\mathbf{x}$ given weights $\mathbf{w}$. For $\mathcal{Y} = \mathbb{R}^m$, d is typically the Euclidean distance metric. As mentioned above, under the assumption that the training data distribution follows $\mathcal{F}$, we can make our objective function the sample average loss. Thus, our objective function becomes

$$L(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} d(f(\mathbf{x}_i; \mathbf{w}), \mathbf{y}_i).$$

When $\mathcal{Y} = \mathbb{R}$, this is typically called the mean-squared error (MSE), since the objective function takes the form of

$$L(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \left(f(\mathbf{x}_i; \mathbf{w}) - \mathbf{y}_i \right)^2 .$$

1.3 Probabilistic Formulation

Given that the deterministic assumption is quite strong, we can relax this assumption with probabilistic models. Probabilistic models attempt to model the DGP by imposing some structure on $\mathcal{F}$. There are two types that differ in their level of granularity when describing $\mathcal{F}$: discriminative and generative models. Before discussing the specifics of each model type, we will introduce the idea of Kullback-Leibler (KL) divergence, which will allow us to motivate a loss function for these probabilistic models.

1.3.1 Kullback-Leibler (KL) Divergence and Cross-Entropy Loss

In this section, we will motivate and present a seemingly different objective function that we will use for training probabilistic models. However, we will show that this alternative objective function coincides exactly with the framework that we have already constructed.

In both discriminative and generative models, the goal is to learn some true distribution $\mathcal{Q}$. We will train some model to learn some approximate distribution $\mathcal{P}$ that is parameterized by the weights $\mathbf{w}$. To construct an objective function, we are interested in the "distance" between our approximate distribution and the true distribution. In other words, we want to know how well $\mathcal{P}$ approximates $\mathcal{Q}$. Thus, we introduce KL divergence, which allows us to measure such "distance".* We define the KL divergence as follows. For some true distribution $\mathcal{Q}$ and some proposed distribution $\mathcal{P}$, we define the KL divergence as

$$D_{\mathrm{KL}}(\mathcal{Q}||\mathcal{P}) = \mathbb{E}_{z \sim \mathcal{Q}} \left[\log \left(\frac{\mathcal{Q}(z)}{\mathcal{P}(z)} \right) \right] .$$

The KL divergence can be interpreted as the expected logarithmic difference between distributions $\mathcal{P}$ and $\mathcal{Q}$ assuming that $\mathcal{Q}$ is the true model.

KL divergence is intimately tied to the field of information theory. Decomposing the KL divergence using properties of logarithms and linearity of expectation allows us to see this

*Note that KL divergence is not a distance metric in the most rigorous sense because it is not symmetric nor does the triangle inequality hold. There exists probability distributions $\mathcal{P}$ and $\mathcal{Q}$ such that $D_{\mathrm{KL}}(\mathcal{Q}||\mathcal{P}) \neq D_{\mathrm{KL}}(\mathcal{P}||\mathcal{Q})$. However, we will still refer to it as an intuitive "distance" metric.

connection and provide some useful intuition.

$$\begin{aligned} D_{\mathrm{KL}}\left(\mathcal{Q}||\mathcal{P}\right) &= \mathbb{E}_{z\sim\mathcal{Q}}\left[\log\left(\frac{\mathcal{Q}(z)}{\mathcal{P}(z)}\right)\right] \\ &= \mathbb{E}_{z\sim\mathcal{Q}}\left[\log\left(\mathcal{Q}(z)\right) - \log\left(\mathcal{P}(z)\right)\right] \\ &= \mathbb{E}_{z\sim\mathcal{Q}}\left[\log\left(\mathcal{Q}(z)\right)\right] - \mathbb{E}_{z\sim\mathcal{Q}}\left[\log\left(\mathcal{P}(z)\right)\right] \end{aligned}$$

In information theoretic contexts, the term $\mathbb{E}_{z\sim\mathcal{Q}}\left[\log\left(\mathcal{Q}(z)\right)\right]$ is referred to as Shannon's entropy [Sha48] or just entropy. Intuitively, this terms measures the amount of "information" needed to describe the randomness of a random variable with distribution $\mathcal{Q}$.[†] $\mathbb{E}_{z\sim\mathcal{Q}}\left[\log\left(\mathcal{P}(z)\right)\right]$, the second term, is called the cross entropy of $\mathcal{P}$ relative to $\mathcal{Q}$. Broadly speaking, it measures the amount of bits needed to encode $\mathcal{P}$ using an encoding scheme that is optimized for $\mathcal{Q}$. We will revisit these ideas when we discuss minimum description length (MDL) theory [Ris78] in Chapter 8.

Now, we return to our goal of constructing an objective function. Recall that the goal is to find the optimal weights $\mathbf{w}$ that parameterize the distribution $\mathcal{P}$ which minimizes $D_{\mathrm{KL}}\left(\mathcal{Q}||\mathcal{P}\right)$, since KL divergence is our "distance" metric. From this decomposition, we can see that regardless of $\mathcal{P}$, we have that $\mathbb{E}_{z\sim\mathcal{Q}}\left[\log\left(\mathcal{Q}(z)\right)\right]$ will be constant. To that end, choosing weights $\mathbf{w}$ that minimize $D_{\mathrm{KL}}\left(\mathcal{Q}||\mathcal{P}\right)$ is equivalent to choosing weights $\mathbf{w}$ that minimize $\mathbb{E}_{z\sim\mathcal{Q}}\left[-\log\left(\mathcal{P}(z)\right)\right]$. Thus, we want to use something of this form as our loss function.

Similar to our previous setup, we cannot directly compute $\mathbb{E}_{z\sim\mathcal{Q}}\left[-\log\left(\mathcal{P}(z)\right)\right]$ because the true distribution $\mathcal{Q}$ is unknown. However, using the assumption that the training data distribution follows $\mathcal{F}$, we can approximate this value by using the sample mean. We have that the "approximate" distribution $\mathcal{P}$ will be parameterized by the weights $\mathbf{w}$. That is, we can write $-\log\left(\mathcal{P}(z)\right)$ as $-\log\left(p(z|\mathbf{w})\right)$. Thus, we have that our objective function becomes

$$L\left(\mathbf{w}\right) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(p(z_i|\mathbf{w})\right).$$

$-\log\left(p(z_i|\mathbf{w})\right)$ is referred to as the negative log-likelihood (NLL), since it is the likelihood of observing z_i given the weights $\mathbf{w}$.

As we alluded to previously, this newly constructed objective function aligns nicely with the previous framework. In particular, we are taking an average over some loss function.

[†]Note that for the purposes of this thesis, log will refer to the natural logarithm. However, in information theory, $\log_2$ is typically used to represent the notion of "bits". We have that $\log(x) = \log(2)\log_2(x)$ by the change of base formula, so these quantities are equal up to a multiplicative constant.

Here, the loss function is the NLL. However, this framework is slightly more general because the "loss" function that we are taking the average over is not restrained to only be a function of the outputs. In particular, we have not specified the distribution we are modelling. In other words, we have not specified what z_i is. This will serve as the main distinction between the discriminative and generative models.

1.3.2 Probabilistic Discriminative Model

Probabilistic discriminative models attempt to model the conditional distribution of $\mathbf{y}$ conditioned on the input $\mathbf{x}$. In other words, we want to learn the distribution $p(\mathbf{y}|\mathbf{x})$, occasionally notated as $\mathcal{Q}_{\mathbf{y}|\mathbf{x}}$. Typically, this conditional distribution is sufficient for inference purposes because we are only interested in predicting some output given some inputs. Using the notation from the previous section, we have that $\mathcal{Q}$ is $\mathcal{Q}_{\mathbf{y}|\mathbf{x}}$ and z_i is $\mathbf{y}_i$. Thus, the weights $\mathbf{w}$ will only parameterize the distribution $\mathcal{Q}_{\mathbf{y}|\mathbf{x}}$.

Here, we should note that the weights $\mathbf{w}$ do not fully specify $\mathcal{F}$, they only parameterize the conditional distribution $p(\mathbf{y}|\mathbf{x})$. We have $\mathcal{F}$ or $p(\mathbf{x}, \mathbf{y})$ as our object of interest. From Bayes' Rule, we can rewrite this as $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x})\, p(\mathbf{y}|\mathbf{x})$. A discriminative model only imposes a probabilistic model on $p(\mathbf{y}|\mathbf{x})$.

To train such a model, we can use the framework established in the previous section. In particular, we have that our objective function is

$$L(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} -\log\left(p(\mathbf{y}_i|\mathbf{x}, \mathbf{w})\right).$$

To evaluate $-\log\left(p(\mathbf{y}_i|\mathbf{x}_i, \mathbf{w})\right)$, we use our input $\mathbf{x}_i$ and weights to $\mathbf{w}$ to produce a conditional distribution. Then, we find the likelihood of the given output $\mathbf{y}_i$.

We can see that the objective function that we constructed mirrors that from the deterministic case nicely. In particular, we have that our loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is defined as $\mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right) = -\log\left(p(\mathbf{y}_i|\mathbf{x}, \mathbf{w})\right)$. Given its connection to the idea of cross-entropy from information theory, this is sometimes referred to as cross-entropy loss.

As the name might suggest, discriminative models are typically used in classification contexts, where the goal is to classify some input into some categories. For example, in an computer vision context, $\mathbf{x}$ may be an image, and $\mathbf{y}$ may be its associated label. The goal is to model the possible outputs given some input image. Here, the task simply involves identification given some input image. Therefore, it is sufficient to solely model the conditional distribution $p(\mathbf{y}|\mathbf{x})$.

1.3.3 Probabilistic Generative Model

Now, we discuss probabilistic generative model, which attempts to parameterize the entire DGP, or $\mathcal{F}$ itself. Previously, we saw that the discriminative model learned the parameters of the conditional distribution $p(\mathbf{y}|\mathbf{x})$. However, it did not fully parameterize $\mathcal{F}$. As such, it is unable to "generate" new data if desired. On the other hand, generative models parameterize the entire distribution. Thus, we can generate new data by simply sampling from this distribution. As the name suggests, generative models can be useful when training data is scarce, and we want to generate more data.

Recall $\mathcal{F}$ refers to the joint distribution of inputs, $\mathbf{x}$, and outputs, $\mathbf{y}$. This is also notated as $p(\mathbf{x}, \mathbf{y})$. Thus, our objective function is

$$L(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} -\log\left(p(\mathbf{x}_i, \mathbf{y}_i|\mathbf{w})\right).$$

Note the distinction between this objective and the one from the discriminative case. Now, we care about the joint distribution $p(\mathbf{x}_i, \mathbf{y}_i|\mathbf{w})$ parameterized by the weights $\mathbf{w}$. In the discriminative case, we only cared about the conditional distribution $p(\mathbf{y}_i|\mathbf{x}_i, \mathbf{w})$. Using the notation from the previous sections, we have that $\mathcal{Q}$ is $\mathcal{Q}_{\mathbf{x},\mathbf{y}}$ and z_i is $(\mathbf{x}_i, \mathbf{y}_i)$. Note that now we are modeling the ordered pair of inputs and outputs. Here, we have extended our framework from the deterministic case slightly, since the loss function involves both the input and outputs, rather than just the outputs.

1.4 Training

While the full details of training these models is beyond the scope of this thesis, we will quickly discuss the key components of training. As we have discussed, the goal for most machine learning models is to minimize some objective function. We have that this objective function is a function of the weights $\mathbf{w}$. Thus, we can move in the direction of steepest descent. This is the opposite direction of the gradient of the objective function with respect to the weights, which is the direction of steepest ascent. Thus, when training these models, we are often interested in finding

$$\mathbf{J}_L(\mathbf{w}) = \nabla_{\mathbf{w}} L = \begin{bmatrix} \frac{\partial L}{\partial \mathbf{w}_1} \\ \frac{\partial L}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial L}{\partial \mathbf{w}_d} \end{bmatrix}.$$

This vector is often called the Jacobian and is an essential element to training a machine learning model. In this thesis, we will follow the convention where gradients are column vectors.

Then, we update the weights using the update policy defined by

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \eta \mathbf{J}_L\left(\mathbf{w}\right),$$

where η is called the learning rate. Typically, this updating continue until the training "converges", or when $\|\mathbf{w}^{(t+1)} - \mathbf{w}^{(t)}\| \leq t$ for some threshold t. Training a model to convergence is quite difficult, and there is a lot of discussion in the literature about optimal learning rates. Furthermore, there have been more complex variants of the update policy to help with convergence.

Calculating the Jacobian for the objective function L is computationally intensive, especially because L typically sums over all N samples in the training dataset. To that end, training procedures typically perform stochastic gradient descent (SGD). At each iteration, a batch of the training data $\mathcal{B} \subseteq \mathcal{D}$ is selected. Then, the objective function is approximated by summing over $|\mathcal{B}|$ observations. This approximation is equivalent to taking the sample mean to approximate the loss over the total training dataset $\mathcal{D}$.

1.5 Evaluation

Once a model has been trained, we want to evaluate the accuracy of a model. In particular, we are interested in $L(\mathbf{w}) = \mathbb{E}_{\mathcal{F}}\left[\mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right)\right]$. We could report the final average loss on the training dataset $\mathcal{D}$. However, this measure would likely be biased. In particular, since the model was trained on the training data, it is likely to perform better on the training data than a random draw from $\mathcal{F}$. Intuitively, optimizing the average training loss would leave models prone to "overfitting", where the model only "memorizes" patterns in the training data and cannot generalize to the full distribution $\mathcal{F}$. This problem is related to the statistical field of "post-selection inference", which studies model evaluation after a model has been selected using some data. We will omit a more thorough technical discussion of post-selection inference and only discuss it at an intuitive level.

A common solution is to split the training data $\mathcal{D}$ into a training and validation (test) split. We will call these $\mathcal{T}$ and $\mathcal{V}$ respectively. The idea is to train the model on $\mathcal{T}$. Then, run the model on $\mathcal{V}$ and report the average loss. This way, we can measure the "generalized" loss. We will use such metric for the remainder of this thesis.

1.6 Artificial Neural Networks (ANNs)

In recent years, artificial neural networks, a class of machine learning models, has exploded in popularity. As the name suggests, the general idea of these artificial neural networks is to mimic the network of neurons found in the brain. As such, ANNs describe the relationship between neurons to achieve a certain task. ANNs can be used for both deterministic and probabilistic models as we will discuss shortly. In this section, we present one of the simplistic ANNs: multi-layer perceptrons (MLPs). First, we define a perceptron, then proceed to construct the MLP. In Chapter 3 and Chapter 4, we will discuss more advanced ANN models.

1.6.1 Perceptron

Original Formulation

The idea of a perceptron was first introduced in seminal work by McCulloch and Pitts [MP43] and later by Rosenblatt [Ros58] as a way to explain how brains can retain information at the neuron level. They proposed that these perceptrons were organized in a feed-forward structure with input and output connections for each perceptron. For a given perceptron, the combination of the input perceptron signals and the strength of the connections with the input perceptron would dictate whether a perceptron would "fire" or not. Figure 1.1 shows a simple diagram of the proposed perceptron.

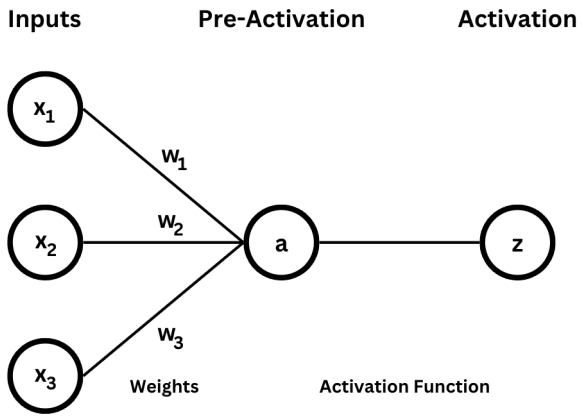

Figure 1.1: A simple perceptron.

Mathematically, we can describe a single perceptron as follows. A perceptron has inputs $\mathbf{x} \in \mathbb{R}^d$, which is a d-dimensional vector. This is analogous to the signals from the preceding perceptrons. Then, we have our weights $\mathbf{w} \in \mathbb{R}^d$, which is also a d-dimensional vector that represents the "strength" of the connection. In the original construction of a perceptron, the

perceptron was "all-or-nothing". It would either "fire" or not. Thus, we take the dot product $\mathbf{x}^\mathsf{T}\mathbf{w}$ and compare it to a threshold θ. If $\mathbf{x}^\mathsf{T}\mathbf{w} \geq \theta$, then the perceptron "fires". Thus, given some input $\mathbf{x}$, we have the output of the perceptron is $H\left(\mathbf{x}^\mathsf{T}\mathbf{w} - \theta\right)$, where $H : \mathbb{R} \to \mathbb{R}$ is the Heaviside function defined as

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}.$$

The $-\theta$ is referred to as the bias and f is called the activation function.[‡] To "train" an "intelligent" system, $\mathbf{w}$ and θ are ideally learnable parameters.

Current Formulation

The current formulation of a perceptron mirrors the original construction quite closely. For some input $\mathbf{x} \in \mathbb{R}^d$, we have learnable weights $\mathbf{w} \in \mathbb{R}^d$ and a learnable bias $b \in \mathbb{R}$. Then, the output is $f\left(\mathbf{x}^\mathsf{T}\mathbf{w} + b\right)$, where $f : \mathbb{R} \to \mathbb{R}$ is any activation function. $\mathbf{x}^\mathsf{T}\mathbf{w} + b$ is sometimes referred to as the pre-activation value. This formulation allows for a more general activation function. Common activation functions include the sigmoid, defined as $\sigma(x) = \frac{1}{1+e^{-x}}$, tanh, and rectified linear unit (ReLU) function defined by

$$\text{ReLU}(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}.$$

The ReLU function mimics the idea of a perceptron "firing", with negative pre-activation values being mapped to 0.

1.6.2 Multi-Layer Perceptron (MLPs)

Formulation

The perceptron serves as the basis for the multi-layer perceptrons, which is also referred to as fully connected (FC) layers in the literature. We can combine the perceptrons in a layer-wise manner to construct a multi-layer perceptron as shown in Figure 5.1.

[‡]More classical papers may refer to f as the squashing function.

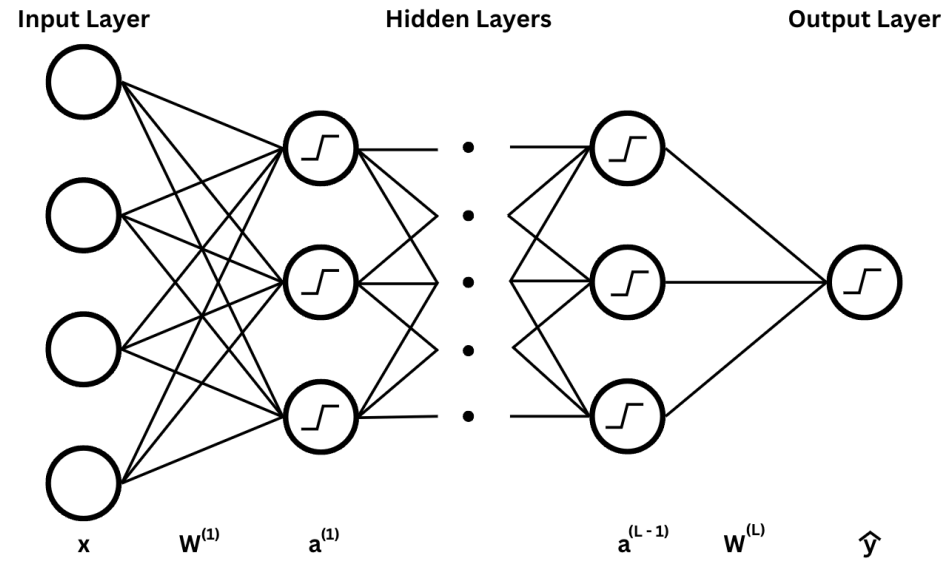

Figure 1.2: A MLP diagram.

Here, we show the basic structure of a MLP. The MLP has an input $\mathbf{x} \in \mathbb{R}^n$ and an output $\mathbf{y} \in \mathbb{R}^m$. Between the input and output layers, there is a series of $L-1$ hidden layers with d_i perceptrons in each layer with $i \in \{1, \ldots, L-1\}$. To simplify notation, we also define $d_0 = n$ and $d_L = m$. We refer to the output of perceptron j in layer ℓ as $\mathbf{a}_j^{(\ell)} \in \mathbb{R}$. The associated weights of a perceptron with output $\mathbf{a}_j^{(\ell)}$ are $\mathbf{w}_j^{(\ell)} \in \mathbb{R}^{d_{\ell-1}}$. The associated bias term is $\mathbf{b}_j^{(\ell)} \in \mathbb{R}$. For an input $\mathbf{a}^{(\ell-1)} \in \mathbb{R}^{d_{\ell-1}}$ to layer ℓ, we have that

$$\mathbf{a}_j^{(\ell)} = f\left(\left(\mathbf{a}^{(\ell-1)}\right)^\mathsf{T} \mathbf{w}_j^{(\ell)} + \mathbf{b}_j^{(\ell)}\right),$$

where $f : \mathbb{R} \to \mathbb{R}$ is the activation function. The $\left(\mathbf{a}^{(\ell-1)}\right)^\mathsf{T} \mathbf{w}_j^{(\ell)}$ term is sometimes called the pre-activation value and notated as $\mathbf{z}_j^{(\ell)}$.

To find the output of an entire layer, we can use matrix notation and the row-major representation for the inputs. Suppose there are N input vectors. First, we can construct $\mathbf{W}^{(\ell)} = \left[\mathbf{w}_1^{(\ell)} \mathbf{w}_2^{(\ell)} \ldots \mathbf{w}_{d_\ell}^{(\ell)}\right] \in \mathbb{R}^{d_{\ell-1} \times d_\ell}$ to be our weight matrix for layer ℓ. The bias term is the vector $\mathbf{b}^{(\ell)} \in \mathbb{R}^{d_\ell}$. Then, the input to layer ℓ in row-major layout is

$$\mathbf{A}^{(\ell-1)} = \begin{bmatrix} \left(\mathbf{a}_1^{(\ell-1)}\right)^\mathsf{T} \\ \left(\mathbf{a}_2^{(\ell-1)}\right)^\mathsf{T} \\ \vdots \\ \left(\mathbf{a}_N^{(\ell-1)}\right)^\mathsf{T} \end{bmatrix} \in \mathbb{R}^{N \times d_{\ell-1}}.$$

This yields the output of layer ℓ, $\mathbf{A}^{(\ell)} \in \mathbb{R}^{N \times d_\ell}$, as

$$\mathbf{A}^{(\ell)} = \mathbf{f}\left(\mathbf{A}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{1}_N \left(\mathbf{b}^{(\ell)}\right)^\mathsf{T}\right).$$

Here, $\mathbf{f} : \mathbb{R}^{N \times d_\ell} \to \mathbb{R}^{N \times d_\ell}$ is the activation function f applied element-wise to the pre-activations. By convention, we will say that the depth of a MLP refers to the number of hidden and output layers L. The width of a layer is the number of perceptrons d_i, and the maximum width is sometimes referred to as the width of the MLP.

Although quite simple in nature, it has been rigorously proven that MLPs are universal approximators. As shown by Hornik, Stinchcombe, and White [HSW89], a single hidden layer MLP with sufficiently many perceptrons and non-polynomial activation function can approximate any Borel-measurable function from one finite dimensional Euclidean space to another (e.g. $\mathbb{R}^n$) "arbitrarily well". Rigorously defining these ideas and proving the statement is beyond the scope of this thesis, but this motivates the use of MLPs for these complex tasks. This theoretical result does have its limitations in a practical sense though. In particular, the number of perceptrons needed is likely not feasible, motivating some of the architectures that we will discuss in Chapter 3 and Chapter 4.

A simple way to get around this challenge in practice is to make the MLP deeper by adding more layers. This gives rise to the field of *deep learning* [LBH15], which studies models with many hidden layers (when L is large). We will combine MLPs with ideas discussed in Chapter 3 and Chapter 4 to get more complex deep learning models. Intuitively, these deep learning models perform better than more classical techniques because they can learn patterns without hand-selecting features. Prior to the rise of deep learning, most classical models required the practitioner to select some sets of features to "search for". For example, for images, there might be certain sets of lines that are crucial for classifying some objects. Thus, these models are quite limited and not generalizable. In contrast, for these deep learning models, hand-engineered features are not required, as the model will learn which features are important through the many hidden layers.

Implementation

To implement a MLP, it suffices to implement matrix multiplication and the application of an activation function $\mathbf{f} : \mathbb{R}^{N \times d_\ell} \to \mathbb{R}^{N \times d_\ell}$. This is to compute

$$\mathbf{A}^{(\ell)} = \mathbf{f}\left(\mathbf{A}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{1}_N \left(\mathbf{b}^{(\ell)}\right)^\mathsf{T}\right).$$

The bias term can be incorporated to the matrix multiplication as follows. Consider

$$\tilde{\mathbf{A}}^{(\ell-1)} = \begin{bmatrix} \mathbf{1}_N & \mathbf{A}^{(\ell-1)} \end{bmatrix}.$$

Furthermore, let

$$\tilde{\mathbf{W}^{(\ell)}} = \begin{bmatrix} \left(\mathbf{b}^{(\ell)}\right)^\mathsf{T} \\ \mathbf{W}^{(\ell)} \end{bmatrix}.$$

Then, we have that

$$\mathbf{A}^{(\ell)} = \mathbf{f}\left(\tilde{\mathbf{A}}^{(\ell-1)}\tilde{\mathbf{W}}^{(\ell)}\right) = \mathbf{f}\left(\mathbf{A}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{1}_N\left(\mathbf{b}^{(\ell)}\right)^\mathsf{T}\right)$$

as desired. We postpone the discussion of the implementation and parallelization of the matrix multiplication and activation function to the next section after more specialized architecture is introduced.

Computational Diagram

To represent these complex models, we often use a computational diagram, which visually shows the relationship between inputs and outputs in the model. Below, we show a simple computational diagram for $\mathbf{AB} + \mathbf{C}$.

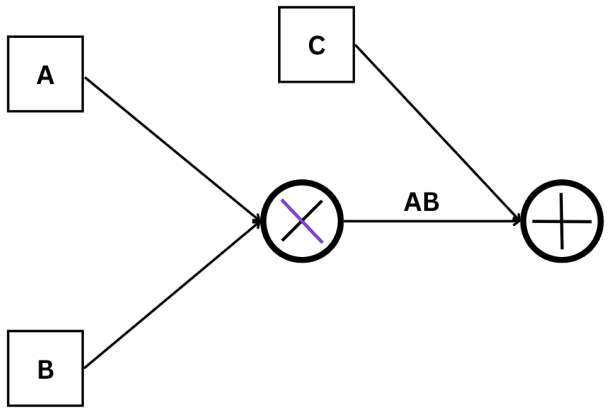

Figure 1.3: A simple computational diagram.

The computational diagram is a directed acyclic graph (DAG). Here, we adopt the convention where the nodes are operations or functions and the edges represent the values. The computational diagram could also be represented with the edges representing operations and functions and nodes representing values, but the former convention is more commonly used in machine learning. There are also input nodes that are sometimes referred to as leaf nodes,

since they do not have any incoming connections. This computational diagram becomes key when discussing training algorithms, which we discuss next.

Training

Although training is not the main focus of this thesis, we will still include some preliminary information on it. As mentioned previously, we are interested in finding the gradient of the objective function with respect to the flattened weight vector $\mathbf{w}$. To find this, the backpropagation (backwards autodifferentiation) [RHW86] algorithm is commonly used. The idea behind the backpropagation algorithm is to fix an output for the model and find the gradient of that output with respect to all weights in the model by using the chain rule. Let v be a parameter of interest. Then, suppose that operations consisting of v has outgoing edges to $u_1, \ldots, u_m$ in the computational diagram. Then, we have that

$$\frac{\partial \mathcal{L}}{\partial v} = \sum_{i=1}^{M} \frac{\partial \mathcal{L}}{\partial u_i} \frac{\partial u_i}{\partial v}$$

from the chain rule. Thus, the gradient can be computed by iteratively traversing the computational graph in the reverse direction.

We will show an example for the MLP, but this idea can be applied to more complex models such as the models seen in Chapter 3 and Chapter 4. Before we do so, we introduce some general notation and results from vector and matrix calculus.

First, suppose we have a function $f : \mathbb{R}^m \to \mathbb{R}^n$. Let $\mathbf{x} \in \mathbb{R}^m$. Then,

$$f(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^n.$$

Then, we have that the Jacobian matrix of f with respect to $\mathbf{x}$ is

$$\mathbf{J}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial \mathbf{x}_1} & \frac{\partial f_1}{\partial \mathbf{x}_2} & \cdots & \frac{\partial f_1}{\partial \mathbf{x}_m} \\ \frac{\partial f_2}{\partial \mathbf{x}_1} & \frac{\partial f_2}{\partial \mathbf{x}_2} & \cdots & \frac{\partial f_2}{\partial \mathbf{x}_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial \mathbf{x}_1} & \frac{\partial f_n}{\partial \mathbf{x}_2} & \cdots & \frac{\partial f_n}{\partial \mathbf{x}_m} \end{bmatrix} \in \mathbb{R}^{n \times m}.$$

In the case, where $n = 1$ (the co-domain of f is $\mathbb{R}$), we can construct a quantity called the

gradient. In particular, we have

$$\nabla_{\mathbf{x}} f = \begin{bmatrix} \frac{\partial f}{\partial \mathbf{x}_1} \\ \frac{\partial f}{\partial \mathbf{x}_2} \\ \vdots \\ \frac{\partial f}{\partial \mathbf{x}_m} \end{bmatrix} \in \mathbb{R}^m.$$

Note that $\nabla_{\mathbf{x}} f = \mathbf{J}_f(\mathbf{x})^\mathsf{T}$.

Now, suppose we have a function $f : \mathbb{R}^{m \times n} \to \mathbb{R}$. In machine learning models, we typically deal with these type of objective functions, where the input is some matrix of data and the output is a real-valued loss. Then, we define the gradient with respect to the matrix as

$$\nabla_{\mathbf{X}} f = \begin{bmatrix} \frac{\partial f}{\partial \mathbf{X}_{1,1}} & \frac{\partial f}{\partial \mathbf{X}_{1,2}} & \cdots & \frac{\partial f}{\partial \mathbf{X}_{1,n}} \\ \frac{\partial f}{\partial \mathbf{X}_{2,1}} & \frac{\partial f}{\partial \mathbf{X}_{2,2}} & \cdots & \frac{\partial f}{\partial \mathbf{X}_{2,n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial \mathbf{X}_{m,1}} & \frac{\partial f}{\partial \mathbf{X}_{m,2}} & \cdots & \frac{\partial f}{\partial \mathbf{X}_{m,n}} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

for $\mathbf{X} \in \mathbb{R}^{m \times n}$. We can continue this gradient function for higher dimensions.

Now, we define the chian rule for a scalar-valued function composed with a vector-valued function. Let $f : \mathbb{R}^m \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$. Then, consider $h(\mathbf{x}) = f \circ g(\mathbf{x})$ for $\mathbf{x} \in \mathbb{R}^n$. Then, we have that

$$\nabla_{\mathbf{x}} h = (\mathbf{J}_g(\mathbf{x}))^\mathsf{T} \left(\nabla_{g(\mathbf{x})} f \right) \in \mathbb{R}^n.$$

Now, we define a special case of the chain rule for matrices. Suppose we have a function $f : \mathbb{R}^{m \times p} \to \mathbb{R}$. Then, let $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{B} \in \mathbb{R}^{n \times p}$, $\mathbf{C} \in \mathbb{R}^{m \times p}$ such that $\mathbf{AB} = \mathbf{C}$. Then, we have that

$$\nabla_{\mathbf{A}} f = (\nabla_{\mathbf{C}} f) \left(\mathbf{B}^\mathsf{T} \right) \in \mathbb{R}^{m \times n}.$$

We also have that

$$\nabla_{\mathbf{B}} f = \left(\mathbf{A}^\mathsf{T} \right) (\nabla_{\mathbf{C}} f) \in \mathbb{R}^{n \times p}.$$

These results will be useful given the ubiquitous use of matrix multiplications throughout machine learning models.

Now, we return to performing backpropagation for an MLP. Then, we are interested in computing $\nabla_{\tilde{\mathbf{W}}^{(\ell)}} L$ for some layer ℓ. Then, we have that

$$\nabla_{\tilde{\mathbf{W}}^{(\ell)}} L = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L}{\partial \mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i)} \nabla_{\tilde{\mathbf{W}}^{(\ell)}} \mathcal{L}.$$

We have that

$$\mathbf{a}^{(\ell)} = \mathbf{f}\left(\left(\tilde{\mathbf{W}}^{(\ell)}\right)^{\mathsf{T}} \tilde{\mathbf{a}}^{(\ell-1)}\right).$$

Suppose we have $\nabla_{\mathbf{a}^{(\ell)}}\mathcal{L}$. Then, we have that

$$\nabla_{\mathbf{z}^{(\ell)}}\mathcal{L} = \left(\mathbf{J}_{\mathbf{f}}\left(\mathbf{z}^{(\ell)}\right)\right)^{\mathsf{T}} \left(\nabla_{\mathbf{a}^{(\ell)}}\mathcal{L}\right).$$

Given that $\mathbf{f}$ is the activation function f applied elementwise, we have that

$$\mathbf{J}_{\mathbf{f}}\left(\mathbf{z}^{(\ell)}\right) = \begin{bmatrix} f'\left(\mathbf{z}_1^{(\ell)}\right) & 0 & \cdots & 0 \\ 0 & f'\left(\mathbf{z}_2^{(\ell)}\right) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'\left(\mathbf{z}_{d_\ell}^{(\ell)}\right) \end{bmatrix} = \operatorname{diag}\left(f'\left(\mathbf{z}_i^{(\ell)}\right)\right).$$

Then, we have that

$$\nabla_{\tilde{\mathbf{a}}^{(\ell-1)}}\mathcal{L} = \left(\tilde{\mathbf{W}}^{(\ell)}\right)\left(\nabla_{\tilde{\mathbf{z}}^{(\ell)}}\mathcal{L}\right) = \left(\tilde{\mathbf{W}}^{(\ell)}\right)\operatorname{diag}\left(f'\left(\mathbf{z}_i^{(\ell)}\right)\right)\left(\nabla_{\mathbf{a}^{(\ell)}}\mathcal{L}\right),$$

since diagonal matrices are symmetric. Furthermore, we have that

$$\nabla_{\left(\tilde{\mathbf{W}}^{(\ell)}\right)^{\intercal}}\mathcal{L} = \left(\nabla_{\tilde{\mathbf{z}}^{(\ell)}}\mathcal{L}\right)\left(\tilde{\mathbf{a}}^{(\ell-1)}\right)^{\mathsf{T}} = \operatorname{diag}\left(f'\left(\mathbf{z}_i^{(\ell)}\right)\right)\left(\nabla_{\mathbf{a}^{(\ell)}}\mathcal{L}\right)\left(\tilde{\mathbf{a}}^{(\ell-1)}\right)^{\mathsf{T}}.$$

Thus, we have shown that it is sufficient to pass $\nabla_{\mathbf{a}^{(\ell-1)}}\mathcal{L}$ from layer ℓ to layer $\ell - 1$. Then, we can compute the Jacobian for all the parameters in the model. Importantly, we have shown that finding the Jacobian for a MLP can be done in linear time with respect to the number of layers. This can be done in a batched fashion for N datapoints using the Hadamard product (element-wise multiplication), but we omit these details as the focus of this thesis is on efficient inference.

2

Architecture Preliminaries

In this chapter, we discuss preliminary background on computer architecture that will form the foundations for this thesis. Again, while these topics are quite broad and it would be hard to give a complete overview of each, we will try to include the most pertinent background for this thesis. While understanding these machine learning models from a theoretical perspective is useful, creating systems that implement these models efficiently is just as important. Given the sheer size of these models, architecting solutions that can actually implement these models in practice has been a large area of study. To that end, most of this thesis focuses on using these theoretical principles to motivate hardware and software solutions. In particular, we focus on hardware-software co-design solutions. This term refers to the process where hardware and software are optimized in sync, allowing optimizations from each to inform each other. We start our discussion with the basic framework for computer systems and build up to more complicated architectures that are commonly used to implement machine learning models. We take a look at these ideas at the computer architecture level, invoking the underlying circuitry and electrical engineering principles where needed. This discussion is guided by [HH12].

2.1 Transistors

Transistors form the basis for most modern electronics because they are cheap and small. Today, we can fit billions of transistors within a single square centimeter, a truly amazing feat of engineering. First, we discuss the fundamental of transistors, then we discuss general trends in transistors in modern electronics. There are two general types of transistors: bipolar junction transistors (BJTs) and metal-oxide-semiconductor field-effect transistors (MOSFETs). Today's electronics are mainly compromised of MOSFETs, so we focus our discussion on these. The goal of a transistor is to create a "on-off switch", when some voltage is applied. Ideally, we want to pass some voltage through a "gate". The "gate" will regulate whether current can pass between the "source" to the "drain". In the following discussion, we take a simplified approach to explaining transistors to give sufficient context for the remainder of the thesis.

2.1.1 Composition

First, we discuss the materials commonly used to construct transistors. Transistors are typically constructed from silicon, which is naturally not a very good conductor of electricity given its 4 valence electrons. Each silicon atom creates four strong covalent bonds with its neighbors, creating a crystalline structure. However, when silicon is carefully "doped" with elements that have 3 or 5 valence electrons, the conductivity properties improve significantly. We call these dopants "impurities".

Let us consider the case where we add impurities with 3 valence electrons. When such an element is added, there are "holes" of "positive" charge, since the impurity atoms only have 3 valence electrons. Thus, electrons can move to fill the holes creating a flow of electricity. We will call this process p-type doping, since it adds holes of "positive" charge. Boron, an element with 3 valence electrons, is a common impurity.

Now, let us consider the case where we add impurities with 5 valence electrons. When such an element is added, there is now an excess of electrons, since the impurity has 5 valence electrons. Thus, the excess electrons can freely move around creating a flow of electricity. We will call this process n-type doping, since it adds additional electrons of negative charge. Phosphorus, an element with 5 valence electrons, is a common impurity.

2.1.2 Construction

Now that we have discussed the composition of these transistors, we discuss the construction of these "on-off switches". Naturally, we consider two types of MOSFETs: nMOS and

pMOS transistors, which use n-type and p-type dopants, respectively. Depending on the use of n-type and p-type dopants we will see the opposite behavior.

The construction of transistors starts from a silicon "wafer". Then, the wafer is iteratively treated with dopants, silicon, and silicon dioxide, SiO_2, and "patterned" to create a desired pattern. We include a diagram of each type of transistor below from [HH12].

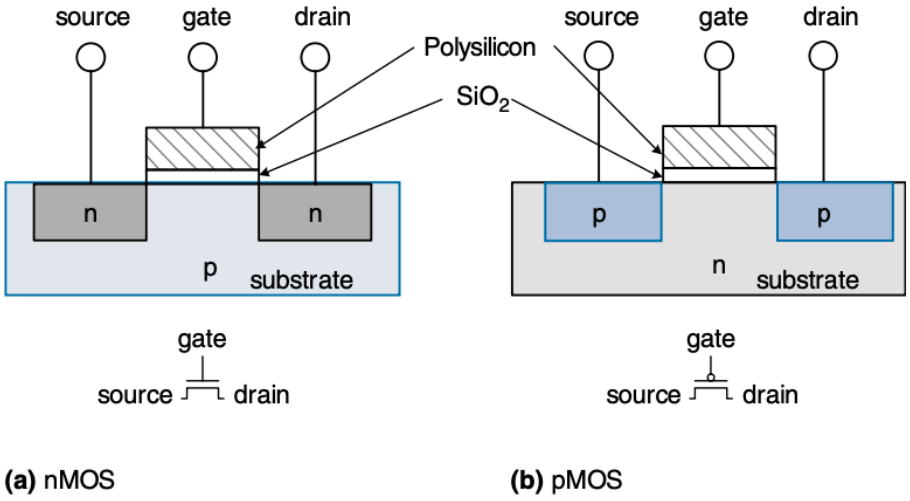

Figure 2.1: nMOS and pMOS. [HH12]

We now discuss the effects of applying some voltage to the gate. First, we consider the case for the nMOS. In this setup, we assume that the source is grounded. Initially, if no voltage is applied to the gate relative to the source, then no current can flow between the source and the drain. This follows from the fact that we have a n-p junction followed by a p-n junction in series. Thus, regardless of the direction of current, one of the junctions will be reverse biased. However, if we apply some voltage V_{DD} to the gate relative to the source, then we get positive charge accumulating at the top of the gate. Then, on the bottom plate, negative charge accumulates creating a "channel" that allows electrons to flow between the two regions with n-type dopants. Thus, for n-type transistors, applying a voltage V_{DD} to the gate relative to the source leads to a "on" state.

The opposite is true for the pMOS. Here, we assume the drain is grounded. When the gate is grounded and a voltage of V_{DD} is applied to the source, there is a flow of holes between the source and drain. This is because the top plate accumulates with negative charge attracting holes to the bottom plate. This creates the channel for the flow of holes. However, when we also apply V_{DD} to the gate, there is no flow of holes, since we have a p-n junction followed by a n-p junction. Therefore, opposite of nMOS, applying no voltage to the gate leads to a "on" state.

We see that a transistor has "capacitor"-like properties, where the silicon dioxide, SiO_2, acts as the dielectric between the polysilicon and silicon plates. A capacitor is a electronic component that consists of two conductive plates separated by a dielectric. When a voltage V_{DD} is applied, charge Q accumulates on one plate and $-Q$ accumulates on the other such that $Q = CV_{DD}$, where C is the "capacitance", or the ability for the capacitor to hold charge.

We have that the capacitance is

$$C = \frac{\epsilon A}{d},$$

where ϵ is the permittivity of the dielectric, A is the area of the plates, and d is the distance between the plates. Furthermore, we have that the "switching" power required to change the state of the transistor is proportional to $CV_{DD}^2 f$, where C is the capacitance, V_{DD} is the voltage difference of the plates, and f is the frequency of changing the state of the transistor. We will return to these ideas shortly.

2.2 Combinatorial Logic

Together, nMOS and pMOS can be combined in a way to represent combinatorial circuits by using the complementary metal-oxide-semiconductor (CMOS) fabrication process. First, we consider their use in combinatorial logic, where the output is only a function of the input and not of past history. Formally, we want to construct a circuit to represent the boolean function $F : \{0,1\}^n \to \{0,1\}^m$. Note that it is sufficient to consider the case for $m = 1$, since we can construct a set of functions $F_i : \{0,1\} \to \{0,1\}$ for $i \in \{1, \ldots, m\}$ and represent each individually. The idea is as follows. We will use the pMOS and nMOS transistors to create "gates" as building blocks for a larger network. Then, we will prove that this "gate" abstraction can be used to represent any such function $F : \{0,1\}^n \to \{0,1\}$.

2.2.1 Sum-of-Products (SOP) Form

Consider an arbitrary function $F : \{0,1\}^n \to \{0,1\}$. Then, we can represent this function as a sum-of-products (SOP) as follows. For a given $\mathbf{y} \in \{0,1\}^n$, we first define a minterm term function $a_{\mathbf{y}} : \{0,1\}^n \to \{0,1\}$ defined by

$$\mathbf{x} \mapsto \bigwedge_{i=1}^{n} \tilde{\mathbf{x}}_i,$$

where $\tilde{\mathbf{x}}_i = \mathbf{x}_i$ iff $\mathbf{y}_i = 1$ and $\tilde{\mathbf{x}}_i = \bar{\mathbf{x}}_i$ otherwise. First, we claim that $a_{\mathbf{y}}(\mathbf{x}) = 1$ iff $\mathbf{x} = \mathbf{y}$. First, we prove the forward direction. We will prove the contrapositive. Suppose that $\mathbf{x} \neq \mathbf{y}$. Then, we have that for some $i \in \{1, \ldots, n\}$, $\mathbf{x}_i \neq \mathbf{y}_i$. Therefore, we claim that $\tilde{\mathbf{x}}_i = 0$. First,

suppose that $\mathbf{y}_i = 0$. Then, we have that $\mathbf{x}_i = 1$. Then, we have that $\tilde{\mathbf{x}}_i = \bar{\mathbf{x}}_i = 0$. Now, suppose that $\mathbf{y}_i = 1$. Then, we have that $\mathbf{x}_i = 0$. Thus, we have that $\tilde{\mathbf{x}}_i = \mathbf{x}_i = 0$. Therefore, we have that $a_{\mathbf{y}}(\mathbf{x}) = 0$.

Now, we prove the backwards direction. Suppose $\mathbf{x} = \mathbf{y}$. Then, we have that

$$a_{\mathbf{y}}(\mathbf{y}) = \bigwedge_{i=1}^{n} \tilde{\mathbf{y}}_i.$$

Then, we claim that $\tilde{\mathbf{y}}_i = 1$ for all $i \in \{1, \ldots, n\}$. First, suppose that $\mathbf{y}_i = 0$. Then, we have that $\tilde{\mathbf{y}}_i = \bar{\mathbf{y}}_i = 1$. Now, suppose that $\mathbf{y}_i = 1$. Then, we have that $\tilde{\mathbf{y}}_i = \mathbf{y}_i = 1$. Thus, we have that

$$a_{\mathbf{y}}(\mathbf{y}) = \bigwedge_{i=1}^{n} 1 = 1.$$

Then, we have that F can be defined as

$$\mathbf{x} \mapsto \bigvee_{F(\mathbf{y})=1} a_{\mathbf{y}}(\mathbf{x}).$$

This is sometimes referred to as disjunctive normal form. Intuitively, we take the inputs that F maps to 1, and we check if the input matches any of them.

2.2.2 Gates

Now, we present the "gate" abstraction by showing the construction of logic gates using nMOS and pMOS transistors. The gate will be a set of functions G that have domain $\{0,1\}^m$ and range $\{0,1\}$ such that any boolean function $F : \{0,1\}^n \to \{0,1\}$ can be written as an expression of functions in G. We say that this set G is functionally complete if this property holds. We have already seen that $\{\text{OR}, \text{AND}, \text{NOT}\}$ are functionally complete, since we can express any boolean function in SOP form. We also claim that $\text{NAND} : \{0,1\}^2 \to \{0,1\}$ is functionally complete. To show that it is functionally complete, it is sufficient to show that OR, AND, and NOT can be expressed in terms of NAND. We have that this is true since

$$\text{OR}\,(x, y) = \text{NAND}\,(\text{NAND}\,(x, x)\,, \text{NAND}\,(y, y))\,,$$

$$\text{AND}\,(x, y) = \text{NAND}\,(\text{NAND}\,(x, y)\,, \text{NAND}\,(x, y))\,,$$

and

$$\text{NOT}\,(x) = \text{NAND}\,(x, x)$$

for $x, y \in \{0,1\}$.

Now, we show the construction of these gates using nMOS and pMOS transistors. The idea is as follows. We will divide the circuits into two parts. One part will connect the ground to the output, and the other will connect the voltage source of V_{DD} to the output. When the output should be a high bit, we want to connect the output to the voltage source and have no connection between the ground and the output. When the output should be the low bit, we want to connect the output to the ground and have no connection to the voltage source. The pMOS requires that the source be connected to a voltage V_{DD}, so we let the portion connected to the voltage source be composed of pMOS transistors. We call this the "pull-up network" (PUN), since it will "pull" the output to the high bit if it is "on" by connecting the output to the voltage source. We use nMOS transistors for the other component and call it the "pull-down network" (PDN), since it will connect the ground to the output. Ideally, we only have one of these components "on" for some input.

Now, we discuss the arrangement of the pMOS and nMOS transistors within the pull-up and pull-down networks. We would like the pull-up to be "on" when F maps the input to 1. pMOS transistors are "on" when the gate is at ground relative to the source. Thus, we can consider $\bar{F} : \{0,1\}^n \to \{0,1\}$ defined by $\mathbf{x} \mapsto F(\mathbf{x})$. We have that $\bar{F}$ is defined by the mapping

$$\mathbf{x} \mapsto \bigwedge_{F(\mathbf{y})=1} \overline{\mathbf{a}_{\mathbf{y}}(\mathbf{x})}.$$

This is a product-of-sum expressions, which is also sometimes called conjunctive normal form. Then, for both PDN and PUN we can arrange the corresponding transistors in parallel for addition terms and in series for multiplicative terms. This follows from the fact that for additive terms we only need a single transistor to be "on", but for multiplicative terms we need all transistors to be "on". Thus, using this formulation, we can represent any boolean function as a circuit of transistors.

Up until this point, we have not discussed efficient implementation. In particular, we have not discussed the optimal configuration of transistors such that the number of transistors used to represent a boolean function is minimized. However, this is a very difficult problem. In fact, the problem is NP-hard, and this can be proven through a reduction from the set cover problem. The field has been well-studied, but we will omit this discussion, as it is beyond the scope of this thesis. Moving forward, we will use this gate abstraction, since we know that we can construct these gates using transistors. Using these gates, we can build complex circuits that integrate many different components onto a single "chip". This will be referred to as integrated circuits (ICs). The ability to combine millions of transistors on a single chip is known as very large scale integration (VLSI).

2.3 Sequential Logic

Now, we consider sequential logic, where the output is both a function of the input and past history. This introduces some complications to the traditional combinatorial logic, since we now need some way of maintaining state. To do so, we introduce the SR latch, D latch and D flip-flop.

2.3.1 SR Latch

The idea of a SR latch is as follows. We have some current state Q. Then, our circuit will take in inputs, Q, $\bar{Q}$, R, and S. S will be referred to as the set input, and R will be referred to as the reset input. Hence, the name SR latch. At most one of these should be true. If $S = 0$ and $R = 1$, then we want to reset the value of Q to 0. If $S = 1$ and $R = 0$, then we should set the value of Q to 1. If $S = 0$ and $R = 0$, then we should just maintain the current value of Q. The key difference in sequential and combinatorial logic is that the output is connected to the input for the computation of the next output. Here, we have that Q and $\bar{Q}$ are both outputs and inputs of the circuit. Thus, we can represent the circuit as follows. Let $Q = \text{NOT}\left(\text{OR}\left(R, \bar{Q}\right)\right)$ and $\bar{Q} = \text{NOT}\left(\text{OR}\left(S, Q\right)\right)$. Then, this achieves the desired properties. This allows us to define $\text{SRL} : \{0,1\}^4 \to \{0,1\}^2$, which yields

$$\left(S, R, Q, \bar{Q}\right) \mapsto \left(\text{NOT}\left(\text{OR}\left(R, \bar{Q}\right)\right), \text{NOT}\left(\text{OR}\left(S, Q\right)\right)\right).$$

2.3.2 D Latch

The D latch makes two improvements on the SR latch. First, it abstracts the SR latch making impossible to have $S = 1$ and $R = 1$. Furthermore, it allows for an input of a clock input (CLK), which regulates when the value is updated. The D latch will take in two inputs: the clock (CLK) and the "data" to set Q to (D). Then, if CLK $= 0$, we want to keep Q in the same state. If CLK $= 1$, then we want to change Q to the value of D. We can do this in the following way.

Let $\left(Q, \bar{Q}\right) = \text{SRL}\left(\text{AND}\left(\text{CLK}, D\right), \text{AND}\left(\text{CLK}, \text{NOT}\left(D\right)\right), Q, \bar{Q}\right)$. Thus, we set when the clock and the data bits are high. We reset when the clock is high and the data bit is low. Otherwise, we hold the same value by setting both S and R to 0. Thus, we have now defined $\text{DL} : \{0,1\}^4 \to \{0,1\}^2$, where

$$\left(\text{CLK}, D, Q, \bar{Q}\right) \mapsto \text{SRL}\left(\text{AND}\left(\text{CLK}, D\right), \text{AND}\left(\text{CLK}, \text{NOT}\left(D\right)\right), Q, \bar{Q}\right).$$

2.3.3 D Flip-Flop

Clocks are typically implemented using oscillators, where the CLK value oscillates between 0 and 1 at a regular frequency. Thus, we would only like the value of Q to be changed when the CLK transitions from 0 to 1. We would not like it to be changed continuously when CLK is at 1. For the rest of the thesis, we will refer to the period of the clock as the clock cycle. Thus, we need additional abstraction compromised of two D latches.

Let DFF still take in CLK and D as input, as well as Q and $\bar{Q}$. Then, we define the output to be

$$\mathrm{DL}\left(\mathrm{CLK}, \mathrm{DL}\left(\overline{\mathrm{CLK}}, D, Q, \bar{Q}\right)_1, Q, \bar{Q}\right),$$

where $\mathrm{DL}\left(\overline{\mathrm{CLK}}, D, Q, \bar{Q}\right)_1$ is the first output of $\mathrm{DL}\left(\overline{\mathrm{CLK}}, D, Q, \bar{Q}\right)$. When $\mathrm{CLK} = 0$, we have that the first D latch will pass on the data D. Then, when $\mathrm{CLK} = 1$, the second D latch pass on the data D. Thus, D only passes through both latches completely on a "positive" edge. With the D flip-flop, we now have a way to maintain a state and update it using the clock.

2.4 Memory

We have introduced a form of "temporary" memory through sequential logic. Note that the circuit always needs to be "on" for the value to be retained. However, now, we would like to introduce more persistent forms of memory that can be used for computation and updated over time, even in the potential absence of power. Thus, in this section, we discuss the key ingredients of modern-day memory hierarchies within computing systems. There are many types of memory that exist, and each serves its own purpose. Below, we discuss three levels of memory that may be desirable, and their implementations. In general, there is a delicate balance between speed, cost, and capacity. We can have memory that lies closer to the computation, but it will likely be more costly and not be that large. Alternatively, we can have memory that lies further away from the computation so it will take longer, but it will be cheaper and have more space. Below, we show a graphical illustration of this.

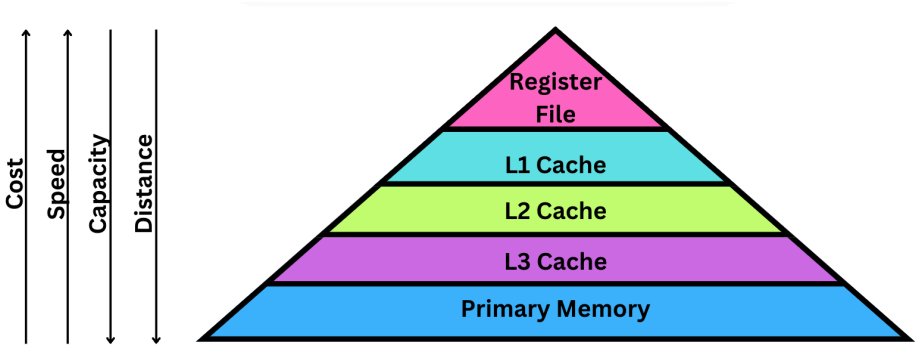

Figure 2.2: Illustration of memory hierarchy.

Memory can be viewed from the following general perspective. Suppose we have an address $A \in \{0,1\}^N$, and we are trying to access data $D \in \{0,1\}^M$ at address A. There are 2^N possible addresses, and each corresponds to a data value of bitwidth M, which we will call a word. We refer to 2^N as the depth and M as the width. Therefore, we will store a total of NM bits. For each bit, we will use a circuit to store the value of the bit. In addition, each bit will be attached to a wordline and a bitline. In read mode, if the wordline that the bit is attached to is high, then the bit should transfer its data to the bitline. In write mode, if the wordline that the bit is attached to is high, then the bit should store the data that is currently in the bitline. Most of the distinction between the memory types that we will discuss boils down to the type of circuitry used to store the bits.

2.4.1 Processor-Level Memory

First, we discuss memory types that lie closest to the processor, which will eventually take instructions and execute computation. The closest memory to the processor is typically the register file. However, register files are typically quite small with only a few hundred bytes of storage. The small size allows very quick memory access times. However, since the size of the register file is quite limited, a more "general" source of memory needs to be established. Ideally, we would like to avoid repeated trips to this "general" source of memory, so the processor also typically employs a cache, which stores recently visited addresses closer to the processor. The cache operates in a hierarchical fashion. Usually, there are multiple levels to the cache, L1 to L$\{n\}$, where n is typically 3 or 4. Each level gets progressively bigger but also progressively costlier to access.

Both the register file and caches are typically implemented using static random access memory (SRAM). SRAM is implemented using 6 transistors for each bit, so we call this a 6T SRAM array. The six transistors are mainly used to create a bistable latch, which retains

the data. SRAM cells are "static" because the transistors allow the cell to steadily hold the data as long as the power is on. This will be in contrast to the dynamic random access memory (DRAM), which we will discuss shortly.

2.4.2 Primary Memory

As discussed in the previous section, we would like to have a more "general" source of memory that can be far from the processor, but store bits in large capacity. Thus, we introduce primary memory as the main mechanism for this. SRAM is a convenient choice for smaller memory sizes. In particular, it works well for storing data with size on the order of megabytes. However, to achieve storage of data on the order of gigabytes, we will use dynamic random access memory (DRAM). DRAM replaces the bistable latch with a single capacitor, only requiring a single additional capacitor. The capacitor stores charge to indicate a high configuration. Although it is cheaper to construct given the smaller number of elements, the downfall of DRAM is that the charge on the capacitor gradually leaks. Therefore, the values need to be "refreshed" regularly every few milliseconds. Furthermore, once the bitline "reads" the value from the capacitor, the charge is completely dissipated, so it must also be refreshed. It is important to note that this recharging adds extra latency that is not present in the design of SRAM. We will leave the details of this recharging process beyond the scope of this thesis.

2.4.3 Non-Volatile Memory

Memory can be classified as volatile or non-volatile. Volatile memory requires a constant power supply for the data to be retained, while non-volatile memory retains data even without a power supply. Up until this point, we have mainly discussed volatile forms of memory. However, for edge devices that may be powered off, non-volatile memory is important. Achieving non-volatility is difficult with a typical transistor, since it is unable to hold charge. We will not delve into the specifics, since it lies beyond the scope of this thesis, but we offer a high level overview.

Non-volatility is typically achieved by using special materials or creating new charge-trapping transistors. For example, flash memory, a common type of non-volatile memory, uses a floating-gate transistor, where the gate is insulated in some special material that allows the gate to retain charge across the plates. Alternatively, one can use magnetism to store the bits without power. In particular, magnetic solutions typically use a special "read-and-write head" to read and alter the magnetization of specific regions of the magnetic tape. We will return to this idea of non-volatile memory in Chapter 9.

2.5 Central Processing Units (CPUs)

In this section, we discuss the core elements of modern-day CPUs. While this topic is quite broad, we will assume some familiarity with these ideas, so we will not give as complete of an overview. The goal is to construct some circuitry that is capable of doing general-purpose calculations, such as arithmetic, as well as handling some basic control flow, like conditionals and loops. For the sake of simplicity, we will assume a von Neumann architecture, where the instructions and data occupy the same memory space. This is as opposed to a Harvard architecture, where they occupy separate memory spaces.

2.5.1 Simplified Overview

The computation of a general-purpose processor can generally be broken down into a five-stage pipeline. They consist of the following five stages.

1. **Instruction Fetch (IF)**: In this stage, the processor uses the current program counter (PC), which points to the current instruction, to fetch the next instruction.
2. **Instruction Decode (ID)**: Here, the instruction, as well as its associated inputs, are decoded.
3. **Execute (EX)**: An arithmetic logic unit (ALU) or other functional unit executes the desired instruction along with the associated input.
4. **Memory Access (MEM)**: If the instruction requires loading or storing memory, it is done during this step.
5. **Write Back (WB)**: The output is written to the register file.

The key logic for a general-purpose processor lies in a control unit, which sends the appropriate signals for the current computation. Ideally, we can separate each of these stages into a pipeline and some form of temporary memory between each. That way, we can "pipeline" the stages by having a new instruction start at the IF stage every clock cycle and progress through the pipeline. However, one might already see that this leads to some issues. First, as we alluded to in the section above, memory accesses are costly, especially if a trip to main memory is necessary. Thus, all the other instructions in the pipeline may need to be stalled until a trip to memory complete. Furthermore, if a future instruction relies on a previous instruction's output, we need to stall the future instruction until the previous one has completed. These "dependency" issues make naive pipelining difficult. Finally, control flow is equally problematic. We do not know when the program counter will jump to a different

instruction, which will require us to flush out all the preemptive calculations. Thus, this motivates many of the optimizations made to processors over the years that we will now discuss.

2.5.2 Recent Optimizations

To accelerate computation on general-purpose processors, we would like to focus on instruction-level parallelism (ILP). We want to execute as many instructions in parallel as possible, without running into the dependency issues laid out above. This motivates a few microarchitectural optimizations that have become popular in more recent CPUs. While we will not cover the full specifics, we will give a general overview of the optimizations.

The main issue we want to deal with is the problem of instructions varying in duration. One key observation is that if there is a costly instruction not all future instructions need to be stalled. In particular, only the ones that depend on the current instruction need to be installed. This motivates the idea of out-of-order (OOO) execution, where instructions are executed out-of-order, while still keeping the semantics of the program intact. There are many ways to go about this. Scoreboarding is one algorithm, which uses a central "scoreboard" to keep track of functional units that are in use and data dependencies. Then, it only issues instructions when there is an available functional unit and the dependency is clear. Tomasulo's algorithm is another algorithm that introduces the idea of a "reservation station" for each functional unit. This spreads the responsibility of the central "scoreboard" over many smaller stations that keep track of dependencies and alert other stations when instructions have completed.

Another difficulty discussed in the previous section is the problems that arise with control flow. In particular, we do not know whether a branch will be taken or not. Thus, we do not know which set of instructions to proceed with. Branch prediction helps us to partially solve this problem by making an informed prediction on the branch that the program is likely to take. This is known as speculative execution. Naively, this can be done based on previous history for that specific branch. More sophisticated algorithms may use MLPs, as discussed in Chapter 1, to predict whether a branch is taken or not. This is a nice application of machine learning to problems in computer architecture. Using these predictions, the processor can then make a "educated guess" on the instructions to continue with, which will make it less likely that instructions need to be flushed if the another branch is taken.

2.5.3 Multi-Core CPUs

With the slowing of Dennard scaling, which will soon be discussed, interest in multi-core CPUs slowly began to rise as a way to increase parallelism. However, adding support for multi-core CPUs is not as easy as stamping out more cores. In particular, the interaction between memory and the processors is something that needs to be reasoned about carefully. While we defer in-depth discussions of these topics, there are many design considerations that accompany the addition of multiple cores. First, we need to consider the placement of the cores and memory tiles. Then, we need to consider the access patterns of each core with respect to a memory tile to construct optimal connections between the cores and memory banks. Furthermore, we also need to consider cache coherence policies, so that shared data is consistent between different cores.

2.6 Graphics Processing Units (GPUs)

In this section, we motivate and detail the use of GPUs in machine learning implementations. While initially motivated for use in graphics, GPUs have been critical to the implementation of machine learning models due to their ability for parallelization. Again, we do not give the full details of their implementations, but we provide important context.

2.6.1 Computing Paradigm

In the construction of CPUs, we focused on adding logic to increase the throughput of instructions on a single core. We want to keep the latency low per instruction. However, GPUs take a different approach. GPUs thrive in data-parallel environments, where data parallelization is natural. In particular, they use the richness of data to mask some of the delays that occur due to memory reads and writes or failed speculative execution. Thus, they have less logic per core, allowing them to greatly multiply the number of cores within a particular circuit. For example, in graphics, each of the display pixel values needs to be computed. However, the computation is likely very similar on each pixel. Therefore, we can expect less complicated control flow logic, and we would prefer more cores for faster computation.

2.6.2 Programming Model

We will now detail the programming model for GPUs, mainly focusing on NVIDIA GPUs. The programming model for GPUs is a hierarchical one, where the most basic execution unit is a thread. Threads operate on a small set of data elements. These threads can be

cooperatively grouped together to form blocks or cooperative thread arrays (CTAs). Threads in the same CTA have access to the same shared memory and can synchronize with each other. Finally, these blocks are grouped together to form a grid that can be partitioned during execution.

The main execution unit of GPUs are streaming multiprocessors (SMs). They are the fundamental building block of the compute unit of GPUs and can be viewed as small processors. Groups of blocks are partitioned between the SMs, which contains its own ALUs, shared memory, and register file. Then, during each cycle, a warp, collection of threads, is executed on the SM. The SM features a scheduler that allows the SM to decide which warp to run. This allows for the "masking" effect that we alluded to earlier. When one warp is waiting on data to be returned from main memory, another warp can run in its place. It is important to note that all threads in a warp need to execute the same instruction. In particular, GPUs follow a SIMT model, which is closely related to the SIMD classification. We will discuss this in a later section. Therefore, to maximize instruction throughput, we would like to limit the amount of thread divergence, where threads in the same warp execute different instructions.

The memory hierarchy follows similarly to the hierarchy for CPUs, with the exception that shared memory is only shared between threads on a block. Main memory is also known as global or device memory, and it is accessible by all threads, as well as the host device that launches the kernel in the GPU. To run a workload on a GPU, a host device, typically a CPU, will launch a compute kernel. Compute kernels are self-contained pieces of code called by a host program that is launched on some specialized computing unit. For example, a compute kernel can be a matrix multiplication operation performed on a GPU. Thus, the goal is to optimize the performance of these individual kernels, so that the total workload is also accelerated. We will see optimizations for matrix multiplications and convolutions shortly.

2.7 Systolic Arrays [Kun82]

In this section, we introduce systolic architectures, particularly systolic arrays. Systolic arrays are a form of hardware design, where processing elements (PEs) are arranged in an array-like fashion. Then, data is "pulsed" through the processing elements. Hence, the name systolic array. Each of the processing elements performs a small computation before passing data to the adjacent cells. [Kun82] argued that systolic architectures are beneficial, particularly for compute-bound tasks, since they can increase the computational throughput while keeping the memory bandwidth constant. Below, we show an example of a systolic

array used in the design of tensor processing units (TPUs).

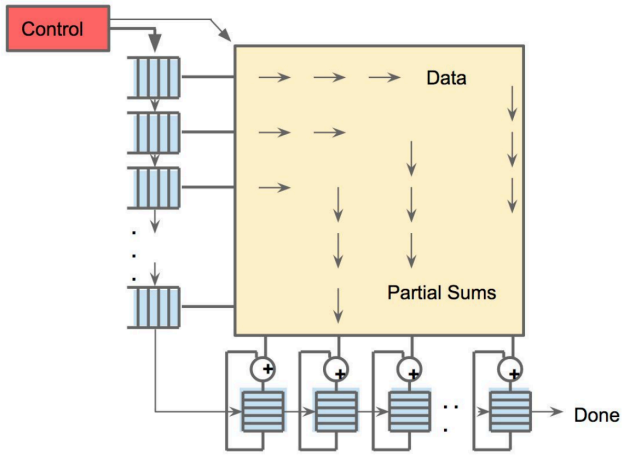

Figure 2.3: Systolic array diagram. [JYP+17]

Now, we motivate the use of systolic arrays for machine learning workloads. As seen in Chapter 1, matrix multiplications are ubiquitous in machine learning workloads. Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{X} \in \mathbb{R}^{n \times p}$. Then, suppose we want to compute $\mathbf{AX} \in \mathbb{R}^{m \times p}$. We can "load" the weights of $\mathbf{A}^\top$ into the cells of the systolic array. Then, on the ith time step, we can begin passing $\mathbf{x}_i$ through the array in a left-to-right fashion. When $\mathbf{x}_i$ reaches column j, the partial sum will just compute the dot product $\mathbf{x}_i^\top \mathbf{a}_j$. Thus, the systolic architecture provides a very natural parallel representation for a matrix-matrix multiplication. The key insight is that the matrix can be continuously passed through the systolic array.

2.8 Implementing Specialized Circuits

Previously, we focused on more general types of computing hardware, such as CPUs and GPUs. Now, we turn our attention to designing specialized circuitry for different downstream applications beyond general-purpose computing. This discussion will form the basis for much of the discussion of Chapter 9.

2.8.1 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs) are a type of integrated circuit that are programmable after fabrication, as the name suggests. FPGAs are very useful for prototyping new hardware, since they are flexible by design. First, we present a high-level overview of the structure of an FPGA. FPGAs contain a grid of configurable logic blocks (CLBs). Each CLB can be thought of as its own function, taking in some inputs and passing on an output.

Below, we show a general schematic of a typical CLB. Note that this varies between FPGAs from different vendors, but they generally share the same structure.

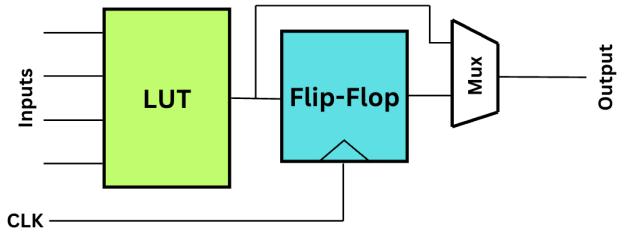

Figure 2.4: Diagram of a general CLB.

CLBs can be composed to model both combinatorial and sequential logic. The lookup table (LUT) allows the user to configure each CLB to a desired function. Then, the D-flip-flop allows the CLB to maintain temporary memory. By composing these CLBs, users can model a wide range of circuits, making FPGAs quite flexible. Given register transfer level (RTL) code written in a hardware description language (HDL) like Verilog, we can use synthesis tools from FPGA vendors to convert the RTL code to netlists, a gate-level representation of the circuit. Then, we can also use vendor tools to convert the netlists into a bitstream that can then be used to program the CLBs. These vendor tools are often referred to as electronic design automation (EDA). Many of these vendor tools are closed-sourced though, so we defer any discussion of the specifics to beyond this thesis.

2.8.2 Application-Specific ICs (ASICs)

Although FPGAs are quite flexible, they pay a price for the flexibility. In particular, the CLBs are quite bulky and not the most optimal representation of a circuit. Therefore, this motivates the needs for application-specific ICs (ASICs), which tradeoff reconfigurability and cost for performance. In particular, given some RTL code, we can again use EDA tools to convert the RTL code to an ASIC. However, since we have a specialized purpose and do not support reconfigurability, many optimizations can be made to further improve performance. We can remove unnecessary logic, as well as test different parameters within the design space. We will discuss many of these design considerations in the next section. Then, the ASIC can be "taped-out" in a fabrication plant.

2.8.3 System-On-Chip (SoCs) and Network-On-Chip (NoCs)

System-On-Chip (SoCs) is another type of integrated circuit that combines both general-purpose and specialized hardware all into a single chip, rather than having them operate separately. It typically features CPUs, GPUs, and memory cells, as well as any other auxiliary modules like networking or I/O interfaces. SoCs are quite common in "edge computing" like in smartphones. In these domains, area and power efficiency metrics are crucial, and an SoCs smaller size allows it to consume less power and have shorter latencies as data does not need to flow as far. The network-on-chip (NoCs) mediates the interactions between different modules in the SoC. We will discuss these much more deeply in Chapter 9.

2.9 Scaling Laws

2.9.1 Dennard Scaling

As we have discussed, these transistors can be combined to represent complex logic functions. Thus, it is desirable to make these transistors as small as possible. Through the evolution of fabrication processes, we have been able to fit more and more transistors within some fixed area. However, as with most things in computer architecture, there are tradeoffs that need to be made during this "miniaturization" process. We are interested in the effect of this "miniaturization" on properties like power consumption. Dennard formalized these tradeoffs in [DGY+74]. We first begin our discussion with "ideal" Dennard scaling, which Dennard discussed in [DGY+74], where we scale the transistor length along all dimensions by $\frac{1}{k}$, scale the voltage by $\frac{1}{k}$, and scale the doping concentration by a factor of k.

This idealized version of Dennard scaling is known as "constant electric field" scaling, since the electric field between the plates remains constant. We have that the electric field is $E = \frac{V}{d}$, where V is the voltage difference between the plates and d is the distance between the plates. Scaling both V and d by a factor of $\frac{1}{k}$ keeps E constant.

Using the equations from the previous section, we have that $A \propto k^{-2}$ and $d \propto k$. Thus, we have $C \propto k^{-1}$. Furthermore, Dennard showed that $f \propto k$. This follows from the fact that there is a shorter distance between the plates allowing charge to move between plates faster. Thus, we have that $P \propto k^{-2}$. Therefore, the power density $\frac{P}{A}$ is constant with respect to k. This observation became known as Dennard scaling. Dennard scaling was powerful because it said that packing more transistors within a fixed area would not change the power consumption. Thus, we could continue to improve performance by packing more transistors while maintaining the same amount of power consumption.

Unfortunately, by the mid-2000s, this idealized version of Dennard scaling began to end.

In particular, the "idealized" version of scaling where voltage was scaled by $\frac{1}{k}$ ran into a few barriers. First, given the smaller size, there was a greater change of "leakage current", where current would flow through a transistor even though it was in a "off" state. Furthermore, the threshold voltage required to switch the state of a transistor does not scale with size. Thus, scaling voltage down too much would lead to unreliable switching behavior.

Therefore, instead of maintaining a constant electric field, V_{DD} began to become constant. As a result, the current scales at a rate of k, leading to $f \propto k^2$. Therefore, $P \propto k$ and $\frac{P}{A} \propto k^3$. With the end of "idealized" Dennard scaling, we could no longer pack transistors more tightly without seeing an increase in power consumption. An increase in power consumption yielded other issues like increased heat generation. This limitation on power consumption is sometimes known as the "power wall". Thus, the field turned to other alternatives to further improve performance.

2.9.2 Moore's Law

In the previous section, we have described physically the effects of scaling on power consumption through Dennard scaling. In this section, we take an economic perspective and discuss the practical implications of these physical relations.

In 1965, Moore predicted, through the below empirical evidence, that the number of transistors within a integrated circuit (IC) would double roughly every two years. The observation came from the following plot. The above diagram plots the manufacturing cost per transistors against the number of transistors in an IC. Initially, as the number of transistors for a single IC increase, the fixed manufacturing cost for each IC is spread amongst more transistors, resulting in a decrease in cost per component. However, at some point, the manufacturing costs of packing the transistors begins to dominate. In particular, it requires more machinery to pack transistors more densely, and the defect rate is likely higher, leading to higher costs. Thus, at any point in time, there is some number of transistors in an IC that results in the minimum cost per component. Over time, Moore showed that the transistor count that resulted in the minimum doubled roughly every two years. Thus, Moore predicted that this trend would continue, and every two years the number of transistors in an IC would double. Therefore, we would likely see an improvement in performance. Furthermore, as a consequence of Dennard scaling, assuming the same area was used, the power consumption would remain constant. Thus, we would be able to achieve better performance with little additional power consumption overhead. This phenomenon was incredibly powerful.

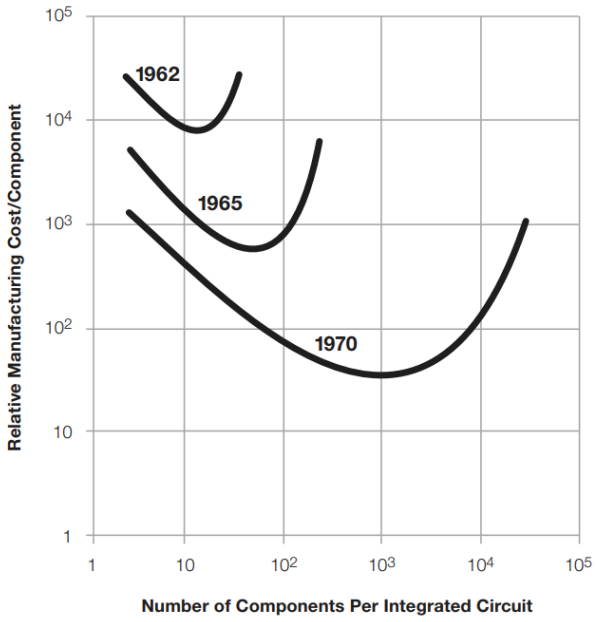

Figure 2.5: Graphical depiction of Moore's Law. [Moo65]

However, like Dennard scaling, Moore's Law also ran into physical limitations that began to slow the trend. Leakage currents, as well as manufacturing challenges have made the doubling difficult. In Chapter 9, we will discuss some alternatives to the typical planar structure of transistors that have allowed for further innovation. However, even with this innovation, the rate of doubling has slowed significantly. This slow of Dennard scaling and Moore's Law will motivate the need for specialized hardware.

2.10 Flynn's Taxonomy

Given this variety of computing systems, we now discuss a general taxonomy of computing systems that helps to contextualize computing today. In particular, Flynn's Taxonomy [Fly66] gives a way of categorizing computer architectures along two axes. We can think of programs as a series of instructions operating on a set of data. Thus, Flynn's Taxonomy categorizes the systems based on number of instruction streams and the data streams on which they operate on. The four categories, which corresponds to each of the combinations are the following.

1. **S**ingle **I**nstruction Stream, **S**ingle **D**ata Stream (SISD)
2. **S**ingle **I**nstruction Stream, **M**ultiple **D**ata Stream (SIMD)
3. **M**ultiple **I**nstruction Stream, **S**ingle **D**ata Stream (MISD)

4. **M**ultiple **I**nstruction Stream, **M**ultiple **D**ata Stream (MIMD)

This taxonomy is useful for describing different computing systems and their components.

2.11 Design Considerations

2.11.1 Timing Considerations

An important design consideration when designing a circuit is the choice of clock frequency. As alluded to previously, the clock allows circuits to temporarily preserve state. However, the state is only updated every cycle. Thus, intuitively, we would like to choose a higher frequency to decrease the latency of a circuit. However, we are limited by the fact that transistors take time to update state. Therefore, we cannot change the clock too quickly, else we will encounter glitches. When choosing a clock frequency, we typically focus on on the propagation delay, t_{pd} the largest delay from input to output. We need to choose a clock frequency such that the length of the period is at least t_{pd}. Else, we will change the clock value before the output of the circuit has stabilized. Typically, to decrease t_{pd} for complex circuits, we use a concept called pipelining. We split a complex circuit into smaller circuits, using temporary memory to store intermediate states. Then, we can decrease the propagation delay, since the complexity of each circuit is reduced.

2.11.2 Performance

Performance can be measured in a variety of ways. Below, we will discuss some commonly used metrics characterizing performance both at different granularities from models to computing systems, particularly for machine learning applications.

MACs

We have already seen that matrix multiplication algorithms are ubiquitous in machine learning models. An essential operation to calculating matrix multiplication outputs is the multiply-accumulate (MAC). Thus, we can characterize the computation complexity of a machine learning model by counting the number of MACs. Given an accumulator a and inputs b and c, the MAC operations performs $a \leftarrow a + b \times c$. Since we typically want to keep a in a consistent datatype, there are two possible implementations based on which quantities are rounded. First, there is the fused multiply-add (FMA) or fused multiply-accumulate (FMAC), which is implemented as $a \leftarrow \text{round}\,(a + b \times c)$, where the rounding mode is chosen by the implementation. Thus, in a FMA, the result of the product is left

in full precision (which takes at most two times the number of bits to represent). In the unfused version, rounding is done twice (once on the product and once on the sum). Thus, we have $a \leftarrow \text{round}\,(a + \text{round}\,(b \times c))$.

For programmable systems, the FMA likely requires its own instruction in the ISA (completed in one clock cycle), while the unfused one can be implemented by using a disjoint multiply and add (completed in two clock cycles). This is seen in the NVIDIA's PTX, which has an `fma` instruction for an FMA, while the unfused version can be implemented using `mul` and `add`.

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times p}$. Then, we have that

$$\mathbf{C}[i][j] = \sum_{k=1}^{n} \mathbf{A}[i][k]\mathbf{B}[k][j].$$

Thus, to compute $\mathbf{C}[i][j]$ for $(i, j) \in \{0, \ldots, m-1\} \times \{0, \ldots, p-1\}$ requires n MACs. Then, to compute $\mathbf{C}$ in total, which consists of mp elements, it requires mnp MACs. Thus, for MLPs, we have that the number of MACs

$$N \sum_{i=1}^{L} d_{i-1} d_i,$$

where d_i is the dimensionality of layer i. Thus, MACs measure the computational complexity of these models, which is directly related to performance. Then, we can use the number of MACs per second, MACs/s, that a processor can execute to measure its performance. Currently, modern computing systems are capable of reaching TMACs/s, where TMACs refers to tera-MACs or 10^{12} MACs. Using MACs/s is a more domain-specific measure of performance as opposed to something like instructions per second, which is more general.

FLOPs

Although MACs are ubiquitous in machine learning models, a more general performance metric that can be used to characterize computational complexity is a floating point operation (FLOP). This typically includes any arithmetic operation, as well as applications of functions. This allows us to include activation functions, element-wise addition, and other operations that we will see in Chapters 3 and 4. Again, we can use FLOPS/s to measure the performance of computing systems. Modern computing systems are also capable of reaching TLOPSs/s.

Roofline Models

The roofline model allows us to characterize the performance of compute kernels. To characterize, the performance of a compute kernel, we want to show the theoretical maximum performance of the kernel and compare it to the current performance. A roofline model allows us to make such comparisons.

In a roofline model, we plot the performance of the compute kernel (in GFLOPs/s) against the arithmetic intensity (in FLOPSs/bytes). The motivation for this choice comes from the fact that performance is typically limited due to computational capabilities or memory bandwidth. Computational limitations refer to limitations in the execution units, such as insufficient tensor cores on a GPU. In this case, the performance is bound by the limited hardware resources, and we say that the kernel is "compute-bound". Compute-bound kernels typically occur when the arithmetic intensity of the kernel is high. Intuitively, this means that the kernel does many FLOPs per unit of memory, so hardware units are under high contention. Alternatively, the kernel's performance could be limited by the main memory bandwidth caused by long main memory access latencies. In such cases, hardware units may be left vacant while memory accesses are being waited on. We call such kernels "memory-bound". We observe memory-bound kernels when the arithmetic intensity is low, since memory access is high. Thus, using arithmetic intensity on the x-axis allows us to identify which is limiting the kernel's performance.

Now, we can plot the theoretical maximum performance. The theoretical maximum is purely a result of the hardware specifications and remains consistent regardless of the workload. As we mentioned, kernels can be either compute-bound or memory-bound. First, we add in the compute-bound constraint (or roofline). For a given computing system, there is likely a peak performance for the hardware units given by π. On the roofline model, we can represent this using a horizontal line at Performance $= \pi$. This is shown by the red dotted line in Figure 2.6. Typically, this peak performance depends on the instruction set being used. For example, a processor with a `fma` instruction may have higher peak performance, since the MAC can be computed in a single clock cycle.

Now, we add the memory-bound constraint (or roofline). At the theoretical maximum, we have that the following equality is true

$$\text{Performance} = \text{Arithmetic Intensity} \cdot \beta.$$

Here, β is the "Memory Bandwidth" and is expressed in $\frac{\text{Byte}}{\text{s}}$. It represents the rate at which data can be read and written between global memory and the registers in the processor under ideal conditions. Actual memory bandwidth usually depends on the type of memory access

operation being done and can be complicated with mechanisms like write coalescing. Using this "Memory Bandwidth" as the slope, we can plot a line that represents this memory-bound roofline. This is shown by the green dotted line in Figure 2.6. This line can be pushed upward depending on the use of caches, which decrease the number of main memory accesses that are needed.

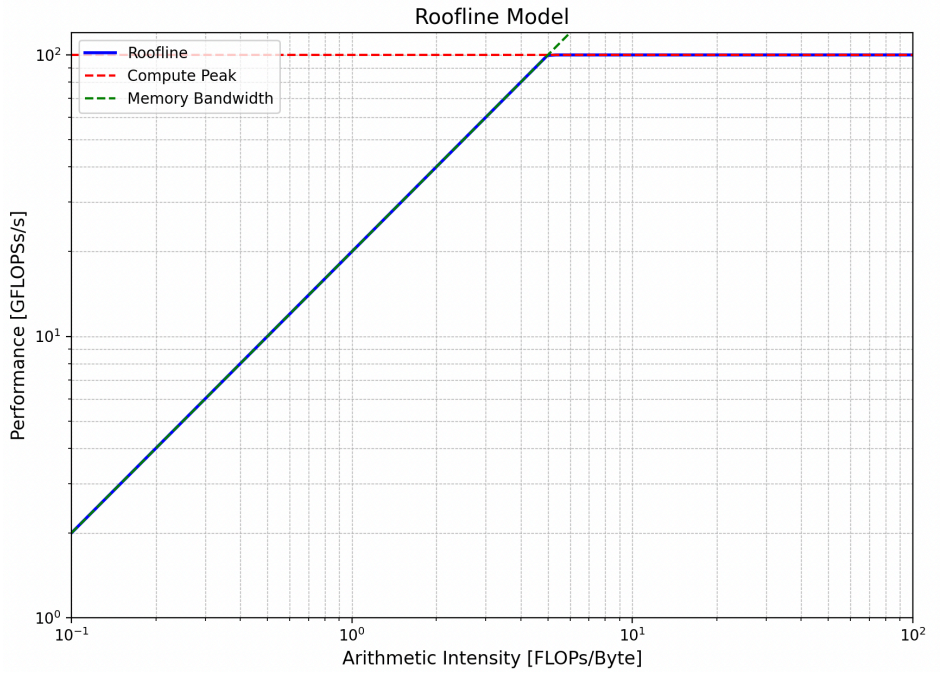

Figure 2.6: Theoretical maximum performance on roofline plot.

The roofline model nicely shows that at different arithmetic intensities there are different limitations. Mathematically, the roofline can be written as

$$R(i) = \min(\pi, i\beta),$$

where the π represents the compute-bound and $i\beta$ represents the memory-bound. For low arithmetic intensities (where $i \leq \frac{\pi}{\beta}$), the kernel is memory-bound. At high arithmetic intensities (where $i > \frac{\pi}{\beta}$), the kernel is compute-bound. The intersection of the two lines at $i = \frac{\pi}{\beta}$ is sometimes referred to as the crossover point.

Now, we will discuss how roofline models can be used to improve kernel performance. Up to this point, we have only discussed the theoretical maximum performance. However, we want to use this information to speed up actual kernels. We can simply add a kernel as a single point in the roofline model, since we can compute the arithmetic intensity and performance through basic profiling. Below in Figure 2.7, we show an example for four

kernels, Kernels A, B, C, and D.

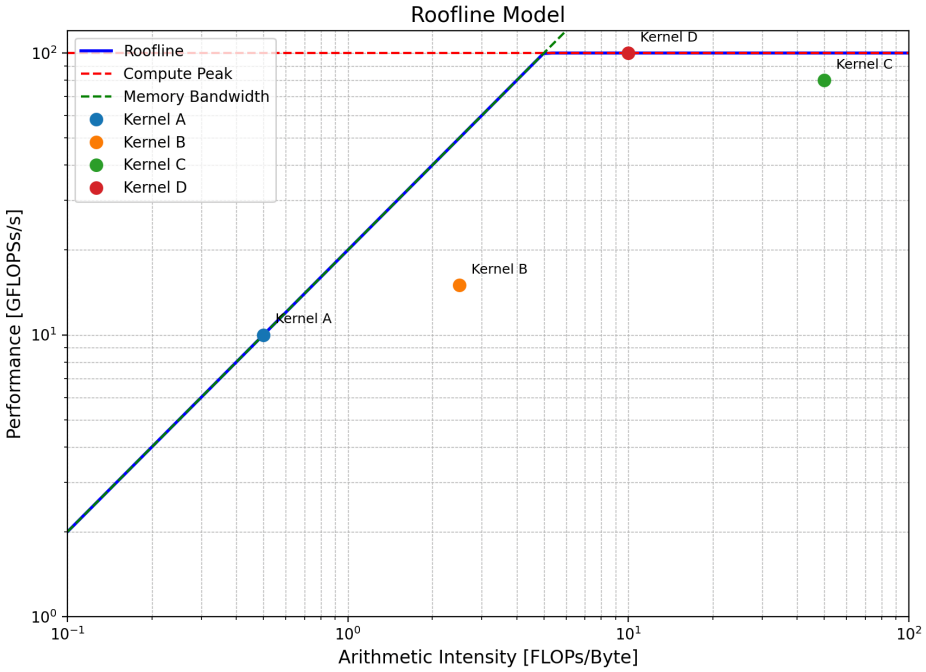

Figure 2.7: An example roofline plot.

If the arithmetic intensity falls below that of the crossover point, then we can conclude that the kernel is memory-bound. The gap in the performance between the theoretical maximum and the actual performance gives us some hints for potential optimizations. If the gap is quite large, one might focus on using memory-based optimizations such as coalesced memory accesses, changing memory access patterns, or reducing memory accesses in general to help make up the gap. For example, Kernel A nearly achieves the theoretical maximum. However, for Kernel B, there is a large performance gap between the actual performance and the theoretical maximum. Thus, memory optimizations may be necessary.

If the arithmetic intensity falls above that of the crossover point, then it indicates that the kernel is compute-bound. If the performance gap between the actual performance and the theoretical maximum is large, then one might consider using different algorithms that increase computational efficiency by reducing the computation necessary. In both the memory-bound and compute-bound cases, we are more concerned with the performance gap between the theoretical maximum and the actual performance. The roofline model gives us a visual way of inspecting this performance gap and also seeing what is the limiting factor. We will return to this roofline model to discuss the performance of convolution and attention kernels, and use it to motivate the need for quantization and sparsity.

2.11.3 Area

Ideally, we want to minimize the area consumed by a circuit to minimize manufacturing costs and power consumption. The area of a circuit is typically correlated with the logical complexity of the underlying circuits. Typically, more complex functions require more complex circuits, leading to larger areas. The area of a circuit is also dictated by the size of transistor used by the manufacturer.

Area can have an adverse effect on performance and energy efficiency. From idealized Dennard scaling, we have that power density remains constant under a constant electric field. Thus, we have that power and area will be linearly correlated. Furthermore, larger areas requires a longer network of wires likely leading to longer latencies. Therefore, it is important to minimize area where possible by making tradeoffs with the complexity of the functions being implemented.

2.11.4 Power

While we have already extensively discussed power consumption during the presentation of transistors and Dennard scaling, power is another axes of consideration when designing a circuit. We saw that

$$P \propto CV_{DD}^2 f.$$

Therefore, while increasing frequency may decrease latency, it will also lead to increased power consumption. Increasing frequency will also require a higher voltage thresholds, which will further increase power consumption. We will return to this principle when discussing dynamic voltage frequency scaling (DVFS) in Chapter 9.

2.12 General Matrix Multiplication (GEMM)

In this section, we discuss the design of hardware solutions specifically for implementing matrix multiplications (matmuls). We will focus on general matrix multiplication which gives a general formulation of the matrix multiplication problem. In particular, suppose you have matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times p}$. We have already discussed that the GPUs and hardware with systolic arrays can quickly compute matrix multiplications for matrices of small size. However, we now consider when the dimensions are large. A simple, but effective, strategy is to "tile" the matrices into smaller matrices and treat each as individual elements. Then, each of the smaller matrices can utilize the dedicated hardware before combining the results back together. This is a "divide-and-combine" algorithm in spirit, as we divide the task of larger matrix multiplications into a set of smaller ones.

Consider the following example. Suppose we tile the matrices $\mathbf{A}$ and $\mathbf{B}$ as follows. Let

$$\mathbf{A} = \left[\begin{array}{c|c} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \hline \mathbf{A}_{21} & \mathbf{A}_{22} \end{array}\right] \text{ and } \mathbf{B} = \left[\begin{array}{c|c} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \hline \mathbf{B}_{21} & \mathbf{B}_{22} \end{array}\right].$$

Then, we have that

$$\mathbf{AB} = \left[\begin{array}{c|c} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{11}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \hline \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{array}\right].$$

Therefore, we have split the matrix multiplication into smaller ones that can then be computed using the specialized hardware.

2.13 Activation Functions

As we have already seen, it is common to apply activation functions to the result of matrix multiplications. We will now discuss how these activation functions are typically implemented. Applying these activations functions is quite parallelizable, since each element is independent of the other. Therefore, there are no data dependencies. Furthermore, since we are implementing matrix multiplication in a tiled fashion, we can just apply the activation function to each tile output. Since the elements are independent of each other, we need not wait for the entire matrix multiplication output to be computed. Thus, we can just leave the tile output in local memory and apply the activation function immediately after. We need not write the tile output back out to global memory and wait for the matrix multiplication to complete before loading it back in. This optimization is known as kernel fusion, since we have "fused" these two operations together. We will see this optimization appear again in future chapters.

Part II

Background

3

Convolutional Neural Networks (CNNs)

Convolutional neural networks (ConvNets or CNNs) have shown great promise for many computer vision tasks. In this section, we discuss background on these models and discuss their implementations on GPU and specialized hardware. We will primarily focus on the application of CNNs to image classification tasks, although there are many other applications where CNNs are suitable.

Now, we present CNNs using the framework presented in Chapter 1. In this section, we will have an input image $\mathbf{x} \in \mathcal{X}$. Typically, $\mathcal{X}$ is $\mathbb{R}^{c \times h \times w}$, where c is the number of channels in the image, h is the height of the image, and w is the width of the image. Our goal is to classify the image with a unique label from some pre-determined set of labels $\mathcal{C}$. We will assume that there are K classes, so $|\mathcal{C}| = K$. As alluded to in Chapter 1, given that we are classifying images, a probabilistic discriminative model may be a natural way to model this problem. Given an image $\mathbf{x}$, we can assume that the conditional distribution of $\mathbf{y}$ is $\text{Cat}(\boldsymbol{\pi}_{\mathbf{x}})$. Thus, we have that $\mathbb{P}(\mathbf{y} = C_k|\mathbf{x}) = \boldsymbol{\pi}_{\mathbf{x},k}$. In other words, given the image, the probability of the image being classified in class k is $\boldsymbol{\pi}_{\mathbf{x},k}$. As we have notated, $\boldsymbol{\pi}_{\mathbf{x}}$ is a function of $\mathbf{x}$. Thus, the primary goal in this chapter is to fit a model $f : \mathcal{X} \to [0,1]^K$ that outputs $\hat{\boldsymbol{\pi}}_{\mathbf{x}} \in [0,1]^K$ given input $\mathbf{x}$. This model will be parameterized by $\mathbf{w}$. As discussed previously, since the conditional distribution is parameterized by $\mathbf{w}$, we can learn the optimal $\mathbf{w}$ by using cross-entropy loss. Given that $\mathbf{y}$ conditioned on $\mathbf{x}$ is distributed $\text{Cat}(\boldsymbol{\pi}_{\mathbf{x}})$, we

have that the log-likelihood function is

$$\ell\left(\mathbf{w};(\mathbf{x},\mathbf{y})\right)=\log\left(\prod_{i=1}^{K}\left(f\left(\mathbf{x};\mathbf{w}\right)_i\right)^{\mathbf{y}_i}\right)=\sum_{i=1}^{K}\mathbf{y}_i\log\left(f\left(\mathbf{x};\mathbf{w}\right)_i\right).$$

Now, we discuss some of the key operations that compose CNNs. We will first offer a mathematical formulation of the operation then discuss the practical implementation of these operations.

3.1 Convolution Operations

As the name suggests, the convolution operation is the backbone operations of CNNs. This operation is derived from the mathematical convolution operation frequently seen in functional analysis. First, we provide the rigorous presentation of the convolution operation. Then, we show the application to CNNs.

3.1.1 Formal Setup

The setup for the convolution operation is as follows. We follow the notation in [SHW57] and [CWV+14].

Continuous Form

In the most general sense, we are given two functions $f:\mathbb{R}^n\to\mathbb{C}$ and $g:\mathbb{R}^n\to\mathbb{C}$. For our purposes, we are not interested in complex-valued functions, so we will assume that the co-domain of f and g is $\mathbb{R}$. Then, we can define the convolution operation as

$$f*g(x)=\int_{\mathbb{R}^n}f(y)g(x-y)dy$$

for $x\in\mathbb{R}^n$.* Thus, the convolution operation, notated as $*$, yields a third function. Intuitively, the convolution operation is presented as an operation that "blends" f and g together. In other words, it describes how f behaves under the shape of g. We can see this in the formulation, since the $g(x-y)$ term provides a weighting of the integral of $f(y)$. f is generally the function we are interested in analyzing, and g is known as the kernel.†

*There are some technical conditions that are required for this integral to exist. In particular, f and g should be Schwartz functions. However, we will omit these details from this thesis.

†Although this convention is typically used, it should be noted that the convolution is actually commutative, so $f*g(x)=g*f(x)$ for all $x\in\mathbb{R}^n$.

Discrete 2D Form

For computer vision tasks, we are generally interested in the discretized version of the convolution operation, since the pixels are discrete objects. Thus, we will modify the functions to be $f : \mathbb{Z}^n \to \mathbb{R}$ and $g : \mathbb{Z}^n \to \mathbb{R}$. Here, the inputs are like the "coordinates" of the pixel. Then, we can define the convolution as

$$f * g(x) = \sum_{y \in \mathbb{Z}^n} f(y)g(x - y)$$

for $t \in \mathbb{Z}^n$. First, we start with the 2D image. We will generalize this to 3D images in the next section. Suppose that $f : \mathbb{Z}^2 \to \mathbb{R}$ represents an image with dimensions $h \times w$ and $g : \mathbb{Z}^2 \to \mathbb{R}$ represents a kernel with dimensions with $r \times s$. Suppose that the pixels for f are represented on the region $\{0, \ldots, h-1\} \times \{0, \ldots, w-1\}$. For $(x, y) \notin \{0, \ldots, h-1\} \times \{0, \ldots, w-1\}$, we will leave $f((x, y))$ undefined. Similarly, suppose the pixels for g are represented on the region $\{-r+1, \ldots, 0\} \times \{-s+1, \ldots, 0\}$. For $(x, y) \notin \{-r+1, \ldots, 0\} \times \{-s+1, \ldots, 0\}$, we will leave $g((x, y))$ undefined. Now, consider the convolution $f * g((a, b))$ for $(a, b) \in \{0, \ldots, h - r\} \times \{0, \ldots, w - s\}$. This will be our "new" image with dimensions $(h - r + 1) \times (w - s + 1)$. For now, we will only consider the pixels where the images fully "overlap", since the other values are undefined. Then, we have that

$$f * g((a, b)) = \sum_{x=a}^{a+r-1} \sum_{y=b}^{b+s-1} f((x, y))g((a, b) - (x, y)).$$

Pictorially, we can represent the convolution operation as shown below. First, we start by evaluating $f * g((0, 0))$ in Figure 3.1. Here, we take the sum of the products of entries in f and g with the same color. We have that

$$f * g((0, 0)) = \sum_{x=0}^{r-1} \sum_{y=0}^{s-1} f((x, y))g((-x, -y)).$$

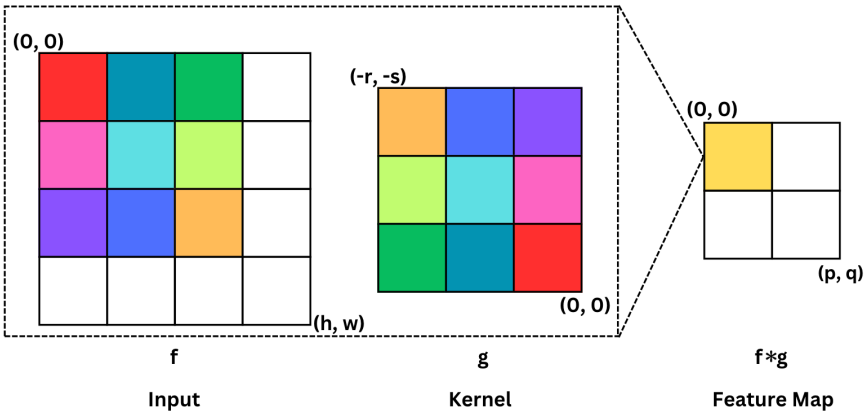

Figure 3.1: Convolution operation at (0, 0).

Similarly, we repeat the same procedure to calculate $f * g((1,0))$. We have that

$$f * g((1,0)) = \sum_{x=0}^{r-1} \sum_{y=1}^{s} f((x,y))g((-x,-y+1)).$$

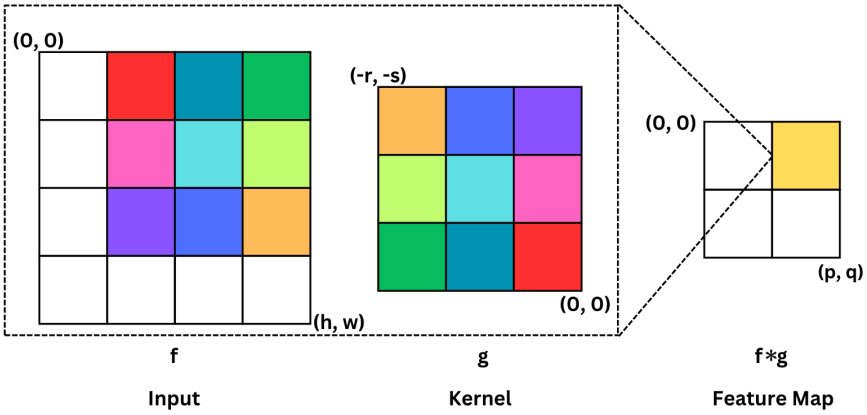

Figure 3.2: Convolution operation at (0, 1).

A minor inconvenience is that the kernel is reflected, which leads to the corresponding entries being reflected. Ideally, we would have an operation $\star$ such that

$$f \star g((a,b)) = \sum_{x=a}^{a+r-1} \sum_{y=b}^{b+s-1} f((x,y))g((x,y)-(a,b)),$$

as depicted in Figure 3.3.

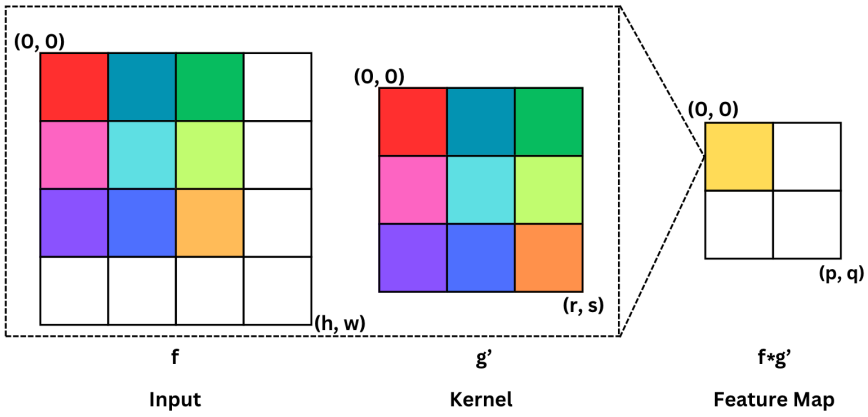

Figure 3.3: Convolution operation with flipped kernel.

We can construct this function by flipping the filter $g : \mathbb{Z}^2 \to \mathbb{R}$ first to get $g' : \mathbb{Z}^2 \to \mathbb{R}$, with $g'(x) = g(-x)$. Then, we can take $f \star g = f * g'$ to get our desired output. The $\star$ operation is called cross-correlation. Confusingly, the CNN literature typically refers to the idea of a convolution and cross-correlation interchangeably for historical reasons. However, they do share a direct connection as explicitly shown above. Typically, in CNN models, cross-correlation is the operation being used.

Thus, with that, we have successfully constructed $f * g(x)$ for $x \in \{0, \ldots, h - r\} \times \{0, \ldots, w - s\}$, which will define the output image. The 2D convolution operates on the "spatial dimension", or the height and width of the input. Later, we will add more dimensions to this operation. The output of the operation is typically called the feature map, and it is the result of a single kernel applied to an entire image.

3D Discrete Form

Now, we finally introduce the 3D discrete form of the convolution, which is most frequently used in CNN models. The 3D form simply generalizes the 2D form with another dimension for the channel. We will call this dimension of the input the channel or depth dimension. We have an image $\mathbf{x} \in \mathbb{R}^{c \times h \times w}$ and a filter $\mathbf{F} \in \mathbb{R}^{c \times r \times s}$. We will assume that the image and the filter share the same number of channels, since the channels typically correspond to the RGB pixel scheme. In the same manner to the previous section, we can extend each of these to functions $\bar{\mathbf{x}} : \mathbb{Z}^3 \to \mathbb{R}$ and $\bar{\mathbf{F}} : \mathbb{Z}^3 \to \mathbb{R}$ respectively. Then, we have that the pixels of $\mathbf{x}$ will be represented by $\bar{\mathbf{x}}(t)$ for $t \in \{0, \ldots, c-1\} \times \{0, \ldots, h-1\} \times \{0, \ldots, w-1\}$. The pixels of $\mathbf{F}$ will be represented by $\bar{\mathbf{F}}(t)$ for $t \in \{-c+1, \ldots, 0\} \times \{-r+1, \ldots, 0\} \times \{-s+1, \ldots, 0\}$. We will also define $\bar{\mathbf{F}}'(t) = \bar{\mathbf{F}}(-t)$ on $t \in \{0, \ldots, c-1\} \times \{0, \ldots, r-1\} \times \{0, \ldots, s-1\}$.

First, we discuss the true convolution operation. We have that

$$\bar{\mathbf{x}} * \bar{\mathbf{F}}\left((i,j,k)\right) = \sum_{\alpha=i}^{i+c-1} \sum_{\beta=j}^{j+r-1} \sum_{\gamma=k}^{k+s-1} \bar{\mathbf{x}}\left((\alpha,\beta,\gamma)\right) \bar{\mathbf{F}}'\left((i,j,k)-(\alpha,\beta,\gamma)\right)$$

for $(i,j,k) \in \{0\} \times \{0,\ldots,h-r\} \times \{0,\ldots,w-s\}$. Intuitively, this is the same as the 2D discrete convolution where we slide the filter over the input image, except now we also sum over all channels. Expressing this in matrix form, we have the following. Let the output be $\mathbf{o} \in \mathbb{R}^{k \times p \times q}$. Since the number of channels are the same, $k = 1$. Then, we have that

$$\mathbf{o}[0,j,k] = \sum_{\alpha=0}^{c-1} \sum_{\beta=0}^{r-1} \sum_{\gamma=0}^{s-1} \mathbf{x}\left[\alpha, a\left(j,r,\beta\right), a\left(k,s,\gamma\right)\right] \mathbf{F}\left[\alpha,\beta,\gamma\right]$$

for $(j,k) \in \{0,\ldots,p-1\} \times \{0,\ldots,q-1\}$. Here, let $p = d(h,r)$ and $q = d(w,s)$. We will call $d : \mathbb{N}^2 \to \mathbb{N}$ the size function and define it as $d(i,j) \triangleq i - j + 1$. The size functions gives the dimensions of the output given the input dimensions. Furthermore, we will call $a : \mathbb{Z}^3 \to \mathbb{Z}$ the accessing functions, which gives the appropriate index to access the image. These give us the index to access for the image. Here, we define $a(i,j,\delta) \triangleq i + j - \delta$.

Now, we discuss the cross-correlation operation. We have that

$$\bar{\mathbf{x}} \star \bar{\mathbf{F}}'\left((i,j,k)\right) = \sum_{\alpha=i}^{i+c-1} \sum_{\beta=j}^{j+r-1} \sum_{\gamma=k}^{k+s-1} \bar{\mathbf{x}}\left((\alpha,\beta,\gamma)\right) \bar{\mathbf{F}}'\left((\alpha,\beta,\gamma)-(i,j,k)\right)$$

for $(i,j,k) \in \{0\} \times \{0,\ldots,h-r\} \times \{0,\ldots,w-s\}$. Again, this is analogous to the 2D cross correlation operation. Expressing this in matrix form, we get the "convolution" form that is commonly discussed in the CNN literature. Let the output be $\mathbf{o}' \in \mathbb{R}^{k \times p \times q}$. Again, $k = 1$. Then, we have that

$$\mathbf{o}'[0,j,k] = \sum_{\alpha=0}^{c-1} \sum_{\beta=0}^{r-1} \sum_{\gamma=0}^{s-1} \mathbf{x}\left[\alpha, a\left(j,r,\beta\right), a\left(k,s,\gamma\right)\right] \mathbf{F}\left[\alpha,\beta,\gamma\right]$$

for $(j,k) \in \{0,\ldots,p-1\} \times \{0,\ldots,q-1\}$. Here, let $p = d'(h,r)$ and $q = d'(w,s)$. We define the size function as $d'(i,j) \triangleq i - j + 1$. Furthermore, we have the accessing function is $a'(i,j,\delta) \triangleq i + \delta$. Thus, we can see that the form of the convolution and cross-correlation operation is the same, except the indexing functions are different.

Thus, when we discuss convolution and cross-correlation, it is generally sufficient to only discuss one operation, since the other operation will be the same except with a flipped kernel. Thus, for the purposes of this thesis, we will focus on the cross-correlation operation, since

it is the most pertinent in the CNN literature.

3D Discrete Form Extension

Now, we can extend this framework to have multiple filters. Suppose $\mathbf{F} \in \mathbb{R}^{k \times c \times r \times s}$. Here, $\mathbf{F}$ has k filters. Each of the filters will yield an output feature map. Similar to the setup in the previous section, there are c input feature maps. Then, for each input $\mathbf{x} \in \mathbb{R}^{c \times h \times w}$, we will apply each of the k filters to the input. Then, we will stack the k output feature maps. Thus, we will have the output $\mathbf{o} \in \mathbb{R}^{k \times p \times q}$. Here, again, we define $p = d(h, r)$ and $q = d(w, s)$, where d is the size function and $d(i, j) \triangleq i - j + 1$. Then, we will define the cross-correlation output

$$\mathbf{o}[z, i, j] = \sum_{\alpha=0}^{c-1} \sum_{\beta=0}^{r-1} \sum_{\gamma=0}^{s-1} \mathbf{x}\left[\alpha, a\left(i, r, \beta\right), a\left(j, s, \gamma\right)\right] \mathbf{F}\left[z, \alpha, \beta, \gamma\right]$$

for $(z, i, j) \in \{0, \ldots, k-1\} \times \{0, \ldots, p-1\} \times \{0, \ldots, q-1\}$.[‡] Note that the general form stays the same, we have just added an extra indexing variable z to index the k output feature maps.

Similar to the matrix multiplication discussed in Chapter 1, we can also batch the inputs into batches of n. We will call this last dimension of the input the batch dimension. Thus, we can have $\mathbf{x} \in \mathbb{R}^{n \times c \times h \times w}$. This will yield an output $\mathbf{o} \in \mathbb{R}^{n \times k \times p \times q}$, where p and q are defined in the same way as above. We have that

$$\mathbf{o}[b, z, i, j] = \sum_{\alpha=0}^{c-1} \sum_{\beta=0}^{r-1} \sum_{\gamma=0}^{s-1} \mathbf{x}\left[b, \alpha, a\left(i, r, \beta\right), a\left(j, s, \gamma\right)\right] \mathbf{F}\left[z, \alpha, \beta, \gamma\right]$$

for $(b, z, i, j) \in \{0, \ldots, n-1\} \times \{0, \ldots, k-1\} \times \{0, \ldots, p-1\} \times \{0, \ldots, q-1\}$.[§] Note that the general form stays the same, we have just added an extra indexing variable b to index the n inputs. Finally, we have fully constructed the cross-correlation operation.

Intuition

Now, given this cross-correlation we will discuss some intuition for their popularity in computer vision tasks. We will focus on the task of image classification. The cross-correlation

[‡]Note that this is no longer a true cross-correlation operation because the dimensions do not match. However, it is still referred to as such.

[§]Note that this is still not a true cross-correlation operation even though the dimensions match. In particular, the filters are not summing up over different inputs. Rather, it is limited to a single input.

operation is used in CNNs for a few reasons. Below, we will highlight the most pertinent ones.

The first advantage comes from the fact that images have strong local structure or "spatial correlations". In particular, neighboring pixels are highly correlated with each other, often forming some type of pattern that makes image classification easier. Intuitively, we, humans, are aware of this fact. When we classify images, we typically look for patterns in specific regions of the image. For example, when looking for an automobile, we might check for wheels or headlights. The cross-correlation allows us to mathematically generate this searching effect by scanning the filter over the image to search for such local patterns. Intuitively, each filter can be viewed as a type of visual pattern that we want to detect in the local neighborhood. By summing over pixels in a neighborhood, the cross-correlation combines the information within a region to detect for the presence of such pattern. Thus, we can view each entry of the feature map as a neuron that is the result of a kernel applied to some patch of the input. The patch of the input that the kernel covers is sometimes referred to as the receptive field on the input. This term comes from this biological motivation that this patch is the field that this neuron is attending to. The size of the kernel, $c \times r \times s$, is then referred to as the receptive field size. This is also why CNNs are sometimes referred to as locally connected networks (LCNs). Rather than using the entire input, each neuron focuses on local patterns.

When searching for such patterns, we are usually less concerned with the location of such pattern, since images can be taken from many different perspectives. Thus, we want the pattern to be detected regardless of its location. The cross-correlation operation achieves this goal nicely, since the filter passes over the entire image. A naive approach to image classification may be to flatten the image $\mathbf{x} \in \mathbb{R}^{c \times h \times w}$ to $\text{vec}(\mathbf{x}) \in \mathbb{R}^{chw}$, and then pass the input through a MLP. However, the MLP lacks this "spatial structure". To see this, note that we could simply change the order of the neurons in a MLP, and this would have no effect on the output. Thus, the MLP would need to redundantly check each subset of neurons for this local structure because it lacks "spatial" understanding. Furthermore, it is not guaranteed that the pattern will always be detected. For example, consider the pattern was translated or slightly distorted within the image. The cross-correlation operation would still detect the pattern because the same kernel is passed over the entire image. However, it is not guaranteed that the new location will result in high activations within the MLP. This property of cross-correlation operations makes them strongly desirable, especially in the image classification setting where images can be taken from various perspectives. Once these patterns are detected, then we can use techniques like pooling, which we will discuss shortly, to combine the detected patterns to form some signal.

Finally, cross-correlation operations are desirable due to the "parameter sharing" that

is used within the operation. Ideally, we want to make these models compact with fewer parameters, so that they can be easily transported. Since each kernel being applied to the entire image, there is a notion of parameters being "shared". This is in direct contrast to the proposed MLP solution. There, the number of weights will grow directly with the size of the image, since the size of the input has increased. Thus, for higher resolution images, more parameters will be needed to find the same local patterns. However, by using the cross-correlation operation, we can avoid such costs since each filter is passed over the entire image. The filters are of constant size. Thus, even though the amount of computation may increase with a larger image, the number of parameters in the model remains the same. This is a desirable property that we will revisit again in future architectures.

Stride

Now, we introduce some variations of the cross-correlation operation that are commonly seen in the literature. In the standard definition presented above, we assumed a horizontal and vertical stride of 1. The filter moved one unit to the right between output features in the same row. Furthermore, the filter moved one unit down between output features in the same column. However, we can make the operation more general by allowing different horizontal and vertical strides. First, we provide some visual intuition for the variation. Like the typical cross-correlation operation, we start with the following for calculating the output activation at $(0, 0)$ as shown in Figure 3.4.

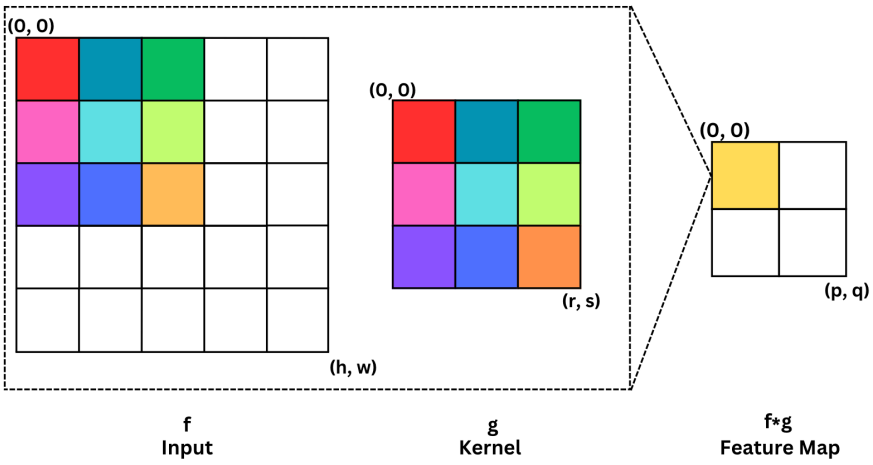

Figure 3.4: Strided cross-correlation operation at (0, 0).

However, when computing the output activation at $(0, 1)$, we shift the kernel two units right on the input image as shown in Figure 3.5.

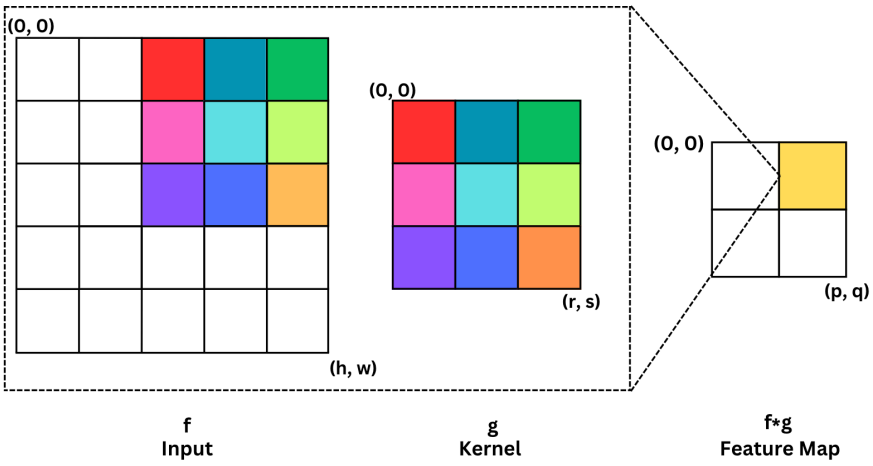

Figure 3.5: Strided cross-correlation operation at (0, 1).

Mathematically, we can update our cross-correlation operation by changing the accessing function, which will change the size function. Previously, we had that $a(i, j, \delta) \triangleq i + \delta$. Now, we add a fourth parameter related to the stride. Thus, the new accessing function is $a(i, j, \delta, m) \triangleq i \cdot m + \delta$. Suppose our horizontal stride is u and vertical stride is v. Then, our new formulation is

$$\mathbf{o}[b, z, i, j] = \sum_{\alpha=0}^{c-1} \sum_{\beta=0}^{r-1} \sum_{\gamma=0}^{s-1} \mathbf{x}\left[b, \alpha, a\left(i, r, \beta, u\right), a\left(j, s, \gamma, v\right)\right] \mathbf{F}\left[z, \alpha, \beta, \gamma\right]$$

for $(b, z, i, j) \in \{0, \ldots, n-1\} \times \{0, \ldots, k-1\} \times \{0, \ldots, p-1\} \times \{0, \ldots, q-1\}$. Since the stride has changed, the size function will also decrease by a factor of the stride. Thus, our new size function is $d\left(i, j, m\right) \triangleq \frac{i-j+1}{m}$. Then, we have that $p = d(h, r, u)$ and $q = d(w, s, v)$. In the case, where $u \nmid h - r + 1$ or $v \nmid w - s + 1$, there is a slight issue because the filter does not fit evenly over the image. This can be resolved through zero padding, which we will introduce next.

Strides of greater than 1 are often used to decrease computational complexity. As shown above, the amount of computation is reduced by a factor of uv. Furthermore, they decrease the spatial dimension size of the output feature map, as represented by p and q. As such, convolutions with strides greater than 1 are a type of "downsampling" operation, where the spatial dimension is scaled down by some factor. This decreases the amount of computation needed in subsequent operations, since the feature map has been scaled down.

Zero Padding

As alluded to in the previous section, there may be instances where the filter does not exactly cover a receptive field of the input, particularly on its borders. In such cases, we may want to "partially" apply the filter to as much of the receptive field as possible. In such cases, we might extend the receptive field with zeros, so that the filter can be fully applied to the region. This motivates the idea of zero padding, which adds zeros to the borders of the input image. We give a visual perspective of this in Figure 3.6. Here, we add zero-padding of height 1, which is applied to both the top and bottom of the image. Similarly, we apply zero-padding of width 1, which is applied to both to the left and right of the image.

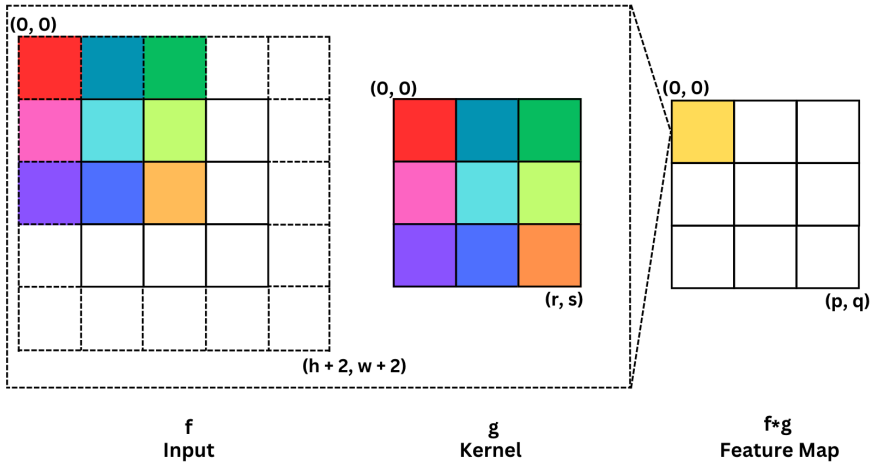

Figure 3.6: Cross-correlation on zero padded input.

Now, we formalize this idea. Given an input batch of images $\mathbf{x} \in \mathbb{R}^{n \times c \times h \times w}$, suppose we add zero-padding of height p_h and width p_w. Then, we define $\tilde{\mathbf{x}} \in \mathbb{R}^{n \times c \times (h+2p_h) \times (w+2p_w)}$, which will be the new zero-padded image where $\tilde{\mathbf{x}}[b, \alpha, \beta + p_h, \gamma + p_w] = \mathbf{x}[b, \alpha, \beta, \gamma]$ for all $b, \alpha, \beta, \gamma \in \{0, \dots, n-1\} \times \{0, \dots, c-1\} \times \{0, \dots, h-1\} \times \{0, \dots, w-1\}$ and 0 otherwise. Then, we can just apply the typical cross-correlation operation to get the output, $\tilde{\mathbf{x}} \star \mathbf{F}$. Thus, we can just treat padding as creating a new input, but the fundamental operation remains the same. Therefore, in all further discussions, we will only consider the padding variation, as the padding variation can essentially be "folded" into the input.

3.1.2 Training

Suppose we have an input $\mathbf{x} \in \mathbb{R}^{n \times c \times h \times w}$ and filter $\mathbf{F} \in \mathbb{R}^{k \times c \times r \times s}$. Then, let $\mathbf{o} = \mathbf{x} \star \mathbf{F} \in \mathbb{R}^{n \times k \times p \times q}$. As discussed in Chapter 1, it is sufficient to find $\nabla_{\mathbf{x}} \mathcal{L}$ and $\nabla_{\mathbf{F}} \mathcal{L}$ in terms of $\nabla_{\mathbf{o}} \mathcal{L}$ for loss function $\mathcal{L}$.

Recall that

$$\mathbf{o}[b,z,i,j] = \sum_{\alpha=0}^{c-1}\sum_{\beta=0}^{r-1}\sum_{\gamma=0}^{s-1}\mathbf{x}\left[b,\alpha,a\left(i,r,\beta,u\right),a\left(j,s,\gamma,v\right)\right]\mathbf{F}\left[z,\alpha,\beta,\gamma\right]$$

for $(b,z,i,j) \in \{0,\ldots,n-1\}\times\{0,\ldots,k-1\}\times\{0,\ldots,p-1\}\times\{0,\ldots,q-1\}$. First, we find $\nabla_{\mathbf{F}}\mathcal{L}$. For some $(b,z,i,j) \in \{0,\ldots,n-1\}\times\{0,\ldots,k-1\}\times\{0,\ldots,p-1\}\times\{0,\ldots,q-1\}$ and $(\alpha,\beta,\gamma) \in \times\{0,\ldots,c-1\}\times\{0,\ldots,r-1\}\times\{0,\ldots,s-1\}$, we have

$$\left(\nabla_{\mathbf{F}}\mathbf{o}[b,z,i,j]\right)\left[z,\alpha,\beta,\gamma\right] = \mathbf{x}[b,\alpha,a\left(i,r,\beta,u\right),a\left(j,s,\gamma,v\right)].$$

Then, summing over all $(b,i,j) \in \{0,\ldots,n-1\}\times\{0,\ldots,p-1\}\times\{0,\ldots,q-1\}$ and applying the chain rule, we get the following.

$$\begin{aligned}\left(\nabla\mathcal{L}\right)\left[z,\alpha,\beta,\gamma\right] &= \sum_{b=0}^{n-1}\sum_{i=0}^{p-1}\sum_{j=0}^{q-1}\mathbf{x}[b,\alpha,a\left(i,r,\beta,u\right),a\left(j,s,\gamma,v\right)]\frac{\partial\mathcal{L}}{\partial\mathbf{o}[b,z,i,j]}\\ &= \sum_{b=0}^{n-1}\sum_{i=0}^{p-1}\sum_{j=0}^{q-1}\mathbf{x}[b,\alpha,a\left(i,r,\beta,u\right),a\left(j,s,\gamma,v\right)]\left(\nabla_{\mathbf{o}}\mathcal{L}\right)\left[b,z,i,j\right]\end{aligned}$$

This formulation looks very similar to a cross-correlation operation. Indeed, for efficient implementation, it can be implemented using a cross-correlation, but we will leave the specifics beyond the scope of this thesis.

Now, we find $\nabla_{\mathbf{x}}\mathcal{L}$. For some $(b,z,i,j) \in \{0,\ldots,n-1\}\times\{0,\ldots,k-1\}\times\{0,\ldots,p-1\}\times\{0,\ldots,q-1\}$ and $(\alpha,\beta,\gamma) \in \{0,\ldots,c-1\}\times\{0,\ldots,h-1\}\times\{0,\ldots,w-1\}$, we have

$$\left(\nabla_{\mathbf{x}}\mathbf{o}[b,z,i,j]\right)\left[b,\alpha,\beta,\gamma\right] = \mathbf{F}[z,\alpha,a^{-1}\left(i,r,\beta,u\right),a^{-1}\left(j,s,\gamma,v\right)].$$

Here, $a^{-1}\left(i,r,\beta,u\right) \triangleq \frac{i+r-\beta}{u}$. We call this function a^{-1}, since it is like a reverse accessing function. Previously a was used to go from a filter location to its corresponding input location. Now, a^{-1} is used to go from the input location to the corresponding filter location. Then, summing over all $(z,i,j) \in \{0,\ldots,k-1\}\times\{0,\ldots,p-1\}\times\{0,\ldots,q-1\}$ and applying the chain rule, we get the following.

$$\begin{aligned}\left(\nabla_{\mathbf{x}}\mathcal{L}\right)\left[b,\alpha,\beta,\gamma\right] &= \sum_{z=0}^{k-1}\sum_{i=0}^{p-1}\sum_{j=0}^{q-1}\mathbf{F}[z,\alpha,a^{-1}\left(i,r,\beta,u\right),a^{-1}\left(j,s,\gamma,v\right)]\frac{\partial\mathcal{L}}{\partial\mathbf{o}[b,z,i,j]}\\ &= \sum_{b=0}^{n-1}\sum_{i=0}^{p-1}\sum_{j=0}^{q-1}\mathbf{F}[z,\alpha,a^{-1}\left(i,r,\beta,u\right),a^{-1}\left(j,s,\gamma,v\right)]\left(\nabla_{\mathbf{o}}\mathcal{L}\right)\left[b,z,i,j\right]\end{aligned}$$

This formulation is actually very similar to the true convolution operation. This is because our accessing function is now inverted, so we iterate over the filter in the reverse direction. Indeed, it is typically implemented as a convolution, but we again leave the specifics beyond the scope of this thesis.

3.1.3 Implementation

Now, we discuss efficient implementation of the convolution operation. There are quite a few options, and the preferred approach is typically dependent on the dimensions of the image and filter. Thus, we will briefly discuss three common approaches. We will leave most of the tedious details beyond the scope of this thesis.

Naive Implementation

Suppose we have an input $\mathbf{x} \in \mathbb{R}^{n \times c \times h \times w}$ and filter $\mathbf{F} \in \mathbb{R}^{k \times c \times r \times s}$. Then, let $\mathbf{o} = \mathbf{x} \star \mathbf{F} \in \mathbb{R}^{n \times k \times p \times q}$. Then, recall that

$$\mathbf{o}[b, z, i, j] = \sum_{\alpha=0}^{c-1} \sum_{\beta=0}^{r-1} \sum_{\gamma=0}^{s-1} \mathbf{x}\left[b, \alpha, a\left(i, r, \beta, u\right), a\left(j, s, \gamma, v\right)\right] \mathbf{F}\left[z, \alpha, \beta, \gamma\right]$$

for $(b, z, i, j) \in \{0, \ldots, n-1\} \times \{0, \ldots, k-1\} \times \{0, \ldots, p-1\} \times \{0, \ldots, q-1\}$. To find $\mathbf{o}$, one could just follow this equation directly. That is for each $(b, z, i, j) \in \{0, \ldots, n-1\} \times \{0, \ldots, k-1\} \times \{0, \ldots, p-1\} \times \{0, \ldots, q-1\}$, this sum is computed. This is often referred to as the naive implementation. As one can see, each sum requires three loops (one over the channel dimension and one for each of the spatial dimensions). Since $\mathbf{o}$ has four dimensions, this naive implementation requires seven nested loops to compute the output $\mathbf{o}$. While this can be optimized, optimizing this operation is a little unnatural, since it is not a commonly used deep learning operation beyond CNNs. This motivates the use of implicit GEMM.

Implicit GEMM

We have already seen in Chapter 1 that matrix multiplications have been heavily optimized. Thus, we can reframe the cross-correlation as a matrix multiplication. This transformation is often called an implicit GEMM. In order for the implicit GEMM to be implemented, the inputs need to be transformed in a way such that matrix multiplication yield the correct output. The key idea is the following. Suppose we have an input $\mathbf{x} \in \mathbb{R}^{n \times c \times h \times w}$ and filter $\mathbf{F} \in \mathbb{R}^{k \times c \times r \times s}$. Then, we will transform the filter matrix to $\tilde{\mathbf{F}} \in \mathbb{R}^{k \times crs}$ and transform the input matrix to $\tilde{\mathbf{x}} \in \mathbb{R}^{crs \times npq}$, where p and q are the output size. The transformation of the

kernel and input image can be done via im2col or im2row transformations. We defer these details to [CWV+14].

Fast Fourier Transform (FFT)

Another potential implementation approach for the cross-correlation operation relies on calculating cross-correlation operation in the Fourier domain by applying the Convolution Theorem. In particular, it uses a fast fourier transform to transform the input and kernel to the Fourier domain, compute a pointwise multiplication in this Fourier domain, then transform the input back to the spatial domain again using a fast fourier transform. We will consider such approach to be beyond the scope of the thesis, as it requires a more extensive background in signal processing and Fourier analysis. For a complete treatment of this optimization, see [MHL14]. It should be noted though that libraries such as cuFFT and cuDNN do include this approach as a potential option for implementing the convolution primitive. However, it is not as frequently used as the naive or im2col implementations due to shared memory constraints.

3.2 Pooling

As motivated in the previous section, pooling allows us to combine neighboring pixels in a feature map. The cross-correlation operation allows us to identify specific spatial patterns. Then, the pooling operation allows us to propagate this signal forward. The motivation for pooling comes from the idea that once a specific pattern in an image is identified, it is generally much less important where it is located given that images can be taken from many different perspectives. For example, detecting the presence of headlights from a car is vital to classification regardless of the location of the headlights in the image. Pooling allows us to "blur" this notion of location.

3.2.1 Formulation

At its core, pooling uses a pooling function to distill the activations of a local receptive field to a single number. In particular, we can think of a pooling function $P : \mathbb{R}^{r \times s} \to \mathbb{R}$. This is similar to the filter that we used for a cross-correlation operation, where it acts on a local receptive field. The way in which the pooling function is defined can yield variants. Below, we will discuss two such variants. There are certainly other variants that can be constructed, but these are the most common in the literature.

Once we have defined a pooling function, we can define a pooling layer as follows. Let $\mathbf{P} : \mathbb{R}^{n \times c \times h \times w} \to \mathbb{R}^{n \times c \times p \times q}$. Using the same notation as the cross-correlation operation, we

will define a horizontal stride u and a vertical stride v. Then, we have that $p = d(h, r, u)$ and $q = d(w, s, v)$, where the size function is defined as $d(i, j, m) \triangleq \frac{i-j+1}{m}$, exactly the same as in the cross correlation operation. Let the output be $\mathbf{o} = \mathbf{P}(\mathbf{x}) \in \mathbb{R}^{n \times c \times p \times q}$ for $\mathbf{x} \in \mathbb{R}^{n \times c \times h \times w}$. Then, we have that

$$\mathbf{o}[b, z, i, j] = P(\mathbf{x}[b, z, a(i, r, u) : a(i, r, u) + r, a(j, s, v) : a(j, s, v) + s]).$$

Here, the accessing function is defined as $a(i, j, m) = i \cdot m$, which is similar in spirit to that for the cross-correlation operation. The only difference is that in pooling the input to the pooling function is a matrix itself. Thus, we adjust the accessing function slightly to only give the upper-left index of the matrix. Intuitively, we can imagine the pooling operation like the cross-correlation operation where a filter traverses the input image over the spatial dimension. In fact, we can view the 2-D cross-correlation as a pooling operation, where the pooling function is defined as

$$\mathbf{A} \mapsto \sum_{(i,j) \in \{1,\ldots,r\} \times \{1,\ldots,s\}} \mathbf{A}[i, j]\mathbf{F}[i, j]$$

for a filter $\mathbf{F} \in \mathbb{R}^{r \times s}$. Thus, the pooling operation "pools" the information of a receptive field.

As with cross-correlation, when the stride is greater than 1, the operation decreases the input feature map in the spatial dimensions. Thus, the pooling operation is often referred to as a "downsampling" or "subsampling" operation. This is the terminology used when discussed in [LBBH98].

Max Pooling

In max pooling, as the name suggests, we take the pooling function to be the maximum over the input receptive field. More formally, we have that $P : \mathbb{R}^{r \times s} \to \mathbb{R}$ to be defined by

$$\mathbf{A} \mapsto \max_{(i,j) \in \{1,\ldots,r\} \times \{1,\ldots,s\}} \mathbf{A}_{(i,j)}.$$

The advantage of max pooling is that it is robust to noise or variation in the input feature map. In particular, activations that are not the maximum in their given receptive fields can be perturbed slightly with no real effect on the output. This insight becomes critical when we discuss ideas like quantization and reliability. On the other hand, max pooling is often seen as an "aggressive" form of pooling, since it essentially discards all other values that are not the greatest within a receptive field. Thus, the cross-correlation operation may

have detected a weaker pattern match, but the max pooling layer will simply discard that information.

Average Pooling

As stated above, a disadvantage of max pooling is that it discards potentially useful information. An alternative is to use average pooling, which takes the average instead of the maximum. Thus, we have the pooling function $P : \mathbb{R}^{r \times s} \to \mathbb{R}$ to be defined by

$$\mathbf{A} \mapsto \frac{1}{rs} \sum_{(i,j) \in \{1,\ldots,r\} \times \{1,\ldots,s\}} \mathbf{A}_{(i,j)}.$$

Although average pooling is seen as a "softer" pooling scheme, it has the disadvantage that it incorporates all the weights in the local receptive field. Thus, it does not have the nice property of max pooling that it is robust to perturbations.

3.2.2 Training

For training, it suffices to find the gradient of the pooling function P with respect to its inputs $\mathbf{A}$, since these pooling functions are not parameterizable.

Max Pooling

For max pooling, we have that

$$\mathbf{A} \mapsto \max_{(i,j) \in \{1,\ldots,r\} \times \{1,\ldots,s\}} \mathbf{A}_{(i,j)},$$

so

$$\left(\nabla_{\mathbf{A}} P\right)[i,j] = \begin{cases} 1 & \mathbf{A}_{(i,j)} = \max_{(i,j) \in \{1,\ldots,r\} \times \{1,\ldots,s\}} \mathbf{A}_{(i,j)} \\ 0 & \mathbf{A}_{(i,j)} \neq \max_{(i,j) \in \{1,\ldots,r\} \times \{1,\ldots,s\}} \mathbf{A}_{(i,j)} \end{cases}$$

for all $(i,j) \in \{1,\ldots,r\} \times \{1,\ldots,s\}$. In other words, if the element is the maximum, then the gradient with respect to that element is 1. Otherwise, the gradient is 0.

Average Pooling

For average pooling, we have that

$$\mathbf{A} \mapsto \frac{1}{rs} \sum_{(i,j) \in \{1,\ldots,r\} \times \{1,\ldots,s\}} \mathbf{A}_{(i,j)},$$

so

$$(\nabla_{\mathbf{A}} P)[i,j] = \frac{1}{rs}$$

for all $(i,j) \in \{1,\dots,r\} \times \{1,\dots,s\}$.

3.2.3 Implementation

Pooling operations are naturally parallelizable, since we can consider each "window" and perform the desired computation. There is no dependency on neighboring windows, so each computation can be executed individually.

3.3 Batch Normalization

Introduced in [IS15], batch normalization (BN) is ubiquitously used in CNN models. Batch normalization was motivated by the observation presented in [LBOM12] that models converge more quickly when the inputs are normalized. Consider a single perceptron with input $\mathbf{x}$, weights $\mathbf{w}$, and output $\mathbf{y}$. Then, we have that

$$\nabla_{\mathbf{w}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \mathbf{x}.$$

If all the entries of $\mathbf{x}$ are the same sign, then all of the updates to $\mathbf{w}$ will be the same sign. This makes convergence slower because the updating of the weights cannot move in different directions. Thus, this motivates normalizing the inputs. Batch normalization addresses the problem of "internal covariate shift", where activations distributions start to change because of updates in previous layers. By using batch normalization, we can allow the input for each layer to be normalized, so that the input distribution is similar.

Normalizing the inputs is sometimes referred to as "whitening", and it is a commonly used technique. However, naively whitening data leads to a few problems. Suppose we have an input to a layer $\mathbf{x}$. Then, let $\mathcal{X}$ be the set of the inputs to the layer for eah data point. Then, we have that the whitened input is

$$\hat{\mathbf{x}} = \text{Norm}(\mathbf{x}, \mathcal{X}) = \hat{\boldsymbol{\Sigma}}^{-\frac{1}{2}} (\mathbf{x} - \hat{\boldsymbol{\mu}}),$$

where $\hat{\boldsymbol{\mu}}$ is the sample mean, defined by

$$\hat{\boldsymbol{\mu}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i,$$

and $\hat{\boldsymbol{\Sigma}}$ is the sample covariance matrix, defined by

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{N}\sum_{i=1}^{N}\left(\mathbf{x}_i - \hat{\boldsymbol{\mu}}\right)\left(\mathbf{x}_i - \hat{\boldsymbol{\mu}}\right)^{\mathsf{T}}.$$

However, this naive formulation is infeasible for a few reasons. First, finding the sample mean and covariance matrix is expensive because we need to iterate over all data points $\mathbf{x}_i \in \mathcal{X}$. Second, the covariance matrix grows quadratically in the number of inputs, making it expensive to compute and store. Finally, in backpropagation, we need to compute $\frac{\partial \operatorname{Norm}(\mathbf{x},\mathcal{X})}{\partial \mathbf{x}}$ as well as $\frac{\partial \operatorname{Norm}(\mathbf{x},\mathcal{X})}{\partial \mathcal{X}}$. Thus, we need to compute gradients for all of the data points even in a single backwards pass. This motivates the idea of batch normalization, where we normalize over a batch $\mathcal{B}$ instead of the entire dataset $\mathcal{D}$.

3.3.1 Formulation

For a channel k, we define $\mathrm{BN}^{(k)}_{\gamma^{(k)},\beta^{(k)}} : \mathbb{R} \times \mathbb{R}^{n\times h\times w} \to \mathbb{R}$ as

$$\mathrm{BN}^{(k)}_{\gamma^{(k)},\beta^{(k)}}(x,\mathcal{B}) = \gamma^{(k)} \cdot \left(\frac{x - \hat{\mu}^{(k)}_{\mathcal{B}}}{\sqrt{\left(\hat{\sigma}^2_{\mathcal{B}}\right)^{(k)} + \epsilon}}\right) + \beta^{(k)},$$

where

$$\hat{\mu}^{(k)}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|hw}\sum_{i=0}^{|\mathcal{B}|-1}\sum_{r=0}^{h-1}\sum_{s=0}^{w-1}\mathbf{x}_{ikrs}$$

is defined as the sample mean of the batch,

$$\left(\hat{\sigma}^2_{\mathcal{B}}\right)^{(k)} = \frac{1}{|\mathcal{B}|hw}\sum_{i=0}^{|\mathcal{B}|-1}\sum_{r=0}^{h-1}\sum_{s=0}^{w-1}\left(\mathbf{x}_{ikrs} - \hat{\mu}^{(k)}_{\mathcal{B}}\right)^2$$

is defined as the sample variance of the batch, and ϵ is some small pre-defined constant.

This formulation makes the following changes to the naive whitening approach. First, we do not normalize over the entire joint distribution of the input vector $\mathbf{x}$. Instead, we whiten over each feature map individually, which we denote as $\mathbf{x}^{(k)}$ for all $k \in \{1,\ldots,c\}$. We also assume that elements in the same feature map come from a similar distribution. This allows us to avoid the problem of constructing the empirical covariance matrix. Thus, the input to the batch normalization function is a single real number. Next, we introduce the parameters $\gamma^{(k)}$ and $\beta^{(k)}$. This gives the model the flexibility to represent the input activations without whitening, so that the new model has the same level of expressivity. If $\gamma^{(k)} = \sqrt{\left(\hat{\sigma}^2_{\mathcal{B}}\right)^{(k)} + \epsilon}$

and $\beta^{(k)} = \hat{\mu}_{\beta}^{(k)}$. Then, we would have that

$$\mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}^{(k)}\left(x, \mathcal{B}\right) = x,$$

giving the batch normalization function the ability to just act as the identity. This is a powerful concept that we will see again in the discussion of residual networks, where we want to give the network the flexibility to do nothing. Finally, we introduce ϵ. This is mainly for numerical stability purposes, since $\hat{\sigma}_{\mathcal{B}}^2$ may be close to zero, which may lead to very unstable whitened outputs. Thus, ϵ ensures that we do not divide by zero or something close to zero.

3.3.2 Training

We have that $\gamma^{(k)}$ and $\beta^{(k)}$ are trainable parameters. Thus, using backpropagation, it suffices to calculate $\frac{\partial \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}^{(k)}(\mathbf{x}_{bkst},\mathcal{B})}{\partial \gamma^{(k)}}$, $\frac{\partial \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}^{(k)}(\mathbf{x}_{bkst},\mathcal{B})}{\partial \beta^{(k)}}$, and $\frac{\partial \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}^{(k)}(\mathbf{x}_{bkst},\mathcal{B})}{\partial \mathbf{x}_{akuv}^{(k)}}$ for all $(b,s,t),(a,u,v) \in \{0,\ldots,|\mathcal{B}|-1\} \times \{0,\ldots,h-1\} \times \{0,\ldots,w-1\}$. We have that

$$\frac{\partial \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}^{(k)}\left(\mathbf{x}_{bkst}, \mathcal{B}\right)}{\partial \gamma^{(k)}} = \frac{\mathbf{x}_{bkst} - \hat{\mu}_{\mathcal{B}}^{(k)}}{\sqrt{\left(\hat{\sigma}_{\mathcal{B}}^2\right)^{(k)} + \epsilon}}$$

and

$$\frac{\partial \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}^{(k)}\left(\mathbf{x}_{bkst}, \mathcal{B}\right)}{\partial \beta^{(k)}} = 1$$

for all $(b,s,t) \in \{0,\ldots,|\mathcal{B}|-1\} \times \{0,\ldots,h-1\} \times \{0,\ldots,w-1\}$.

Note that

$$\frac{\partial \hat{\mu}_{\mathcal{B}}^{(k)}}{\partial \mathbf{x}_{bkst}} = \frac{1}{|\mathcal{B}|hw}$$

for all $(b,s,t) \in \{0,\ldots,|\mathcal{B}|-1\} \times \{0,\ldots,h-1\} \times \{0,\ldots,w-1\}$. Furthermore, we have that

$$\frac{\partial \left(\hat{\sigma}_{\mathcal{B}}^2\right)^{(k)}}{\partial \mathbf{x}_{bkst}} = \frac{2}{|\mathcal{B}|hw}\left(\mathbf{x}_{bkst} - \hat{\mu}_{\mathcal{B}}^{(k)}\right)$$

for all $(b,s,t) \in \{0,\ldots,|\mathcal{B}|-1\} \times \{0,\ldots,h-1\} \times \{0,\ldots,w-1\}$.

Then, we have the following.

$$\frac{\partial \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}^{(k)}\left(\mathbf{x}_{bkst}, \mathcal{B}\right)}{\partial \mathbf{x}_{bkst}} = \frac{\gamma^{(k)}}{\sqrt{\left(\hat{\sigma}_{\mathcal{B}}^2\right)^{(k)} + \epsilon}}\left(1 - \frac{\partial \hat{\mu}_{\mathcal{B}}^{(k)}}{\partial \mathbf{x}_{bkst}}\right) + \left(\mathbf{x}_{bkst} - \hat{\mu}_{\mathcal{B}}^{(k)}\right) \cdot \left(-\frac{1}{2} \cdot \frac{1}{\left(\left(\hat{\sigma}_{\mathcal{B}}^2\right)^{(k)} + \epsilon\right)^{\frac{3}{2}}} \cdot \frac{\partial \left(\hat{\sigma}_{\mathcal{B}}^2\right)^{(\ell)}}{\partial \mathbf{x}_{bkst}}\right)$$

Furthermore, we have the following for $(a,u,v) \neq (b,s,t)$.

$$\frac{\partial \mathrm{BN}^{(k)}_{\gamma^{(k)},\beta^{(k)}}\left(\mathbf{x}_{bkst},\mathcal{B}\right)}{\partial \mathbf{x}_{akuv}} = -\frac{\gamma^{(k)}}{\sqrt{\left(\hat{\sigma}^2_{\mathcal{B}}\right)^{(k)}+\epsilon}}\frac{\partial \hat{\mu}^{(k)}_{\mathcal{B}}}{\partial \mathbf{x}_{akuv}} + \left(\mathbf{x}_{bkst}-\hat{\mu}^{(k)}_{\mathcal{B}}\right)\cdot\left(-\frac{1}{2}\cdot\frac{1}{\left(\left(\hat{\sigma}^2_{\mathcal{B}}\right)^{(k)}+\epsilon\right)^{\frac{3}{2}}}\cdot\frac{\partial\left(\hat{\sigma}^2_{\mathcal{B}}\right)^{(\ell)}}{\partial \mathbf{x}_{akuv}}\right)$$

This gives all of the necessary partial derivatives required to execute backpropagation.

3.3.3 Implementation

Running Inference

Now, we discuss efficient implementation of the forward pass. Suppose that we have trained the model over batches $\{\mathcal{B}_1,\ldots,\mathcal{B}_m\}$. Then, when running inference, we do not want to require that the data is batched. Thus, for $\hat{\mu}^{(k)}_{\mathcal{B}}$ and $\left(\hat{\sigma}^2_{\mathcal{B}}\right)^{(k)}$, we need to replace these with constant values. Thus, we use

$$\hat{\mu}^{(k)} = \frac{1}{m}\sum_{i=1}^{m}\hat{\mu}^{(k)}_{\mathcal{B}_i}$$

and

$$\left(\hat{\sigma}^2\right)^{(k)} = \frac{1}{m-1}\sum_{i=1}^{m}\left(\hat{\sigma}^2_{\mathcal{B}_i}\right)^{(k)}.$$

Here, we used the unbiased estimates for the mean and variance over the batches. Now, we define $\mathrm{BN}^{(k)}_{\gamma,\beta,\hat{\mu},\hat{\sigma}^2}:\mathbb{R}\to\mathbb{R}$ by

$$\mathrm{BN}^{(k)}_{\gamma,\beta,\hat{\mu},\hat{\sigma}^2}\left(x\right) = \gamma^{(k)}\frac{x-\hat{\mu}^{(k)}}{\sqrt{\left(\hat{\sigma}^2\right)^{(k)}+\epsilon}}+\beta^{(k)}$$

to be used for inference. Then, we apply $\mathrm{BN}^{(k)}_{\gamma^{(k)},\beta^{(k)},\hat{\mu}^{(k)},\left(\hat{\sigma}^2\right)^{(k)}}$ to the all the elements in the kth feature map, which we will refer to as $\mathrm{BN}_{\gamma^{(k)},\beta^{(k)},\hat{\mu}^{(k)},\left(\hat{\sigma}^2\right)^{(k)}}$.

Folding

To speed up BN inference, the prevailing technique is to use BN folding. BN folding arises from the following observation. Typically, in CNN architectures, the BN operation follows a cross-correlation operation and a bias operation. Thus, suppose, we have some input $\mathbf{x}\in\mathbb{R}^{n\times c\times h\times w}$, some filter $\mathbf{F}\in\mathbb{R}^{k\times c\times r\times s}$, and some bias $\mathbf{b}\in\mathbb{R}^{k\times p\times q}$. Now, suppose we apply BN to the jth feature map. Then, for the bth input and jth feature map, we have

that the output is

$$\mathrm{BN}_{\gamma^{(j)},\beta^{(j)},\hat{\mu}^{(j)},(\hat{\sigma}^2)^{(j)}}\left(\mathbf{x}[b,:,:,:] \star \mathbf{F}[j,:,:,:] + \mathbf{b}[j,:,:,:]\right),$$

where $\mathbf{x}[b,:,:,:]$ is the bth input of the batch, $\mathbf{F}[j,:,:,:]$ is the jth filter and $\mathbf{b}[j,:,:]$ is the bias for the jth feature map. Then, we have the following for the output at index (b,j,s,t) for $(s,t) \in \times\{0,\ldots,p-1\} \times \{0,\ldots,q-1\}$. Let $\mathbf{o} = \mathbf{x} \star \mathbf{F} \in \mathbb{R}^{n\times k\times p\times q}$.

$$\begin{aligned}
&\mathrm{BN}_{\gamma^{(j)},\beta^{(j)},\hat{\mu}^{(j)},(\hat{\sigma}^2)^{(j)}}\left(\mathbf{x} \star \mathbf{F}[j,:,:] + \mathbf{b}[j,:,:]\right)[b,j,s,t] \\
&\quad = \gamma^{(j)} \frac{\mathbf{o}[b,j,s,t] + \mathbf{b}[j,s,t] - \hat{\mu}^{(j)}}{\sqrt{(\hat{\sigma}^2)^{(j)} + \epsilon}} + \beta^{(j)} \\
&\quad = \frac{\gamma^{(j)}}{\sqrt{(\hat{\sigma}^2)^{(j)} + \epsilon}} \mathbf{o}[b,j,s,t] + \left(\frac{\mathbf{b}[j,s,t]}{\sqrt{(\hat{\sigma}^2)^{(j)} + \epsilon}} - \frac{\hat{\mu}^{(j)}}{\sqrt{(\hat{\sigma}^2)^{(j)} + \epsilon}} + \beta^{(j)} \right)
\end{aligned}$$

Thus, to get our new output, we see that we can just linearly transform the original output. In particular, this means that we can scale all the weights in the jth feature map of filter $\mathbf{F}$ by $\frac{\gamma^{(j)}}{\sqrt{(\hat{\sigma}^2)^{(j)}+\epsilon}}$ and change the bias by dividing by $\sqrt{(\hat{\sigma}^2)^{(j)} + \epsilon}$ and adding the term $-\frac{\hat{\mu}^{(j)}}{\sqrt{(\hat{\sigma}^2)^{(j)}+\epsilon}} + \beta^{(j)}$. Thus, this BN operation can be "folded" into the previous cross-correlation layer by updating the weights. Updating the weights can be done offline, so having a BN operation adds no additional cost in the inference routine. Therefore, we do not need to discuss efficient implementation of the BN operation, since it is done entire in the cross-correlation operation with no additional cost.

Due to this folding property, BN can be seen as a type of "weight normalization", assuming that the input is whitened. Assuming that the input is whitened, then we have that the variance of the output is proportional to the variance of the weights. Since we divide out by the variance of the output for the filter weights, this is equivalent to normalizing the weights. While we do not explore this idea rigorously in this thesis, one can refer to [SK16] for a more detailed treatment of this topic.

3.4 Softmax

As mentioned previously, given $\mathbf{x}$, we want to find $\boldsymbol{\pi}_{\mathbf{x}}$, which is the parameter for a categorical probability distribution. Thus, we need each $\boldsymbol{\pi}_{\mathbf{x},i} \in (0,1)$ for $i \in \{1,\ldots,K\}$. Furthermore, we need that

$$\sum_{i=1}^{K} \boldsymbol{\pi}_{\mathbf{x},i} = 1,$$

since the probabilities should sum to 1. The softmax operation, often notated as σ allows us to perform this normalization of a real-valued output vector into a categorical probability distribution.

3.4.1 Formulation

More rigorously, we have that $\sigma : \mathbb{R}^K \to (0,1)^K$ is defined by

$$\left(\sigma\left(\mathbf{z}\right)\right)_i = \frac{\exp\left(\mathbf{z}_i\right)}{\sum_{j=1}^{K} \exp\left(\mathbf{z}_j\right)}$$

for all $i \in \{1, \ldots, K\}$. Intuitively, we have that the softmax function normalizes the output value using the exponential values. From inspection, we can see that $0 < \left(\sigma\left(\mathbf{z}\right)\right)_i < 1$, since $0 < \exp\left(\mathbf{z}_i\right) < \sum_{j=1}^{K} \exp\left(\mathbf{z}_j\right)$ for all $i \in \{1, \ldots, K\}$. Furthermore, we have that

$$\sum_{i=1}^{K} \left(\sigma\left(\mathbf{z}\right)\right)_i = \sum_{i=1}^{K} \frac{\exp\left(\mathbf{z}_i\right)}{\sum_{j=1}^{K} \exp\left(\mathbf{z}_j\right)} = \frac{\sum_{j=1}^{K} \exp\left(\mathbf{z}_j\right)}{\sum_{j=1}^{K} \exp\left(\mathbf{z}_j\right)} = 1$$

as desired. Thus, we can simply have our model output real-valued outputs. Then, we can apply the softmax function to get $\boldsymbol{\pi}_{\mathbf{x}}$.

3.4.2 Implementation

Numerical Stability

Due to the nature of exponential functions, there is a good chance of encountering underflow ($\mathbf{z}_i$ is negative and large in magnitude) or overflow ($\mathbf{z}_i$ is positive and large in magnitude) issues with the softmax function when using any numerical format with a finite number of bits as will be discussed in Chapter 5. The overflow issues tend to be more problematic, since the underflow issues will simply cause the probabilities to go to 0. To resolve this issue, the softmax can be calculated in a more numerically stable way by shifting the output values. In particular, we have the following.

$$\begin{aligned}
\left(\sigma\left(\mathbf{z}\right)\right)_i &= \frac{\exp\left(\mathbf{z}_i\right)}{\sum_{j=1}^{K} \exp\left(\mathbf{z}_j\right)} \\
&= \frac{\exp\left(-\max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right) \exp\left(\mathbf{z}_i\right)}{\exp\left(-\max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right) \sum_{j=1}^{K} \exp\left(\mathbf{z}_j\right)} \\
&= \frac{\exp\left(\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right)}{\sum_{j=1}^{K} \exp\left(\mathbf{z}_j - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right)}
\end{aligned}$$

This formulation is desirable because we have that $\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k \leq 0$ for all $k \in \{1,\ldots,K\}$. Therefore, we have that $0 < \exp\left(\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right) \leq 1$ for all $k \in \{1,\ldots,K\}$. Thus, we will not encounter any overflow issues. There may still be underflow issues, since $\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k \leq 0$ is not bounded below. However, we have that $\mathbf{z}_{k'} - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k = 0$ for $k' = \arg\max_{k\in\{1,\ldots,K\}} \mathbf{z}_k$. Therefore, we have that $\exp\left(\mathbf{z}_{k'} - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right) = 1$. Thus, we will not encounter a divide by zero error where $\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k$ are all rounded down to 0.

To help avoid underflow issues, we typically take the logarithm and do operations on that quantity. The log function helps stabilize the softmax output.

$$\begin{aligned}
\log\left(\left(\sigma\left(\mathbf{z}\right)\right)_i\right) &= \log\left(\frac{\exp\left(\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right)}{\sum_{j=1}^{K} \exp\left(\mathbf{z}_j - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right)}\right) \\
&= \log\left(\exp\left(\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right)\right) - \log\left(\sum_{j=1}^{K} \exp\left(\mathbf{z}_j - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right)\right) \\
&= \left(\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right) - \log\left(\sum_{j=1}^{K} \exp\left(\mathbf{z}_j - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right)\right)
\end{aligned}$$

In this way, we avoid the instability of $\exp\left(\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right)$ in the initial numerator when $\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k$ has large magnitude. Although the term is still present in the second term, $-\log\left(\sum_{j=1}^{K} \exp\left(\mathbf{z}_j - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right)\right)$, its effect on the sum is likely negligible when $\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k$ has large magnitude. If desired, we can retrieve the original softmax values by simply exponentiating this quantity. Then, we have

$$\left(\sigma\left(\mathbf{z}\right)\right)_i = \exp\left(\left(\mathbf{z}_i - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right) - \log\left(\sum_{j=1}^{K} \exp\left(\mathbf{z}_j - \max_{k\in\{1,\ldots,K\}} \mathbf{z}_k\right)\right)\right),$$

which is the most commonly used implementation of the softmax function.

Parallelization

Softmax operations typically only occur after the very last layer. Thus, typically it is only applied to vectors of length K, where K is the number of classes. K is typically small, so the runtime of the softmax operation is relatively negligible. However, in Chapter 4, we will discuss an efficient piecemeal implementation of the softmax operation on vectors $\mathbf{x} \in \mathbb{R}^d$, where d is large.

Training

While the softmax function does not have any trainable parameters, we still need to find the Jacobian of the output vector $\sigma(\mathbf{z})$ with respect to the inputs $\mathbf{z}$ for the backpropagation algorithm. We have that

$$\mathbf{J}_{\sigma}(\mathbf{z}) = \begin{bmatrix} \frac{\partial(\sigma(\mathbf{z}))_1}{\partial \mathbf{z}_1} & \frac{\partial(\sigma(\mathbf{z}))_1}{\partial \mathbf{z}_2} & \cdots & \frac{\partial(\sigma(\mathbf{z}))_1}{\partial \mathbf{z}_K} \\ \frac{\partial(\sigma(\mathbf{z}))_2}{\partial \mathbf{z}_1} & \frac{\partial(\sigma(\mathbf{z}))_2}{\partial \mathbf{z}_2} & \cdots & \frac{\partial(\sigma(\mathbf{z}))_2}{\partial \mathbf{z}_K} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial(\sigma(\mathbf{z}))_K}{\partial \mathbf{z}_1} & \frac{\partial(\sigma(\mathbf{z}))_K}{\partial \mathbf{z}_2} & \cdots & \frac{\partial(\sigma(\mathbf{z}))_K}{\partial \mathbf{z}_K} \end{bmatrix}.$$

First, consider $\frac{\partial(\sigma(\mathbf{z}))_i}{\partial \mathbf{z}_i}$. Then, we have the following.

$$\begin{aligned} \frac{\partial(\sigma(\mathbf{z}))_i}{\partial \mathbf{z}_i} &= \frac{\left(\sum_{j=1}^{K} \exp(\mathbf{z}_j)\right) \exp(\mathbf{z}_i) - \exp(\mathbf{z}_i) \exp(\mathbf{z}_i)}{\left(\sum_{j=1}^{K} \exp(\mathbf{z}_j)\right)^2} \\ &= \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)} \cdot \frac{\sum_{j=1}^{K} \exp(\mathbf{z}_j) - \exp(\mathbf{z}_i)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)} \\ &= (\sigma(\mathbf{z}))_i (1 - (\sigma(\mathbf{z}))_i) \end{aligned}$$

Now, consider $\frac{\partial(\sigma(\mathbf{z}))_i}{\partial \mathbf{z}_j}$ for $i \neq j$. Then, we have the following.

$$\begin{aligned} \frac{\partial(\sigma(\mathbf{z}))_i}{\partial \mathbf{z}_j} &= -\frac{\exp(\mathbf{z}_i)}{\left(\sum_{k=1}^{K} \exp(\mathbf{z}_k)\right)^2} \exp(\mathbf{z}_j) \\ &= -\frac{\exp(\mathbf{z}_i)}{\sum_{k=1}^{K} \exp(\mathbf{z}_k)} \cdot \frac{\exp(\mathbf{z}_j)}{\sum_{k=1}^{K} \exp(\mathbf{z}_k)} \\ &= -(\sigma(\mathbf{z}))_i (\sigma(\mathbf{z}))_j \end{aligned}$$

Then, we have the following.

$$\begin{aligned} \mathbf{J}_{\sigma}(\mathbf{z}) &= \begin{bmatrix} (\sigma(\mathbf{z}))_1 - (\sigma(\mathbf{z}))_1 (\sigma(\mathbf{z}))_1 & -(\sigma(\mathbf{z}))_1 (\sigma(\mathbf{z}))_2 & \cdots & -(\sigma(\mathbf{z}))_1 (\sigma(\mathbf{z}))_K \\ -(\sigma(\mathbf{z}))_2 (\sigma(\mathbf{z}))_1 & (\sigma(\mathbf{z}))_2 - (\sigma(\mathbf{z}))_2 (\sigma(\mathbf{z}))_2 & \cdots & -(\sigma(\mathbf{z}))_2 (\sigma(\mathbf{z}))_K \\ \vdots & \vdots & \ddots & \vdots \\ -(\sigma(\mathbf{z}))_K (\sigma(\mathbf{z}))_1 & -(\sigma(\mathbf{z}))_K (\sigma(\mathbf{z}))_2 & \cdots & (\sigma(\mathbf{z}))_K - (\sigma(\mathbf{z}))_K (\sigma(\mathbf{z}))_K \end{bmatrix} \\ &= \operatorname{diag}((\sigma(\mathbf{z}))_i) - \sigma(\mathbf{z}) \sigma(\mathbf{z})^{\mathsf{T}} \end{aligned}$$

Thus, we can use this Jacobian to proceed with backpropagation as described in Chapter 1. Furthermore, rather conveniently, we can express this Jacobian in terms of matrix operations

on the original output value.

3.5 Common CNN Datasets

Given our discussion of the building blocks of CNN models and their implementations, we now discuss CNN models in full. CNN models are typically characterized by the architecture used and the training data set. First, we provide a brief overview of common datasets that have been used in the literature of quantization, sparsity, and reliability.

There are two main datasets that we will focus on: CIFAR-10 [KNH+14] and ImageNet Large Scale Visual Recognition Challenge (ILSVRC) dataset [RDS+15]. Both datasets are used for image classification tasks, where the goal is to select exactly one category for each image.

3.5.1 CIFAR-10 [KNH+14]

The CIFAR-10 is a relatively compact dataset of $60,000$ 32×32 RGB images, split evenly among ten classes with $10,000$ images per class. The ten classes are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each image is categorized in exactly one class. The dataset can be randomly partitioned into a training set consisting of $50,000$ images and a testing set of $10,000$ images. Figure 3.7 shows some sample images from CIFAR-10.

There is also a CIFAR-100 variant that contains the same $60,000$ images hierarchically categorized into 20 superclasses with 5 classes in each superclass. Thus, there are a total of 100 classes, and each image is given a "coarse" label that denotes the superclass and a "fine" label that denotes the class. The CIFAR-10 and CIFAR-100 datasets are subsets of a larger Tiny Images dataset [TFF08] that consists of 80 million images. The Tiny Images dataset was constructed based on WordNet [Mil94], a collection of English words, which groups synonymous words into synonym sets (synsets). Tiny Images is the images that result from searching a collection of noun synsets into an internet search engine.

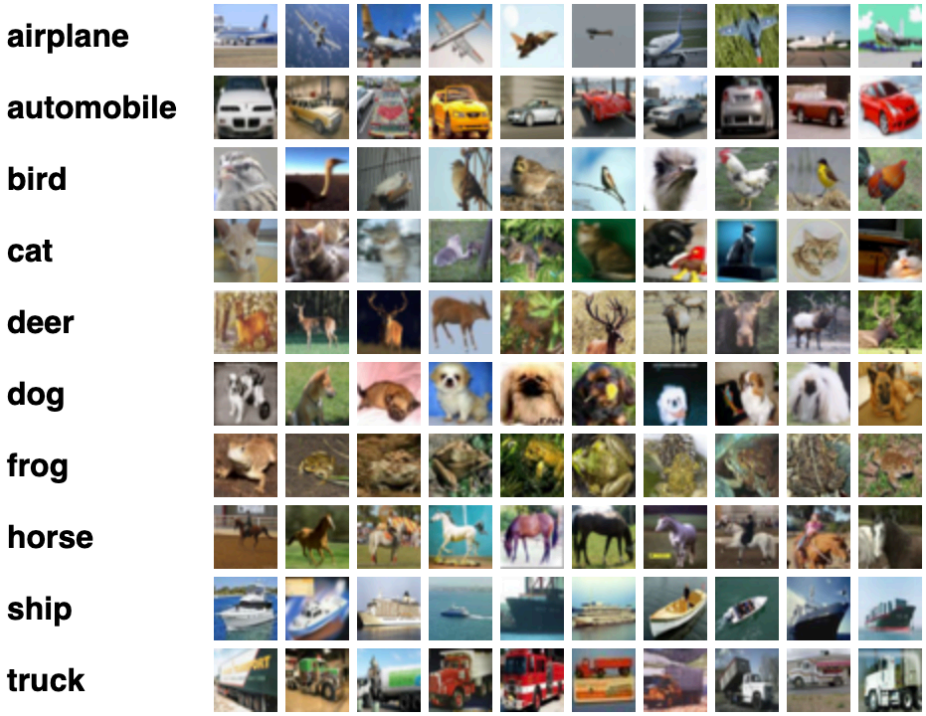

Figure 3.7: Example of CIFAR-10 dataset. [KNH+14]

3.5.2 ILSVRC Dataset [RDS+15]

The ILSVRC contains a much larger set of images compared to CIFAR-10, making it more suitable for thorough benchmarking of CNN models. It contains $1,281,167$ training images, $50,000$ validation images, and $10,000$ test images that can be categorized into $1,000$ object classes. This is significantly larger than the $60,000$ images spanning 10 classes for CIFAR-10. Furthermore, the images of ILSVRC15 are at full resolution compared to the lower resolution of CIFAR-10. The images are typically pre-processed to a size of 256×256. ILSVRC also has nice properties like bounding boxes for evaluating object detection algorithms. These types of models fall beyond the scope of this thesis.

Similar to CIFAR-10, the ILSVRC dataset is a subset of a larger dataset called ImageNet [DDS+09]. In addition, it is also based off the WordNet database. However, ImageNet boasts images of higher resolution and a broader range of synsets compared to Tiny Images. Furthermore, ImageNet used Amazon Mechanical Turk (AMT) to task human users with verifying each image making it more accurate and specific than Tiny Images which relied on search engines.

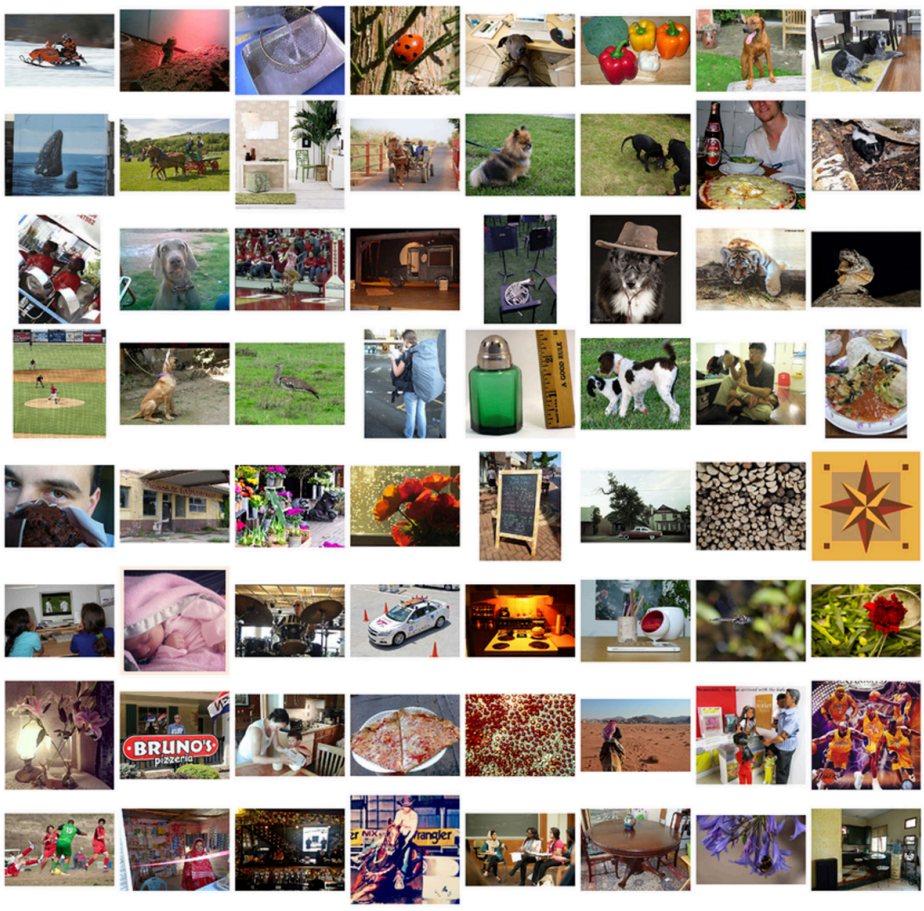

Figure 3.8: Example of ImageNet dataset. [DDS+09]

3.6 Common CNN Architectures

Now, we discuss common CNN architectures that are used for image classification. We discuss them roughly in chronological order based on when they were introduced, such that we can use drawbacks of prior models to motivate improvements in future models.

3.6.1 LeNet [LBBH98]

Architecture

Introduced in 1998, LeNet is typically cited as the first use of convolutions applied to digit recognition. Below, we show the computational diagram from [LBBH98], and then we discuss the key features. LeNet was used to classify digits from 0 to 9 using a 32×32 bitmap input.

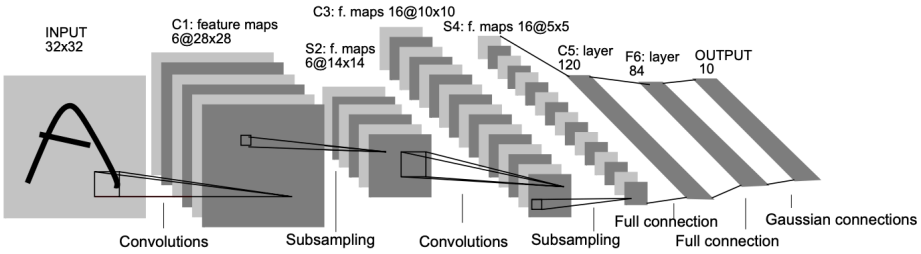

Figure 3.9: Computational diagram for LeNet. [LBBH98]

The convolution and subsampling operations are the same operations that have already been discussed above. Note that the "full connection" is a convolution with 120 filters of filter size 5×5, which yields a flattened vector in $\mathbb{R}^{120}$. LeNet uses Euclidean Radial Basis Function units (RBF) for the output layer. Let the input of the output layer be $\mathbf{x} \in \mathbb{R}^{84}$. Then, we have that

$$\mathbf{y}_i = \sum_{j=1}^{84} \left(\mathbf{x}_j - \mathbf{w}_{ij}\right)^2.$$

LeNet was probabilistic in spirit, with these output values corresponding to unnormalized Gaussian densities. In particular, we have that

$$\mathcal{N}\left(x; \mu, \sigma^2\right) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right).$$

This also explains the use of "Gaussian connections" to refer to the last layer.

Loss Function

LeNet also uses a probabilistic flavor of loss function with

$$\mathcal{L}\left(f\left(\mathbf{x}; \mathbf{w}\right), y\right) = f\left(\mathbf{x}; \mathbf{w}\right)_y + \log\left(\exp\left(-j\right) + \sum_{i=0}^{9} \exp\left(-f\left(\mathbf{x}; \mathbf{w}\right)_i\right)\right)$$

for some positive constant $j \in \mathbb{R}_{>0}$.

First, we verify that $\mathcal{L}\left(f\left(\mathbf{x}; \mathbf{w}\right), y\right) \geq 0$ to ensure that it is a valid loss function. We have the following.

$$\exp\left(-j\right) + \sum_{i=0}^{9} \exp\left(-f\left(\mathbf{x}; \mathbf{w}\right)_i\right) \geq \exp\left(-f\left(\mathbf{x}; \mathbf{w}\right)_y\right)$$

$$\log\left(\exp\left(-j\right)+\sum_{i=0}^{9}\exp\left(-f\left(\mathbf{x};\mathbf{w}\right)_i\right)\right)\geq -f\left(\mathbf{x};\mathbf{w}\right)_y$$

$$f\left(\mathbf{x};\mathbf{w}\right)_y+\log\left(\exp\left(-j\right)+\sum_{i=0}^{9}\exp\left(-f\left(\mathbf{x};\mathbf{w}\right)_i\right)\right)\geq 0$$

$$\mathcal{L}\left(f\left(\mathbf{x};\mathbf{w}\right),y\right)\geq 0$$

Now, we discuss the interpretation of each term. First, we have that $f\left(\mathbf{x};\mathbf{w}\right)_y$ is the associated output of the RBF for the correct label. Thus, we want this to be the output of the RBF to be small for the correct output. However, we also would like push the output of incorrect RBF units. Otherwise, we could just output low values for each RBF unit. We want to maximize

$$\frac{\exp\left(-f\left(\mathbf{x};\mathbf{w}\right)_y\right)}{\exp\left(-j\right)+\sum_{i=1}^{9}\exp\left(-f\left(\mathbf{x};\mathbf{w}\right)_i\right)},$$

which is the relative output of the correct class. One could also interpret this as the softmax-like operation applied to the outputs appended with the value j for the "rubbish" class. Note that this is not exactly the softmax function, since we first negate the inputs. Thus, we want the output of the RBF to be low for the correct label. Thus, one can view the loss function through a probabilistic lens as the likelihood of observing the correct class given by the probability above. Taking the negative log of this expression, we get the following desired expression.

$$\log\left(\frac{\exp\left(-f\left(\mathbf{x};\mathbf{w}\right)_y\right)}{\exp\left(-j\right)+\sum_{i=1}^{9}\exp\left(-f\left(\mathbf{x};\mathbf{w}\right)_i\right)}\right)=f\left(\mathbf{x};\mathbf{w}\right)_y+\log\left(\exp\left(-j\right)+\sum_{i=0}^{9}\exp\left(-f\left(\mathbf{x};\mathbf{w}\right)_i\right)\right)$$

Here, e^{-j} represents the weight of a "rubbish" class, and it pushes the true RBF output down and the incorrect RBF output upward.

3.6.2 AlexNet [KSH12]

AlexNet revived interest in CNNs by using commodity GPUs to improve training times, allowing them to train larger models and achieve higher accuracy. AlexNet revolutionized the field by training a large-scale model that had impressive performance on ILSVRC, a significantly more complex task than digit recognition. While LeNet had tens of thousands parameters, AlexNet had on the order of tens of millions of parameters. AlexNet employed various elements of the parallelization techniques that we have discuss in previous sections of this chapter to drastically training times.

Group Convolution

Before discussing the architecture of AlexNet, we discuss one of the key optimizations in AlexNet that allowed for its success. Prior to the introduction of AlexNet, training large models on GPUs was difficult because it was hard to execute training on a multi-GPU system. AlexNet solved this problem by introducing group convolutions. Let g be the number of groups. Then, AlexNet divides the input $\mathbf{x} \in \mathbb{R}^{n \times c \times h \times w}$ along the channel dimension, such that there are g inputs $\mathbf{x}_1, \ldots, \mathbf{x}_g \in \mathbb{R}^{n \times \frac{c}{g} \times h \times w}$. Similarly, we divide the filter along the channel dimension. Previously, we had that $\mathbf{F} \in \mathbb{R}^{k \times c \times r \times s}$. Now, we have g filters $\mathbf{F}_1, \ldots, \mathbf{F}_g \in \mathbb{R}^{k \times \frac{c}{g} \times r \times s}$. Then, we compute $\mathbf{x}_i \star \mathbf{F}_i$ for all $i \in \{1, \ldots, g\}$. The upshot of group convolutions is that each of these $\mathbf{x}_i \star \mathbf{F}_i$ computations can be executed independently. In particular, each can be executed on a separate GPU, and then the result can be concatenated back together in a future layer.

Local Response Normalization

AlexNet also used local response normalization, which is an alternative form of normalization. While batch normalization is predominantly used in most modern CNN architectures, we include an explanation of local response normalization for completeness. Local response normalization occurs within a single example. For an activation $\mathbf{a}[i, j, k]$, we have that the local response normalized output is

$$\mathbf{b}[i, x, y] = \frac{\mathbf{a}[i, x, y]}{\left(k + \alpha \sum_{j=\max\left(0, i - \frac{n}{2}\right)}^{\min\left(N-1, i+\frac{n}{2}\right)} \mathbf{a}[j, x, y]^2\right)^{\beta}},$$

where N is the number of kernels and k, n, α, β are pre-determined hyperparameters. Intuitively, local response normalization scales the output of kernel i at location (x, y) by all the other kernel outputs at location (x, y). Like batch normalization, the motivation for local response normalization has its roots in efficient training.

3.6.3 VGG [SZ15]

VGG experimented with the idea of making CNNs deeper by adding more layers, while using very small filters in each layer. By pushing the number of layers up to 19, they were able to achieve very competitive accuracies on ILSVRC. The underlying setup of VGG are very similar to that of AlexNet, except with much more layers. We will return to this observation shortly when introducing ResNet.

3.6.4 Residual Networks (ResNet) [HZRS15]

As mentioned in the discussion of VGG, increasing the depth of a CNN was shown to improve performance. However, [HZRS15] showed that training deeper models to convergence is particularly difficult. In particular, we typically expect deeper models to outperform shallower models, since they have strictly more expressive power. In particular, the deeper model can just use the identity map for all future layers and achieve the same accuracy as the shallower model. However, [HZRS15] observes that this does not occur. Therefore, they propose that layers should learn some representation $\mathcal{H}$ such that the output is $\mathbf{x} + \mathcal{H}(\mathbf{x})$. This is as opposed to the conventional setup, where the layers try to learn some representation $\mathcal{G}$ such that the output is $\mathcal{G}(\mathbf{x})$. This proposal stems from the observation that it is typically easier to push the weights towards zero than the identity matrix. Thus, in the new setup, we can just push the weights to zero to learn the identity function.

To that end, [HZRS15] proposes adding "skip connections" between an input to a layer and the output of a future layer. In this way, the model can try to learn the representation $\mathcal{H}$ with output $\mathbf{x} + \mathcal{H}(\mathbf{x})$. Using these skip connections, they can increase the depth of the model and achieving higher accuracy. The remainder of the architecture of ResNet models is similar to models we have previously discussed. The idea of "skip connections" can be extended to any model, and it will appear again in Chapter 9.

3.7 Evaluation of CNNs

Now, we discuss metrics for evaluating CNN models on image classification tasks. A natural metric that we can use is the loss on some test set. However, the loss is rather uninterpretable, since it is the negative log likelihood of the data parameterized by the weights. However, creating a more "interpretable" evaluation metric is complicated by the fact that we are assuming a probabilistic model, but we only observe the crystallized image label. For example, we could have an image that has 0.05 probability of being labeled as an automobile, but the crystallized label is an automobile. The likelihood captures this notion well by using probabilities, but extending beyond that is challenging.

Naively, one might reduce the model to a deterministic one for this purpose, and assume that the label with the highest probability is the one that is chosen. Then, we can use top-1 accuracy by finding the proportion of images such that the model predicts the true label has the highest probability of being observed. We should note that this is not a completely precise evaluation scheme, given that the model was trained assuming a probabilistic setting. However, it is an intuitive measure of performance. For datasets with more classes, like ImageNet, top-5 accuracy is sometimes used, where we relax the constraint and find the

proportion of images where the model predicts that the probability the true label is within the top 5 over all of the classes.

3.8 Implementation on GPUs

We have discussed most of the basic optimizations used to parallelize CNNs on high-performance systems above. These optimizations are often implemented for GPUs in close-sourced libraries such as cuDNN [CWV+14], which write hand-optimized kernels for commonly used training and inference routines. These libraries take computational subgraphs as inputs. Then, these libraries determine the best implementation approach given the current problem size and choose a kernel that is expected to run fastest on the given problem size. The flexibility of GPUs allows these libraries to use different approaches for different problem sizes. By taking in the computational subgraph, they can take also advantage of optimizations like kernel fusion and optimized memory layouts to improve performance. These libraries are often called into by popular machine learning frameworks like PyTorch [PGM+19], TensorFlow [ABC+16], and JAX [FJL18], which gives practitioners an easy way of implementing accelerated kernels on commodity hardware like GPUs.

4

Language Models (LMs)

Language models have recently exploded in popularity largely due to the Transformer architecture. In this chapter, we discuss the theoretical and practical underpinnings of these architectures, common architecture choices, and the datasets used to train these models.

The field of natural language processing (NLP) is quite broad, and there are many tasks that fall under this category, such as sentiment analysis, code generation, machine translation, summarization, question answering. A big breakthrough in this field stemmed from the observation that most of these tasks center around language modeling. At a high level, this refers to the relationship between words in a language. Using this relationship, we can then begin to understand things like meaning, which will allow us to produce new text. This observation stems from the fact that humans are typically not taught these tasks directly. Instead, they pick them up by virtue of deeply understanding the structure of language. Furthermore, this approach was motivated by the fact that there is an abundance of unlabeled corpora of text, but little labeled data tailored to these specific tasks. Thus, we can first train a language model on these unlabeled corpora of text. Then, we can fine-tune the model to tailor it to a specific task. This chapter will follow in a similar spirit. We will first construct a language model through generative pre-training. Then, we will show the application of this model to specific language tasks.

4.1 Tokenization

First, we discuss tokenization. While it will not be the primary emphasis of this thesis, we feel it is important to discuss, since it is an integral part of the language modeling framework. Our first concern in modeling language is changing a text input from a string of characters into covariates that can then be inputted to the model. This task of changing human-readable text to a machine-readable format is typically referred to as tokenization. The field of tokenization itself is quite complex, since it is not at all clear how to best represent language using real numbers. Thus, this is still a very pertinent research topic that is being pursued by linguists and computer scientists. In this section, we discuss tokenization strategies that are commonly used in the literature today.

The main difficulty of tokenization lies in deciding the granularity to be used when separating a string of text. Choosing the level of granularity is a difficult task given the complexity and diversity of language. This task is often referred to as segmentation.

There are a few design considerations. The first is sequence length. Sequence length refers to the number of tokens for a given input. We will call it L. Initially, one might naively suggest a character-level granularity, where each character is a token. However, this leads to a large sequence length, since every character is assigned an embedding vector. Thus, the amount of computation increases drastically. Furthermore, defining what constitutes a character is also non-trivial given the diversity of languages. We need to make sure that each "character" in the input matches a known character in our "dictionary". The "dictionary" is the set of allowable tokens, and we will refer to it as $\mathcal{V}$. If we are only dealing with a single language, defining the vocabulary may be relatively straightforward. However, if there are multiple languages involved or special characters are used, then the problem quickly becomes complicated.

The second design consideration is vocabulary size. Assuming characters from a single language, character-level tokenization has the nice attribute that $|\mathcal{V}|$ is bounded. However, if this is not the case, then some type of character encoding standard like UTF-8 needs to be used. In such cases $|\mathcal{V}| = 1,112,064$. Furthermore, word-level tokenization suffers from a similar problem even for a single language. The vocabulary size of a language is quite large, leading to many embedding vectors $\mathbf{z}_i$ needing to be stored. This makes storage and indexing extremely expensive. Word-level tokenization also has the problem that misspellings can lead to "out-of-vocabulary" situations, where the token is not in the vocabulary. Sometimes, a special `<unk>` character is used for these "out-of-vocabulary" tokens.

The last design consideration is the interpretability of the tokens. Words are the typical unit of language that we use, so it seems naturally to segment sentences at the word-level.

The character-level may be too fine-grained in that some of the meaning is lost because the characters are viewed individually. We want the tokens to be units that have some meaning associated with them.

This problem of tokenization boils down to two parts. First, we need to define a vocabulary $\mathcal{V}$. Then, given a new input sequence, we need to specify how to match the sequence to the vocabulary. Matching the input sequence to the vocabulary is straightforward for character-level or word-level granularities, since they are already segmented. However, for more complex tokenization algorithms, this may be less clear. We will mainly focus on defining the vocabulary.

Given the word-level and character-level granularities, a nice middle ground is to use "subword" tokenization, where the tokens are subsets of full words. This form of tokenization delicately balances the size of the vocabulary with the sequence length.

Once a granularity has been chosen, then each token can be mapped to a unique identification number i, which can then be mapped to a unique vector embedding vector $\mathbf{z}_i \in \mathbb{R}^{d_{\text{emb}}}$. More concretely, for a token t with identification number i, we can encode it as a one-hot encoded vector $\mathbf{v}_t \in \mathbb{R}^{|\mathcal{V}|}$. In particular, we have that $\mathbf{h}_{t,i} = 1$ and $\mathbf{h}_{t,j} = 0$ for all $j \neq i$. Then, we can construct a matrix

$$\mathbf{E} = \begin{bmatrix} \mathbf{z}_1^\top \\ \mathbf{z}_2^\top \\ \vdots \\ \mathbf{z}_{|\mathcal{V}|}^\top \end{bmatrix} \in \mathbb{R}^{|\mathcal{V}| \times d_{\text{emb}}},$$

where $\mathbf{z}_i$ is the embedding vector for the identification number i. Thus, we have that the embedding of token t is

$$\mathbf{h}_t^\top \mathbf{E} \in \mathbb{R}^{1 \times d_{\text{emb}}}.$$

We can imagine a series of tokens with length L. Then, we can construct the design matrix as

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}_1^\top \\ \mathbf{h}_2^\top \\ \vdots \\ \mathbf{h}_L^\top \end{bmatrix} \in \{0, 1\}^{L \times |\mathcal{V}|},$$

where $\mathbf{h}_i$ is the one-hot encoding of token i. Then, we have that $\mathbf{HE} \in \mathbb{R}^{L \times d_{\text{emb}}}$.

Below, we discuss two strategies for defining the vocabulary $\mathcal{V}$. Both generate the vocabulary by iterating over some given training dataset to extract all the tokens that will constitute $\mathcal{V}$.

4.1.1 Byte Pair Encoding (BPE)

Byte Pair Encoding (BPE), initially introduced for data compression in [Gag94] and adapted for segmentation in [SHB16], is one solution to the tokenization problem. First, given some reference text, we generate a vocabulary $\mathcal{V}$. This vocabulary is generated iteratively. First, we start with $\mathcal{V}$ as the set of characters in the language, and the set of character preceded by the special symbol "##". The "##" will denote that the characters are not at the beginning of the word, and it replaces the need for a space. Thus, the word "cat" will initially be tokenized into ["c", "##a", "##t"]. Then, within the reference text, we find the most frequent adjacent pair of tokens. Then, we add this pair to the vocabulary as a "merge operation". Then, we repeat this process, until m merge operations have been defined. m is a hyperparamter that can be set by the user. Thus, we have successfully defined $\mathcal{V}$.

To tokenize a new input sequence, we start by separating the sequence to the word level by using whitespaces. We call this initial splitting process pre-tokenization. Then, we tokenize each word into the character level using the ## symbol. Finally, we iterate over the ordered list of merge operations, merging all instances of the pairs of tokens. Thus, merging typically occurs within word boundaries. Alternatively, we can remove the need for the ## symbol by removing the pre-tokenizing step and treating any whitespace as another character. This is particularly useful for languages that lack the use of spaces.

4.1.2 WordPiece

WordPiece, initially introduced in [SN12], follows the same structure as BPE. However, rather than merging the most frequent adjacent pair, it merges the pair with the lowest score as defined by the following score function. For a pair of characters A and B, we define the score to be

$$s(A,B) = \frac{\text{freq}\,(\text{AB})}{\text{freq}\,(\text{A}) \cdot \text{freq}\,(\text{B})},$$

where $\text{freq}\,(t)$ is the frequency of t in the training corpus. The motivation for this algorithm comes from the following observation. Suppose the text is currently composed by a sequence of tokens $\{s_1, s_2, \ldots, s_N\}$. We start off with the assumption that each of the tokens is generated independently from some probability distribution over the tokens in a given vocabulary $\mathcal{V}$, so

$$\mathbb{P}\,(s_1, s_2, \ldots, s_N|\mathcal{V}) = \mathbb{P}\,(s_1|\mathcal{V})\,\mathbb{P}\,(s_2|\mathcal{V}) \cdots \mathbb{P}\,(s_n|\mathcal{V}) = \prod_{i=1}^{n} \mathbb{P}\,(s_i|\mathcal{V})\,.$$

This independence assumption is derived from the unigram or bag-of-words assumption, which assumes that a token does not depend on the previous tokens. This is a good as-

sumption for finding the right granularity because ideally tokens should be as independent from each other as possible. Although they are still correlated, and we will try to learn this correlation through the language model, this gives us a good granularity to work with.

However, the initial vocabulary construction, where each character is a token, is likely not the correct granularity. In particular, there are likely tokens that are likely to appear together. The intuition behind WordPiece is that we should group such tokens together. Suppose we have s_t and s_{t+1} as adjacent tokens. Then, we could combine them to form a larger token $s_t s_{t+1}$ in vocabulary $\mathcal{V}'$. Then, we would have that

$$\mathbb{P}\left(s_1, s_2, \ldots, s_N | \mathcal{V}'\right) = \prod_{i=1}^{t-1} \mathbb{P}\left(s_i | \mathcal{V}'\right) \cdot \mathbb{P}\left(s_t s_{t+1} | \mathcal{V}'\right) \cdot \prod_{i=t+1}^{n} \mathbb{P}\left(s_i | \mathcal{V}'\right).$$

Taking the ratio of these likelihoods, we get

$$\frac{\mathbb{P}\left(s_1, s_2, \ldots, s_N | \mathcal{V}'\right)}{\mathbb{P}\left(s_1, s_2, \ldots, s_N | \mathcal{V}\right)} = \frac{\mathbb{P}\left(s_t s_{t+1} | \mathcal{V}'\right)}{\mathbb{P}\left(s_t | \mathcal{V}\right) \mathbb{P}\left(s_{t+1} | \mathcal{V}\right)} = \frac{\frac{\text{freq}(s_t s_{t+1})}{N'}}{\frac{\text{freq}(s_t)}{N} \frac{\text{freq}(s_{t+1})}{N}} = \frac{\text{freq}\left(s_t s_{t+1}\right)}{\text{freq}\left(s_t\right) \text{freq}\left(s_{t+1}\right)} \cdot \frac{N}{N'} \cdot N.$$

Assuming that $N \approx N'$, since we are only merging two tokens, this is proportional to our desired score. Intuitively, we are merging two tokens that appear frequently together and do not appear as much individually.

4.1.3 Implementation

Implementations of efficient tokenization algorithms is quite challenging, and there have been many optimizations made for each of the above algorithms. These algorithms often involve using complex data structures like tries to search for tokens. Since tokenization is not the primary focus of this thesis, we leave this topic as one beyond the scope of this thesis. For a more detailed treatment of some of these optimizations, one can refer to [KR18] or [SSS+21].

4.1.4 Positional Embeddings

In LMs, we also want to encode the order of tokens, as the order that the tokens appear in is important to meaning. As we will soon see, the attention operator itself is symmetric with respect to token ordering. Thus, we need some way mechanism to "add" the position of a token to its embeddings. There is a rich literature surrounding the best ways to encode the position of a token. We will introduce a few below. For the sake of brevity, we will give a high-level overview and refer the interested reader to the associated papers.

Fixed Positional Embeddings

Fixed positional embeddings adopt the following setup. Suppose we have a sequence $\mathbf{s}_{1:m}$ with embeddings $[\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_m]$. Then, for token i, [VSP+23] proposes that we add the vector $\mathbf{p}_i \in \mathbb{R}^{d_{\text{emb}}}$ with

$$\mathbf{p}_{i,2j} = \sin\left(\frac{i}{10000^{\frac{2j}{d_{\text{emb}}}}}\right)$$

and

$$\mathbf{p}_{i,2j+1} = \cos\left(\frac{i}{10000^{\frac{2j}{d_{\text{emb}}}}}\right).$$

Then, the new embeddings are $[\mathbf{s}_1 + \mathbf{p}_1, \mathbf{s}_2 + \mathbf{p}_2, \ldots, \mathbf{s}_m + \mathbf{p}_m]$.

Learned Positional Embeddings

Instead of adopting fixed positional embeddings, we can also use learned positional embeddings, similar to how the token embeddings themselves are learned. Thus, for each position i, we have an associated vector $\mathbf{p}_i \in \mathbb{R}^{d_{\text{emb}}}$. Then, the new positional embeddings for a sequence $\mathbf{s}_{1:m}$ with embeddings $[\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_m]$ is $[\mathbf{s}_1 + \mathbf{p}_1, \mathbf{s}_2 + \mathbf{p}_2, \ldots, \mathbf{s}_m + \mathbf{p}_m]$. Then, $\mathbf{p}_i$ are learned during the training of the model.

Rotary Position Embeddings (RoPE) [SLP+23]

Many state-of-the-art models use RoPE as the preferred method of positional embedding. RoPE applies a rotational transformation to the query and key vectors of the attention mechanism based on their position. We leave the full mathematical details to [SLP+23].

4.2 Attention

Now, we discuss the attention mechanism, which forms the backbone of most state-of-the-art language models today. Previously, recurrent neural networks (RNN) were the predominant model used for language modeling. However, following the introduction of the attention mechanism in [BCB16] and its popularization in the transformer architecture from [VSP+23]. Per usual, we discuss the attention mechanism and efficient implementation.

4.2.1 Formulation

In this section, we focus on scaled dot-product attention (SDPA), which was explicitly constructed in [VSP+23]. However, the intuition draws from the ideas described in [BCB16].

In language processing, we are frequently interested in understanding meaning and how different tokens relate to each other. Attention allows us to model this idea.

Suppose that you have a sequence $\mathbf{s}_{1:m}$ of token embeddings $[\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_m]$ and another sequence $\mathbf{t}_{1:n}$ composed of $[\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n]$, with $\mathbf{s}_i, \mathbf{t}_j \in \mathbb{R}^{d_{\text{emb}}}$ for all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$. First, we might be interested in the relation $\mathbf{t}_j$ to $\mathbf{s}_i$ for some $j \in \{1, \ldots, n\}$ and $i \in \{1, \ldots, m\}$. There are many ways that $\mathbf{t}_j$ and $\mathbf{s}_i$ may relate, so we define a "query" that addresses the type of relation that we are focusing on. We can generate this query $\mathbf{q}_j \in \mathbb{R}^{1 \times d_k}$ by creating a weight matrix $\mathbf{W}^{\mathbf{Q}} \in \mathbb{R}^{d_{\text{emb}} \times d_k}$. Then, we have that $\mathbf{q}_j = \mathbf{W}^{\mathbf{Q}}\mathbf{t}_j$. Given the query, we now construct a key for the token $\mathbf{s}_i$, so that we can measure how the query and key interact. To construct the key, we follow a similar procedure where we construct a weight matrix $\mathbf{W}^{\mathbf{K}} \in \mathbb{R}^{d_{\text{emb}} \times d_k}$. Then, we define the key to be $\mathbf{k}_i = \mathbf{W}^{\mathbf{K}}\mathbf{s}_i \in \mathbb{R}^{1 \times d_k}$. Thus, we have constructed a query vector $\mathbf{q}_j \in \mathbb{R}^{1 \times d_k}$ and a key vector $\mathbf{k}_i \in \mathbb{R}^{1 \times d_k}$. To find the "interaction", we take the dot product, $\mathbf{q}_j\mathbf{k}_i^{\mathsf{T}}$ and normalize by $\sqrt{d_k}$.

We can repeat this procedure for all $i \in \{1, \ldots, m\}$. To get the interaction between $\mathbf{t}_j$ and $\mathbf{s}_i$ for all $i \in \{1, \ldots, m\}$. To see this construction in matrix form, let

$$\mathbf{S} = \begin{bmatrix} \mathbf{s}_1^{\mathsf{T}} \\ \mathbf{s}_2^{\mathsf{T}} \\ \vdots \\ \mathbf{s}_m^{\mathsf{T}} \end{bmatrix} \in \mathbb{R}^{m \times d_{\text{emb}}}.$$

Then, we have that

$$\mathbf{K} = \mathbf{S}\mathbf{W}^{\mathbf{K}} \in \mathbb{R}^{m \times d_k}$$

is the key matrix. Then, we can get the "attention" terms through

$$\mathbf{q}_j\mathbf{K}^{\mathsf{T}} \in \mathbb{R}^{1 \times m}.$$

This yields the attention that $\mathbf{t}_j$ pays to each of the tokens $\mathbf{s}_i$ for all $i \in \{1, \ldots, m\}$.

Given these interaction terms, we want to use these to combine the responses from each token in $\mathbf{s}_{1:m}$. Similar to the query and key vectors, to get the responses, or the value vector, of a token $\mathbf{s}_i$, we define a weight matrix $\mathbf{W}^{\mathbf{V}} \in \mathbb{R}^{d_{\text{emb}} \times d_v}$. Then, we have that $\mathbf{V} = \mathbf{S}\mathbf{W}^{\mathbf{V}} \in \mathbb{R}^{m \times d_v}$ yields the value vectors for each of the tokens in $\mathbf{s}_{1:m}$. Thus, we would like to take a weighted sum over these value vectors given the "attention" terms. Therefore, we apply the softmax operation (as defined in Chapter 3) to the row vector $\mathbf{q}_j\mathbf{K}^{\mathsf{T}}$ and use

these as the weights for the value vectors. Thus, we get the final expression is

$$\text{softmax}\left(\frac{\mathbf{q}_j\mathbf{K}^\intercal}{\sqrt{d_k}}\right)\mathbf{V} \in \mathbb{R}^{1\times d_v}.$$

Now, we can imagine extending this analysis to all tokens $\mathbf{t}_j$ for $j \in \{1,\dots,n\}$. Then, we construct

$$\mathbf{T} = \begin{bmatrix}\mathbf{t}_1^\intercal \\ \mathbf{t}_2^\intercal \\ \vdots \\ \mathbf{t}_n^\intercal\end{bmatrix} \in \mathbb{R}^{n\times d_{\text{emb}}}.$$

Then, we have that

$$\mathbf{Q} = \mathbf{T}\mathbf{W}^{\mathbf{Q}} \in \mathbb{R}^{n\times d_k}.$$

Thus, we define the attention function as $\text{Attention} : \mathbb{R}^{n\times d_k} \times \mathbb{R}^{m\times d_k} \times \mathbb{R}^{m\times d_v} \to \mathbb{R}^{n\times d_v}$ defined by

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

where the softmax is applied row-wise.

In our current setup, we assume that $\mathbf{s}_{1:m}$ and $\mathbf{t}_{1:n}$ are two distinct sequences. Thus, we call this cross-attention. We could also have that $\mathbf{s}_{1:m} = \mathbf{t}_{1:n}$, in which case, we call it self-attention. Together, we call the Attention operation an attention head.

4.2.2 Masking

Now, we motivate the idea of masking. We may not want a token in $\mathbf{t}_{1:n}$ to "attend" to all tokens in $\mathbf{s}_{1:m}$. Then, we may want to apply an additive mask to $\mathbf{Q}\mathbf{K}^\intercal$ such that we add a large negative value to $\left(\mathbf{Q}\mathbf{K}^\intercal\right)_{ij}$ iff we do not want $\mathbf{t}_i$ to attend to $\mathbf{s}_j$. More precisely, we will define a mask matrix $\mathbf{M} \in \mathbb{R}^{n\times m}$ such that $\mathbf{M}_{ij} = C$, where C is a large negative constant, if we do not want $\mathbf{t}_i$ to attend to $\mathbf{s}_j$ and $\mathbf{M}_{ij} = 0$ otherwise. Since we apply the softmax operation immediately after, adding a large negative constant will push the "attention" that $\mathbf{t}_i$ pays to $\mathbf{s}_j$ to 0. Then, we redefine $\text{MaskedAttention} : \mathbb{R}^{n\times d_k} \times \mathbb{R}^{m\times d_k} \times \mathbb{R}^{m\times d_v} \times \mathbb{R}^{n\times m} \to \mathbb{R}^{n\times d_v}$ defined by

$$\text{MaskedAttention}\left(\mathbf{Q},\mathbf{K},\mathbf{V},\mathbf{M}\right) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal + \mathbf{M}}{\sqrt{d_k}}\right)\mathbf{V},$$

where the softmax is applied row-wise. We apply the mask prior to the softmax operation, so that the sum of the outputs of the softmax operation, per row, is still 1. Otherwise, we

destroy the probabilistic nature of the softmax operation.

A common masking scheme is known as causal masking, which is commonly used in a self-attention operation. In particular, consider a sequence $\mathbf{s}_{1:m}$. Then, in a text generation setting, we may only want $\mathbf{s}_j$ to attend to $\mathbf{s}_i$ for all $i \leq j$. Then, we define $\mathbf{M}_{\text{causal}} \in \mathbb{R}^{m \times m}$ such that $(\mathbf{M}_{\text{causal}})_{i,j} = 0$ iff $i \leq j$.

4.2.3 Caching

One nice property of the self-attention mechanism is that computation can be reused even after adding tokens to $\mathbf{s}_{1:m}$ when causal masking is being used. This strategy is commonly used in text generation, where we are adding new tokens to the sequence. Although text generation is not the main focus of this thesis, we include this optimization for completeness. Suppose we add $\mathbf{s}_{m+1}$ to the sequence. Then, consider

$$\mathbf{S}' = \left[\begin{array}{c} \mathbf{S} \\ \hline \mathbf{s}_{m+1}^\mathsf{T} \end{array}\right] \in \mathbb{R}^{(m+1) \times d_{\text{emb}}}.$$

Then, we have that

$$\mathbf{Q}' = \mathbf{S}'\mathbf{W}^{\mathbf{Q}} = \left[\begin{array}{c} \mathbf{S}\mathbf{W}^{\mathbf{Q}} \\ \hline \mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{Q}} \end{array}\right] = \left[\begin{array}{c} \mathbf{Q} \\ \hline \mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{Q}} \end{array}\right] \in \mathbb{R}^{(m+1) \times d_k},$$

$$\mathbf{K}' = \mathbf{S}'\mathbf{W}^{\mathbf{K}} = \left[\begin{array}{c} \mathbf{S}\mathbf{W}^{\mathbf{K}} \\ \hline \mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{K}} \end{array}\right] = \left[\begin{array}{c} \mathbf{K} \\ \hline \mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{K}} \end{array}\right] \in \mathbb{R}^{(m+1) \times d_k},$$

and

$$\mathbf{V}' = \mathbf{S}'\mathbf{W}^{\mathbf{V}} = \left[\begin{array}{c} \mathbf{S}\mathbf{W}^{\mathbf{V}} \\ \hline \mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{V}} \end{array}\right] = \left[\begin{array}{c} \mathbf{V} \\ \hline \mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{V}} \end{array}\right] \in \mathbb{R}^{(m+1) \times d_v}.$$

Thus, we have the following.

$$\begin{aligned}
&\text{MaskedAttention}\left(\mathbf{Q}', \mathbf{K}', \mathbf{V}'\right) \\
&\quad = \text{softmax}\left(\frac{\mathbf{Q}'\mathbf{K}'^\mathsf{T}}{\sqrt{d_k}}\right)\mathbf{V}' \\
&\quad = \text{softmax}\left(\frac{\left[\begin{array}{c} \mathbf{Q} \\ \hline \mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{Q}} \end{array}\right]\left[\begin{array}{c|c} \mathbf{K}^\mathsf{T} & \left(\mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{K}}\right)^\mathsf{T} \end{array}\right]}{\sqrt{d_k}}\right)\left[\begin{array}{c} \mathbf{V} \\ \hline \mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{V}} \end{array}\right]
\end{aligned}$$

$$= \text{softmax}\left(\frac{\left[\begin{array}{c|c} \mathbf{Q}\mathbf{K}^\mathsf{T} & \mathbf{Q}\left(\mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{K}}\right)^\mathsf{T} \\ \hline \left(\mathbf{t}_{n+1}^\mathsf{T}\mathbf{W}^{\mathbf{Q}}\right)\mathbf{K}^\mathsf{T} & \left(\mathbf{t}_{n+1}^\mathsf{T}\mathbf{W}^{\mathbf{Q}}\right)\left(\mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{K}}\right)^\mathsf{T} \end{array}\right]}{\sqrt{d_k}}\right)\left[\begin{array}{c} \mathbf{V} \\ \hline \mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{V}} \end{array}\right]$$

Assuming we are using causal masking, then the first m rows of the output of the softmax operation remain unchanged. In particular, the $\mathbf{Q}\left(\mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{K}}\right)^\mathsf{T}$ term is irrelevant, since a mask will be applied to it. Thus, it suffices to compute $\mathbf{t}_{n+1}^\mathsf{T}\mathbf{W}^{\mathbf{Q}}$ and $\mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{K}}$, assuming that we have stored $\mathbf{K}$. Then, we have the following.

$$\begin{aligned} &\text{MaskedAttention}\left(\mathbf{Q}', \mathbf{K}', \mathbf{V}'\right) \\ &= \left[\begin{array}{c|c} \frac{\text{softmax}\left(\mathbf{Q}\mathbf{K}^\mathsf{T}\right)}{\sqrt{d_k}} & \mathbf{0}_m \\ \hline \frac{\text{softmax}\left(\left(\mathbf{t}_{n+1}^\mathsf{T}\mathbf{W}^{\mathbf{Q}}\right)\mathbf{K}'^\mathsf{T}\right)}{\sqrt{d_k}} & \end{array}\right]\left[\begin{array}{c} \mathbf{V} \\ \hline \mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{V}} \end{array}\right] \end{aligned}$$

Thus, we can cache $\mathbf{V}$ and $\mathbf{K}$ to improve performance. Then, we only need to compute $\mathbf{t}_{n+1}^\mathsf{T}\mathbf{W}^{\mathbf{Q}}$, $\mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{K}}$, and $\mathbf{s}_{m+1}^\mathsf{T}\mathbf{W}^{\mathbf{V}}$. We keep $\mathbf{K}$ and $\mathbf{V}$ in a "KV cache" in GPU memory during inference for text generation.

4.2.4 Multi-Headed Attention

We can create multiple attention heads to capture different relationships between token sequences. Suppose we have h attention heads. Each of the attention heads yields its own result $\mathbf{r}_i \in \mathbb{R}^{n \times d_v}$ for all $i \in \{1, \ldots, h\}$. Then, we can concatenate each of the heads along the rows to yield a matrix

$$\mathbf{R} = \left[\begin{array}{c|c|c|c} \mathbf{r}_1 & \mathbf{r}_2 & \cdots & \mathbf{r}_h \end{array}\right] \in \mathbb{R}^{n \times hd_v}.$$

Finally, we pass $\mathbf{R}$ through a linear layer with weights $\mathbf{W}^{\mathbf{O}} \in \mathbb{R}^{hd_v \times d_{\text{emb}}}$. Thus, we get a final output of $\mathbf{R}\mathbf{W}^{\mathbf{O}} \in \mathbb{R}^{n \times d_{\text{emb}}}$.

4.2.5 Training

We have already discussed all necessary components for the weights to be trained. The necessary components consist of backpropagation through matrix multiplication, which is discussed in Chapter 1, and backpropagation through softmax, which is discussed in Chapter 3. Thus, no further discussion of training is necessary for this section.

4.2.6 Implementation

Now, we discuss efficient implementations of an attention head. The attention head is the core computational mechanism of a language model, so optimizing it will be quite important.

Naive Implementation

Recall that

$$\text{MaskedAttention}\left(\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{M}\right) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathsf{T}} + \mathbf{M}}{\sqrt{d_k}}\right)\mathbf{V}.$$

Naively, we could use optimized matrix multiplication algorithms to compute the above quantity. However, the difficulty is that matrix multiplications are typically implemented in a tiled fashion as described in Chapter 2. Therefore, we cannot fuse the matrix multiplication and softmax kernels, since the softmax requires that the full row be completely calculated. However, this motivates the idea of FlashAttention [DFE+22], where the softmax need not the entire row for computation.

FlashAttention [DFE+22]

As motivated above, we would like to perform a softmax on individual tiles. [DFE+22] makes the following mathematical insight that allows such tiling.

We follow the notation provided in [DFE+22]. Suppose we have two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$. Then, consider the concatenated vector

$$\mathbf{z} = \begin{bmatrix}\mathbf{x} \\ \mathbf{y}\end{bmatrix} \in \mathbb{R}^{2d}.$$

Then, we want to relate

$$\text{softmax}\left(\begin{bmatrix}\mathbf{x} \\ \mathbf{y}\end{bmatrix}\right)$$

to

$$\text{softmax}\left(\mathbf{x}\right) \text{ and } \text{softmax}\left(\mathbf{y}\right).$$

Let $m_{\mathbf{x}} = \max_{i \in \{1,\ldots,d\}} \mathbf{x}_i$ and $m_{\mathbf{y}} = \max_{i \in \{1,\ldots,d\}} \mathbf{y}_i$. Then, we have that

$$m_{\mathbf{z}} = \max_{i \in \{1,\ldots,2d\}} \mathbf{z}_i = \max\left(m_{\mathbf{x}}, m_{\mathbf{y}}\right).$$

Then, we have the following.

$$\begin{aligned}
\operatorname{softmax}(\mathbf{z}) &= \operatorname{softmax}\left(\begin{bmatrix}\mathbf{x}\\ \mathbf{y}\end{bmatrix}\right) \\
&= \frac{\begin{bmatrix}\exp(\mathbf{x}_1 - m_{\mathbf{z}})\\ \vdots \\ \exp(\mathbf{x}_d - m_{\mathbf{z}}) \\ \exp(\mathbf{y}_1 - m_{\mathbf{z}}) \\ \vdots \\ \exp(\mathbf{y}_d - m_{\mathbf{z}})\end{bmatrix}}{\sum_{i=1}^{d}\exp(\mathbf{x}_i - m_{\mathbf{z}}) + \sum_{i=1}^{d}\exp(\mathbf{y}_i - m_{\mathbf{z}})} \\
&= \frac{\begin{bmatrix}\exp(m_{\mathbf{x}} - m_{\mathbf{z}})\exp(\mathbf{x}_1 - m_{\mathbf{x}})\\ \vdots \\ \exp(m_{\mathbf{x}} - m_{\mathbf{z}})\exp(\mathbf{x}_d - m_{\mathbf{x}}) \\ \exp(m_{\mathbf{y}} - m_{\mathbf{z}})\exp(\mathbf{y}_1 - m_{\mathbf{y}}) \\ \vdots \\ \exp(m_{\mathbf{y}} - m_{\mathbf{z}})\exp(\mathbf{y}_d - m_{\mathbf{y}})\end{bmatrix}}{\exp(m_{\mathbf{x}} - m_{\mathbf{z}})\sum_{i=1}^{d}\exp(\mathbf{x}_i - m_{\mathbf{x}}) + \exp(m_{\mathbf{y}} - m_{\mathbf{z}})\sum_{i=1}^{d}\exp(\mathbf{y}_i - m_{\mathbf{y}})}
\end{aligned}$$

Then, we have that

$$\mathbf{x}' = \begin{bmatrix}\exp(\mathbf{x}_1 - m_{\mathbf{x}})\\ \vdots \\ \exp(\mathbf{x}_d - m_{\mathbf{x}})\end{bmatrix} \text{ and } \mathbf{y}' = \begin{bmatrix}\exp(\mathbf{y}_1 - m_{\mathbf{y}})\\ \vdots \\ \exp(\mathbf{y}_d - m_{\mathbf{y}})\end{bmatrix}$$

are computed in the $\operatorname{softmax}(\mathbf{x})$ and $\operatorname{softmax}(\mathbf{y})$ respectively. Therefore, we need only $m_{\mathbf{x}}$, $m_{\mathbf{y}}$, $\mathbf{x}'$, and $\mathbf{y}'$ to compute $\operatorname{softmax}(\mathbf{z})$. Therefore, we can compute the softmax in a piecemeal fashion.

Using the above observation, [DFE+22] provides an efficient fused kernel for computing an attention head. We refer the interested reader to [DFE+22] for the full details and proofs on the asymptotic runtime of the algorithm.

Batch Size

As in previous chapters, we can implement the attention mechanism used batched inputs. However, we run into a few bottlenecks that make large batch sizes more difficult.

First, these language models have large memory footprints, since they are quite large, as we will soon discuss. In addition, as we already have discussed, the computational costs grow quadratically with the sequence length. Furthermore, these models are typically deployed in interactive environments that require low latency. For example, language models might be used for chatbots, where users may expect an immediate response, which does not allow the system to batch requests. Therefore, smaller batch sizes are typically used, and it is not uncommon to see batch sizes of 1. Therefore, for most of this chapter, we focus our discussion on the case where the batch size is 1.

4.3 Layer Normalization

As seen in Chapter 3, normalization is a common strategy to reduce training times. In Chapter 3, we introduced batch normalization, a technique commonly used in CNNs to normalize the inputs. However, one drawback of batch normalization is that the mean and variance statistics were computed over the batch. In particular, we need a sufficiently large batch size for these estimates to be reasonable. As we have discussed previously, batch sizes for language models are typically smaller due to their large memory footprint. [BKH16] introduced a new normalization framework called layer normalization that normalizes each data point independently. Instead of normalizing across the batch, they normalize across the layer, as the name suggests. However, most of the formulation from batch normalization is still pertinent. We formalize the above intuition below.

4.3.1 Formulation

For a layer ℓ, we define $\mathrm{LN}^{(\ell)}_{\gamma^{(\ell)},\beta^{(\ell)}} : \mathbb{R} \times \mathbb{R}^{hd_v} \to \mathbb{R}$ as

$$\mathrm{LN}^{(\ell)}_{\gamma^{(\ell)},\beta^{(\ell)}}(a, \mathbf{x}) = \gamma^{(\ell)} \cdot \left(\frac{a - \hat{\mu}^{(\ell)}}{\sqrt{(\hat{\sigma}^2)^{(\ell)} + \epsilon}} \right) + \beta^{(\ell)},$$

where

$$\hat{\mu}^{(\ell)} = \frac{1}{hd_v} \sum_{i=0}^{hd_v} \mathbf{x}_{\ell i}$$

is defined as the sample mean of the layer,

$$\left(\hat{\sigma}^2_{\mathcal{B}}\right)^{(\ell)} = \frac{1}{hd_v} \sum_{i=0}^{hd_v} \left(\mathbf{x}_{\ell i} - \hat{\mu}^{(\ell)}\right)^2$$

is defined as the sample variance of the layer, and ϵ is some small pre-defined constant. Similar to batch normalization, we have learnable parameters $\gamma^{(\ell)}$ and $\beta^{(\ell)}$.

4.3.2 Training

We have that $\gamma^{(\ell)}$ and $\beta^{(\ell)}$ are trainable parameters. Thus, using backpropagation, it suffices to calculate $\frac{\partial \mathrm{LN}^{(\ell)}_{\gamma^{(\ell)},\beta^{(\ell)}}(\mathbf{x}_{ij},\mathbf{x}_i)}{\partial \gamma^{(\ell)}}$, $\frac{\partial \mathrm{LN}^{(\ell)}_{\gamma^{(\ell)},\beta^{(\ell)}}(\mathbf{x}_{ij},\mathbf{x}_i)}{\partial \beta^{(\ell)}}$, and $\frac{\partial \mathrm{LN}^{(\ell)}_{\gamma^{(\ell)},\beta^{(\ell)}}(\mathbf{x}_{ij},\mathbf{x}_i)}{\partial \mathbf{x}_{i,k}}$ for all $j,k \in \{0,\ldots,n-1\}$. Here, we assume that $\mathbf{x} \in \mathbb{R}^{n \times hd_v}$, and $\mathbf{x}_i$ is the transpose of the ith row of $\mathbf{x}$. We have that

$$\frac{\partial \mathrm{LN}^{(\ell)}_{\gamma^{(\ell)},\beta^{(\ell)}}(\mathbf{x}_{ij},\mathbf{x}_i)}{\partial \gamma^{(\ell)}} = \frac{a-\hat{\mu}^{(\ell)}}{\sqrt{\left(\hat{\sigma}^2\right)^{(\ell)}+\epsilon}}$$

and

$$\frac{\partial \mathrm{LN}^{(\ell)}_{\gamma^{(\ell)},\beta^{(\ell)}}(\mathbf{x}_{ij},\mathbf{x}_i)}{\partial \beta^{(\ell)}} = 1$$

for all $j \in \{0,\ldots,n-1\}$.

Note that

$$\frac{\partial \hat{\mu}^{(\ell)}}{\partial \mathbf{x}_{ij}} = \frac{1}{hd_v}$$

for all $j \in \{0,\ldots,n-1\}$. Furthermore, we have that

$$\frac{\partial \left(\hat{\sigma}^2\right)^{(\ell)}}{\partial \mathbf{x}_{ij}} = \frac{2}{hd_v}\left(\mathbf{x}_{ij}-\hat{\mu}^{(\ell)}\right)$$

for all $j \in \{0,\ldots,n-1\}$.

Then, we have the following.

$$\frac{\partial \mathrm{LN}^{(\ell)}_{\gamma^{(\ell)},\beta^{(\ell)}}(\mathbf{x}_{ij},\mathbf{x}_i)}{\partial \mathbf{x}_{ij}} = \frac{\gamma^{(\ell)}}{\sqrt{\left(\hat{\sigma}^2\right)^{(\ell)}+\epsilon}}\left(1-\frac{\partial \hat{\mu}^{(\ell)}}{\partial \mathbf{x}_{ij}}\right)+\left(\mathbf{x}_{ij}-\hat{\mu}^{(\ell)}\right)\cdot\left(-\frac{1}{2}\cdot\frac{1}{\left(\left(\hat{\sigma}^2\right)^{(\ell)}+\epsilon\right)^{\frac{3}{2}}}\cdot\frac{\partial\left(\hat{\sigma}^2\right)^{(\ell)}}{\partial \mathbf{x}_{ij}}\right)$$

Furthermore, we have the following for $k \neq j$.

$$\frac{\partial \mathrm{LN}^{(\ell)}_{\gamma^{(\ell)},\beta^{(\ell)}}(\mathbf{x}_{ij},\mathbf{x}_i)}{\partial \mathbf{x}_{ik}} = -\frac{\gamma^{(\ell)}}{\sqrt{\left(\hat{\sigma}^2\right)^{(\ell)}+\epsilon}}\frac{\partial \hat{\mu}^{(\ell)}}{\partial \mathbf{x}_{ik}}+\left(\mathbf{x}_{ij}-\hat{\mu}^{(\ell)}_{\mathcal{B}}\right)\cdot\left(-\frac{1}{2}\cdot\frac{1}{\left(\left(\hat{\sigma}^2\right)^{(\ell)}+\epsilon\right)^{\frac{3}{2}}}\cdot\frac{\partial\left(\hat{\sigma}^2\right)^{(\ell)}}{\partial \mathbf{x}_{ik}}\right)$$

This gives all of the necessary partial derivatives required to execute backpropagation.

4.3.3 Implementation

Unlike batch normalization, $\hat{\mu}^{(\ell)}$ and $\hat{\sigma}^{(\ell)}$ are not pre-computed during training. Instead, they are "computed on the fly". This is because the variation is measured per sample, unlike batch normalization, where the variation measured over all samples. Since these values are not pre-computed, we cannot perform batch normalization folding as discussed in Chapter 3. However, the time that it takes for layer normalization operations to complete is usually negligible compared to that for attention heads. Thus, layer normalization can be implemented naively with little performance degradation.

4.4 Common LM Architectures

First, we will discuss a general framework for language models, then we will discuss more specific model architectures. The core idea of language modeling is to model the interaction between tokens. In particular, given some textual input, we want to first understand the input by creating an internal representation of it. Then, using this representation of the input to guide the output of the model. The neat idea of language modeling is that we can use text to represent a lot of downstream tasks. For example, if a user wants to summarize a piece of text, they can simply add "Summarize this text." to their input. Similarly, if they are interested in translating the text from one language to another, they can add "Translate to [insert language]." to their prompt. This flexibility is also convenient because models do not need to task-specific training data to learn. Instead, they can use the already widely available corpus of text available on the internet to simply learn the interaction between tokens. We will return to this idea shortly.

Below, we will discuss some of the most common language modeling architectures in the literature today. Typically, they differ in their computational structure, as well as their training objectives. As with Chapter 3, we primarily focus on the key structure of these models and less on their training methodology.

4.4.1 Encoder

As alluded to above, given some textual input, we first want to understand the input and create a deep internal representation of the input. This is the role of the encoder. The output of the encoder should be a sequence of vectors of the same length as the initial sequence of tokens, where each vector is a "hidden state" for that token. This sequence of richer representations can then be used for a variety of tasks. As we have seen previously, we can use multi-headed attention heads for this, as the output has the dimensions $\mathbb{R}^{n \times d_{\text{emb}}}$, which

is the same dimensions as the original embedded input.

4.4.2 Decoder

The goal of the decoder will be to predict the distribution of the next token. In particular, consider a sequence of tokens $\mathbf{x}_{1:N}$. Then, we want to model the joint probability

$$\mathbb{P}(\mathbf{x}_{1:N}) = \prod_{i=1}^{N} \mathbb{P}(\mathbf{x}_i \mid \mathbf{x}_{i-1}, \ldots, \mathbf{x}_1).$$

Under this factorization from Bayes' Rule, it is sufficient to consider $\mathbb{P}(\mathbf{x}_i \mid \mathbf{x}_{i-1}, \ldots, \mathbf{x}_1)$. Thus, the goal of the decoder is to output this distribution over the vocabulary given some sequence of input tokens and optional output from a decoder.

4.4.3 Encoder-Decoder Models

As we have presented above, together, the encoder and decoder components can be combined to generate the next token in a sequence given some initial prompt. The prompt is fed into the encoder, which results in some richer embedding. Then, the current response is fed into the decoder along with the embedding from the encoder.

Transformers [VSP+23]

The Transformer architecture forms the backbone of most modern LM architectures today. Introduced in [VSP+23], it consists of both an encoder and decoder module. Below, we show that the "canonical" computational diagram of a transformer, as presented in [VSP+23].

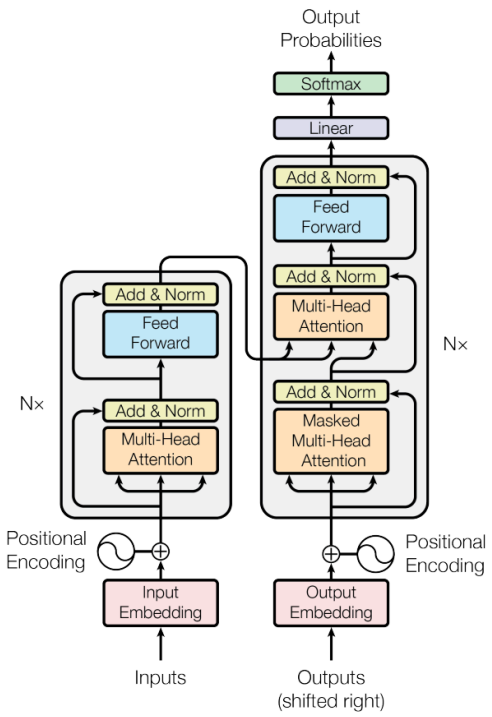

Figure 4.1: Computational diagram for Transformer architecture. [VSP+23]

As seen in the computational diagram, the multi-headed attention do not use any masks. They are "non-causal", since each token can attend to any other token in the input sequence. This is because the input is already provided in full. This is in contrast to the first output multi-headed attention, which uses causal masking. This is because the output sequence is generated one token at a time. However, to compute the next token, the model should only attend to the previous output tokens. Thus, the decoder portion forms a "auto-regressive" model, where the tokens only attend to tokens that come before it. Since the Transformer architecture contains both an encoder and decoder the training objective is to predict the next token. Thus, a cross-entropy loss function can be used, as discussed in Chapter 1.

4.4.4 Encoder-Only Models

Now, we discuss encoder-only models, where we only have the encoder portion of a transformer. A common technique to derive some meaning from the encoder output is to prepend the `[CLS]` token to the beginning of a input sentence. Then, the "hidden state" of the `[CLS]` token is intended to capture some aggregate meaning of the sentence. Then, this can be passed through a feedforward network for downstream tasks. The key idea mentioned above still remains though. Once we have a pre-trained encoder, we can simply take the output and use it for some downstream task without needing to re-train an entire model. We can still

"finetune" the model for the downstream task, but we can "transfer" the pre-trained model to the new application. Encoder-only models are particularly good for tasks that require a rich representation of its input, like sentiment analysis. Below, we introduce BERT, which is a common encoder-only architecture that we will see again in Chapter 9.

Bidirectional Encoder Representations from Transformers (BERT) [DCLT19]

As opposed to the autoregressive models that we have briefly discussed, BERT takes a bidirectional approach to the problem of language modeling. BERT is an encoder-only model, featuring a stack of N layers of the encoder portion of the Transformer architecture. Since BERT only has an encoder layer, we cannot use a next token objective. Instead, BERT introduces two tasks to train the model simultaneously. The first is a "masked" training objective, where a single token is randomly masked. Then, using the surrounding tokens, a hidden state is produced that can then be used to classify the token. The second training objective is a next sentence prediction (NSP) task. Here, two sentences are separated with a [SEP] token. The goal of the model is to predict whether the two sentences follow each other sequentially by using the hidden state of the `[CLS]` token. Unlike in autoregressive models, BERT's encoder-only structure allows tokens to attend to all other tokens in the input sequence. A common BERT model is $\mathbf{BERT_{BASE}}$, which features $N = 12$ layers, 12 attention heads per multi-head attention, and $d_{\text{emb}} = 768$.

4.4.5 Decoder-Only Models

Finally, we discuss decoder-only models, which is the predominant model architecture used for large-scale language models like GPT. The model architecture does not feature an encoder. Instead, it simply uses the embedding of the input sequence. Then, it predicts the next sequence using those embeddings and the current output sequence. Most decoder-only models are autoregressive in nature, since they predict the next token only based on the preceding tokens.

[ZRG+22] introduced a family of OPT models that vary in the number of layers N, the number of attention heads per multi-head attention, and d_{emb}. The models range in size from 125M parameters to 175B parameters. [Wor23] also released a family of BLOOM models that attempts to extend large language models to multilingual tasks. We leave the full details to the individual papers.

4.5 Common LM Datasets

There are many datasets that are used to train language models collected from a variety of sources, since almost any piece of text can be used as training data. As such, these datasets are typically collected from webpages, books, news articles, and other human-written sources of content. Furthermore, unlike vision data, which we discussed in Chapter 3, textual data used to train language models does not necessarily need to be annotated. Therefore, there are no "canonical" datasets used for evaluation purposes, and researchers often generate new data for training larger and larger language models. The goal of adding more data is to increase exposure to vocabulary and language styles, as well as preserve data quality. Below, we list a small subset of those. It should be noted that there are also many private datasets used to train state-of-the-art language models. As we will soon discuss in Chapters 5 and 6, this is a concern for quantization and sparsity methods, which may require additional calibration data.

- The Penn Treebank (PTB) [MSM93] dataset is a common dataset used for sequence labelling tasks. It is a collection of roughly three thousand articles from the Wall Street Journal over a period of three years. The main advantage of the PTB dataset is that the words are annotated with their parts-of-speech (POS), making it a nice choice for labelling tasks.

- WikiText-2 [MXBS16] is another commonly used dataset derived from roughly thirty thousand articles on Wikipedia. WikiText, which refers to the entire collection of these articles, is roughly one hundred million words in size. Thus, [MXBS16] provides a truncated version, which they call WikiText-2, which is double the size of PTB.

- WebText [RWC+19], introduced with the release of GPT-2, uses web scrapes to collect meaningful text data. In particular, it scrapes outbound links on Reddit. In total, WebText contains the contents of tens of millions of outbound links.

- Colossal Cleaned Common Crawl (C4) [RSR+23] also uses data from web scrapes, but it attempts to clean up the data to preserve data quality. The full details can be found in [RSR+23].

4.6 Evaluation

As mentioned in the outset of the chapter, there are a plethora of applications of these language models. Thus, evaluating language models themselves, as well as their use in

these specific tasks, is not as straightforward as for the image classification task discussed in Chapter 3. In this section, we discuss a couple of techniques for evaluating these language models.

4.6.1 Perplexity (PPL)

Perplexity, a concept drawn from information theory, is a common metric to evaluate the underlying language model. For a sequence of tokens $\mathbf{x}_{1:n}$, we have that perplexity is defined by

$$\text{PPL} = \exp\left(-\frac{1}{n}\sum_{i=1}^{n}\log\left(\mathbb{P}\left(\mathbf{x}_i|\mathbf{x}_{i-1},\ldots,\mathbf{x}_1\right)\right)\right).$$

Here, $-\frac{1}{n}\sum_{i=1}^{n}\log\left(\mathbb{P}\left(\mathbf{x}_i|\mathbf{x}_{i-1},\ldots,\mathbf{x}_1\right)\right)$ is the average negative log likelihood (NLL). Thus, this is akin to reporting to the loss of a model. However, as we discussed in Chapter 3, the loss is rather uninterpretable. However, in the language model setting, using classification accuracy is undesirable, since the vocabulary is quite large and the distribution may be quite spread out. Perplexity is good for evaluating architectures that contain decoders and have a predict-next-token objective.

4.6.2 General Language Understanding and Evaluation (GLUE) [WSM+19]

Since language models are typically used for downstream tasks, GLUE provides a set of tasks that can be used to assess a language model. The intuition is that these language models should be able to extend to a variety of language tasks. The tasks consists of single-sentence tasks, similar and paraphrasing tasks, and inference tasks. Single-sentence tasks evaluate the model's ability to predict the acceptability of an English sentence (CoLA) and perform sentiment analysis (SST-2). Similarity and paraphrasing tasks require models to assess whether sentences are "semantically equivalent" (MRPC and QQP) or assign a similarity score (STS-B). Finally, inference tasks require the model to decide the relationship between Sentence A and Sentence B (MNLI, QNLI, RTE) and identify the pronoun reference in a sentence (WNLI). For full details on each of these tasks, the interested reader can refer to [WSM+19]. Together, these tasks can be used to evaluate a base language model with potentially some extra layers. GLUE is good for evaluating the performance of encoder-only architectures like BERT, where the "hidden states" are typically used for some later downstream task.

5 Quantization

Given the popularity of machine learning models, there is a demand to deploy these models in smaller, energy constrained environments. Thus, soon after the rise of deep neural networks, there has been a lot of research on model compression and efficiency. Quantization is one example of such model compression technique. Quantization attempts to represent weights and activations in a more compressed number format with a smaller bitwidth. The hope is that the loss of accuracy is minimal and the efficiency benefits are quite large. The type of quantization that we will be primarily interested in is that applied at inference time after training has occurred. The assumption is that the model will be trained in full floating-point precision, but it can be subsequently quantized and deployed with a much lower memory and energy footprint.

5.1 Mathematical Formulation

First, we start with a mathematical formulation of quantization. As discussed in Chapter 1, let $f : \mathcal{X} \to \mathcal{Y}$ be our model of interest, and let it be parameterized by weights $\mathbf{w} \in \mathbb{R}^d$. We will assume that the optimal weights $\hat{\mathbf{w}} \in \mathbb{R}^d$ minimizes L. We will assume that the training procedure returns these optimal weights $\hat{\mathbf{w}}$. This is a strong assumption because it assumes that the training converges, and there is also a limited amount of precision even for

full-precision numbers.

However, when we quantize, we reduce the accuracy of our trained weights limiting them to some pre-chosen values, which we will refer to as $\tilde{\mathbf{w}}$. We want to minimize $L(\tilde{\mathbf{w}})$, where L is the objective function from Chapter 1. Furthermore, we will require that $\tilde{\mathbf{w}}_i \in \mathcal{Q}_{i,b_i}$, where $\mathcal{Q}_{i,b_i}$ has at most 2^{b_i} elements when the bitwidth is b_i. We will refer to $\mathcal{Q}_{i,b_i}$ as the quantization grid, since it is the set of values that we are quantizing to. In other words, each weight, $\tilde{\mathbf{w}}_i$ only can take on the values in $\mathcal{Q}_{i,b_i}$. The quantization grid can be parameterized itself by $\hat{\mathbf{w}}$. Thus, $\mathcal{Q}_{i,b_i}$ can be a function of $\hat{\mathbf{w}}$. Without any constraints on $\mathcal{Q}_{i,b_i}$, the solution to this problem is trivial for any $b_i \geq 1$, since each weight can have a different quantization grid. Since we know that $\hat{\mathbf{w}}$ minimizes $\mathbb{R}^d$, we can just choose the quantization grid $\mathcal{Q}_{i,b_i}$ such that $\mathbf{w}_i \in \mathcal{Q}_{i,b_i}$. Thus, $\hat{\mathbf{w}}_i = \tilde{\mathbf{w}}_i$. However, we will soon add constraints such that the trivial solution no longer holds.

One can view quantizing the weights as adding a slight perturbations to the original trained weights to move it to the quantization grid. Thus, we can use a second-order Taylor expansion to approximate the change in expected loss. In particular, we have that

$$\mathcal{L}\left(f(\mathbf{x}; \hat{\mathbf{w}} + \Delta\mathbf{w}), \mathbf{y}\right) \approx \mathcal{L}\left(f(\mathbf{x}; \hat{\mathbf{w}}), \mathbf{y}\right) + \Delta\mathbf{w}^\mathsf{T} \nabla_\mathbf{w} \mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right)\Big|_{\mathbf{w}=\hat{\mathbf{w}}} + \frac{1}{2}\Delta\mathbf{w}^\mathsf{T} \mathbf{H}_\mathcal{L}{}^{(\hat{\mathbf{w}})} \Delta\mathbf{w},$$

where $\mathbf{H}_\mathcal{L}{}^{(\hat{\mathbf{w}})}$ is the Hessian matrix of $\mathcal{L}$ evaluated at $\hat{\mathbf{w}}$. Thus, we have that

$$\mathcal{L}\left(f(\mathbf{x}; \hat{\mathbf{w}} + \Delta\mathbf{w}), \mathbf{y}\right) - \mathcal{L}\left(f(\mathbf{x}; \hat{\mathbf{w}}), \mathbf{y}\right) \approx \Delta\mathbf{w}^\mathsf{T} \nabla_\mathbf{w} \mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right)\Big|_{\mathbf{w}=\hat{\mathbf{w}}} + \frac{1}{2}\Delta\mathbf{w}^\mathsf{T} \mathbf{H}_\mathcal{L}{}^{(\hat{\mathbf{w}})} \Delta\mathbf{w}.$$

Taking the expectation of both sides, we get the following.

$$\begin{aligned}
&\mathbb{E}\left[\mathcal{L}\left(f(\mathbf{x}; \hat{\mathbf{w}} + \Delta\mathbf{w}), \mathbf{y}\right) - \mathcal{L}\left(f(\mathbf{x}; \hat{\mathbf{w}}), \mathbf{y}\right)\right] \\
&\quad\approx \mathbb{E}\left[\Delta\mathbf{w}^\mathsf{T} \nabla_\mathbf{w} \mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right)\Big|_{\mathbf{w}=\hat{\mathbf{w}}} + \frac{1}{2}\Delta\mathbf{w}^\mathsf{T} \mathbf{H}_\mathcal{L}{}^{(\hat{\mathbf{w}})} \Delta\mathbf{w}\right] \\
&\quad\approx \mathbb{E}\left[\Delta\mathbf{w}^\mathsf{T} \nabla_\mathbf{w} \mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right)\Big|_{\mathbf{w}=\hat{\mathbf{w}}}\right] + \mathbb{E}\left[\frac{1}{2}\Delta\mathbf{w}^\mathsf{T} \mathbf{H}_\mathcal{L}{}^{(\hat{\mathbf{w}})} \Delta\mathbf{w}\right] \\
&\quad\approx \Delta\mathbf{w}^\mathsf{T} \mathbb{E}\left[\nabla_\mathbf{w} \mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right)\Big|_{\mathbf{w}=\hat{\mathbf{w}}}\right] + \frac{1}{2}\Delta\mathbf{w}^\mathsf{T} \mathbb{E}\left[\mathbf{H}_\mathcal{L}{}^{(\hat{\mathbf{w}})}\right] \Delta\mathbf{w}
\end{aligned}$$

By differentiation under the integral sign, under some regularity conditions that we will assume hold, we have that

$$\mathbb{E}\left[\nabla_\mathbf{w} \mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right)\right] = \nabla_\mathbf{w} \mathbb{E}\left[\mathcal{L}\left(f(\mathbf{x}; \mathbf{w}), \mathbf{y}\right)\right].$$

Assuming that $\hat{\mathbf{w}}$ is the trained weights with full precision, we can assume that $\mathbf{w}$ has achieved a local maximum on $\mathbb{E}\left[\mathcal{L}\left(f(\mathbf{x};\mathbf{w}),\mathbf{y}\right)\right]$ and

$$\nabla_{\mathbf{w}}\mathbb{E}\left[\mathcal{L}\left(f(\mathbf{x};\mathbf{w}),\mathbf{y}\right)\right]\Big|_{\mathbf{w}=\hat{\mathbf{w}}} = \nabla_{\mathbf{w}}L\left(\mathbf{w}\right)\Big|_{\mathbf{w}=\hat{\mathbf{w}}} = \mathbf{0}.$$

Thus, we can approximate the expected change in loss to be

$$\mathbb{E}\left[\mathcal{L}\left(f(\mathbf{x};\hat{\mathbf{w}}+\Delta\mathbf{w}),\mathbf{y}\right)-\mathcal{L}\left(f(\mathbf{x};\hat{\mathbf{w}}),\mathbf{y}\right)\right] \approx \frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]\Delta\mathbf{w}.$$

Therefore, we can reduce quantization to choosing a $\Delta\mathbf{w}$ such that $\frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]\Delta\mathbf{w}$ is minimized and $\hat{\mathbf{w}}_i + \Delta\mathbf{w}_i \in \mathcal{Q}_{i,b_i}$. Assuming that the training data comes from the same distribution as the true data distribution, we have that $\mathbf{H}_L{}^{(\hat{\mathbf{w}})}$ is an unbiased estimate of $\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]$, where L is the objective function.

The above observation is the backbone of most of the state-of-the-art quantization methods today. Over the years, researchers have presented many variants of this base framework. Below, we will discuss a few of the most pertinent ones.

5.2 Quantization Variants

Below, we discuss some variants to the general quantization approach outlined above. Each of the variants is a design decision that has its tradeoffs, which we will also discuss.

5.2.1 Post-Training Quantization vs. Quantization-Aware Training

The first type of variation in quantization methods is when it occurs. Post-training quantization (PTQ) assumes that the model is trained in full precision. Then, it assumes that there is some function $Q : \mathbb{R}^d \to \mathcal{B}_b$ for each weight, where each of the values of $\mathcal{B}_b$ are mapped to $\mathcal{Q}_b$. We call this function the quantization function. We will call $\mathcal{B}_b$ the basis, and $|\mathcal{B}_b| = 2^b$, since we are quantizing to a bitwidth of b elements. We have been assuming this approach in the previous section. Importantly, q is a function of all of the weights, but it outputs the quantized weight at a single position. This is because the value of the other weights may influence how $\mathbf{w}_i$ should be quantized. We will call a quantization function naive iff q is only be a function of $\mathbf{w}_i$. We will explore this idea more deeply in Chapter 10.

Then, we have a function that maps from the quantized data type to $\mathcal{Q}_b$. Typically, we refer to $\mathcal{Q}_b$ as the "quantization grid", since it is the set of values that we can quantize to. We will call this mapping $D : \mathcal{B}_b \to \mathcal{Q}_b$ or the dequantization function. One can think of D as similar to the idea of number formats as a whole, where we use a number format for

$\mathcal{B}_b$, and we map it to some real numbers $\mathcal{Q}_b$. We will view the dequantization as fixed. In particular, once it has been constructed, it will not change regardless of the weights. The quantization grid may depend on a fully trained set of weights $\hat{\mathbf{w}}$, but it will be fixed after that. We will discuss optimal methods for setting a quantization grid in a later section.

The goal of PTQ is to find such a q and d for each weight $\mathbf{w}_i$ such that $D \circ Q : \mathbb{R}^d \to \mathcal{Q}_b$ minimizes the second-order error term. Together, a quantization and dequantization scheme for each weight fully characterizes a quantization scheme. Thus, $\{(Q_i, D_i)\}_{i=1}^d$ will be referred to as a quantization scheme. Recently, PTQ has garnered a lot of attention because we usually have pre-trained models in full precision. Thus, we would prefer to use this model instead of retraining the weights.

Another variant of quantization is the more accurate quantization-aware training (QAT). In QAT, quantization is applied during training, and it is incorporated into the loss function. While we are still looking for a mapping from $\mathbb{R}^d$ to $\mathcal{Q}_b$, the $\mathcal{Q}_b$ can essentially be learned through training on data. The advantage of this is that it has more flexibility than PTQ, since we are not constrained to the weights from the pre-trained model and have access to training data. Thus, QAT can achieve more optimal weight combinations. However, QAT comes at the expense of retraining the entire model, an often costly procedure especially for large models. Furthermore, acquiring the data that was used to train a model is sometimes difficult due to privacy concerns.

Typically, in the literature, we see some combination of these methods, where we use a pretrained model as a starting point, but we still do some further training with a calibration set. The training is not as extensive as retraining the model from scratch like QAT though, so it requires less data and is less energy intensive. We will further explore some of these schemes in the following sections. Most of the subsequent literature review will be focused on this flavor of quantization.

5.2.2 Weight vs. Activation

The framework that we established above focuses on weight-only quantization, where we try to quantize the weights to minimize the expected loss. Currently, most of the literature focuses on weight-only quantization, since it is an easier task than quantizing both weights and activations. However, it should be noted that we can also quantize the activations to lower bitwidths. This process is more complex because the activations are dependent on the input. We will see this idea return in Chapter 8. Thus, the dynamic range is dependent on the input. However, the literature does explore possible dynamic quantization schemes. For most parts of this thesis though, we will focus on weight-only quantization though.

5.2.3 Choosing Quantization Granularity

As mentioned previously, we first need to choose the granularity of quantization. The granularity refers to the level at which weights must share a quantization and dequantization function. More rigorously, it is the partition, $P_1, \ldots, P_g$, of the weights such that for $i, j \in P_k$, we have that $Q_i = Q_j$ and $D_i = D_j$. We have that $g \leq d$, where d is the total number of weights.

If $g = d$, then we have a different quantization and dequantization function for each weight. Thus, we can just choose $\mathcal{Q}_{i,b_i}$ such that $\mathbf{w}_i \in \mathcal{Q}_{i,b_i}$. Then, we select $Q_i(\mathbf{w}) = 0$ and $D_i(0) = \mathbf{w}_i$. Then, we have no information loss. However, we should note that there is a hidden cost that makes this no better than storing the weights in full precision. When $g = d$, we need to store all $\mathbf{w}_i$ in full precision, since we need to store $\mathcal{Q}_{i,b_i}$ for any quantization scheme. Thus, quantization provides no benefit. Therefore, it is more interesting to consider the case where $g < d$.

There are many possible granularities that can be chosen, and it is typically dependent on the model architecture. For example for weight quantization in CNN models, there are a few choices for granularity. First, one could do per-tensor quantization, which requires that all weight elements in a tensor share the same quantization and dequantization functions. Another possible option is per-channel quantization, where weights that correspond to the same output channel share the same quantization and dequantization functions. We can also do per-group quantization, which refers to the group convolution idea discussed in Chapter 3. This is a compromise between per-tensor and per-channel quantization.

For LLMs, since they are composed of multi-head attention (MHA) blocks, we have a notion of per-head quantization, which groups together weights for the same attention head. We can also do per-embedding quantization, where weights that correspond to the same embedding are grouped together. Note that we can again extend these scheme to the "group" level, where we take groups of heads or groups of embeddings.

5.2.4 Uniform vs. Non-Uniform

Another type of variation depends on the choice of $\mathcal{B}_b$ for a bitwidth b. Assuming we are doing PTQ, we want to find a map $D : \mathcal{B}_b \to \mathcal{Q}_b$, where $\mathcal{Q}_b \subseteq \mathbb{R}$. We say a quantization scheme is uniform if the following property holds. For all $x, y \in \mathcal{B}_b$, we have that

$$D(x) - D(y) = s(x - y)$$

for some $s \in \mathbb{R}$. Let the smallest element of $\mathcal{B}_b$ be b_0. Then, suppose that D maps $b_0 \mapsto d_0$. Then, we have that D maps $x \mapsto s(x - b_0) + d_0$ for a uniform quantization scheme. In other

words, we can reduce storing D to storing s and d_0, instead of needing to store the entire mapping from $\mathcal{B}_b \to \mathcal{Q}_b$. This is quite advantageous, since the size of $\mathcal{Q}_b$ grows exponentially with the number of bits.

Intuitively, a quantization scheme is uniform if the space between adjacent points in the quantization grid is proportional to their actual distance. In the case where $\mathcal{B}_b = \text{INT}_n$, the distance between adjacent points in the quantization grid is s. However, we should note that this need not be true. We will see this more when $\mathcal{B}_b$ has a floating-point representation. In that case, the distance between two points in the quantization grid are not uniform, since the distance between two adjacent points in $\mathcal{B}_b$ is not constant. The advantage of uniform quantization though is that matrix multiplication can be without needing to store the entirety of $\mathcal{Q}_b$.

Mathematically, we have the following. Suppose we have vectors $\mathbf{w} \in \mathbb{R}^n$ and $\mathbf{x} \in \mathbb{R}^n$. Furthermore, suppose $D : \mathcal{B}_b \to \mathbb{R}$ is a uniform dequantization function with scaling products s. Let $Q : \mathbb{R}^d \to \mathcal{B}_b$ be the quantization function. Let $\mathbf{D} : {\mathcal{B}_b}^n \to \mathbb{R}^n$ be d applied element-wise. Similarly, let $\mathbf{Q} : \mathbb{R}^n \to {\mathcal{B}_b}^n$ be q applied element-wise. Then, we have the following.

$$\begin{aligned}
(\mathbf{D} \circ \mathbf{Q}(\mathbf{w})) \cdot \mathbf{x} &= \sum_{i=1}^{n} D \circ Q\left(\mathbf{w}\right) \mathbf{x}_i \\
&= \sum_{i=1}^{n} \left(s\left(Q\left(\mathbf{w}_i\right) - b_0\right) + d_0\right) \mathbf{x}_i \\
&= \sum_{i=1}^{n} sQ\left(\mathbf{w}_i\right) \mathbf{x}_i + \sum_{i=1}^{n} \left(d_0 - sb_0\right) \mathbf{x}_i \\
&= s\mathbf{Q}\left(\mathbf{w}\right) \cdot \mathbf{x} + \sum_{i=1}^{n} \left(d_0 - sb_0\right) \mathbf{x}_i
\end{aligned}$$

Thus, we have that

$$(\mathbf{D} \circ \mathbf{Q}(\mathbf{w})) \cdot \mathbf{x} = s\mathbf{Q}\left(\mathbf{w}\right) \cdot \mathbf{x} + \sum_{i=1}^{n} \left(d_0 - sb_0\right) \mathbf{x}_i.$$

This has the nice property that we only need to store s_0 and d_0. Then, to do operations on $\mathbf{D} \circ \mathbf{Q}\left(\mathbf{w}\right)$, the quantized version of $\mathbf{w}$, we can use $\mathbf{Q}(\mathbf{w})$, which has a smaller memory footprint. On the other hand, for non-uniform quantization, we need to store the mapping $D : \mathcal{B}_b \to \mathcal{Q}_b$, since it does not satisfy any nice relationship. Thus, the size to story D grows exponentially with bitwidth b. Furthermore, non-uniform quantization requires repeated memory accesses to dequantize, whereas we can use computations to compute D for uniform quantization. This becomes critical when we assess whether workloads are either memory

or compute bound.

5.2.5 Symmetric vs. Unsymmetric

For uniform quantization schemes, we can further classify them as symmetric or unsymmetric. We will call a uniform quantization scheme symmetric iff $sx = D(x)$ for all $x \in \mathcal{B}_b$. Note that it is sufficient for $sb_0 = d_0$. Intuitively, this is true because the gap between adjacent points on the quantization grid are equally separated by s. We will now show this formally. Let $x \in \mathcal{B}_b$. Then, we have that $D(x) = s(x - b_0) + d_0 = sx - sb_0 + d_0$. Assuming $sb_0 = d_0$, we have that $D(x) = sx$. Thus, we have that

$$(\mathbf{D} \circ \mathbf{Q}(\mathbf{w})) \cdot \mathbf{x} = s\mathbf{Q}(\mathbf{w}) \cdot \mathbf{x} + \sum_{i=1}^{n} (d_0 - sb_0)\, \mathbf{x}_i = s\mathbf{Q}(\mathbf{w}) \cdot \mathbf{x}.$$

Symmetry is a desirable property because we can forgo any addition/subtraction in the dequantization step and simply scale the $x \in \mathcal{B}_b$ by the factor of s. This limits the amount of computation and may be helpful in compute-bound environments.

5.2.6 Choice of $\mathcal{B}_b$

The choice of $\mathcal{B}_b$ is typically limited by hardware considerations. Below, we discuss a few viable options for $\mathcal{B}_b$. Ideally, we want $\mathcal{B}_b = \{0, 1\}^b$, so that the quantized numbers can be stored in binary.

Integer Number Formats

For a non-negative number $n \in \mathbb{Z}_+$, we can naturally express it through it's binary representation. Thus, for a bitwidth of b, we can represent the integers between 0 and $2^b - 1$. To include negative numbers, we can use two's complement, which allows us to represent all integers between -2^{b-1} and $2^{b-1} - 1$ with a bitwidth of b.

Fixed-Point Number Formats

Fixed-point number formats allow us to begin to represent real numbers. Suppose we have a bitwidth of b. Then, we can choose the number of integer bits i and fractional bits f such that $i + f + 1 = b$. Then, for a sign bit b_s, integer bits b_i, and fractional bits b_f, the corresponding number represented is $(-1)^{b_s} \frac{b_i b_f}{2^f}$, where $b_i b_f$ is the concatenation of b_i and b_f. Thus, we can express $\frac{j}{2^f}$ for all $-2^{i-1} - 1 \leq j \leq 2^{i-1} - 1$.

Floating-Point Number Formats

Finally, we come to floating-point number formats, which are the most common representation of real numbers. Suppose we have a bitwidth of b. Then, we assign one sign bit, m mantissa bits, and e exponent bits, where $1 + m + e = b$. The sign bit is used to indicate whether the real number is positive or negative. Suppose we have that the sign bit is b_s, the mantissa bits are b_m, and the exponent bits are b_e.

Then, we have the following conditions. If $b_s = b_m = b_e = 0$, then we have that the number represented is 0. If $b_s = 0$, $b_m = 0$, and $b_e = 2^e - 1$, then we have that the number represented is $-\infty$. If $b_s = 1$, $b_m = 0$, and $b_e = 2^e - 1$, then we have that the number represented is $+\infty$. If $b_m \neq 0$ and $b_e = 2^e - 1$, then we have that the number represented is NaN. If $b_e = 0$, then the represented number is $(-1)^{b_s} \times 0.b_m \times 2^{1-(2^e-1)}$. This case is known as "subnormal" numbers, since the exponent bits are all zero, so it is used for representing numbers small in magnitude. Finally, in the general case, we have that the represented number is $(-1)^{b_s} \times 1.b_m \times 2^{b_e-(2^e-1)}$. This is all defined within the IEEE754 standard.

Application to Quantization

The literature mainly focuses on using integers of bitwidth b to be $\mathcal{B}_b$, which we will denote as INT_b. In this setting, we can either choose the integers to be signed or unsigned. Other alternatives include using floating-point based $\mathcal{Q}_b$, such as AdaptivFloat, which we will introduce in 8. Floating-point based quantization introduces a new layer of complexity, as adjacent elements of the quantization grid are not evenly spaced, due to the use of exponent bits. Despite this, all of the previously discussed properties are still valid descriptions, since they do not assume anything about $\mathcal{B}_b$. In particular, we can still say that a floating-point quantization scheme is uniform, since the only condition is that the distance in the quantization grid is proportional to the distance in the basis.

5.2.7 Homogeneous vs. Heterogeneous Bitwidths

In our construction, the choice of bitwidth b_i can vary for weights $\mathbf{w}_i$. We will call a quantization scheme homogeneous in bitwidth if b_i is the same for all $i \in \{1, 2, \ldots, d\}$. Otherwise, we will call it heterogeneous. From a hardware perspective, homogeneous widths are likely the easiest to deal with, since MACs only need to accommodate one datatype. However, there is existing literature that points to the fact that some weights may be more important than others. In that case, it may be desirable to use larger bitwidths to help preserve the accuracy of these weights. We will explore this idea in much greater depth in Chapter 8.

5.2.8 Different Rounding Methods

Assuming that $\mathcal{B}_b$ and $\mathcal{Q}_b$ are known and $D : \mathcal{B}_b \to \mathcal{Q}_b$ is fixed, we still can decide how to choose our quantization function $Q : \mathbb{R}^d \to \mathcal{B}_b$. In this part, we focus on naive quantization functions, where Q is only a function of $\mathbf{w}_i$. Thus, we can define $\tilde{Q} : \mathbb{R} \to \mathcal{B}_b$. This can be effectively seen as "rounding", as for a given $r \in \mathbb{R}$, we are choosing to representing it with $D \circ \tilde{Q}(r) \in \mathcal{Q}_b$.

Naturally, one might choose a rounding scheme where they choose the point on the quantization grid that is closest to the original r. Thus, we have that

$$D \circ \tilde{Q}(r) = \arg\min_{q \in \mathcal{B}} |r - q| .$$

We will call this rounding scheme "rounding-to-nearest" (RTN). If $r < \min \mathcal{Q}_b$, then we have that $D \circ \tilde{Q}(r) = \min \mathcal{Q}_b$. Similarly, if $r > \max \mathcal{Q}_b$, then we have that $D \circ \tilde{Q}(r) = \max \mathcal{Q}_b$. This phenomenon is known as clamping, where the numbers beyond the quantization grid are set to their closest extreme. If $\min \mathcal{Q}_b < r < \max \mathcal{Q}_b$, then, we know that $a \leq r \leq b$ for adjacent points $a, b \in \mathcal{Q}_b$. Intuitively, it will sit between two points on the quantization grid. If $D \circ \tilde{Q}(r) = b$, then we call this "rounding up" and if $D \circ \tilde{Q}(r) = a$, then we call this "rounding down".

However, as we will soon see, "rounding-to-nearest" is not always the most preferred strategy. In fact, it might be better to round up even though rounding down might result in a closer point on the quantization grid. Another type of "rounding" scheme is stochastic rounding. As the name suggests, rounding is done randomly. In particular, we choose whether to round up or round down with some probability.

We will define a stochastic rounding scheme as introduced in [GAGN15]. This rounding scheme was initially for training deep neural networks in low-precision environments, but the same methodology can be applied to quantization. If $r < \min \mathcal{Q}_b$, then we will have that $D \circ \tilde{Q}(r) = \min \mathcal{Q}_b$. Similarly, if $r > \max \mathcal{Q}_b$, then we will have that $D \circ \tilde{Q}(r) = \max \mathcal{Q}_b$. This is the same clamping that we do for rounding-to-nearest. Again, if $\min \mathcal{Q}_b < r < \max \mathcal{Q}_b$, then, we know that $a \leq r \leq b$ for some adjacent points $a, b \in \mathcal{Q}_b$. We will choose $D \circ \tilde{Q}(r) = b$ with probability $1 - \frac{b-r}{b-a} = \frac{r-a}{b-a}$ and $D \circ \tilde{Q}(r) = a$ with probability $1 - \frac{r-a}{b-a} = \frac{b-r}{b-a}$. Intuitively, the probability of rounding up or down is inversely proportional to how close the original point is to adjacent points of the quantization grid. Thus, if r is closer to b than a, there is a higher probability of rounding to b. However, this will not always occur. On the other hand, rounding-to-nearest will always choose b. Let $\tilde{r}$ be the value that r is rounded to. This

stochastic rounding scheme has the nice property that the expected value of $\tilde{r}$ is

$$\mathbb{E}[\tilde{r}] = \frac{b-r}{b-a} \cdot a + \frac{r-a}{b-a} \cdot b = \frac{ba - ra + rb - ab}{b-a} = r.$$

Thus, the rounding scheme is unbiased, since the expected error of $\tilde{r} - r$ is zero. In a few sections, we will discuss the advantages of a stochastic rounding scheme.

5.3 Finding $\mathcal{Q}$ (Finding D)

Now, we turn to finding optimal quantization schemes. The goal is to find optimal Q_i and D_i for all weights $\mathbf{w}_i$. Optimizing for both simultaneously is quite difficult, so the literature often takes a two-pronged approach. In particular, they choose an optimal D function. Then, given that quantization grid, they choose an optimal set of Q functions.

For this thesis, we will focus primarily on symmetric, uniform quantization schemes. Thus, choosing an optimal D function simply amounts to choosing some scaling factor s_i for each weigh $\mathbf{w}_i$. For non-uniform quantization schemes, we refer the reader to recent works like SqueezeLLM [KHG+24], which typically take a k-means clustering-based approach to finding an optimal quantization grid.

5.3.1 Max-Scaled

Suppose we want a set of weights $S = \{\mathbf{w}_1, \ldots, \mathbf{w}_c\}$ to all share the same quantization scheme. One natural way to choose a scaling factor is to simply choose

$$s = \frac{\max_{\mathbf{w}_i \in S} |\mathbf{w}_i|}{\max_{x \in \mathcal{B}_b} |x|}.$$

We will refer to this as max-scaled quantization, and we will return to this notion in Chapter 10. Intuitively, we scale the quantization grid, such that we can represent the maximum magnitude weight. In particular, there is no "clipping" error, since we do not have any weights that lie outside of the range of the quantization grid.

5.3.2 Loss Aware Post-Training Quantization (LAPQ) [NCB+20]

The scheme that we presented above is rather naive. In particular, it allows us to represent all weights within the bounds of the quantization grid. However, this may not always be optimal. Intuitively, imagine that you have an outlier weight with all other weights in the set clustered closely. Then, you may want to choose a scale that only attends to the non-

outlier weights to provide more precision for those. Thus, [NCB+20] introduced the LAPQ framework to choose the optimal step size. Their setup is as follows.

First, they will assume that all layers will share the same quantization and dequantization functions. Suppose there are n layers. Then, the goal is to find

$$\mathbf{s} = \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix}$$

such that

$$\frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]\Delta\mathbf{w}$$

is minimized. LAPQ does this in the following. First, it assumes a rounding-to-nearest quantization scheme. While we will soon discuss the non-optimality of rounding-to-nearest quantization schemes, as we mentioned previously, solving for both Q and D simultaneously is numerically difficult. LAPQ first solves for the optimal $\mathbf{s}$ in a layer-wise fashion. Suppose we are optimizing for layer i. Then, we select a set $P \subseteq \mathbb{R}$. For each $p \in P$, we choose the $s_{i,p}$ such that

$$\left\|Q_{s_i}\left(\mathbf{w}^{(i)}\right) - \mathbf{w}^{(i)}\right\|_p$$

is minimized. Here, Q_{s_i} is the quantization function with scale s_i. Thus, we minimize the L_p norm for each value of P. Then, for layer i, we have "candidates" for the optimal s_i. In particular, we have $\{s_{i,p} : p \in P\}$. We will notate

$$\mathbf{s}^{(p)} = \begin{bmatrix} s_{1,p} \\ \vdots \\ s_{n,p} \end{bmatrix}$$

as the set of candidates for a given $p \in P$.

Next, LAPQ uses quadratic interpolation to find the optimal $\mathbf{s}$ given the candidates. This stems from the following observation. First, we will write $\mathcal{L}$ as a function of p, assuming that $\mathbf{y}$ and $\hat{\mathbf{y}}$ are fixed. In particular, we define

$$R(p) \triangleq \mathcal{L}\left(f(\mathbf{x}; \hat{\mathbf{w}} + \Delta_p\mathbf{w}), \mathbf{y}\right),$$

where $\Delta_p\mathbf{w}$ corresponds to the change from quantizing the weights using scale factors $\mathbf{s}^{(p)}$

and a rounding-to-nearest scheme. Then, again by a Taylor expansion, we have that

$$R(\hat{p}) \approx R(p^*) + (\hat{p} - p^*) \left.\frac{\partial R}{\partial p}\right|_{p=p^*} + \frac{1}{2} \left.\frac{\partial^2 R}{\partial p^2}\right|_{p=p^*} (\hat{p} - p^*)^2 .$$

Thus, we have that

$$R(\hat{p}) - R(p^*) \approx (\hat{p} - p^*) \left.\frac{\partial R}{\partial p}\right|_{p=p^*} + \frac{1}{2} \left.\frac{\partial^2 R}{\partial p^2}\right|_{p=p^*} (\hat{p} - p^*)^2 .$$

Here, p^* is the optimal value of p that minimizes L. Taking the expectation of both sides, we get the following.

$$\begin{aligned}
\mathbb{E}\left[R(\hat{p}) - R(p^*)\right] &\approx \mathbb{E}\left[(\hat{p} - p^*) \left.\frac{\partial R}{\partial p}\right|_{p=p^*} + \frac{1}{2} \left.\frac{\partial^2 R}{\partial p^2}\right|_{p=p^*} (\hat{p} - p^*)^2\right] \\
&\approx (\hat{p} - p^*)\, \mathbb{E}\left[\left.\frac{\partial R}{\partial p}\right|_{p=p^*}\right] + \frac{1}{2} (\hat{p} - p^*)^2\, \mathbb{E}\left[\left.\frac{\partial^2 R}{\partial p^2}\right|_{p=p^*}\right]
\end{aligned}$$

By differentiation under the integral sign, under some regularity conditions that we will assume hold, we have that

$$\mathbb{E}\left[\frac{\partial R}{\partial p}\right] = \frac{\partial L}{\partial p}.$$

Assuming that p^* is the optimum, we have that

$$\left.\frac{\partial L}{\partial p}\right|_{p=p^*} = 0.$$

Thus, we can approximate the expected change in loss to be

$$\mathbb{E}\left[R(\hat{p}) - R(p^*)\right] \approx \frac{1}{2} (\hat{p} - p^*)^2\, \mathbb{E}\left[\left.\frac{\partial^2 R}{\partial p^2}\right|_{p=p^*}\right].$$

Thus, we can see model the change in loss as a quadratic function with respect to p. Thus, given the set of P, we can fit a quadratic function to find the optimal p^*. Then, this yields a set of optimal scaling factors $\mathbf{s}^{(p^*)}$. This gives a pretty decent approximation for the optimal scaling factors. LAPQ takes one last step by optimizing the scaling factors jointly. Up until this point, we have derived the optimal scaling factors on a per-layer basis. Thus, LAPQ adds one additional joint optimization step. They use Powell's iteration, an iterative method for finding minima of a function to update $\mathbf{s}^{(p^*)}$. For the full details of Powell's iteration and LAPQ, we refer the reader to [Pow64] and [NCB+20] respectively.

5.4 Finding Optimal Q

We will now explore a couple of quantization methods for finding Q, which were seminal works in the field of quantization. They were developed in the context of CNNs, but their principles can be extended to other types of deep learning models. While there are many other methods in the literature today, these are probably the most influential, as they guide the motivation between most other quantization algorithms. Similar to the previous section, we will assume a uniform quantization scheme. Thus, D is already determined by the scaling factor that we have chosen above.

5.4.1 Adaptive Rounding (AdaRound) [NAvB+20]

Prior to AdaRound, most quantization methods used a rounding-to-nearest scheme for the quantization function q, once they had fixed $\mathcal{Q}$. They would choose to "round" to the nearest element in $\mathcal{Q}$. $\mathcal{Q}$ was typically defined using the minimum and maximum element as the lower and upper bound, respectively. Then, uniform quantization was used to fill in the quantization grid. While this is the most optimal for minimizing the loss within a single weight value, it might not be the most optimal when considering the model as a whole. In particular, this analysis neglects the fact that weights "interact" with each other when they multiply an input. Intuitively, we might want to "redistribute" some of the "information" among the weights. For example, we might want to pass some of the "information" from weights with high magnitude to low magnitude, so that they are not clamped by the range. AdaRound verifies this by showing that 100 trials of stochastic rounding does better at minimizing the second-order term as opposed to rounding to nearest. This is captured below in a plot from [NAvB+20], where they show that the second order error term can be decreased by using stochastic rounding, leading to higher accuracy.

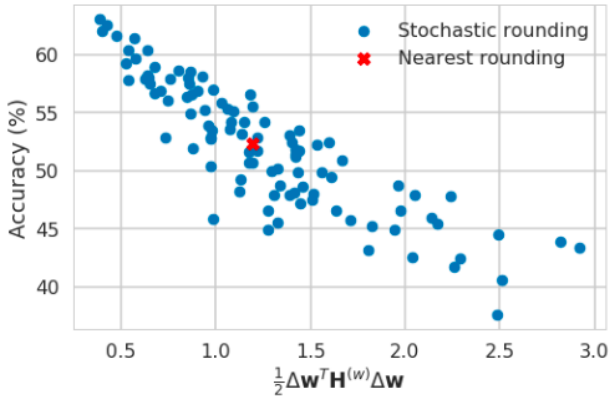

Figure 5.1: Effect of stochastic rounding on accuracy. [NAvB+20]

This intuition is captured more formally by the second-order approximation that we derived previously. Recall that we have

$$\mathbb{E}\left[\mathcal{L}\left(f(\mathbf{x};\hat{\mathbf{w}}+\Delta\mathbf{w}),\mathbf{y}\right)-\mathcal{L}\left(f(\mathbf{x};\hat{\mathbf{w}}),\mathbf{y}\right)\right]\approx\frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]\Delta\mathbf{w}.$$

If we assumed that $\mathbf{H}_{L}{}^{(\mathbf{w})}$ were a diagonal matrix, then we would only be concerned with minimizing the loss of each individual weight. Let $\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]=\operatorname{diag}\left(\{d_i\}\right)$. Then, we have that

$$\mathbb{E}\left[\mathcal{L}\left(f(\mathbf{x};\hat{\mathbf{w}}+\Delta\mathbf{w}),\mathbf{y}\right)-\mathcal{L}\left(f(\mathbf{x};\hat{\mathbf{w}}),\mathbf{y}\right)\right]\approx\frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbf{H}_{L}{}^{(\hat{\mathbf{w}})}\Delta\mathbf{w}=\sum_{i=1}^{d}d_i\left(\Delta\mathbf{w}_i\right)^2.$$

Thus, we would only want to minimize $\Delta\mathbf{w}_i$ for each weight individually.

However, this assumption is unlikely to hold for most deep neural networks, given that weights in the same layer interact with each other given the non-linearities. Dropping this assumption the $\mathbf{H}_{L}{}^{(\mathbf{w})}$ is diagonal leads us to the conclusion that minimize $\Delta\mathbf{w}_i$ for each i individually may not be the most optimal.

To remedy this, AdaRound presents the following framework for thinking about quantization. Suppose that the deep neural network has n layers, with weights $\mathbf{W}^{(i)}\in\mathbb{R}^{d_i\times d_{i+1}}$ and flattened weights $\mathbf{w}^{(i)}\in\mathbb{R}^{d_i\cdot d_{i+1}}$ for layer i. Then, consider the flattened weight vector

$$\mathbf{w}=\begin{bmatrix}\left(\mathbf{w}^{(1)}\right)^{\mathsf{T}} & \left(\mathbf{w}^{(2)}\right)^{\mathsf{T}} & \cdots & \left(\mathbf{w}^{(n)}\right)^{\mathsf{T}}\end{bmatrix}.$$

Furthermore, assume that $\mathbf{z}^{(i)}=\mathbf{a}^{(i-1)^{\mathsf{T}}}\mathbf{W}^{(i)}\in\mathbb{R}^{d_{i+1}}$ are the pre-activation values of layer i and $\mathbf{a}^{(i)}\in\mathbb{R}^{d_{i-1}}$ is the output of layer $i-1$. Since the Hessian matrix grows quadratically in the size of the number of weights, storing the Hessian matrix is infeasible from a memory footprint perspective. Thus, AdaRound assumes that the interaction between weights in different layers is negligible and focuses on a "layer-wise" reconstruction. Thus, for each layer ℓ, they choose $\mathbf{w}^{(\ell)}$ that minimizes their objective function. We claim that this assumption is fairly reasonable. Consider the value of the Hessian matrix of the loss with respect to the weights, $\mathbf{H}_{\mathcal{L}}{}^{(\mathbf{w})}$. In particular, we want to take the Hessian with respect to weights $\mathbf{w}_{i,j}^{(a)}$ and $\mathbf{w}_{k,\ell}^{(b)}$. Without loss of generality, assume $a<b$, so that the weights are from different layers.

$$\begin{aligned}\mathbf{H}_{\mathcal{L}}{}^{(\mathbf{w})}_{(a,(i,j)),(b,(k,\ell))}&=\frac{\partial^2\mathcal{L}}{\partial\mathbf{w}_{i,j}^{(a)}\mathbf{w}_{k,\ell}^{(b)}}\\&=\frac{\partial}{\partial\mathbf{w}_{j}^{(i)}}\left[\frac{\partial\mathcal{L}}{\mathbf{w}_{k,\ell}^{(b)}}\right]\end{aligned}$$

$$
\begin{aligned}
&= \frac{\partial}{\partial \mathbf{w}_{i,j}^{(a)}} \left[\frac{\partial \mathcal{L}}{\partial \mathbf{z}_k^{(b)}} \frac{\partial \mathbf{z}_k^{(b)}}{\mathbf{w}_{k,\ell}^{(b)}} \right] \\
&= \frac{\partial}{\partial \mathbf{w}_{i,j}^{(a)}} \left[\frac{\partial \mathcal{L}}{\partial \mathbf{z}_k^{(b)}} \mathbf{a}_\ell^{(b-1)} \right] \\
&= \frac{\partial}{\partial \mathbf{z}_i^{(a)}} \left[\frac{\partial \mathcal{L}}{\partial \mathbf{z}_k^{(b)}} \mathbf{a}_\ell^{(b-1)} \right] \cdot \frac{\partial \mathbf{z}_i^{(a)}}{\partial \mathbf{w}_{i,j}^{(a)}} \\
&= \frac{\partial}{\partial \mathbf{z}_i^{(a)}} \left[\frac{\partial \mathcal{L}}{\partial \mathbf{z}_k^{(b)}} \mathbf{a}_\ell^{(b-1)} \right] \cdot \mathbf{a}_j^{(a-1)} \\
&= \frac{\partial^2 \mathcal{L}}{\partial \mathbf{z}_i^{(a)} \partial \mathbf{z}_k^{(b)}} \cdot \mathbf{a}_\ell^{(b-1)} \mathbf{a}_j^{(a-1)}
\end{aligned}
$$

We claim that $\frac{\partial \mathcal{L}}{\partial \mathbf{z}_i^{(a)} \partial \mathbf{z}_k^{(b)}}$ is negligible for $a < b$. Thus, we have the following.

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{z}_i^{(a)} \partial \mathbf{z}_k^{(b)}} &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}_i^{(a)}} \left[\frac{\partial \mathcal{L}}{\partial \mathbf{z}_k^{(b)}} \right] \\
&= \frac{\partial \mathcal{L}}{\partial \mathbf{z}_k^{(b)}} \left[\frac{\partial \mathcal{L}}{\partial \mathbf{z}_k^{(b)}} \right] \cdot \frac{\partial \mathbf{z}_k^{(b)}}{\partial \mathbf{z}_i^{(a)}} \\
&= \frac{\partial^2 \mathcal{L}}{\partial \left(\mathbf{z}_k^{(b)} \right)^2} \cdot \frac{\partial \mathbf{z}_k^{(b)}}{\partial \mathbf{z}_i^{(a)}}
\end{aligned}
$$

$\frac{\partial \mathbf{z}_k^{(b)}}{\partial \mathbf{z}_i^{(a)}}$ involves repeated multiplication, which will tend to yield negligible values. Intuitively, the output of a neuron will likely not have a significant interaction with neurons in future layers. This effect tends to decline as the neurons are farther apart.

Thus, we can assume that we only need to calculate the Hessian for each layer. Thus, we will focus on computing $\mathbf{H}_{\mathcal{L}}{}^{\left(\mathbf{w}^{(\ell)}\right)}$. From above, we have that

$$
\mathbf{H}_{\mathcal{L}}{}^{\left(\mathbf{w}^{(\ell)}\right)}_{(a,b),(c,d)} = \frac{\partial^2 \mathcal{L}}{\partial \mathbf{z}_a^{(\ell)} \partial \mathbf{z}_c^{(\ell)}} \cdot \mathbf{a}_b^{(\ell-1)} \mathbf{a}_d^{(\ell-1)}.
$$

Thus, we have that

$$
\mathbf{H}_{\mathcal{L}}{}^{\left(\mathbf{w}^{(\ell)}\right)} = \nabla^2_{\mathbf{z}^{(\ell)}} \mathcal{L} \otimes \mathbf{a}^{(\ell-1)} \mathbf{a}^{(\ell-1)\mathsf{T}}.
$$

We get the following for our minimization problem.

$$
\arg\min_{\Delta \mathbf{w}^{(\ell)}} \frac{1}{2} \Delta \mathbf{w}^{(\ell)\mathsf{T}} \mathbb{E} \left[\mathbf{H}_{\mathcal{L}}{}^{\left(\hat{\mathbf{w}}^{(\ell)}\right)} \right] \Delta \mathbf{w} = \arg\min_{\Delta \mathbf{w}^{(\ell)}} \mathbb{E} \left[\Delta \mathbf{w}^{(\ell)\mathsf{T}} \left(\mathbf{H}_{\mathcal{L}}{}^{\left(\mathbf{z}^{(\ell)}\right)} \otimes \mathbf{a}^{(\ell-1)} \mathbf{a}^{(\ell-1)\mathsf{T}} \right) \Delta \mathbf{w}^{(\ell)} \right]
$$

$$= \arg\min_{\Delta \mathbf{w}^{(\ell)}} \mathbb{E}\left[\left(\Delta \mathbf{W}^{(\ell)} \mathbf{a}^{(\ell-1)}\right)^{\mathsf{T}} \mathbf{H}_{\mathcal{L}}{}^{\left(\mathbf{z}^{(\ell)}\right)} \left(\Delta \mathbf{W}^{(\ell)} \mathbf{a}^{(\ell-1)}\right)\right]$$

Calculating $\mathbf{H}_{\mathcal{L}}{}^{\left(\mathbf{z}^{(\ell)}\right)} = \nabla^2_{\mathbf{z}^{(\ell)}} \mathcal{L}$ is still infeasible given the quadratic nature of the Hessian. Thus, AdaRound makes the assumption that $\nabla^2_{\mathbf{z}^{(\ell)}} \mathcal{L}$ is diagonal with diagonal entries c_i that are constant with respect to the data. This is a big assumption that will later be relaxed in brecq.

Under this assumption, we maximize the following objective.

$$\arg\min_{\Delta \mathbf{w}^{(\ell)}} \frac{1}{2} \Delta \mathbf{w}^{(\ell)\mathsf{T}} \mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{\left(\hat{\mathbf{w}}^{(\ell)}\right)}\right] \Delta \mathbf{w} \approx \arg\min_{\Delta \mathbf{w}^{(\ell)}} \mathbb{E}\left[\sum_{i=1}^{d_i} c_i \left(\Delta \mathbf{w}_i^{(\ell)} \mathbf{a}^{(\ell-1)}\right)^{\mathsf{T}} \left(\Delta \mathbf{w}_i^{(\ell)} \mathbf{a}^{(\ell-1)}\right)\right]$$

We can see that minimizing the second-order term can be done in a piecemeal fashion by minimizing $\hat{\mathbf{w}}_i^{(\ell)}$ or each row of the weight matrix for layer ℓ separately. Furthermore, we choose the $\hat{\mathbf{w}}_i^{(\ell)}$ that minimizes the $\Delta \mathbf{z}_i^{(\ell)}$. This is a very strong result. It says that we should quantize such that the approximate pre-activation difference is minimized, as opposed to minimizing the magnitude of $\Delta \mathbf{w}$.

However, the problem still remains that we need to choose $\Delta \mathbf{w}$ such that $\hat{\mathbf{w}} + \Delta \mathbf{w} \in \mathcal{Q}_b$. To do this, AdaRound takes $\mathcal{Q}_b$, the quantization grid, to be fixed. Then, they choose whether to round up or down for each weight, and they want the combination that minimizes the above objective. However, solving this discrete optimization is proven to be NP-hard. Thus, they use approximate methods to determine the optimal rounding scheme. We leave the full details to [NAvB+20].

Note that AdaRound does not quite fit the naive quantization schemes described in the previous section. In particular, although we are still deciding whether we should round up or down, this decision is informed by the other weights. In particular, that is AdaRound's key thesis that the interaction between weights should not be ignored.

5.4.2 brecq [LGT+21]

brecq extends the analysis from AdaRound by relaxing a couple of assumptions. First, AdaRound assumes that the Hessian with respect to the pre-activations is diagonal with constants that are independent of the data. brecq uses the Fisher Information Matrix (FIM) to approximate $\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{\left(\mathbf{z}^{(\ell)}\right)}\right]$ instead. The Fisher Information Matrix is defined to be the negative expected value of the Hessian, $-\bar{\mathbf{H}}_{\mathcal{L}}$. Assuming that the training data comes from the same distribution as the true data distribution, we have that the Hessian $\mathbf{H}_L$ with respect to the objective function is an unbiased estimate of $\bar{\mathbf{H}}_{\mathcal{L}}$. Furthermore, it is a well-known

result that

$$\mathbb{E}\left[\left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right)^2\right] = -\mathbb{E}\left[\frac{\partial^2 \mathcal{L}}{\partial \mathbf{z}^{(\ell)2}}\right] = -\bar{\mathbf{H}}_{\mathcal{L}}^{\left(\mathbf{z}^{(\ell)}\right)} \approx \mathbf{H}_{L}^{\left(\mathbf{z}^{(\ell)}\right)}$$

for a cross-entropy loss function. brecq still assume that $\mathbf{H}_{\mathcal{L}}{}^{\left(\mathbf{z}^{(\ell)}\right)}$ is diagonal, but they do not assume data independence. Thus, we have the natural approximation

$$\mathbf{H}_{L}{}^{\left(\mathbf{z}^{(\ell)}\right)} \approx \operatorname{diag}\left(\left\{\frac{\partial^2 L}{\partial \mathbf{z}^{(\ell)2}}\right\}\right) \approx \operatorname{diag}\left(\left\{\left(\frac{\partial L}{\partial \mathbf{z}^{(\ell)}}\right)^2\right\}\right).$$

This follows by the linearity of the expectation and derivative operator. This gradient has already been computed when training the model, and its size is linear with the number of the weights.

AdaRound also made the assumption that interactions between layers were negligible and took a "layer-wise" reconstruction approach. However, brecq argues that there are other granularities that we can choose other than the layer level. For example, in CNNs, there is a notion of a "block" which contains a series of layers and a skip connection. They claim that quantizing using a "block-wise" reconstruction strikes the right balance in terms of bias and variance. Rather surprisingly, they show that reconstruction using a Hessian for the entire network performs worse than layer or block-wise reconstruction. The authors argue that using the entire network leads to high generalization error. They illustrate this idea in Table 5.1, which shows the accuracy for a ResNet-18 model trained using each reconstruction approach to a 2-bit quantized model. From the table, we observe that the best accuracy occurs when using a block-wise reconstruction approach.

Table 5.1: Block-Wise Reconstruction [LGT+21]

Model	Layer	Block	Stage	Net
ResNet-18	65.19	**66.39**	66.01	54.15

The final distinction between AdaRound and brecq is that $\Delta \mathbf{w}_i$ is restricted to two values, since the weight can only be rounded up or down. However, brecq gives a more general framework, where we can freely choose $\Delta \mathbf{w}_i$ such that $\mathbf{w}_i + \Delta \mathbf{w}_i \in \mathcal{Q}_{i,b_i}$. Learning this $\Delta \mathbf{w}_i$ falls beyond the scope of the thesis, but one can refer to [LGT+21] for more detailed learning strategies.

Together, AdaRound and brecq use a robust theoretical framework to achieve very little degradation while quantizing to extremely low bitwidth settings. Most of this analysis still guides the literature today. Even with the shift to LLMs, we still see algorithms like GPTQ

attempting to optimize this second-order term, albeit with looser assumptions [FAHA23]. We will have a complete discussion of these methods in Chapter 10.

As seen in the results below, these methods can achieve up to two-bit quantization while losing less than 10% accuracy compared to models that use full precision. Note that both of these algorithms require additional passes to approximate the change in pre-activation values. However, given a pre-trained model, these overhead times are usually negligible. Furthermore, this is all still a one-time expense, since we can then reuse these quantized weights for all inference workloads.

Table 5.2: Results [LGT+21]

Methods	Bits (Weights/Activations)	ResNet-18	ResNet-50
Full Prec.	32/32	71.08	77.00
AdaRound	4/32	68.71	75.23
brecq	4/32	**70.70**	**76.29**
AdaRound	3/32	68.07	73.42
brecq	3/32	**69.81**	**75.61**
AdaRound	2/32	55.96	47.95
brecq	2/32	**66.30**	**72.40**

5.5 Hardware Considerations

Now, we discuss hardware concerns related to quantization. The motivation for quantization lies with memory-bound tasks, as discussed in Chapter 2. In particular, many machine learning workloads are memory-bound, since there is a lot of data movement for the activations and weights. Therefore, quantization allows us to increase the arithmetic intensity by decreasing the number of bytes that need to be read to and from memory. However, accelerating quantized machine learning models presents some of its own challenges that we will now discuss.

5.5.1 Arithmetic Units

This thesis mainly focuses on weight-only quantization. Therefore, once the weights are loaded into local memory, they are dequantized to the precision of the compute fabric (typically FP16/FP32) prior to computation. Therefore, we can continue to use FP16/FP32 arithmetic units to perform all computation. However, if both weights and activations are quantized and the quantization scheme is uniform, then we can use arithmetic units to compute the matrix multiplication results natively in $\mathcal{B}$, as discussed in a prior section. In such

cases, we need to add arithmetic units to support arithmetic in the quantized data type. We will discuss the options for this much more thoroughly in Chapter 9.

5.5.2 Packing and Unpacking Kernels

Typically, instruction set architectures (ISAs) do not support loading and storing of data of arbitrary bitwidth. Therefore, especially for low bitwidth quantization schemes, we need to "pack" the bits within a larger load or store format that is supported by the ISA. Packing kernels "pack" the bits into the larger number format, while unpacking kernels "unpack" the bits from the larger number format into the desired number format. We will later discuss [FAHA23] in Chapter 10. We refer the interested reader to [FAHA23] for examples of packing and unpacking kernels.

6 Sparsity

Similar to quantization, sparsity is another model compression method that helps to deploy machine learning models in constrained environments. Whereas quantization decreased the precision of the weights by using a less precise data type, sparsity decreases the precision of the weights by requiring that $p\%$ of the values be zero. Having zeros in these values is desirable for a few reasons. First, zero values are easy to store, since no precision is needed. Furthermore, in computation, we have the convenient property that $a+0 = a$ and $a \cdot 0 = 0$ for all $a \in \mathbb{R}$. Therefore, zeros make computation easier, since the computation can be essentially skipped. Thus, similar to the previous chapter, in this chapter, we discuss sparsity strategies, as well as their hardware implications. We take inspiration from [HABN+21].

6.1 Mathematical Formulation

We approach sparsity with the same setup as quantization. In this section, we will present a "naive" formulation, which we will later complicate. As discussed in Chapter 1, let $f : \mathcal{X} \to \mathcal{Y}$ be our model of interest, and let it be parameterized by weights $\mathbf{w} \in \mathbb{R}^d$. We will assume that the optimal weights $\hat{\mathbf{w}} \in \mathbb{R}^d$ minimizes L. We will assume that the training procedure returns these optimal weights $\hat{\mathbf{w}}$.

When we sparsify, similar to quantization, we reduce the accuracy of our trained weights

by limiting them to some pre-chosen values, which we will refer to as $\tilde{\mathbf{w}}$. However, instead of using a less precise data type, we map some $\mathbf{w}_i$ to 0 to generate $\tilde{\mathbf{w}}$. We want to minimize $L(\tilde{\mathbf{w}})$, where L is the objective function from Chapter 1. Furthermore, we will require that $|\{i \in \{1, \ldots, d\} : \tilde{\mathbf{w}}_i = 0\}| \geq p\% \cdot d$. In other words, $p\%$ of the weights are 0.

In many ways, we can view the setup of sparsity to be analogous to quantization. This is not the canonical approach, but it illuminates some underlying similarities between them. Like quantization, it assumes that there is some function $Q_i : \mathbb{R}^d \to \{0, 1\}$ for each weight. Let $\mathbf{Q} : \mathbb{R}^d \to \{0, 1\}^d$ be defined by $\mathbf{Q}(\mathbf{w})_i = Q_i(\mathbf{w})$. However, to get the "desparsified" values, we use the output of Q applied to every element of $\mathbf{w}$ as a mask. In particular, we have that $D_i : \mathbb{R} \times \{0, 1\} \to \mathbb{R}$ defined by $(\mathbf{w}_i, \mathbf{m}_i) \mapsto \mathbf{w}_i \times \mathbf{m}_i$. Given weights $\mathbf{w}$ and mask $\mathbf{m}$, let $\mathbf{D} : \mathbb{R}^d \times \{0, 1\}^d$ be defined by $\mathbf{w} \odot \mathbf{m}$, where $\odot$ is the elementwise product of $\mathbf{w}$ and $\mathbf{m}$. We can think of $\mathbf{m}$ as a mask of $\{0, 1\}$, such that it maps $\mathbf{w}_i$ to either 0 or itself. Then, our final output is $\tilde{\mathbf{w}} = \mathbf{D}(\mathbf{w}, \mathbf{Q}(\mathbf{w}))$. The Q function is similar between quantization and sparsity. In particular, we can view sparsity's Q function as a more specific version of the quantization version, where $\mathcal{B} = \text{INT}_1$, since we choose either 0 or 1.

Once the Q function is defined, the two methods diverge. In particular, quantization has a "dequantization" function D that undoes the work of the Q function. However, sparsity uses the output of the Q as a mask. This distinction is quite important, and we discuss its implications in Chapter 10. When searching for the optimal sparsity scheme, it suffices to define Q for each weight $\mathbf{w}_i$. Thus, we can define a sparsity scheme as $\{Q_i\}_{i=1}^d$. A "valid" sparsity scheme is one such that $|\{i \in \{1, \ldots, d\} : Q_i(\mathbf{w}) = 0\}| \geq p\% \cdot d$ for all $\mathbf{w} \in \mathbb{R}^d$. We sometimes refer to the method of finding the optimal valid sparsity scheme as pruning.

A key observation is that defining a valid $\{Q_i\}_{i=1}^d$ is equivalent to defining a score (or saliency) function $S : \mathbb{R}^d \to \mathbb{R}$ for each weight $\mathbf{w}_i$. The intuition is that we can calculate the score for each weight $\mathbf{w}_i$. Now, consider the set $S(\mathbf{w}) = \{S_i(\mathbf{w})\}_{i=1}^d$. Then, we select some $k \geq p\% \cdot d$. Let $S(\mathbf{w})_{(k)}$ be the kth element when $S(\mathbf{w})$ is sorted by magnitude. Then, for all $S_i(\mathbf{w}) \leq S(\mathbf{w})_{(k)}$, we map the weight $\mathbf{w}_i$ to 0. Thus, we define $\{S_i\}_{i=1}^d$ to be a scoring scheme. Typically, we want the same score function to be used for each weight, so we can do a "apples-to-apples" comparison, but this is not strictly necessary.

For notational simplicity, let $\mathbf{S} : \mathbb{R}^d \to \mathbb{R}^d$ be the result of applying S_i to the corresponding weight $\mathbf{w}_i$. Then, let $\mathbf{T}_k : \mathbb{R}^d \to \mathbb{R}^d$ be the result of applying $\mathbb{I}\left(S_i(\mathbf{w}_i) > S(\mathbf{w})_{(k)}\right)$ to each element. Intuitively, it maps the weights that have scores greater than the kth highest score to 1, and the rest to 0.

We will now prove this equivalence between sparsity and scoring schemes rigorously. We will prove that for a "valid" sparsity scheme, there exists a scoring scheme that produces the same output. Then, we prove the reverse direction that for a scoring scheme, there exists a

"valid" sparsity scheme that produces the same output.

First, we prove the forward direction. Let $\{Q_i\}_{i=1}^d$ be a "valid" sparsity scheme. Then, consider the following scoring scheme. Define S_i by the mapping $\mathbf{w} \mapsto Q_i(\mathbf{w})$. Let $k = |i \in \{1, \ldots, d\} : S_i(\mathbf{w}) = 0|$. Since the sparsity scheme is valid, we have that $k \geq p\% \cdot d$. Then, we map $\mathbf{w}_i$ to 0 iff $Q_i(\mathbf{w}) = 0$, as desired.

Now, we prove the reverse direction. Suppose we have a scoring scheme $\{S_i\}_{i=1}^d$. Then, define Q_i to be the map such that $\mathbf{w} \mapsto \mathbb{I}\left(S_i(\mathbf{w}) > S(\mathbf{w})_{(k)}\right)$. Then, $\mathbf{w}_i = 0$ iff $S_i(\mathbf{w}) > S(\mathbf{w})_{(k)}$ as desired. By showing that these two perspectives are equivalent, we can use them interchangeably and choose the one that is more convenient to reason about.

6.2 Sparsity Variants

Like in Chapter 5, we present variations of the above formulation for sparsity. Many of these variations follow very similarly to the variants for quantization. However, it should be noted that there are fewer degrees of freedom in the sparsity problem. In particular, assuming a naive setup, we should note that there are a finite number of sparsity schemes. Given a weight vector of $\mathbf{w} \in \mathbb{R}^d$ there are 2^d possible outputs of $\tilde{\mathbf{w}}$. However, for quantization, there are infinitely many $\tilde{\mathbf{w}}$ that can be outputted, since we can choose any real value to include in the quantization grid. This can be seen from our formulation above, since we have essentially restricted the quantization problem by enforcing that $\mathcal{B} = \text{INT}_1$.

6.2.1 Model vs. Ephemeral

Sparsity is a very general term that can be applied in many different contexts with respect to deep learning. In this thesis, we focus on weight sparsity, which attempts to induce sparsity given some weights to model. One can consider this as a form of "offline" sparsity, where sparsity is performed prior to deployment. However, we might also consider inducing sparsity for specific examples in a "online" fashion. [HABN+21] refers to this as ephemeral sparsification, where the sparsification is applied to specific examples. The use of activation functions is an example of this. For some examples, depending on the input value, the activation function may push the output closer to 0. For example, using a ReLU activation function pushes the activation to 0 if the input is negative. In addition, we can consider changing the model architecture to create some "sparsity" within the architecture. For example, CNNs can be viewed as a sparsified version of a fully-connected MLP, since the kernel weights are shared.

6.2.2 Structured vs. Unstructured

One of the most important variations on sparsity schemes is the use of "structure" within the sparsity scheme. This has important hardware ramifications that we will soon discuss. Intuitively, we want to define some notion of structure, where a sparsity scheme is more structured if the output mask has some "pattern". First, we formalize this mathematically. Let $T \subseteq \{1, \ldots, d\}$. Then, we will say that a sparsity scheme $\{Q_i\}_{i=1}^d$ is structured on a set T if for all $\hat{\mathbf{w}} \in \mathbb{R}^d$, we have that $|\{i \in T : \mathbf{m}_i = 0\}| \geq t\% \cdot |T|$ for some $t \in (0, 100]$, where $\mathbf{m}$ is the associated mask. This definition requires that for a given set of indices, at least $t\%$ of them must be zero regardless of the input weight $\hat{\mathbf{w}}$. Then, we know that there is some "structure" on this set of indices.

S essentially gives us the freedom to choose the granularity of the "pattern" within sparsification. At the extreme, we could have $T = \{1, \ldots, d\}$. However, this does add any "structure", since it simply requires that at least $t\%$ of all weights be 0 for some $t \in (0, 100]$, which is always true if the sparsity scheme is valid. On the other extreme, we could have $|T| = 1$. Then, we require that a single weight always be zero regardless of the input $\hat{\mathbf{w}}$. We can quickly see that choosing a smaller T and larger T enforces more "structure". Thus, like in quantization, choosing an appropriate S and t is one of the main challenges. The accuracy of a structured sparsity scheme will be at most the accuracy of an unstructured sparsity scheme, since the structure enforces some additional conditions. However, the upshot is that the added structure allows the weights to be efficiently stored and computed on. We will discuss these ideas shortly.

It should be noted that given a scoring scheme $\{S_i\}_{i=1}^d$ we can easily transform a scheme from unstructured to structured on T. In particular, for the set T, we can select some $\ell \geq t\%|T|$. Then, let $S_T(\mathbf{w})_{(\ell)}$ be the ℓth element when $S_T(\mathbf{w}) = \{S_i(\mathbf{w}) : i \in T\}$ is sorted by magnitude. Then, for $i \in T$, we can map $\mathbf{w}_i$ to 0 iff $S_i(\mathbf{w}_i) \leq S_T(\mathbf{w})_\ell$. Intuitively, we take all weights with score below the ℓth score in numerical order and map all of the corresponding weights to 0.

For a more concrete example, consider adding structure at the tensor-level, which is similar in spirit to per-tensor quantization. One desirable trait of a sparsity scheme is that for every tensor we have that at least $t\%$ of that tensor will be 0. Suppose that the parameters consists of tensors $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(m)}$. Then, let T_i be the corresponding indices of $\mathbf{w}$ for tensor i. Then, we can require that the sparsity scheme be structured on T_i for all $1 \leq i \leq m$. One can quickly see that we can extend this characterization to any granularity, as discussed in Chapter 5. An example that we will soon discuss is 2:4 sparsity, which was popularized by the release of NVIDIA A100 GPUs. In this case, the granularity T is $\{4(i-1)+1, 4(i-1)+2, 4(i-1)+3, 4(i-1)+4\}$ for all $i \in \{1, \ldots, \lfloor \frac{d}{4} \rfloor\}$ and $t = 50$. Here,

for an aligned set of four consecutive weights, we require that 50% of them be zero.

6.2.3 Timing of Sparsity

Similar to the distinction of PTQ and QAT, sparsity can also be applied at different times in the training schedule. In particular, like PTQ, sparsity may be applied after training has fully completed. On the other hand, like QAT, sparsity may be applied during the training process. For example, the model may be trained incrementally with sparsity being applied between episodes of training. However, the framework that we have presented above works for either situation. Regardless of when sparsity is applied, we are always faced with the same decision of choosing the connections to remove or preserve. Knowing the model has been fully trained, and hence converged, will allow us to make additional assumptions. Thus, like in the previous chapter, we mainly focus on the case where sparsity is applied after training has completed.

6.2.4 Data Dependent vs. Independent

Finally, like in quantization, training data can be used to find an optimal sparsity scheme. As discussed previously, these data-dependent pruning strategies may be undesirable because they require additional training and access to training data that may not be publicly available. However, it may sometimes be necessary to preserve more accuracy. We will see pruning methods of both variations, data-free and data-dependent, shortly.

6.3 Sparsity Methods

Now, we discuss pruning methods that we feel are most pertinent to this thesis. This is by no means a comprehensive review of sparsity methods, but it illuminates the key methods that will be useful for later parts of the thesis. For a fuller review of sparsity methods, we suggest the interested reader review [HABN+21].

6.3.1 Magnitude-Based Methods

One of the simplest, yet most intuitive sparsity methods, is magnitude-based pruning. As its name suggests, we use the magnitude of a weight to measure its saliency. Thus, let $S_i : \mathbb{R}^d \to \mathbb{R}$ be defined by $\mathbf{w} \mapsto |\mathbf{w}_i|$. Then, we will prune all weights less than or equal to $\mathbf{w}_{(k)}$, where $\mathbf{w}_{(k)}$ is the kth element when $\mathbf{w}$ is sorted by magnitude. This pruning method is motivated by the fact that large weights tend to have more of a "effect" on the output of the network and maintaining them is necessary to preserving performance. Although quite

simple, we will see that magnitude-based pruning falls naturally from other more complex methods that we will soon discuss. Another upshot of magnitude-based methods it that it is data-free, since it solely depends on the weights.

6.3.2 Second-Order Methods

Similar to our discussion of quantization methods in Chapter 5, we can use second-order information as a proxy for salience. In this section, we will continue to illuminate the many similarities that quantization and sparsity share by viewing them under this unified framework.

Recall from Chapter 5 that we viewed quantization as a method of weight perturbation. Using this perspective, we were able to approximate the expected change in loss, via a Taylor expansion, to be

$$\mathbb{E}\left[\mathcal{L}\left(f(\mathbf{x};\hat{\mathbf{w}}+\Delta\mathbf{w}),\mathbf{y}\right)-\mathcal{L}\left(f(\mathbf{x};\hat{\mathbf{w}}),\mathbf{y}\right)\right]\approx\frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]\Delta\mathbf{w}.$$

This allowed us to reduce the quantization problem to simply choosing a $\Delta\mathbf{w}$ such that $\frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]\Delta\mathbf{w}$ was minimized and $\hat{\mathbf{w}}_i+\Delta\mathbf{w}_i\in\mathcal{Q}_{i,b_i}$. We can view sparsity under a similar framework. Similar to quantization, sparsity can be viewed as another method of "perturbing" weights, since we are simply forcing some weights to be zero. Thus, we can again use the Taylor expansion as an approximation of the expected change in loss. Again, we want to choose a $\Delta\mathbf{w}$ such that $\frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]\Delta\mathbf{w}$ is minimized. However, instead of requiring that $\hat{\mathbf{w}}_i+\Delta\mathbf{w}_i\in\mathcal{Q}_{i,b_i}$, we now want $p\%$ of the weights to be zero. Thus, we require that $|\{i\in\{1,\ldots,d\}:\hat{\mathbf{w}}_i+\Delta\mathbf{w}_i=0\}|=p\%\cdot d$.

Optimal Brain Damage (OBD) [LDS89]

OBD makes the simplifying assumption that $\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]=\operatorname{diag}\left(\{h_{ii}\}\right)$ is a diagonal matrix. Then, we have that

$$L\left(\hat{\mathbf{w}}+\Delta\mathbf{w}\right)-L\left(\hat{\mathbf{w}}\right)\approx\frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]\Delta\mathbf{w}=\frac{1}{2}\sum_{i=1}^{d}h_{ii}\Delta\mathbf{w}_i^2.$$

Intuitively, this is similar to the naive rounding-to-nearest quantization scheme that preceded AdaRound. This is because we ignore the interactions between weights. Instead, we try to minimize the sum of squared difference between the original and perturbed weights. Thus, we can assign a score defined by

$$S_i\left(\mathbf{w}\right)\triangleq\frac{1}{2}h_{ii}\mathbf{w}_i^2.$$

Again, we can assume that the training data comes from the same distribution as the true data generating process, so we can approximate $\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]$ as $\mathbf{H}_L^{(\hat{\mathbf{w}})}$.

We claim that computation of the diagonal elements of the second-order term has the same asymptotic computational complexity as backpropagation itself. This comes from the following observation. Suppose we want to compute $\frac{\partial^2 \mathcal{L}}{\partial \mathbf{w}_i^2}$ for some $i \in \{1, \ldots, d\}$. Let z be some intermediate variable, and v be the parameter of interest. From the chain rule, we have that

$$\frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial z} \frac{\partial z}{\partial v}.$$

Then, we have that

$$\frac{\partial^2 \mathcal{L}}{\partial v^2} = \frac{\partial \mathcal{L}}{\partial z} \frac{\partial^2 z}{\partial v^2} + \frac{\partial z}{\partial v} \frac{\partial \frac{\partial \mathcal{L}}{\partial z}}{\partial v} = \frac{\partial \mathcal{L}}{\partial z} \frac{\partial^2 z}{\partial v^2} + \left(\frac{\partial z}{\partial v}\right)^2 \frac{\partial^2 \mathcal{L}}{\partial z^2}$$

by the product rule. Thus, we can run an initial backwards pass, where we "pass back" $\frac{\partial \mathcal{L}}{\partial z}$ for all intermediate values z. Then, we can run a second backwards pass, where we "pass back" $\frac{\partial^2 \mathcal{L}}{\partial z^2}$ for all intermediate values z. Thus, while we do two backwards passes, the asymptotic computational complexity remains the same.

If $\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right] = \mathbf{I}_{d \times d}$, we have that the saliency score reduces to

$$S_i(\mathbf{w}) = \frac{1}{2}\mathbf{w}_i^2,$$

which we claim yields the same result as magnitude-based pruning. To show two scoring schemes $\{S_i\}_{i=1}^d$ and $\{R_i\}_{i=1}^d$, it suffices to show that for all $i, j \in \{1, \ldots, d\}$, we have that the order is preserved. More formally, we can show that there exists a strictly increasing function $f : \mathbb{R} \to \mathbb{R}$ such that $S_i(\mathbf{w}) = f(R_i(\mathbf{w}))$. In this case, we have that $f : \mathbb{R} \to \mathbb{R}$ defined by $x \mapsto \frac{1}{2}x^2$ suffices, since $\frac{1}{2}|\mathbf{w}_i|^2 = \frac{1}{2}\mathbf{w}_i^2$. This gives further credence to the claim that magnitude-based pruning is a simply but relatively good pruning method.

Optimal Brain Surgeon (OBS) [HS92]

Similar to AdaRound, OBS relaxes the assumption that $\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]$ is a diagonal matrix. Furthermore, it complicates the naive sparsity scheme by not only applying a mask but also adjusting the weight values through an additive bias. As in OBD, we are trying to minimize the following quantity

$$L(\hat{\mathbf{w}} + \Delta\mathbf{w}) - L(\hat{\mathbf{w}}) \approx \frac{1}{2}\Delta\mathbf{w}^\mathsf{T}\mathbb{E}\left[\mathbf{H}_{\mathcal{L}}{}^{(\hat{\mathbf{w}})}\right]\Delta\mathbf{w} \approx \frac{1}{2}\Delta\mathbf{w}^\mathsf{T}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}.$$

Suppose we want to prune a single weight $\mathbf{w}_q$. Then, we choose a $\Delta\mathbf{w}$, such that $\Delta\mathbf{w}_q + \mathbf{w}_q = 0$. Here, note that we will find an optimal vector $\Delta\mathbf{w}$ instead of simply choosing a weight to prune. This will allow us to later update all the other weights. Equivalently, we have that $\mathbf{e}_q^\mathsf{T}\Delta\mathbf{w} + \mathbf{w}_q = 0$, where $\mathbf{e}_q$ is the standard basis vector where the qth entry is 1 and all other entries are 0. Thus, we want to solve for

$$\min_{\Delta\mathbf{w}} \frac{1}{2}\Delta\mathbf{w}^\mathsf{T}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w} \quad \text{s.t.} \quad \mathbf{e}_q^\mathsf{T}\Delta\mathbf{w} + \mathbf{w}_q = 0.$$

Then, we can use a Lagrangian to solve for the problem. In particular, we can construct the following Lagrangian function

$$G\left(\Delta\mathbf{w}, \lambda\right) = \frac{1}{2}\Delta\mathbf{w}^\mathsf{T}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w} + \lambda\left(\mathbf{e}_q^\mathsf{T}\Delta\mathbf{w} + \mathbf{w}_q\right).$$

Then, we have that

$$\frac{\partial G}{\partial \Delta\mathbf{w}} = \mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w} + \lambda\mathbf{e}_q = \mathbf{0}$$

by the first-order condition. Thus, we have that

$$\Delta\mathbf{w} = -\lambda\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\mathbf{e}_q$$

for the optimal $\Delta\mathbf{w}$. Furthermore, we also have that

$$\frac{\partial G}{\partial \lambda} = \mathbf{e}_q^\mathsf{T}\Delta\mathbf{w} + \mathbf{w}_q = \mathbf{0}.$$

Substituting, we have that

$$\mathbf{e}_q^\mathsf{T}\left(-\lambda\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\mathbf{e}_q\right) + \mathbf{w}_q = -\lambda\left(\mathbf{e}_q^\mathsf{T}\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\mathbf{e}_q\right) + \mathbf{w}_q = -\lambda\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq} + \mathbf{w}_q = 0.$$

Thus, we have that

$$\lambda = \frac{\mathbf{w}_q}{\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}$$

at the optimum. Thus, we have that

$$\Delta\mathbf{w} = -\frac{\mathbf{w}_q\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\mathbf{e}_q}{\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}.$$

Thus, we have the following for $L(\hat{\mathbf{w}} + \Delta\mathbf{w}) - L(\hat{\mathbf{w}})$ at the optimum.

$$
\begin{aligned}
L(\hat{\mathbf{w}} + \Delta\mathbf{w}) - L(\hat{\mathbf{w}}) &= \frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w} \\
&= \frac{1}{2}\left(-\frac{\mathbf{w}_q\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\mathbf{e}_q}{\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}\right)^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\left(-\frac{\mathbf{w}_q\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\mathbf{e}_q}{\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}\right) \\
&= \frac{1}{2}\left(\frac{\mathbf{w}_q}{\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}\right)^2\left(\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\mathbf{e}_q\right)^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\left(\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\mathbf{e}_q\right) \\
&= \frac{1}{2}\left(\frac{\mathbf{w}_q}{\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}\right)^2\left(\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\mathbf{e}_q\right)^{\mathsf{T}}\mathbf{e}_q \quad (\mathbf{H}_L^{(\hat{\mathbf{w}})}\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1} = \mathbf{I}) \\
&= \frac{1}{2}\left(\frac{\mathbf{w}_q}{\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}\right)^2\mathbf{e}_q^{\mathsf{T}}\left(\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right)^{\mathsf{T}}\mathbf{e}_q \\
&= \frac{1}{2}\left(\frac{\mathbf{w}_q}{\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}\right)^2\mathbf{e}_q^{\mathsf{T}}\left(\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{\mathsf{T}}\right)^{-1}\mathbf{e}_q \\
&= \frac{1}{2}\left(\frac{\mathbf{w}_q}{\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}\right)^2\mathbf{e}_q^{\mathsf{T}}\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\mathbf{e}_q \qquad (\mathbf{H}_L^{(\hat{\mathbf{w}})} \text{ is symmetric}) \\
&= \frac{1}{2}\left(\frac{\mathbf{w}_q}{\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}\right)^2\cdot\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq} \\
&= \frac{\mathbf{w}_q^2}{2\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}
\end{aligned}
$$

Note that this is the optimum for choosing to prune weight $\mathbf{w}_q$. Thus, we will refer to $\Delta\mathbf{w}$ as $\Delta\mathbf{w}^{(q)}$, since there is one for every weight $\mathbf{w}_q$. First, we need to decide which q to choose.

As we have shown above, we can assign a score function defined by

$$S_q(\mathbf{w}) \triangleq \frac{\mathbf{w}_q^2}{2\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{qq}}.$$

We want to choose the weight that has the smallest saliency. In this setting, the smallest saliency results in the smallest change in loss. Again, we can see that under the assumption $\mathbf{H}_L^{(\hat{\mathbf{w}})} = \mathbf{I}$, then, we again have that the saliency reduces to simply magnitude-based pruning.

Suppose the weight with the smallest saliency is weight $\mathbf{w}_{q'}$. Then, we can add

$$\Delta \mathbf{w}^{(q')} = -\frac{\mathbf{w}_{q'}\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\mathbf{e}_{q'}}{\left[\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}\right]_{q'q'}}$$

to all the weights. Thus, not only, do we prune weight $\mathbf{w}_{q'}$, we also update other weights as well.

Given this setup, we need to slightly update the naive formulation that we have previously presented. First, we need to account for the fact that the other weights can be updated. Previously, we defined a "desparsification" function $\mathbf{D} : \mathbb{R}^d \times \{0,1\}^d \to \mathbb{R}^d$ defined by $(\mathbf{w}, \mathbf{m}) \mapsto \mathbf{w} \odot \mathbf{m}$. However, we will extend this framework by adding a bias term. In particular, we define $\mathbf{D}' : \mathbb{R}^d \times \mathbb{R}^d \times \{0,1\}^d \to \mathbb{R}^d$ defined by $(\mathbf{w}, \mathbf{b}, \mathbf{m}) \mapsto (\mathbf{w} + \mathbf{b}) \odot \mathbf{m}$. Then, we define a function $\mathbf{B} : \mathbb{R}^d \to \mathbb{R}^d$ such that $\tilde{\mathbf{w}} = \mathbf{D}'(\mathbf{w}, \mathbf{B}(\mathbf{w}), \mathbf{Q}(\mathbf{w}))$. Intuitively, we add the bias then apply the mask. Note that the application of the mask in OBS is unnecessary, since we already require that $\mathbf{e}_q^\intercal \Delta \mathbf{w} + \mathbf{w}_q = 0$. However, it may be useful in other frameworks where the $\mathbf{w} + \mathbf{b}$ may not have the desired sparsity.

Second, we need to account for the fact that OBS takes an iterative approach to the pruning process. In particular, at each step j, using a set of score function $\{S^{(j)}\}_{i=1}^d$, it choose some weight $\mathbf{w}_{q'}$ to prune and map to 0. Then, it updates all the other un-pruned weights. However, we can just view this as an iterated version of the framework we presented previously. At each time step, we define a score function, and then apply a transformation to the weights $\mathbf{w}$. Note that the score function needs to be recomputed at every time step, since $\mathbf{H}_L^{(\hat{\mathbf{w}})}$ will change at every iteration.

The main bottleneck in OBS is calculating $\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}$ for an entire network. OBS was suggested at a time when compute was limited and models contained thousands of weights. Now, models contain billions of weights making the construction of and inversion of the Hessian quite difficult. However, [HS92] proposes some optimizations to calculating the inverse

Hessian. Furthermore, we will revisit this idea in Chapter 10 to see further optimizations for CNNs and LLMs.

The following approach is very similar in spirit to the FIM approximation approach that we saw in Chapter 5. Assume a squared error loss with $\mathbf{y} \in \mathbb{R}^m$, such that $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2}(\mathbf{y} - \hat{\mathbf{y}})^\mathsf{T}(\mathbf{y} - \hat{\mathbf{y}})$. The assumption of a squared error loss is restricting, as we have seen cross-entropy loss as a common form for the probabilistic models. However, as we will discuss in Chapter 10, we will typically apply this framework at a layer level, where the squared error loss is more reasonable. We will use $\mathbf{y}^{[i]}$ and $\hat{\mathbf{y}}^{[i]}$ to refer to the true response and the estimated response for the ith observation, respectively. Then, we have that

$$L(\mathbf{w}) = \frac{1}{2N}\sum_{i=1}^{N}\left(\mathbf{y}^{[i]} - \hat{\mathbf{y}}^{[i]}\right)^\mathsf{T}\left(\mathbf{y}^{[i]} - \hat{\mathbf{y}}^{[i]}\right).$$

Thus, we have that

$$\frac{\partial L}{\partial \mathbf{w}} = -\frac{1}{N}\sum_{i=1}^{N}\frac{\partial \hat{\mathbf{y}}^{[i]}}{\partial \mathbf{w}}\left(\mathbf{y}^{[i]} - \hat{\mathbf{y}}^{[i]}\right).$$

Furthermore, we have that

$$\mathbf{H}_L^{(\hat{\mathbf{w}})} = \frac{\partial^2 L}{\partial \mathbf{w}^2} = \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{j=1}^{m}\frac{\partial \hat{\mathbf{y}}_j^{[i]}}{\partial \mathbf{w}}\frac{\partial \hat{\mathbf{y}}_j^{[i]}}{\partial \mathbf{w}}^\mathsf{T} - \sum_{j=1}^{m}\frac{\partial^2 \hat{\mathbf{y}}_j^{[i]}}{\partial \mathbf{w}^2}\left(\mathbf{y}_j^{[i]} - \hat{\mathbf{y}}_j^{[i]}\right)\right).$$

We will assume the second term is negligible. [HS92] justifies this for two reasons. First, assuming that the model is sufficiently well-trained $\left(\mathbf{y}_j^{[i]} - \hat{\mathbf{y}}_j^{[i]}\right)$ will be small for all $1 \leq j \leq m$. Furthermore, even if it is not negligible, we can view sparsity as a way of "generalizing" our model. However, this term encourages that we incorporate some of the "noise" from our original model. Thus, we are left with

$$\mathbf{H}_L^{(\hat{\mathbf{w}})} = \frac{\partial^2 L}{\partial \mathbf{w}^2} = \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{j=1}^{m}\frac{\partial \hat{\mathbf{y}}_j^{[i]}}{\partial \mathbf{w}}\frac{\partial \hat{\mathbf{y}}_j^{[i]}}{\partial \mathbf{w}}^\mathsf{T}\right).$$

This is very similar in spirit to the FIM approximation.

Using this observation, we can construct $\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{-1}$ iteratively using Kailath inversion. First, we define $\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j}$ for $i \in \{1, \ldots, N\}$ and $j \in \{1, \ldots, m\}$. Then, we define

$$\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j+1} = \left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j} + \frac{1}{N}\frac{\partial \hat{\mathbf{y}}_{j+1}^{[i]}}{\partial \mathbf{w}}\frac{\partial \hat{\mathbf{y}}_{j+1}^{[i]}}{\partial \mathbf{w}}^\mathsf{T}$$

and

$$\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i+1,1} = \left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,N} + \frac{1}{N}\frac{\partial \hat{\mathbf{y}}_1^{[i+1]}}{\partial \mathbf{w}}\frac{\partial \hat{\mathbf{y}}_1^{[i+1]}}{\partial \mathbf{w}}^{\mathsf{T}}.$$

Here, we have just unrolled the sum. Ideally, we would like to define $\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{0,0} = \mathbf{0}_{d\times d}$. However, we will see shortly that this is not feasible. Using this unrolling, we have that $\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{N,m}$ is our desired quantity of $\mathbf{H}_L^{(\hat{\mathbf{w}})}$, assuming that $\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{0,0} = \mathbf{0}_{d\times d}$.

By unrolling the sum, we can now apply Kailath inversion. In particular, Kailath inversion for matrices $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\mathbf{B} \in \mathbb{R}^{n\times r}$, $\mathbf{C} \in \mathbb{R}^{r\times r}$, and $\mathbf{D} \in \mathbb{R}^{r\times n}$, we have that

$$(\mathbf{A}+\mathbf{BCD})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{B}\left(\mathbf{C}^{-1}+\mathbf{DA}^{-1}\mathbf{B}\right)^{-1}\mathbf{DA}^{-1}.$$

The upshot of the Kailath inversion is that $(\mathbf{A}+\mathbf{BCD})^{-1}$ can be written in terms of $\mathbf{A}^{-1}$, so we do not need to invert the matrix from scratch. First, consider

$$\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j+1} = \left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j} + \frac{1}{N}\frac{\partial \hat{\mathbf{y}}_{j+1}^{[i]}}{\partial \mathbf{w}}\frac{\partial \hat{\mathbf{y}}_{j+1}^{[i]}}{\partial \mathbf{w}}^{\mathsf{T}}.$$

Let $\mathbf{A} = \left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j} \in \mathbb{R}^{d\times d}$, $\mathbf{B} = \frac{\partial \hat{\mathbf{y}}_{j+1}^{[i]}}{\partial \mathbf{w}} \in \mathbb{R}^{d\times 1}$, $\mathbf{C} = \frac{1}{N} \in \mathbb{R}^{1\times 1}$, and $\mathbf{D} = \frac{\partial \hat{\mathbf{y}}_{j+1}^{[i]}}{\partial \mathbf{w}}^{\mathsf{T}} \in \mathbb{R}^{1\times d}$. Then, by Kailath inversion, we have that

$$\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j+1}^{-1} = \left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j}^{-1} - \frac{\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j}^{-1}\frac{\partial \hat{\mathbf{y}}_{j+1}^{[i]}}{\partial \mathbf{w}}\frac{\partial \hat{\mathbf{y}}_{j+1}^{[i]}}{\partial \mathbf{w}}^{\mathsf{T}}\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j}^{-1}}{N + \frac{\partial \hat{\mathbf{y}}_{j+1}^{[i]}}{\partial \mathbf{w}}^{\mathsf{T}}\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j}^{-1}\frac{\partial \hat{\mathbf{y}}_{j+1}^{[i]}}{\partial \mathbf{w}}}.$$

Similarly, we now consider

$$\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i+1,1} = \left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,N} + \frac{1}{N}\frac{\partial \hat{\mathbf{y}}_1^{[i+1]}}{\partial \mathbf{w}}\frac{\partial \hat{\mathbf{y}}_1^{[i+1]}}{\partial \mathbf{w}}^{\mathsf{T}}.$$

Let $\mathbf{A} = \left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,N} \in \mathbb{R}^{d\times d}$, $\mathbf{B} = \frac{\partial \hat{\mathbf{y}}_1^{[i+1]}}{\partial \mathbf{w}} \in \mathbb{R}^{d\times 1}$, $\mathbf{C} = \frac{1}{N} \in \mathbb{R}^{1\times 1}$, and $\mathbf{D} = \frac{\partial \hat{\mathbf{y}}_1^{[i+1]}}{\partial \mathbf{w}}^{\mathsf{T}} \in \mathbb{R}^{1\times d}$. Then, by Kailath inversion, we have that

$$\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i+1,1}^{-1} = \left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,N}^{-1} - \frac{\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,N}^{-1}\frac{\partial \hat{\mathbf{y}}_1^{[i+1]}}{\partial \mathbf{w}}\frac{\partial \hat{\mathbf{y}}_1^{[i+1]}}{\partial \mathbf{w}}^{\mathsf{T}}\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,N}^{-1}}{N + \frac{\partial \hat{\mathbf{y}}_1^{[i+1]}}{\partial \mathbf{w}}^{\mathsf{T}}\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,N}^{-1}\frac{\partial \hat{\mathbf{y}}_1^{[i+1]}}{\partial \mathbf{w}}}.$$

Now, we return to the issue of defining $\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{0,0} = \mathbf{0}_{d\times d}$. The problem is that this matrix

is not invertible, so the first iteration of Kailath inversion will fail. Instead, we choose to let $\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{0,0} = \alpha\mathbf{I}$ for some small constant α. Then, we are essentially finding the inverse of $\mathbf{H}_L^{(\hat{\mathbf{w}})} + \alpha\mathbf{I}$. Thus, we are now minimizing

$$\frac{1}{2}\Delta\mathbf{w}^\mathsf{T}\left(\mathbf{H}_L^{(\hat{\mathbf{w}})} + \alpha\mathbf{I}\right)\Delta\mathbf{w} = \frac{1}{2}\Delta\mathbf{w}^\mathsf{T}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w} + \frac{1}{2}\alpha\Delta\mathbf{w}^\mathsf{T}\Delta\mathbf{w}.$$

Thus, we have essentially added a regularization term that penalizes $\Delta\mathbf{w}$ for being too large in magnitude. However, the effects are negligible given that α is quite small.

The upshot of the work we did above is that the Hessian can be computed iteratively and with a single pass over the data points. Thus, we can consider each data point separately without needing to consider their interactions. The downside is that we still need to store the full $\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)_{i,j}^{-1}$ in memory. This is difficult for models with billions of parameters. We will discuss solutions for this problem in Chapter 5.

6.3.3 Regularization Methods

We can also change the objective function when training our model to induce sparsity. These give rise to "regularization" methods, which add some regularization term to the objective function. In particular, the new objective function $L'(\mathbf{w})$ will be defined as the sum of the original objective function $L(\mathbf{w})$ and a regularization term $R(\mathbf{w})$. Therefore, for these regularization methods, it suffices to define $R : \mathbb{R}^d \to \mathbb{R}^d$. Then, the model can be trained using this new objective function. While regularization terms do not always guarantee $p\%$ sparsity, they can help push a model towards have sparse entries. Regularization terms are typically problematic since they are not differentiable. We will see a few examples of that.

The most natural type of regularization is L_p regularization, where we take the L_p norm of the weights. More concretely, we define $R(\mathbf{w}) = \lambda \|\mathbf{w}\|_p$ for some $\lambda \in \mathbb{R}_{>0}$. The most common variants of L_p regularization are for $p = 0, 1, 2$, since the regularization terms are most interpretable.

The L_0 norm is equal to the number of non-zeros entries. Mathematically, we can define it by

$$\|\mathbf{w}\|_0 = \sum_{i=1}^{d} \mathbb{I}(\mathbf{w}_i \neq 0).$$

However, the most immediate drawback of L_0 regularization is the fact that the regularization term is not differentiable everywhere. Therefore, we cannot use our standard gradient descent algorithms to solve for an optimum. Furthermore, [GJY11] showed that finding the optimal for such a regularization term is NP-hard. This is similar to the binary optimization

problem presented in AdaRound from Chapter 5, which has also been proven to be NP-hard. Similar to AdaRound, [LWK18] presents an approximation for the optimum by turning the regularization term into a differentiable function. We leave the details beyond the scope of this thesis. We will see parallels between L_0 regularization and movement-based pruning methods in Chapter 9.

L_1 regularization, often referred to as Least Absolute Shrinkage and Selection Operator (LASSO), uses the sum of the absolute values of the weights as the regularization term. Therefore, weights are pushed towards 0 to minimize this term. Concretely, we have that

$$\|\mathbf{w}\|_1 = \sum_{i=1}^{d} |\mathbf{w}_i| .$$

Again, this regularization term is not differentiable everywhere, and requires special optimizations methods like cyclic coordinate descent to find optimums.

L_2 regularization, often referred to as ridge regularization, uses the sum of the squares of the weights. We have that

$$\|\mathbf{w}\|_2 = \sum_{i=1}^{d} (\mathbf{w}_i)^2 .$$

Unlike the L_0 and L_1 regularization, this regularization term is indeed differentiable. However, L_2 regularization tends to push weights close to 0 but never exactly to 0. This is because the magnitude is squared, which already makes the penalty for weights close to 0 quite small.

6.4 Hardware Considerations

Like in Chapter 5, we now turn our attention to hardware considerations for sparse inference. Again, we focus mainly on weight-only sparsification. The upshot of sparsity is that we need not store all the bits for the elements that are zero. Thus, we will discuss some popular compression formats to remove unnecessary information. Then, we will discuss the support needed for fast sparse computation.

6.4.1 Compression Formats

Bitmask Encoding

A common compression strategy is bitmask encoding, where a row of $\mathbf{a}_i$ of a vector $\mathbf{a} \in \mathbb{R}^n$ is separated into a value vector $\mathbf{v} \in \mathbb{R}^{\text{nnz}(\mathbf{a}_i)}$ and a mask vector $\mathbf{m} \in \{0,1\}^n$. $\text{nnz}(\mathbf{a})$ is the number of non-zero elements in $\mathbf{a}$. $\mathbf{v}$ will only contain the non-zero values of $\mathbf{a}$. $\mathbf{m}$ will

contain a 0 at position j if $\mathbf{a}_j = 0$ and a 1 if $\mathbf{a}_j \neq 0$. $\mathbf{a}$ can be retrieved from $\mathbf{v}$ and $\mathbf{m}$ in the following fashion. Starting from the lowest index, we can iterate over $\mathbf{m}$. If $\mathbf{m}_i = 0$, then we can set $\mathbf{a}_i = 0$. Else, we can pop the top element of $\mathbf{v}$ and set $\mathbf{a}_i$ equal to that.

To extend this to a full matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, we can simply store a mask matrix $\mathbf{M} \in \{0, 1\}^{m \times n}$. Then, we can use a flattened value vector

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_m \end{bmatrix} \in \mathbb{R}^{\text{nnz}(\mathbf{A})},$$

where $\mathbf{v}_i$ is the value vector for the row $\mathbf{a}_i$.

Compressed Sparse Row (CSR)/Compressed Sparse Column (CSC)

Another common compression strategy is compressed sparse row or compressed sparse column. The two strategies are analogous, except CSR considers the non-zero values by row and CSC considers the non-zero values by column. Again, suppose we have a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$. Then, we will decompose the matrix into three vectors. First, we have that $\mathbf{v} \in \mathbb{R}^{\text{nnz}(\mathbf{A})}$, which again is just the vector of non-zero values. Then, we have the index vector $\mathbf{i} \in \mathbb{R}^{\text{nnz}(\mathbf{A})}$, which stores the row (for CSR) or column (for CSC) index. Finally, we have the index vector $\mathbf{d} \in \mathbb{R}^{\text{nnz}(\mathbf{A})+1}$. For row (for CSR) or column (for CSC) i, we have that the row or column contains the values from $\mathbf{v}_{\mathbf{d}_i}$ to $\mathbf{v}_{\mathbf{d}_{i+1}-1}$ inclusive.

Again, we claim that we can reconstruct $\mathbf{A}$ from $\mathbf{v}$, $\mathbf{i}$, and $\mathbf{d}$. First, we initialize all elements to 0. Then, for row (for CSR) or column (for CSC) i of $\mathbf{A}$, we can first find $\mathbf{d}_i$ and $\mathbf{d}_{i+1}$. Then, for the values from $\mathbf{v}_{\mathbf{d}_i}$ to $\mathbf{v}_{\mathbf{d}_{i+1}-1}$, we can place them at index $\mathbf{i}_{\mathbf{d}_i}$ to $\mathbf{i}_{\mathbf{d}_{i+1}-1}$. CSC/CSR is a more compressed version of bitmask encoding. In particular, in bitmask encoding, we still need to store a mask matrix that has size $m \times n$. However, in CSR/CSC, we have that all vectors are length $\text{nnz}(\mathbf{A})$. Therefore, in high sparsity domains, CSR/CSC is preferred. However, typically, neural network weights have moderate amounts of density, so bitmask encoding is sufficient, as we will discuss in Chapter 9.

6.4.2 Computation

Once the weights are loaded in, we will typically decode the weights to their full representation and perform computation per usual. However, depending on the level of sparsification, different approaches to matrix multiplication arithmetic units are desirable. For example, for dense matrices, the typical inner product kernels are preferred. However, for high sparsity

matrices, inner product kernels are not actually preferred. We will not discuss the full details of this, but we refer the interested reader to [YES24].

7 Reliability

In this section, we discuss reliability as it pertains to computer architecture. In particular, we are interested in the effects of errors on the performance of computing systems. We will try to understand reliability from the transistor level to the system level to the software level. Then, we will apply these insights to the realm of machine learning, which will allow us to later connect reliability with other fields of research like adversarial machine learning. Throughout this entire chapter, we take great inspiration from [Mah20], which lays out the foundations for reliability applied to machine learning.

7.1 Motivation

As modern-day computing systems grow in size, "resilience" has become more and more critical. In particular, we will mainly focus on transient hardware errors that can have broad-scale implications on the performance of the system. We will call these soft errors. Our goal is to try to detect these soft errors as soon as possible and raise some exception to the user-level program. However, these soft errors often go undetected. We will refer to undetected soft errors as silent errors. Silent errors can either not affect the final output of the program, in which case we call them masked. However, in other cases, these soft errors may change the program output, leading to silent data corruptions (SDCs).

Note that these soft errors can occur for a variety of reasons. Below, we offer a few examples. This exposition takes inspiration from [CAG+14]. We refer the interested reader to [CAG+14] for a more thorough treatment of current perspectives on reliability.

A common motivating example used in the field of reliability is the potential for a high-energy particle strike. Cosmic rays—particles that travel through space—and alpha particles—particles emitted during radioactive decay—are common naturally-occurring high-energy particles. When these particles come into contact with a transistor, they can displace electrons and potentially change the state of the transistor. If the particle strikes a memory cell, it may "flip" the value held within the memory cell. We refer to this as a bitflip. Another example of a soft error is a voltage droop. Voltage droops occur when the power supply voltage experiences a temporary dip. If the voltage falls below a critical threshold, transistors may not receive enough energy to switch properly, leading to transient logic errors. Finally, we can imagine that the longer a system has been used the more prone it is to sudden hardware failures. This is known as a wear out, since the electrical components begin to "wear out" after prolonged use.

Although the probability of any such failure is quite low, given that systems have rapidly increased in size, the probability that at least one component experiences a failure within a time frame has increased. We will detail these ideas formally in the next section. Recently, we have seen many advances in high performance computing (HPC) that has made supercomputers that are capable of exascale (10^{18} FLOPSs/s) computing, such as Frontier [AZL+23] from Oak Ridge Leadership Computing Facility (OLCF) and El Capitan from Lawrence Livermore National Laboratory. These supercomputers have millions of CPU cores and run billions of threads simultaneously.

The first step is often identifying parts of the stack that are most prone to the effects of these transient hardware errors. In essence, we want to determine the "reliability" of each of the components. However, once that has been determined, the next step is "hardening" the component so that they are "resilient" to these types of errors. Currently, the most common technique for dealing with these failures is a simple checkpoint-and-restart procedure, where the system's state logs regular checkpoints. Then, if a system failure occurs and is detected, the system restarts at the latest stable checkpoint. While simple, this approach is quite costly, since it needs to log all system details at each checkpoint. Furthermore, for exascale systems, it is predicted that the time it takes to checkpoint is longer than the mean time to failure (MTTF). The MTTF is defined as the average time between failures. Thus, it will be hard for systems to even make any forward progress. Thus, we need to design cheap and efficient "hardening" approaches that be implemented on these exascale systems.

Typically, these hardware errors go undetected. We will call these silent errors, since they

do not have a noticeable effect on program execution. Software can usually be written to hide a lot of these hardware errors by returning errors for anomalous behavior. However, software introduces its own problems. First, this requires some coordination between software and hardware vendors on the responsibility for detecting anomalous behavior. Second, software can be unreliable itself, often containing buggy behavior. Finally, software is typically built hierarchically, where an application uses underlying libraries. Thus, even if the application itself is fault-resilient, the underlying libraries may not be. Thus, while promising, using software to combat hardware errors does not always solve the problem.

We can also use hardware itself to correct for these errors. For example, error-correcting codes (ECC) can be used to store data in a redundant and efficient fashion. However, the reliability of hardware components has stagnated over the past years. This is mainly true since fabrication facilities use a standard fabrication process for all of their components. Thus, there is no preference for the design of supercomputers, which need more resilient hardware given their size.

Therefore, many have turned to algorithmic insights to "harden" these computing systems. In particular, they want to design algorithms that are insensitive to hardware errors and have the ability to recover from them. This area is very application-dependent and different techniques can be used in different settings. Thus, this motivates the use of reliability-guided analysis in machine learning models. In particular, as machine learning models grow, their execution will be susceptible to similar potential errors. Therefore, understanding the "reliability" and "resilience" of these models can guide more model design and selection. In this thesis, we primarily focus on the application of reliability principles to machine learning models.

7.2 Applications to Machine Learning

From our exposition in the previous section, we have seen that the field of reliability is turning to domain-specific algorithmic methods to combat uncertainty of transient hardware errors. Thus, when constructing a machine learning model, we would like to choose a model that is ideally resilient to these hardware errors. This motivates the area of applying reliability to machine learning. When assessing a model selection, we can try to model the change in model output of a given model under a set of "error injections". In our work, we will focus on single bitflips, as this is the most natural and easiest to model. There are two natural ways of modeling the change in model output under these perturbations. First, in vision settings, we can count the number of mismatches between the final classification output (most likely class) of the unperturbed model and the perturbed model. However, this analysis is rather

limited, as not all tasks are classification tasks. Furthermore, perturbations can change the classification probability without changing the classification class as outlined in Chapter 3. Therefore, a stronger metric is ΔLoss or the absolute change in loss between the two models. We will call the unperturbed the "golden" model.

Ideally, we are interested in the number of mismatches, since we only typically care about the classification output and not the probabilities. This is the true "corruption" that we want to avoid. However, ΔLoss can give us a more continuous metric that allows us to compare the vulnerability (inverse of reliability) of models in ways that mismatch frequency cannot. Thus, we will primarily focus on modeling the ΔLoss under our bitflip perturbation model.

7.3 Mathematical Formulation

Mathematically, we can model the problem as follows. To the best of our knowledge, this is an original formulation. First, we will make the simplifying assumption that all transient hardware errors can be treated equivalently. This is a rather large assumption, since it assumes that they occur at the same frequency and have the same effects. However, this allows us to develop a cleaner statistical model. Furthermore, we will assume that these hardware transient errors only apply to the weights, similar to the focus of Chapters 5 and 6. While this is a big assumption, it fits nicely with the other mathematical theory formed in previous chapters. Suppose that we have trained weights $\hat{\mathbf{w}} \in \mathbb{R}^d$. As mentioned in Chapter 5, we have that a datatype aligns with a function $Q : \mathbb{R} \to \mathcal{B}$, where $\mathcal{B} = \{0,1\}^b$. Thus, consider let $\tilde{\mathbf{w}} \in \{0,1\}^{d \times b}$ be the result of Q applied to $\hat{\mathbf{w}}$ element-wise. Then, we will suppose that $\hat{\mathbf{w}}$ is stored somewhere in memory.

We will model the transient hardware error rate for a bit (i,j) as a Poisson process with rate parameter $\lambda_{i,j}$. In our model, we have a total of db bits. Recall a sequence of arrivals is a Poisson process if the following two conditions hold. First, the number of transient hardware errors in a time of length t is distributed as $\text{Pois}\left(\lambda_{i,j} t\right)$. This is a reasonable assumption, since the Poisson distribution is used to model events with extremely low probability. Thus, $\lambda_{i,j}$ is an indication of the error rate for each bit. We model this on the bit level, since different bits of $\tilde{\mathbf{w}}$ may be stored in different locations. Therefore, they may be more or less susceptible to transient hardware errors, which will change the rate parameter $\lambda_{i,j}$. The second condition is that the number of transient hardware errors in disjoint time intervals are independent. This assumption is fairly reasonable, since it assumes that these events are not likely to be clustered.

Then, we will assume that the transient hardware errors for each bit are independent of

each other. This is also a simplifying assumption, since a particle strike in one area will likely affect the neighboring bits as well. However, it allows us to combine the Poisson processes nicely. By the superposition property of Poisson processes (Chapter 13 of [BH19]), we can model the total number of transient hardware in the weights as a Poisson process with rate parameter $\sum_{i=1}^{d}\sum_{j=1}^{b}\lambda_{i,j}$. Furthermore, we can model the error per bit in the following fashion. For each occurrence of a transient hardware error in any of the weights, we can assign it to bit (i,j) with probability $\frac{\lambda_{i,j}}{\sum_{y=1}^{d}\sum_{z=1}^{b}\lambda_{(y,z)}}$. We will assume that when a transient hardware error occurs, it will flip the state of the bit.

For evaluating the "reliability" of a model, we are interested in the average loss under a time period of t. t will be the amount of time it takes for an inference pass, and it should grow as the model becomes deeper. Let L_t be the random variable for the change in loss for a model that runs for a time t. Then, by the Law of Iterated Expectations (Adam's Law), we have that

$$\mathbb{E}\left[L_t\right] = \mathbb{E}\left[\mathbb{E}\left[L_t \,\middle|\, \left\{\left\{N_{i,j}(t)\right\}_{j=1}^{b}\right\}_{i=1}^{d}\right]\right],$$

where $N_{i,j}(t)$ is the number of transient hardware errors for bit (i,j) over time t. We will give empirical methods for estimating this quantity in Chapter 7. Note that the above formulation is evaluating the reliability of a model under some data type. However, we may also want to calculate the reliability of individual components of a model, such as individual activations. We will explore this idea in the subsequent section.

To get an unbiased estimate of $\mathbb{E}\left[L_t\right]$, we can do the following. Using the superposition property, we can sample $N \sim \text{Pois}\left(\sum_{i=1}^{d}\sum_{j=1}^{b}\lambda_{i,j}t\right)$. Then, for each of the N events, we select a bit (i,j) with probability $\frac{\lambda_{i,j}}{\sum_{y=1}^{d}\sum_{z=1}^{b}\lambda_{(y,z)}}$. Then, we flip the bit. After N bits have been flipped, then we compute the change in loss. We can repeat this process for M trials and take the sample mean. This gives us an unbiased estimate by the Law of Iterated Expectations. We will call this model-level resilience with model-wide timing.

In the above models, we have assumed that all bitflips have the same effect during the interval of length time t. However, this may not true. In fact, for neural networks, the computation is done in a layer-wise manner. Therefore, a bitflip in a layer that has already been passed likely has no effect. Thus, one way to make our model more precise is to consider time intervals for each bit individually. In particular, suppose we have a model that runs and uses bit (i,j) for time $\mathbf{T}_{i,j}$. Then, let $L_{\mathbf{T}}$ be the random variable for the change in loss for that model. Then, we are interested in the quantity

$$\mathbb{E}\left[L_{\mathbf{T}}\right] = \mathbb{E}\left[\mathbb{E}\left[L_{\mathbf{T}} \,\middle|\, \left\{\left\{N_{i,j}(\mathbf{T}_{i,j})\right\}_{j=1}^{b}\right\}_{i=1}^{d}\right]\right],$$

where again $N_{i,j}(\mathbf{T}_{i,j})$ is the number of transient hardware errors for bit (i,j) over time $\mathbf{T}_{i,j}$. For weights $\hat{\mathbf{w}}_i$ and $\hat{\mathbf{w}}_j$ that are in the same layer, we have that the time $\mathbf{T}_{i,k} = \mathbf{T}_{j,\ell}$ for all $k, \ell \in \{1, \ldots, b\}$, since the time that the bits are relevant for is the same.

To get an empirical unbiased estimate for $\mathbb{E}[L_{\mathbf{T}}]$, we can do the following. We will assume that the time is the same for all bits in a layer. We will only consider a bit for time $\mathbf{T}_{i,j}$. We can again use the superposition property among the "live" bits. Thus, let $\lambda^{(\ell)} = \sum_{i \in \mathbf{w}^{(\ell)}} \sum_{j=1}^{b} \lambda_{i,j}$, where $\mathbf{w}^{(\ell)}$ is the set of indices of weights in layer ℓ. Then, for each layer ℓ, we sample $N^{(\ell)} \sim \text{Pois}\left(\lambda^{(\ell)} \mathbf{T}^{(\ell)}\right)$, where $\mathbf{T}^{(\ell)}$ is the time spent on layer ℓ. Then, for each $N^{(\ell)}$ errors, we can sample the bits in that layer and select a bit (i,j) in the layer with probability $\frac{\lambda_{i,j}}{\lambda^{(\ell)}}$. Then, we can do this for each layer ℓ. We can only do this under the assumption that the layers are executed in disjoint time intervals. Then, we can use the Poisson process assumption that the number of hardware errors in disjoint time intervals are independent. Then, we can compute the total change in loss. Again, we can do this for M trials and take the sample mean. We will call this model-level resilience with layer-wise timing. We will return to this idea in Chapter 7.

Note that while we have mainly restricted our analysis to weights, this analysis extends in the same manner to activations as well. In particular, we can just consider the bits for both activations and weights in our analysis. Furthermore, it is important to note that all of this analysis is done under the assumption of a data type function Q. In particular, this analysis depends on the selection of Q, which motivates the implementation of tools to simulate different number formats. We will explore this idea more deeply in Chapter 8. To the best of our knowledge, this mathematical formulation is an original perspective on modeling transient hardware errors. It gives us a mathematically sound way of modelling hardware errors with respect to machine learning models.

7.4 Granularity

Reliability and redundancy go hand-in-hand. Once we have identified the reliability of a model, we would like to decide the necessary redundancy to ensure the limitation of impact from soft errors. This ranges from the extremes of no redundancy to full model redundancy. Redundancy is important in making large-scale systems more reliable, but they come at a steep price. For example, double-modular redundancy (DMR) and triple-modular redundancy (TMR) are frequently used in safety-critical settings. This refers to the process where the entire module is duplicated and run for each input. Then, the results are compared to detect areas. However, using ideas from the design considerations discussed in Chapter 2, redundancy can multiply common cost metrics like area and power. Thus, we need to tradeoff

the value added by redundancy and its associated costs. Similar to quantization and sparsity, the key question is the granularity to operate at. Here, we use the term granularity to refer to the level that the hardening techniques is applied at. Below, we discuss two granularities introduced in [Mah20]: feature-level resilience (FLR) and inference-level resilience (ILR). Finally, they can be combined to yield feature and inference-level resilience (FILR).

7.4.1 Feature-Level Resilience (FLR)

As mentioned above, we want to measure the average change in loss given the number of hardware errors in each bit. Naturally, some weights (and their corresponding bits) may have a larger effect on loss than others. Thus, [MSHF+21] propose feature-level resilience (FLR) as a technique to harden vulnerable feature maps. Recall from Chapter 3 that feature maps correspond to the output of a single filter. Thus, for every feature map $f_i \in \mathbb{R}^{h \times w}$, we will assign a corresponding vulnerability V_{fmap_i} defined as

$$V_{\text{orig}_i} = V_{\text{fmap}_i} \cdot P_{\text{prop}_i}.$$

V_{orig_i} is the "origination vulnerability" for feature map f_i. It captures the likelihood that a hardware error can originate in feature map f_i. Using the terminology from the previous section, it is roughly analogous to $\lambda_{i,j}\mathbf{T}_{i,j}$. In particular, [MSHF+21] notes that the longer the feature map is needed for computation, the more likely it is for a transient hardware error to occur. We have added another $\lambda_{i,j}$ term to capture that inherently some feature maps may be more vulnerable than others, depending on where they are stored. [MSHF+21] uses the number of MACs that f_i is involved in as a proxy for V_{orig_i}.

P_{prop_i} is the "propagation probability", which represents the likelihood that the silent error transforms into a data corruption. This is rather hard to quantify. In particular, a single bitflip may not have much of an impact, but together some set of bitflips may have large impact. However, one simplifying assumption that we can make is that change in loss is additive across feature maps. In particular, for each feature map, we will solely focus on the change in loss when error injections occur in that feature map. Therefore, we can perform M error injections in f_i over N trials to approximate the expected ΔLoss.

Together, we have some metric for the "vulnerability" of each feature map. Then, we can sort these in descending order and choose to duplicate the top k feature maps. Then, during computation, for each of these feature maps, the duplicated outputs are checked for any differences. If a difference is identified, then an exception is raised. Note that one drawback of FLR is that it considers feature maps in isolation. In particular, two feature maps may not be vulnerable individually, but if perturbed simultaneously, they may have

a larger collective effect on the change in loss. For example, one might imagine that two feature maps in the same layer will contain different information about the previous input, but if both are altered, then the information flow in the network is disrupted.

7.4.2 Inference-Level Resilience (ILR)

FLR takes a "static" approach to the hardening problem. In particular, it determines the most vulnerable feature maps offline, and it duplicates those that it deems to be most vulnerable. However, once that decision is made, duplication occurs for all inputs. Therefore, it is "static" in the sense that the hardening strategy does not change depending on the input. ILR propose a more "dynamic" hardening scheme, where recomputation is only necessary for inputs that are susceptible to silent data corruptions. ILR takes a step back and focuses on the model output, rather than focusing on individual feature maps like FLR.

The motivation for ILR is as follows. Recall that we are interested in SDCs, which occur when there is a mismatch between the classification of the golden and perturbed models. If a model is very confident in its prediction, then silent errors may not appreciably change its output. In particular, the silent error may perturb its confidence slightly, but it us unlikely to change its final output, since the model has a good understanding of the image. Thus, ILR only reaches for recomputation when the output of the model is not "confident".

We will use two potential metrics to quantify confidence, assuming a classification task setup. The first is the highest prediction probability for any given class, which [MSHF+21] refers to as Top1-Conf. As discussed in Chapter 3, this corresponds to an estimate of the probability that the given image is in the class. If the model predicts that it is extremely likely the image is in a specific class, a small perturbation is unlikely to change that it remains the most likely class. As stated previously, we only care about the highest output probability as it compares to the other probabilities.

The second metric quantifies this "relative" probability ranking more exactly. In particular, it takes the difference between the top two output probabilities. [MSHF+21] calls this Top2Diff. If this difference is high, then perturbations are unlikely to change the relative ordering, which makes recomputation unnecessary. On the other hand, if the top class has a slight advantage over the other, then the perturbation may change the relative rankings.

For both of these criterion, we can choose $\text{Top1-Conf}_{\text{threshold}}$ and $\text{Top2Diff}_{\text{threshold}}$ as a threshold. Then, for any image where the criterion falls below the threshold, recomputation is triggered and the output of both passes are compared. In this way, we only trigger recomputation when we question the validity of the output based on these metrics.

7.4.3 Feature and Inference-Level Resilience (FILR)

Given the presentation of FLR and ILR, we can see that the two methods are naturally disjoint from each other. In particular, given that FLR is "static" in flavor and ILR is "dynamic", we can apply both simultaneously to achieve even better results. In particular, we can first set thresholds $\text{Top1-Conf}_{\text{threshold}}$ and $\text{Top2Diff}_{\text{threshold}}$ for ILR. Then, for our offline analysis for FLR, we can focus on images where their output does not meet the ILR criterion. Then, we can rank the feature maps vulnerability and choose the feature maps to harden. In this way, we protect some inference at the ILR-level. Then, for those that are not protected by ILR, we use FLR to mitigate the effects of data corruption. [MSHF+21] shows that this two-pronged approach can achieve nearly full error coverage in that almost all potential corruptions are detected.

Part III

Our Contributions

8

GoldenEye [MTA+22]

In this section, we present our joint work on GoldenEye, done in collaboration with Abuldrahman Mahmoud, Thierry Tambe, and Tarek Aloui. For the full details of GoldenEye, please refer to [MTA+22]. In this chapter, we give an overview of GoldenEye, detail our contributions, and extend the analysis of GoldenEye using the mathematical framework presented in Chapter 7.

8.1 Problem Statement

Ideally, we want to construct a library capable of simulating fault injections into a given machine learning model. Fault injection simulators are defined by the following features.

1. First, we want to specify the type of fault injection model that we want to use. In Chapter 7, we mainly focused on a bitflip model. However, other models do exist. For example, one could imagine that instead of doing a single bitflip, we replace the current value with some random value sampled uniformly from a given range.

2. Then, we need to specify where the fault injections will occur. We can choose to inject faults in either the activations or weights. However, we might also want to choose the layers of the network that the fault injection occurs at.

3. Regardless of the location of fault injection, we also need to support the representation of various data types. Recall that a fault injection is only well-defined under some data type, since the fault occurs at the bit level. Thus, we need to be able to apply error injection to a broad range of data types. In addition, if the computational data type differs from the storage data type, then the simulator needs to support some notion of quantization and dequantization. We should distinguish emulation and quantization. Emulation is a subset of quantization, where no scaling factor is used. In particular, after quantization and dequantization, a real number is mapped to the nearest point on the quantization grid, which is simply the expressable numbers of the number format. Emulation assumes that Q is a rounding-to-nearest scheme, and D is simply the mapping from a bitstring to a real value. However, quantization and dequantization are more general than that. In particular, we can choose D to have some scaling factor, so that it is just not directly mapped from a bitstring to its real value.

4. Finally, we want some metric to measure the "effect" of the fault injections. As discussed in Chapter 7, mismatch frequency and ΔLoss are two natural metrics.

Before we discuss specific implementations, we will make a few comments about the necessary components for the above problem statement. First, we should recognize that fault injection, emulation, and quantization of the weights is typically much easier than that for the activations. This is because the activations are input-dependent, while weights are static in nature. For the weights, fault injections can be performed offline and a model with perturbed weights can simply be loaded in prior to any computation. Similarly, emulation and quantization can be applied offline prior to any computation. Thus, these simulators typically focus on dealing with the dynamic activations. Second, our main goal for a simulator is generalizability. In particular, as we have discussed in Chapters 5 and 7, there are many evolving data types, fault injection models, and resilience metrics. Thus, portability is the name of the game, as we hope to use the simulator regardless of these user-level choices. Now, we discuss some of the precursors to GoldenEye that motivate its creation.

8.2 PyTorchFi [MAN+20]

8.2.1 Motivation

Before giving an overview of GoldenEye, we will first introduce PyTorchFi [MAN+20], which serves as a precursor to GoldenEye. Prior to PyTorchFi's release, Ares [RGP+18] and TensorFI [CNF+20] were two existing fault injection simulators. Ares was built on top of Keras, while TensorFI was built directly on top of TensorFlow. Keras is an easy to use

API that sits atop backends like Theano [TARA+16] and TensorFlow [ABC+16]. Since the release of TensorFlow v2.0, Keras has been fully integrated into TensorFlow, allowing for programmers to use both the high-level API, as well as low-level implementation details from TensorFlow.

While both of these tools allowed researchers to make great progress, each had their drawbacks, which motivated the rise of PyTorchFi. First, since Keras is a high-level API, it does not natively support hooks. Hooks are callbacks that are called before or after computation of a node in a computational graph. As discussed in Chapter 1, computational graphs allow libraries to easily model dataflow. These callbacks can inspect the data and modify or act on the information in some way. Since hooks are not natively supported in Keras, Ares overrode the `call()` function of internal classes to allow for dynamic perturbations on the activations. However, this becomes hard to maintain as the Keras API is updated. Instead, PyTorchFi takes advantage of PyTorch's native `hook` support, which mitigates the concern of maintainability. PyTorch's `hook` allows users to insert a callback, `func`, either prior to the computation with `register_forward_pre_hook(func)` or after the computation with `register_forward_hook(func)`.

When using `register_forward_pre_hook(func)`, `func` must accept `module` and `input`. `module` is the layer that the hook is registered to and `input` is the input to the layer. If `func` returns anything other than `None`, then that will be used as the input to the module. When using `register_forward_hook(func)`, `func` must accept three arguments: `module`, `input`, and `output`. `module` and `input` are the same for `register_forward_pre_hook(func)`. `output` is the output of the layer, and it is an argument since `register_forward_hook(func)` occurs after the completion of the layer.

Using this built-in support, we can add the functionality of error injections by injecting a forward hook for the error injection. [MAN+20] gives empirical data that also supports the claim that little to no overhead is added.

PyTorchFi's implementation atop PyTorch is also more desirable than implementations atop TensorFlow. In particular, PyTorch uses a dynamic computational graph, which is constructed on the fly, as opposed to TensorFlow, which creates the computational graph prior to execution. An alternative to the Ares workaround detailed above would be to change the computational graph by adding nodes for error injections. However, due to TensorFlow's static nature, prior to v2.0, such node insertions require generating a copy of the initial graph. Then, at inference time, additional logic needs to be added to choose between the original and the altered graph. PyTorch's dynamic nature and support for hooks makes it a more natural choice and serves as the main motivation for PyTorchFi.

8.2.2 Code Structure

Now, we give a detailed overview of the code structure of PyTorchFi. The code structure will motivate the structure that will later be presented in GoldenEye. The goal of PyTorchFi is to be as flexible as possible with respect to different error injection models. This addresses the first bullet point of our problem statement. In particular, the goal is to provide the user flexibility to define their own error injection models.

8.2.3 `core.py`

In `core.py`, PyTorchFi defines a base class `fault_injection`. `fault_injection` constructs an object from a user-provided `model`. It also accepts `batch_size` for the current batch size of the inference routine, `input_shape` for the input shape of the image, and `layer_types` for the type of layers to perform fault injections on. When the object is constructed, the model architecture is traversed to find all layers that satisfy the `layer_types` and find the output size of the layer, `output_size`, and the number of dimensions in the output, `layers_dim`.

Then, `fault_injection` defines the following APIs that can be exposed to the user. First, we discuss `declare_weight_fault_injection`, which attempts to corrupt weights in the model. The function accepts lists of layer numbers, `corrupt_layer`, filter indices, `corrupt_k`, channel indices, `corrupt_c`, height indices, `corrupt_kH`, and width indices, `corrupt_kW`, of the desired weights that should be corrupted. Then, it either takes in a custom fault injection model, `custom_injection`, to apply to the initial value or a corrupted value, `corrupt_value`. This design allows users to easily pass their own custom error injection models. Then, `declare_weight_fault_injection` will create a corrupted copy of the original model with the corrupted weight replaced with the corrupted value. Note that no hooks are needed for weight corruption, since this is a static, one-time operation. It is not input-dependent, so we can just modify the model directly.

`declare_neuron_fault_injection` performs a similar operation to the weight variant, except by corrupting the neuron activation. Since this is a dynamic operation, we need to use the built-in support of hooks. Like `declare_weight_fault_injection`, the function takes in the layer indices, `corrupt_layer`, batch indices, `corrupt_batch`, and the channel and spatial indices via `corrupt_dim1`, `corrupt_dim2`, `corrupt_dim3`. It again gives the user the flexibility to define a `custom_injection` function that will be applied to the current value, or just a pre-defined corrupt value. Then, it adds a forward hook to the desired layers to corrupt the corresponding indices.

8.2.4 `neuron_error_models.py` and `weight_error_models.py`

`neuron_error_models.py` and `weight_error_models.py` provide examples of using instances of `fault_injection`. We will leave the details of these files out, since we will soon discuss GoldenEye, which uses `fault_injection` in a very similar way to these examples.

8.3 New Number Formats

PyTorchFi does not support error injections in custom number formats natively. In particular, the user cannot easily specify a number format for the weights to be stored in. While they can define their own custom injection functions, GoldenEye adds another layer of abstraction, building the machinery to test various number formats. Thus, the motivation of GoldenEye is to build on top of PyTorchFi to allow easy experimentation on various number formats. As mentioned in our problem statement, we would like to model the reliability of models under different data types. We can assume that the computation is still being done in FP32, but we would like to add emulation support for alternative number formats. Furthermore, we would like to support integer quantization, which is a commonly used quantization scheme.

Before discussing the support for these data types, we will discuss a few more data types that have become popularized by the DNN literature. These are in addition to those that were discussed in Chapter 5. The two data types presented below have a key distinguishing feature. In particular, not only do they contain element-wise data, they also have metadata shared between multiple weights in a tensor. These types of number formats are especially useful for block-wise quantization, which we discuss extensively in Chapter 10. GoldenEye is the first functional simulator to support error injections on both the data itself as well as the metadata.

8.3.1 Block Floating Point (BFP)

We have already seen the notion of block-based quantization, where elements within a "block" share the same quantization and dequantization functions. We will explore this idea more deeply in Chapter 10. This idea motivates the creation of the Block Floating Point (BFP) data type. In particular, for each "block" of elements, we force them to share the same exponent bits. However, each individual element has its own "mantissa" bits. Below, we include an example of block floating points, with 1 sign bit, 3 shared exponent bits, and 7 "mantissa" bits. These "mantissa" bits follow the floating-point standard.

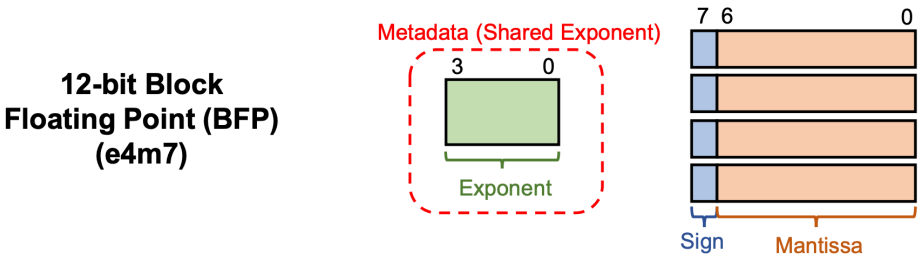

Figure 8.1: Block floating point illustration. [MTA+22]

We call these "mantissa" bits, since they need not be in the $1.M$ format that we discussed in Chapter 5. Although it is true for the above example, we could use an integer-based format for the individual data. The shared exponent represents the "shared" scaling parameter for quantization. Thus, block floating point is natural to combine with block-based quantization. Let b_m be the "mantissa" bits, and b_e be the shared exponent bits. Then, we have that the corresponding real number that is represented is $t\left(b_m\right) \times 2^{b_e}$, where t is some transformation of the mantissa bits. For a true floating point representation, we have that t uses the standard floating point convention. However, for an integer-based format, we would have that t is the identity function and $b_m \mapsto b_m$.

Given this data representation, we now discuss the standard choices for the quantization function Q for this data type. Unlike the previous data types that we have discussed, BFP, and later AFP, are unique in that the quantization grid depends on the choice of the shared exponent bits. For the basic data types that we discussed in Chapter 5, we could use a simple rounding-to-nearest scheme. However, now, we need to choose the shared exponent bits. Once we have done so, we can then use a rounding-to-nearest scheme. We will assume that regardless of the choice of transformation the mantissa bits can represent a range $[-c, c]$. Then, a common quantization function is the max-scaled function. Let $\mathbf{x} \in \mathbb{R}^d$. Then, let

$$s = \max_{i \in \{1,\ldots,d\}} \left|\mathbf{x}_i\right|.$$

Then, we will choose exponent bits $b_e = \lfloor \log_2\left(s\right) \rfloor$, which rounds down to the closest integer expressible by the exponent bits. Then, for an element $\mathbf{x}_i$, we will round $\frac{\mathbf{x}_i}{b_e}$ to the nearest point on the mantissa quantization grid. This will yield the mantissa elements b_m.

BFP suffers from the fact that the largest magnitude dictates the quantization grid. Therefore, if the largest magnitude is high, then precision tends to be lost. We will explore AFP as an alternative to this in the next section.

8.3.2 Adaptive Floating Point (AFP) [TYW+20]

Adaptive Floating Point (AFP) takes inspiration from BFP by using a block-based number format. However, instead of having a shared exponent, AFP has a shared exponent bias. In particular, each element has its own exponent bits. However, the shared exponent bias is added to each of the exponent bits. The "adaptive" shared exponent bias gives the data type more flexibility in representing a wider range of weights.

First, we provide some motivation for AFP, as presented in [TYW+20]. As seen in the figure below, compared to CNNs, transformer-based models display a much wider distribution of weights.

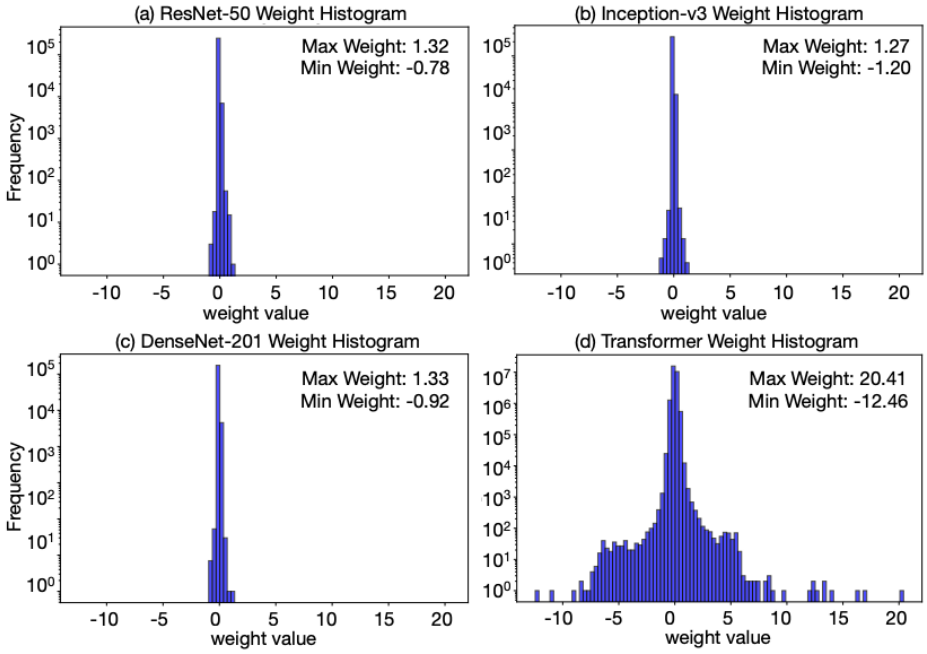

Figure 8.2: Distribution of LLM weights. [TYW+20]

This can largely be explained by the fact that CNNs typically feature batch normalization layers. As discussed in Chapter 3, these batch normalization layers function as weight normalization. However, in transformer-based models, layer normalization is used. As discussed in Chapter 4, layer normalization is input dependent. Thus, it does not create the same weight normalization effect as batch normalization. Therefore, we see a wider distribution of weights. Previous number formats were limited by the range of the exponent bits. In particular, the "exponent quantization grid" is fixed. The possible exponent values are fixed in a certain range for all values. However, we would like to alter this range depending on the magnitude of the values in the block. Furthermore, by altering the range, we can

concentrate on the dynamic range of the current block and allow for better precision.

First, we discuss the conversion of elements in AFP to the real numbers. This defines the dequantization scheme. Suppose we have an element with sign bit b_s, mantissa bits b_m, exponent bits b_e, and shared exponent bias bits b_b. AFP shares most of the conventions with IEEE754 floating-point. In particular, it uses the $1.b_m$ format for the mantissa bits. However, the difference between AFP and the IEEE754 standard is that AFP does not features denormals. Thus, it always uses the $1.b_m$ format. To represent 0, it uses $b_m = 0$ and $b_e = 0$. The corresponding real number for the above representation is

$$(-1)^{b_s} \times 1.b_m \times 2^{b_e + b_b}.$$

Below, we show a graphical representation of AFP.

Figure 8.3: Adaptive floating point illustration. [MTA+22]

Now, we again discuss quantization schemes. Again, like BFP, our goal is to find the shared exponent bias. Let $\mathbf{x} \in \mathbb{R}^d$. Then, let

$$s = \max_{i \in \{1,\ldots,d\}} |\mathbf{x}_i| .$$

Then, we will first choose e' such that $2^{e'} \leq s < 2^{e'+1}$. The goal is that for the largest magnitude s, we set all the exponent bits to 1. Thus, we let $b_b = e' - (2^e - 1)$, where e is the number of exponent bits. Once the shared exponent bias is set, then we can simply use a rounding-to-nearest quantization scheme.

Note that this quantization scheme allows for more efficient hardware design as we will explore more thoroughly in Chapter 9. We should also note that for both BFP and AFP, we view elements at a block granularity, since the shared metadata is between blocks. The block granularity can range from the vector-level to the tensor-level. A block need just be a collection of elements. We will explore this idea more deeply in the next section and in Chapter 9.

8.4 Code Structure

Now, we discuss the emulation, quantization, and dequantization support that GoldenEye adds atop PyTorchFi. Similar to our discussion of the structure of PyTorchFi, we give a complete overview of the important parts of the library. GoldenEye supports emulation of many custom-designed number formats, making it portable and flexible as new number systems are presented. Furthermore, it supports integer quantization and dequantization using a max-scaled layer-wise quantization scheme. One can refer to Chapter 10 for a more detailed explanation of this scheme.

8.4.1 `num_sys_class.py`

As mentioned above, we would like to emulate the storage of weights and activations in different number formats. Furthermore, we want to make it easy for researchers to model their own number formats. Thus, the implementation of `num_sys_class.py` attempts to make this process as seamless as possible. In `num_sys_class.py`, the `_number_sys` class is defined, which provides a general framework from which all custom number system classes can be derived from. The `_number_sys` class features four virtual methods that need to be implemented.

- `real_to_format(value)`: `real_to_format` converts the real number `value` approximated in the compute fabric, typically FP32, to a bitstring defined by the number system. Mathematically, we can view it as converting an element to $\mathbb{R}$ to an element in $\mathcal{B}$.
- `real_to_format_tensor(tensor)`: `real_to_format_tensor` takes in a tensor and does the same as `real_to_format` at the tensor level. This is useful for number formats with metadata that is shared over a tensor.
- `format_to_real(bitstring)`: `format_to_real` takes a bitstring in the number format and converts it back to the real number line, approximated by the compute fabric. Mathematically, it takes an element of $\mathcal{B}$ and maps it to $\mathcal{Q}$.
- `format_to_real_tensor(tensor)`: `format_to_real_tensor` takes in a tensor and does the same as `format_to_real`, but at the tensor level. Again, this is useful for number formats with metadata that is shared over a tensor.

Furthermore, if the number system contains metadata, then the implementation of the method `real_to_format_tensor_meta(tensor, **kwargs)` is necessary. The function quantizes `tensor`, while also performing an error injection into the metadata.

`_number_sys` implements other functions that are typically shared between all derived classes.

- `bit_flip(bitstring, bitind)`: `bit_flip` flips the `bitind` least significant bit of `bitstring`.
- `convert_numsys_flip(value, bitind, flip)`: `convert_numsys_flip` calls the `real_to_format(` method. Then, if `flip == True`, a bitflip occurs in the `bitind` least significant bit. This function is useful for converting to a number system and applying a bitflip for a single value.
- `convert_numsys_tensor(tensor, meta_inj)`: `convert_numsys_tensor` calls the method `real_to_format_tensor(tensor)` if `meta_inj == False` and the method `real_to_format_tensor` otherwise. Thus, if `meta_inj == True`, then we convert the tensor to the number system and apply error injection to the metadata. Otherwise, we just convert to the number system. This function is useful for converting to a number system and applying a bitflip to the metadata. Note that applying a bitflip to metadata only makes sense when there is a tensor.

`num_sys_class.py` contains standard formats like the IEEE754 floating-point format and fixed-point formats. Furthermore, it also contains the two datatypes discussed above: BFP and AFP.

8.4.2 `goldeneye.py`

In `goldeneye.py`, we implement the "core" class of GoldenEye called `goldeneye`. It is derived from the `fault_injection` class from PyTorchFi. Like `fault_injection`, it constructs an object from a user-provided `model`. Like `fault_injection`, it also accepts `batch_size`, `input_shape`, `layer_types`. `goldeneye` also accepts the following arguments.

- `precision`: This is the precision of the "compute" fabric for which the computation is performed in. This is typically either FP16 or FP32, since these are heavily optimized for on the GPU.
- `num_sys`: `num_sys` is expected to be a tuple of a `_number_sys` class and a number system name. This defines the number system used to represent the weights before and after computation.
- `quant`: `quant` is a Boolean that indicates whether integer quantization will occur. This is equivalent to checking if the number system name is an integer type. Integer-based

quantization is handled separately because it requires the max-scale, which is derived from `preprocess.py`. However, for other number systems, there is no scaling factor. Future work may want to consider adding a scaling factor for other number formats.

- `layer_max`: `layer_max` is a list of the maximum magnitude activation in each layer. This is useful for max-scaled layer-wise quantization, where we scale the value by the maximum magnitude activation in the layer.

- `inj_order`: `inj_order` specifies the location of the error injection. GoldenEye supports the following injection orders.

  - `inj_order == 0`: This corresponds to no error injections. Thus, the only role of GoldenEye is to emulate the provided number formats.
  - `inj_order == 1`: This corresponds to error injections in the value data.
  - `inj_order == 2`: This corresponds to error injections in the metadata for number formats that support such metadata.

The overarching goal is to define `apply_goldeneye_transformation` in `goldeneye` using these attributes. Then, we can use this function as a custom fault injection in the function `declare_neuron_fault_injection` from the `fault_injection` base class in PyTorchFi. Ideally, `apply_goldeneye_transformation` will perform quantization and dequantization or emulation, while also injecting the appropriate bitflip requested by the user.

If `inj_order == 1`, then `apply_goldeneye_transformation` will apply error injections to all the locations for that layer, designated by `self.corrupt_batch` and the set provided by `self.corrupt_dim{i}`. If the number system is integer-based, then integer quantization will be performed. In particular, for some $x \in \mathbb{R}$, we will compute $\left\lfloor \frac{x}{s} \right\rceil$ in two's complement, where s is the maximum of the layer. Then, it will perform a bitflip and scale it back. If the number system is not integer-based, then we will perform emulation. In particular, we will call `convert_numsys_format` to convert the number to the format, perform a bitflip, and convert it back to the original compute fabric type. Note that quantization and emulation are very similar. The only difference is that quantization involves the scaling parameter s.

It should be noted that GoldenEye implements a range detector. Thus, if the bitflip changes the value such that it becomes undefined or has a magnitude larger than the largest value of the layer, then the value is set to $\text{sign}(x)\, s$, where x is the output of the bitflip and s is the largest magnitude in the layer.

If `inj_order == 2`, then it just sets `meta_inj_en = True`. This will be used when converting the output tensor to its emulated version.

Regardless of the `inj_order`, we want to change the output to the quantized or emulated version. Thus, if we are quantizing, we quantize and dequantize the output inplace, so that future layers use the new values. If we are emulating, then we will call `convert_numsys_tensor(output, meta_inj = meta_inj_en)`, where `output` is the output tensor.

8.4.3 `preprocess.py`

`preprocess.py` finds the minimum and maximum activation values for each layer. These values are useful for the integer-based quantization discussed above. For `torch.nn.Conv2d` and `torch.nn.Linear` layers, we first register a `save_activations` hook that appends the activations to a global array `activations`. Then, for the data inside the iterator, `data_iter`, we do a forward pass on the input and collect the outputs in the `activations` array. Then, we find the smallest and largest element in each layer. Then, we repeat for each input and continue to update these values. We also find the highest absolute value activation of each layer by taking the maximum of the absolute values of the minimum and maximum activation in each layer. This preprocessing data is all written out to a folder in `networkRanges`.

8.4.4 `profile_model.py`

`profile_model.py` performs initial profiling of the model. It sets `inj_order == 0`, so that no error injections are performed. Then, it computes the accuracy, average loss, average max probability, and average Top2Diff as defined in Chapter 7. These results are outputted in a directory called `networkProfiles`.

8.4.5 `split_data.py`

`split_data.py` splits the dataset (either CIFAR or ImageNet) into a analysis set (AS) and a deployment set (DS). Furthermore, it only selects images that are originally correctly identified with a unique class that has the maximum probability. This is because we are generally interested in the adverse effects of bitflips, so we only want to model its effect on passes that were initially correct.

8.4.6 `injections.py`

`injections.py` performs the error-injection campaign on a given model. It takes in a number system, number of injections per layer, and the injection order, discussed above. Then, for each layer, it does the following. For every image in a batch, it chooses a position in the feature map for that layer for the error injection to occur. Then, it calls the function

`declare_neuron_fault_injection` using `apply_goldeneye_transformation`, which registers the layer hooks that will perform the transformation on the selected elements. Then, it runs the model and simply prints the injection output and the loss to a file in the `injections` folder. It repeats this process `inj_per_layer` times.

8.4.7 `postprocess.py`

`postprocess.py` calculates basic summary statistics given the data collected by running the `injections.py` script. For each layer i, it calculates the following. First, it calculates the average change in loss

$$\overline{\Delta\text{Loss}}^{(i)} = \frac{1}{N}\sum_{j=1}^{N}\Delta\text{Loss}_j^{(i)},$$

where $\Delta\text{Loss}_j^{(i)}$ is the change in loss for the jth trial in layer i and $N =$ `inj_per_layer`. Then, it calculates the sample unbiased variance

$$\sigma^{(i)} = \frac{1}{N-1}\sum_{j=1}^{N}\left(\Delta\text{Loss}_j^{(i)} - \overline{\Delta\text{Loss}}^{(i)}\right)^2.$$

Note that these empirical estimates are done over all losses that are not `NaN`. Then, it calculates the number of mismatches, p.

`postprocess.py` also reports the following confidence intervals. First, it reports

$$F^{-1}\left(0.995\right)\frac{\sigma^{(i)}}{\sqrt{N}},$$

where F^{-1} is the inverse $\mathcal{N}\left(0,1\right)$ CDF. Then, we have that

$$\left[\overline{\Delta\text{Loss}}^{(i)} - F^{-1}\left(0.995\right)\frac{\sigma^{(i)}}{\sqrt{N}}, \overline{\Delta\text{Loss}}^{(i)} + F^{-1}\left(0.995\right)\frac{\sigma^{(i)}}{\sqrt{N}}\right],$$

gives an approximate 99% confidence interval for the average change in loss from an injection. This uses the Central Limit Theorem (CLT) assumption that there are sufficiently many N such that the average roughly follows a normal distribution. Similarly, it reports

$$F^{-1}\left(0.995\right)\sqrt{\frac{p(1-p)}{n}},$$

which can be used to approximate a 99% confidence interval on the average number of mismatches. We assume that each trial with probability p of getting a mismatch. Then, the variance is that of a Bernoulli, which is $p\left(1-p\right)$. Thus, again using CLT, we get that an

approximate 99% interval given by

$$\left[p - F^{-1}(0.995)\sqrt{\frac{p(1-p)}{n}}, p + F^{-1}(0.995)\sqrt{\frac{p(1-p)}{n}}\right].$$

8.4.8 Scripts

There are end-to-end scripts that make it easy for the user to run GoldenEye on a given model in a simplified manner. The general pipeline runs `preprocess.py` to get the max magnitude per layer. Then, `profile_model.py` is run to get baseline measurements. Then, `split_data.py` creates an analysis/deployment set for the error injection campaign to use for the forward passes. Then, `injections.py` performs the actual error injection campaign. Then, `postprocess.py` summarizes the outputs of the error injection campaign.

8.5 Use Cases

[MTA+22] proposes two overarching potential use cases for GoldenEye for future research. Here, we detail each of these use cases.

8.5.1 Number Format Analysis

First, [MTA+22] proposes that GoldenEye be used for number format analysis. In particular, by disabling error injections, we can functionally simulate running inference on different models under different emulation and quantization schemes. This is critical for choosing the quantization data type $\mathcal{B}$ when designing specialized hardware like in Chapter 9. GoldenEye's flexibility to support various number systems makes it a good candidate for such exploration. This helps hardware designers to choose between different data types.

GoldenEye also supports the choice of various hyperparameters like radix point location and metadata width. Therefore, GoldenEye can also be a useful tool when deciding the appropriate bitwidth for a given data type. [MTA+22] proposes the following binary search algorithm. The algorithm is as follows. Start from some nominal bitwidth like 32 bits. Then, for some current bitwidth n with parent bitwdith m, consider $n - \frac{m-n}{2}$ and $n + \frac{m-n}{2}$. For each bitwidth, we find the optimal radix point location and its associated accuracy. Then, choose $n - \frac{m-n}{2}$ if accuracy does not degrade by more than $t\%$ for some threshold $t\%$. Otherwise, choose $n + \frac{m-n}{2}$. Repeat this process until we reach the leaf of the binary tree. This algorithm operates under the assumption that more bits will not lead to accuracy degradation. This observation is true, since more bits have strictly more expressivity. Therefore, this approach

allows a user to find the optimal bitwidth for a given data type. [MTA+22] illustrates this binary search approach with the diagram below.

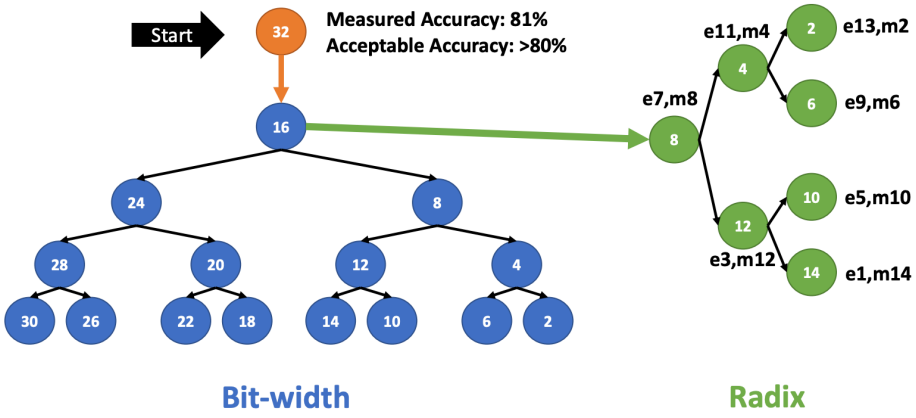

Figure 8.4: Schematic for binary search algorithm for optimal bitwidth. [MTA+22]

8.5.2 Layer-Level Resiliency Analysis

As described in the problem statement, we would like to simulate fault injections to test the resiliency of different machine learning models. This will allow us to build "algorithmic" resilience as discussed in Chapter 7. [MTA+22] reasons that GoldenEye allows us to model the layer-level resilience of models under different data types. In particular, we can see the average change in loss under a single injection for every layer. Therefore, this tells us the resiliency of each layer and which layers should be hardened. However, [MTA+22] falls short in generalizing this to a model-level. In particular, they use the average change in loss across the layers to encapsulate this idea, but it lacks mathematical backing. Therefore, in the next section, we will propose modifications to GoldenEye that will allow us to reason better about these metrics.

8.6 Further Layer-Level Resiliency Analysis

In this section and the next, we detail our additional contributions to the project. First, we will provide some mathematical backing to the resiliency analysis in the spirit of Chapter 7. In particular, we will extend the poisson process model that we presented in Chapter 7.

Recall that in Chapter 7 we modeled model-level resilience with model-wide or layer-wise timing. We can make this resilience metric more granular by considering it at a layer level. Here, we see the flexibility of the Poisson model shine. Suppose we want to measure the resilience of layer ℓ. Then, for bit (i, j) not in layer ℓ, we can simply exclude them from our

model. Let $L_t^{(\ell)}$ be the change in loss from layer ℓ during time t. Then, we are interested in modeling

$$\mathbb{E}\left[L_t^{(\ell)}\right] = \mathbb{E}\left[\mathbb{E}\left[L_t^{(\ell)} \,\middle|\, \left\{\{N_{i,j}(t)\}_{j=1}^{b}\right\}_{i\in \mathbf{s}^{(\ell)}}\right]\right],$$

where $s^{(\ell)}$ is the indices of the weights in layer ℓ and $N_{i,j}(t)$ is the number of transient hardware errors for bit (i,j) over time t.

Again, we can follow the same procedure detailed in Chapter 7 to get an unbiased estimate for $\mathbb{E}\left[L_t^{(\ell)}\right]$. Using the superposition property, we can sample

$$N \sim \text{Pois}\left(\sum_{i\in \mathbf{s}^{(\ell)}} \sum_{j=1}^{b} \lambda_{i,j} t\right).$$

Then, for each of the N events, we select a bit (i,j) with probability $\frac{\lambda_{i,j}}{\sum_{y\in \mathbf{s}^{(\ell)}} \sum_{z=1}^{b} \lambda_{(y,z)}}$. Then, we flip the bit. After N bits have been flipped, then we compute the change in loss. We can repeat this process for M trials and take the sample mean.

The key distinction between this proposed error metric and the one used in [MTA+22] is two-fold. First, we assume variable number of bitflips per layer, rather than always assuming a single bitflip. Furthermore, the number of bitflips depends on both the number of weights in the layer and the time it takes for the computation of the layer (proportional to the number of MACs). While the number of bitflips per layer can still be small, this introduces another dimension that is not considered in [MTA+22]. The resiliency of a layer likely depends on the size, since the larger the layer the more chances of having a bitflip. Furthermore, it should depend on the time that the values reside in memory and are moved around. Thus, we claim that this is a more statistically sound estimate for layer-level resilience.

8.7 Reliability-Aware Quantization

In this section, we will present reliability-aware quantization as a potential final use case of GoldenEye. In particular, we claim that the guiding goals of quantization also hold true for building reliable systems. To motivate these parallels, we first discuss minimum description length (MDL) theory [Ris78]. Then, we will motivate the use of GoldenEye with Hessian-Aware Quantization (HAWQ) [DYG+19].

8.7.1 Minimum Description Length (MDL) Theory [Ris78]

First, we discuss classical MDL theory, a information theoretic approach to model selection. The work of MDL theory and the relevant literature largely preceded the rise of machine

learning models. However, the guiding principles translate directly to machine learning.

The setup for MDL theory is as follows. Suppose we have models $\mathcal{M}_1, \ldots, \mathcal{M}_n$. Naturally, more complex models will perform better on our training set, but we want to choose a generalizable model that will minimize generalization error. Intuitively, more complex models will yield overfitting to the training dataset. Thus, MDL is tightly coupled with another idea in machine learning called Occam's Razor, which says the "simplest" explanation of a dataset is the best. MDL formally describes "simplicity" by positioning the idea in the information theory world.

While we will avoid the rigorous specifics of MDL, we will highlight the key takeaways from MDL theory in a simplified manner. In particular, we focus on crude MDL theory, which takes a two-part approach to compressing the model and the data. The model is viewed as a way of "encoding" the data. Thus, we measure the "code length" of the model and data based on two parts. First, we focus on the encoding of the model itself. This represents the "complexity" of the model itself. Then, we "encode" the data using the model. This represents how well the model approximates the data and is analogous to the idea of entropy. It represents the amount of bits that need to be transmitted to compensate for the misfit between the model and the data. Given a model $\mathcal{M}$, we assign a "code length" for the model data that is equal to

$$L(\mathcal{M}) + (-\log(p(\mathbf{y}|\mathbf{x}, \mathbf{w}))),$$

where $\mathbf{w}$ is the parameters of the model $\mathcal{M}$. The first term $L(\mathcal{M})$ represents the encoding of the model itself, and the second term $-\log(p(\mathbf{y}|\mathbf{x}, \mathbf{w}))$ represents the negative log-likelihood of the data given the model. Thus, the model that minimizes this "total code length" is viewed to be optimal under MDL theory.

MDL theory is also tightly coupled with a Bayesian approach to training deep neural networks as explored in [HvC93]. In particular, a Bayesian prior is a nice framework to model the "complexity" of the model itself. In particular, they use a Gaussian prior for the weights and train a model that minimizes the expected "code length" of the model and the data. While we leave this Bayesian perspective beyond the scope of this thesis, this insight that less "imprecise" weights are easier to represent is critical for our next section.

8.7.2 Hessian Aware Quantization (HAWQ) [DYG+19]

Now, we make the connection between MDL theory, quantization, and reliability. Intuitively, models with flat minima may be preferrable under the MDL framework. For models with flat minima, we have many parameters within a region that roughly correspond to the same "compression" of the data. Thus, we can require less precision when specifying the

model itself while maintaining high compression of the data. Thus, we can decrease the first term, $L(\mathcal{M})$, while keeping the second term relatively constant. Furthermore, MDL theory tells us that these models will likely generalize better, since they have a shorter "code length". Therefore, for models with flatter minima, we may be interesting in finding more "compressed" ways to represent the model.

Weight-based quantization is one way that we can compress such models, since we decrease the precision of the weights. When the objective function is "flatter" with respect to a parameter, that parameter can be more aggressively quantized. MDL theory gives us the reassurance that the generalization error will likely still remain the same. Thus, our goal is to identify the parameters for which the objective function is flat with respect to that parameter. This motivates the idea of HAWQ.

HAWQ is a mixed-precision quantization scheme, which attempts to give a relative ordering of precision for each unit (typically layers) of a deep neural network. Ideally, we want weights in the same unit of a deep neural network to share the same data type as discussed in Chapter 5. Thus, HAWQ divides the neural network into a collection of B "blocks". Then, they use the largest eigenvalue of the Hessian matrix with respect to a block as a proxy for the curvature of the loss function with respect to the parameters in that block. The largest magnitude eigenvalue is a natural proxy for curvature, since it captures the largest rate of change of the loss with respect to the parameters in any arbitrary direction. A high magnitude indicates a large change in loss and a relatively steep loss landscape, whereas a small magnitude indicates a relatively flat landscape.

More formally, let $\mathbf{H}_b = \frac{\partial^2 L}{\partial \mathbf{w}_b^2}$ be the Hessian matrix for block b, where $\mathbf{w}_b$ are the flattened weights of block b. Then, we will assign block b with a score of $\frac{|\lambda_b|}{|\mathbf{w}_b|}$, where $|\lambda_b|$ is the largest magnitude eigenvalue of $\mathbf{H}_b$. Then, this yields a relative order of precision, where blocks with the lowest score will be assigned the lowest precision. We divide by $|\mathbf{w}_b|$ to account for the fact that blocks with more parameters will yield a larger memory benefit if quantized to a smaller precision.

Suppose we have some base data type d. Then, let d_b be the data type with a bitwidth of b. Furthermore, suppose that we are limited to data types $\{d_{b_1}, \ldots, d_{b_n}\}$. Then, initially, we have n^B options for a mixed-precision quantization scheme for a model with B blocks. This is exponential in the number of blocks. However, once we enforce the relative ordering, we claim that we have reduced the number of options to $\binom{n+B-1}{B}$. Given n labels, let x_i be the number of B blocks that are assigned to label i. Then, we have that $x_1 + x_2 + \ldots + x_B = n$. Furthermore, given $(x_1, x_2, \ldots, x_B)$, we have fully determined the assigned labels for the B blocks, since we have a relative ordering. Thus, by a sticks-and-stones argument, we have

that the number of orderings is

$$\binom{n+B-1}{B} = \binom{n+B-1}{n-1} = \frac{(n+B-1)\cdot(n+B-1)\cdots(B+1)}{(n-1)!}.$$

Then, we have the following bounds.

$$\frac{B^{n-1}}{(n-1)!} \leq \frac{(n+B-1)\cdot(n+B-1)\cdots(B+1)}{(n-1)!} \leq \frac{(n+B-1)^{n-1}}{(n-1)!}$$

For $B > n-1$, we have that $n+B-1 \leq B$. Thus, we can update the bounds for $B > n-1$ as follows.

$$\frac{1}{(n-1)!}B^{n-1} \leq \frac{(n+B-1)\cdot(n+B-1)\cdots(B+1)}{(n-1)!} \leq \frac{2^{n-1}}{(n-1)!}B^{n-1}$$

Assuming that n is a fixed constant, we have now shown that $\binom{n+B-1}{B} \in \Theta\left(B^{n-1}\right)$, which is polynomial in B. This is a significant improvement from a previously exponential search space. Thus, we can apply a brute force search to this search space to yield the optimal quantization scheme.

8.7.3 Computation in HAWQ

Now, we detail the computation require in HAWQ. We will use this as motivation to show that using GoldenEye is likely more desirable and interpretable than HAWQ. First, we are interested in calculating the largest real eigenvalue for $\mathbf{H}_b \in \mathbb{R}^{n_b \times n_b}$, the Hessian for block b. Since $\mathbf{H}_b$ is a Hessian matrix and we assume that L is sufficiently "nice", we will assume that $\mathbf{H}_b$ is symmetric ($\mathbf{H}_b = \mathbf{H}_b^\intercal$). Then, by the Spectral Theorem, we are guaranteed that $\mathbf{H}_b$ will have all real eigenvalues. Calculating the largest eigenvalue for a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is quite difficult. Therefore, in this section, we discuss the methods we use to compute the largest eigenvalue. We want to find all eigenvectors $\mathbf{v} \in \mathbb{R}^n$ and associated eigenvalues $\lambda \in \mathbb{R}$ such that $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$. Then, we have that

$$(\mathbf{A} - \lambda\mathbf{I})\,\mathbf{v} = \mathbf{0}.$$

We assume that $\mathbf{v} \neq \mathbf{0}$. Thus, we have that λ is an eigenvalue of $\mathbf{A}$ iff $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$. Thus, we can compute the largest eigenvalue directly by solving $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ and finding the largest such λ. However, this yields a few problems.

First, computing the discriminant is computationally expensive. The best known algorithms run in $O\left(n^\omega\right)$ time, where $\omega > 2$. Given that n is the number of weights in a block

b, n can be quite large. Furthermore, once we have computed the characteristic polynomial, finding the roots is mathematically difficult. The Abel-Ruffini theorem states that for any polynomial with degree greater than or equal to 5, there does not exist a closed form solution for the roots. Thus, numerical approximations need to be made to find the largest root. This motivates the implementation of other eigenvalue approximations that we will now discuss.

Power Iteration [DYG+19]

First, we discuss power iteration, which is a commonly used approximate method for computing the largest real eigenvalue in magnitude. Assuming that $\mathbf{H}_b$ is symmetric, we have by the Spectral Theorem that $\mathbf{H}_b$ is diagonalizable. In particular, we have that

$$\mathbf{H}_b = \mathbf{V}\Lambda\mathbf{V}^{-1}$$

for some $\mathbf{V} \in \mathbb{R}^{n_b \times n_b}$. Here, $\Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_{n_b})$, where $|\lambda_1| > \cdots > |\lambda_{n_b}|$. Since $\mathbf{V}$ is invertible, we have that its column space is $\mathbb{R}^{n_b}$. Therefore, consider a random vector $\mathbf{v} \in \mathbb{R}^{n_b}$. Then, we have that

$$\mathbf{b} = \mathbf{V}\mathbf{c} = \mathbf{c}_1\mathbf{v}_1 + \mathbf{c}_2\mathbf{v}_2 + \cdots + \mathbf{c}_{n_b}\mathbf{v}_{n_b}$$

for some $\mathbf{c} \in \mathbb{R}^{n_b}$.

Now, consider repeatedly multiplying $\mathbf{v}$ by $\mathbf{H}_b$. Then, we have the following assuming $\lambda_1 \neq 0$.

$$\begin{aligned}
\mathbf{H}_b^k\mathbf{b} &= \left(\mathbf{V}\Lambda\mathbf{V}^{-1}\right)^k \mathbf{b} && (\mathbf{H}_b^k = \mathbf{V}\Lambda\mathbf{V}^{-1}) \\
&= \mathbf{V}\Lambda^k\mathbf{V}^{-1}\mathbf{b} \\
&= \mathbf{V}\Lambda^k\mathbf{V}^{-1}\mathbf{V}\mathbf{c} && (\mathbf{b} = \mathbf{V}\mathbf{c}) \\
&= \mathbf{V}\Lambda^k\mathbf{c} \\
&= \sum_{i=1}^{n_b} \lambda_i^k \mathbf{c}_i\mathbf{v}_i \\
&= \lambda_1^k \sum_{i=1}^{n_b} \left(\frac{\lambda_i}{\lambda_1}\right)^k \mathbf{c}_i\mathbf{v}_i
\end{aligned}$$

Then, as $k \to \infty$, we have that

$$\mathbf{H}_b^k\mathbf{b} \to \lambda_1^k\mathbf{c}_1\mathbf{v}_1,$$

since $|\lambda_i| \leq |\lambda_1|$ for all $i > 1$. Thus, we have shown that

$$\frac{\mathbf{H}_b^k \mathbf{b}}{||\mathbf{H}_b^k \mathbf{b}||_2}$$

will yield the eigenvector with the largest eigenvalue in magnitude.

Therefore, we can follow the following algorithm to obtain the largest eigenvector through I iterations.

Algorithm 1 Power Iteration

1: Choose a random vector $\mathbf{b}_0 \in \mathbb{R}^{n_b}$
2: **for** $k = 1, 2, \ldots,$ I **do**
3: $\quad \mathbf{b}_k = \frac{\mathbf{H}_b \mathbf{b}_{k-1}}{||\mathbf{H}_b \mathbf{b}_{k-1}||_2}$
4: **end for**
5: **return** $\mathbf{b}_I$

I is a hyperparameter that can be chosen prior to runtime. It should be noted that the rate of convergence (decrease in error) is exponential in $\left|\frac{\lambda_2}{\lambda_1}\right|$. Every iteration decreases the contribution of $\mathbf{v}_i$ to $\mathbf{H}_b^k \mathbf{b}$ by a factor of $\frac{\lambda_i}{\lambda_1}$ for $i \neq 1$. Since $|\lambda_2| > \cdots > |\lambda_{n_b}|$, we have that the slowest rate of convergence is $\left|\frac{\lambda_2}{\lambda_1}\right|$. Since the error decreases exponentially, it is typically sufficient for I to be on the order of 10^2 to get a decent estimate. Furthermore, in our analysis, it is necessary to assume that $\mathbf{c}_1 \neq 0$, else $\mathbf{H}_b^k \mathbf{b} \to \mathbf{0}$. However, since the vector $\mathbf{b}_0$ is chosen randomly, we can state that this occurs with probability 0.

Then, we can approximate the associated eigenvalue by using the Rayleigh quotient $\mathbf{b}_I^\intercal \mathbf{H}_b \mathbf{b}_I \in \mathbb{R}$. Intuitively, this metric is reasonable because it is the dot product between $\mathbf{b}_I$ and $\mathbf{H}_b \mathbf{b}_I$. If $\mathbf{b}_I$ is an eigenvector such that $\mathbf{H}_b \mathbf{b}_I = \lambda \mathbf{b}_I$, then we get that the Rayleigh quotient simplifies to the true eigenvalue, since

$$\mathbf{b}_I^\intercal \mathbf{H}_b \mathbf{b}_I = \lambda \mathbf{b}_I^\intercal \mathbf{b}_I = \lambda,$$

assuming $||\mathbf{b}_I||_2 = 1$.

Thus, to find the largest eigenvalue, it suffices to multiply a random vector repeatedly by $\mathbf{H}_b$. Multiplying this vector can be done in at most $O(n_b^2)$ time, which already outperforms finding the eigenvalue through the discriminant.

Computing $\mathbf{H}_b \mathbf{b}$

As noted in the section above, it is sufficient to repeatedly compute $\mathbf{H}_b \mathbf{b}$ for some random vector $\mathbf{b}$. This is sometimes called the Hessian Vector Product (HVP). However, constructing

and storing the Hessian in memory can be quite expensive particularly for DNNs, since there are n_b^2 elements that need to be stored. Thus, we can perform one more optimization, such that power iteration can be run in linear time with respect to n_b.

Let $\mathbf{g}_b = \frac{\partial L}{\partial \mathbf{w}_b}$. Then, we have that

$$\frac{\partial \mathbf{g}_b^\mathsf{T} \mathbf{b}}{\partial \mathbf{w}_b} = \frac{\partial \mathbf{g}_b^\mathsf{T}}{\partial \mathbf{w}_b} \mathbf{b} = \mathbf{H}_b \mathbf{b},$$

since $\mathbf{b}$ is a randomly chosen vector and not a function of $\mathbf{w}_b$. Thus, to compute $\mathbf{H}_b \mathbf{b}$, it suffices to compute $\mathbf{g}_b^\mathsf{T} \mathbf{b} \in \mathbb{R}$, and then compute $\frac{\partial \mathbf{g}_b^\mathsf{T} \mathbf{b}}{\partial \mathbf{w}_b}$ through backpropagation. The upshot is that we no longer need to construct the $\mathbf{H}_b$ matrix in memory. Furthermore, the amount of computation required is linear in the number of weights n_b. This optimization method was first introduced in [Pea94], and is referred to as the reverse-over-reverse method of computing the HVP. There exists other methods for computing the HVP like forward-over-reverse or reverse-or-forward, but we leave these beyond the scope of the thesis.

First, we can compute a forward pass of the deep neural network, which has a linear runtime in n_b. In terms of the computation graph, we have $O(n_b)$ nodes. Then, we can use the backpropagation algorithm outlined in Chapter 1, which also runs in linear time with respect to n_b, to get $\mathbf{g}_b$. We can now create another computational graph that contains the original parameters $\mathbf{w}_b$ as well as the newly calculated gradient values $\mathbf{g}_b$. This computational graph still has $O(n_b)$ nodes. Computing $\mathbf{g}_b^\mathsf{T} \mathbf{b}$ can also be computed in linear time with respect to n_b. Finally, we can run backpropagation again on $\mathbf{g}_b^\mathsf{T} \mathbf{b}$ with respect to $\mathbf{w}_b$. Since the number of nodes in the computational graph is still linear in n_b and the output is a scalar, this can be done in $O(n_b)$ time as utlined in Chapter 1. Therefore, this entire algorithm can be run in $O(n_b)$, and it need not construct the Hessian matrix explicitly.

PyHessian [YGKM20]

PyHessian is a library that efficiently implements numerical algorithms for finding summary statistics of the Hessian of a deep neural network like largest eigenvalue. PyHessian is implemented in PyTorch and uses PyTorch's autodifferentiation algorithm to implement the reverse-over-reverse method outlined above.

8.7.4 Relation to Reliability

We claim that the layer-wise resiliency metric presented in the above section is an equally good, if not better, alternative to the eigenvalue for describing the curvature of the loss function with respect to a given layer. In particular, the eigenvalue captures the largest

rate of change in the loss in any arbitrary direction from the optimal weights. By randomly performing bitflip perturbations to the weights and finding the change in loss we can also model this effect. In particular, flat minima will tend to have very small changes in loss with respect to these perturbations, while sharp minima will tend to have very large changes in loss with respect to these perturbations. Therefore, eigenvalues and layer-wise resilience model very similar notions.

The advantage of GoldenEye lies in the fact that gradients need not be taken. Our extensive discussion of computing eigenvalues showed that we need not construct the Hessian $\mathbf{H}_b$, but we still need to take the derivative of $\mathbf{g}_b^\top \mathbf{b}$ with respect to $\mathbf{w}_b$. While this can still be done in linear time, it has much more overhead than a forward pass. Furthermore, it needs to be done iteratively until convergence of power iteration. Alternatively, to calculate the layer-wise resilience using GoldenEye we only need to run a forward pass on the model with some error injections. Taking the average over a few trials allows us to find an approximate for the mean layer-wise resilience, which measures a similar notion to the eigenvalue.

8.8 Future Work

First, our analysis above focuses on evaluating the resiliency of a model at a layer-level. As discussed, we would ideally want to evaluate the model at a model-level. As introduced in Chapter 7, we would like to model model-level resilience with model-wide timing or layer-wise timing. However, the current setup of GoldenEye makes it difficult to model such model-wide metrics, since a single injection is performed per every layer. We can easily extend GoldenEye though to a multi-injection multi-layer framework. Then, using the model and methodology described in Chapter 7, we can calculate the model-level resilience with layer-wise or model-wide timing.

Second, GoldenEye focuses on fault injection from a model perspective. It abstracts the details of the actual computation that is done on a CPU and GPU, and it assumes that they can simply be modeled by perturbing the weights or activations. However, this may not always be true. For example, a fault injection may only affect weight for a certain computation and not for its entire lifetime. The fault could occur when the weight is loaded into a register, while its contents in main memory are preserved. In this way, GoldenEye is not a "cycle-accurate", as it does not model the exact hardware at every clock cycle. However, we should note that making such cycle-accurate simulators is quite difficult, since we need to precisely model the internal hardware implementation. Therefore, future research should seek to understand the tradeoffs between these modeling strategies and potential solutions for it.

Finally, GoldenEye can be applied to many other domains that extends far beyond the fields of dependability and reliability. We will see one such examples in the subsequent chapter (Chapters 9). Another promising area of future research is in the realm of adversarial machine learning. In particular, we can imagine that the perturbations applied to our models are not the result of some transient hardware error, but rather some adversary seeking to corrupt the output of the model. In that case, we can replace the standard bitflip model that we have presented with alternatives error injection models. Then, we can use GoldenEye to better understand the effects of these adversarial attacks. Furthermore, we have mainly focused on the application of faults to the forward inference pass. However, we should note that transient hardware errors are just as likely during training as in the inference pass. Therefore, GoldenEye can be applied to better understand the effect of fault injections on convergence behavior during training. Ideally, faults would cause slight deviations that further training iterations could recover from, but this is not immediately obviously. It may also allow us to determine the "best" optimizer or training algorithms, by evaluating their robustness to such error injections.

9

EdgeBERT [THP+21]

In this section, we present our joint work on EdgeBERT, advised by Thierry Tambe and David Kong. For the full details on EdgeBERT and the EPOCHS SoC, please refer to [THP+21] and [DSJZ+24] respectively. EdgeBERT is an accelerator designed to accelerate transformer-based inference workloads using a few key optimizations. Recently, EdgeBERT was integrated onto the EPOCHS SoC [DSJZ+24], which features a total of fourteen accelerators with four Ariane RISC-V cores and systolic array tiles. In this section, we detail our profiling of EdgeBERT using silicon performance data collected via the EPCOHS SoC.

9.1 EdgeBERT Optimizations

First, we discuss the key optimizations used in the construction of EdgeBERT. Most of these optimizations build off of ideas previously discussed in Chapters 4, 5, and 6.

9.1.1 Early Exit

Formulation

As discussed in 4, encoder-only models feature N attention layers. However, a key observation from [XTL+20] is that not all inputs may need to run through all N layers. In

particular, one might expect "simpler" inputs to only need some fraction of the N attention layers before the hidden state is sufficient for classification. Thus, [XTL+20] propose an early-exit (EE) mechanism, where a few linear layers are added atop the output for the `[CLS]` token for each Transformer layer. For some input, let the output of these linear layers at layer ℓ be $\mathbf{x}^{(\ell)} \in \mathbb{R}^n$. Then, $\mathbf{z}^{(\ell)} = \text{softmax}\left(\mathbf{x}^{(\ell)}\right)$ can be used as the classification probabilities. These classification layers have the same objective as the classification layers after the final attention layer. Therefore, we can use $\mathbf{z}^{(\ell)}$ as a proxy for whether we can suspend inference at the current layer. In particular, we are interested in measure how "stable" the current classification output is. In order to measure this, we draw on ideas from information theory that are detailed in Chapter 1. In particular, we calculate the self-entropy of $\mathbf{z}^{(\ell)}$.

[XTL+20] proposes this exact methodology. Suppose we are at layer ℓ. Then, let the output of the classification layers at layer ℓ be $\mathbf{z}^{(\ell)} \in \mathbb{R}^n$. Then, we calculate the "self-entropy" of $\mathbf{z}^{(\ell)}$ as

$$H\left(\mathbf{z}^{(\ell)}\right) = -\sum_{i=1}^{n} \mathbf{z}_i^{(\ell)} \log\left(\mathbf{z}_i^{(\ell)}\right).$$

Note that

$$\mathbf{z}_i^{(\ell)} = \frac{\exp\left(\mathbf{x}_i^{(\ell)}\right)}{\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)}.$$

Thus, we have the following.

$$\begin{aligned}
H\left(\mathbf{z}^{(\ell)}\right) &= -\sum_{i=1}^{n} \mathbf{z}_i^{(\ell)} \log\left(\mathbf{z}_i^{(\ell)}\right) \\
&= -\sum_{i=1}^{n} \frac{\exp\left(\mathbf{x}_i^{(\ell)}\right)}{\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)} \log\left(\frac{\exp\left(\mathbf{x}_i^{(\ell)}\right)}{\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)}\right) \\
&= -\sum_{i=1}^{n} \frac{\exp\left(\mathbf{x}_i^{(\ell)}\right)}{\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)} \left(\log\left(\exp\left(\mathbf{x}_i^{(\ell)}\right)\right) - \log\left(\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)\right)\right) \\
&= -\sum_{i=1}^{n} \frac{\exp\left(\mathbf{x}_i^{(\ell)}\right)}{\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)} \mathbf{x}_i^{(\ell)} + \sum_{i=1}^{n} \frac{\exp\left(\mathbf{x}_i^{(\ell)}\right)}{\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)} \log\left(\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)\right) \\
&= -\sum_{i=1}^{n} \frac{\exp\left(\mathbf{x}_i^{(\ell)}\right)}{\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)} \mathbf{x}_i^{(\ell)} + \log\left(\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)\right) \frac{\sum_{j=1}^{n} \exp\left(\mathbf{x}_i^{(\ell)}\right)}{\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)} \\
&= -\sum_{i=1}^{n} \frac{\exp\left(\mathbf{x}_i^{(\ell)}\right)}{\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)} \mathbf{x}_i^{(\ell)} + \log\left(\sum_{j=1}^{n} \exp\left(\mathbf{x}_j^{(\ell)}\right)\right)
\end{aligned}$$

Then, we can compare $H\left(\mathbf{z}^{(\ell)}\right)$ to some entropy threshold E_T. If $H(x) < E_T$, then we can suspend inference at that layer. In particular, it follows the following algorithm.

Algorithm 2 Conventional Early Exit Inference [THP+21]

```
for input i = 1 to n do
    for layer ℓ = 1 to 12 do
        z^(ℓ) ← f(x_i; w)
        if H(z^(ℓ)) < E_T then
            exit
        end if
    end for
end for
```

Hardware Implications

EdgeBERT makes a few notable modifications to the proposed early exit algorithm above. Abruptly ending inference is inconvenient, especially if we would like to give some uniform guarantees on latency. In particular, we would like to run the program for T seconds. Therefore, [THP+21] proposes the following modifications. First, it uses the entropy of the first transformer layer classification to try to predict the approximate layer that inference will be stopped at. The approximation is done again via linear layers. However, one can also view this as a lookup table (LUT), since given a threshold and an entropy, the model returns the expected exit layer.

Then, given this expected end time, we can scale the frequency down. As discussed in Chapter 2, scaling the frequency down allows us to lower the voltage. To find the optimal frequency and voltage, EdgeBERT does the following. First, to determine the frequency, we let

$$f' = \frac{N}{T - T_{\text{curr}}},$$

where N is the number of cycles needed to complete the inference up until the early exit layer, T is the target time, and T_{curr} is the current elapsed time that was already used to compute the first layer and predict the exit layer. Intuitively, we just divide the number of remaining cycles by the amount of remaining time until the target. Then, EdgeBERT uses a lookup table to find the optimal voltage V'_{DD} for the corresponding f'.

The voltage is controlled using a low-dropout voltage regulator (LDO) to modulate the supply voltage. In particular, the LDO featured on EdgeBERT can supply voltage between $0.5V$ and $0.8V$. The frequency is controlled by an all-digital phase-locked loop (ADPPL). The ADPPL uses some reference clock frequency to generate a stable clock signal that can

change in frequency. Both of these components are digital components that are synthesizable. In particular, this means that they can be specified directly into the HDL.

Therefore, the algorithm is modified to the following.

Algorithm 3 EdgeBERT Early Exit Inference [THP+21]

for input $i = 1$ to n **do**
  for layer $\ell = 1$ **do**
    $\mathbf{z}^{(\ell)} \leftarrow f\left(\mathbf{x}_i; \mathbf{w}\right)$
    if $H\left(\mathbf{z}^{(\ell)}\right) < E_T$ **then**
      exit
    end if
    $L \leftarrow \mathrm{LUT}_{\mathrm{DVFS}}\left(H\left(\mathbf{z}^{(\ell)}\right), E_T\right)$
    $f' \leftarrow \frac{N}{T - T_{\mathrm{curr}}}$
    $V'_{DD} \leftarrow \mathrm{LUT}_{\mathrm{EE}}\left(f'\right)$
  end for
  for layer $\ell = 2$ to L **do**
    $\mathbf{z}^{(\ell)} \leftarrow f\left(\mathbf{x}_i; \mathbf{w}\right)$
    if $H\left(\mathbf{z}^{(\ell)}\right) < E_T$ **then**
      exit
    end if
  end for
end for

We make the following notes. First, once an exit layer has been predicted, inference will terminate at that layer regardless of whether or not the entropy of the classification at that layer is less than the threshold. This is because we have already scaled the frequency to reach the target time. Furthermore, we should note that there are two lookup tables. The first, $\mathrm{LUT}_{\mathrm{EE}}$ is for determining the expected exit layer. The second, $\mathrm{LUT}_{\mathrm{DVFS}}$ determines the optimal V'_{DD} given a frequency f'.

9.1.2 Adaptive Attention Span

EdgeBERT supports masking, similar to that described in Chapter 4. Recall that we presented masked attention as $\mathrm{MaskedAttention} : \mathbb{R}^{n \times d_k} \times \mathbb{R}^{m \times d_k} \times \mathbb{R}^{m \times d_v} \times \mathbb{R}^{n \times m} \to \mathbb{R}^{n \times d_v}$ defined by

$$\mathrm{MaskedAttention}\left(\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{M}\right) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathsf{T}} + \mathbf{M}}{\sqrt{d_k}}\right)\mathbf{V}.$$

EdgeBERT uses a similar masking scheme, but uses multiplicative masking rather than additive masking.

In particular, we define it as follows. Let $\mathrm{MultMaskedAttention} : \mathbb{R}^{n \times d_k} \times \mathbb{R}^{m \times d_k} \times \mathbb{R}^{m \times d_v} \times$

$\mathbb{R}^{n\times m} \to \mathbb{R}^{n\times d_v}$ be defined by

$$\text{MaskedAttention}\left(\mathbf{Q},\mathbf{K},\mathbf{V},\mathbf{M}\right) = \text{Normalize}\left(\mathbf{M}\odot\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathsf{T}}}{\sqrt{d_k}}\right)\right)\mathbf{V}.$$

Here, $\text{Normalize}:\mathbb{R}^{n\times m}\to\mathbb{R}^{n\times m}$ is defined by the following. We have that

$$\text{Normalize}\left(\mathbf{X}\right)_{ij} = \frac{\mathbf{X}_{ij}}{\sum_{k=1}^{m}\mathbf{X}_{ik}}.$$

Intuitively, we simply divide each row by the sum of the row. This allows us to retain the nice property of the softmax that the sum of the rows was always 1.

Adaptive Attention Span

Now, we provide a potential implementation strategy for determining an appropriate attention span. Note that one need not use this to take advantage of EdgeBERT's masking. Any mask can be used, but we just propose a strategy for determining the attention span. This implementation is based on that proposed in [SGBJ19]. [SGBJ19] proposes an adaptive attention span for a causal masking scheme. However, since we will focus on BERT models, which are bi-directional, as discussed in Chapter 4, we will adapt it to be bi-directional.

First, we will define a function $d:[0,S]\to\mathbb{R}$ parameterized by $z\in\mathbb{R}$. Then, the map is defined by

$$d_z(x) \triangleq \min\left(\max\left(\frac{1}{R}\left(R+z-x\right),0\right),1\right),$$

where R is some pre-selected hyperparamter. Graphically the function looks as follows.

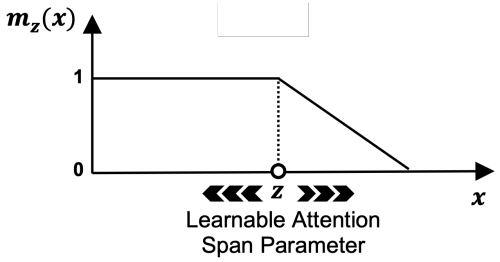

Figure 9.1: m_z function. [THP+21]

Note that the function's range is $[0,1]$. Now, we will choose $\mathbf{M}_{ij}$ such that

$$\mathbf{M}_{ij} = d_z\left(|j-i|\right).$$

Intuitively, we want the "attention" paid to tokens to decrease as they are further from the current token. Using this mask, we have that weights for the weighted sum of the values is equivalent to the following. Let $\mathbf{X} = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}$ for notational convenience.

$$\begin{aligned}
\left(\text{Normalize}\left(\mathbf{M} \odot \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\right)\right)_{ij} &= \frac{\left(\mathbf{M} \odot \text{softmax}\left(\mathbf{X}\right)\right)_{ij}}{\sum_{k=1}^{m}\left(\mathbf{M} \odot \text{softmax}\left(\mathbf{X}\right)\right)_{ik}} \\
&= \frac{d_z\left(|j-i|\right) \cdot \frac{\exp(\mathbf{X}_{ij})}{\sum_{k=1}^{m}\exp(\mathbf{X}_{ik})}}{\sum_{\ell=1}^{m} d_z\left(|\ell-i|\right) \cdot \frac{\exp(\mathbf{X}_{i\ell})}{\sum_{k=1}^{m}\exp(\mathbf{X}_{ik})}} \\
&= \frac{\frac{1}{\sum_{k=1}^{m}\exp(\mathbf{X}_{ik})} d_z\left(|j-i|\right)\exp\left(\mathbf{X}_{ij}\right)}{\frac{1}{\sum_{k=1}^{m}\exp(\mathbf{X}_{ik})}\sum_{\ell=1}^{m} d_z\left(|\ell-i|\right)\exp\left(\mathbf{X}_{i\ell}\right)} \\
&= \frac{d_z\left(|j-i|\right)\exp\left(\mathbf{X}_{ij}\right)}{\sum_{\ell=1}^{m} d_z\left(|\ell-i|\right)\exp\left(\mathbf{X}_{i\ell}\right)}
\end{aligned}$$

This matches the form presented in [SGBJ19]. Thus, the goal during training is to learn the optimal attention span parameter z.

Hardware Implications

EdgeBERT supports the multiplicative mask within the `SMax` module, which computes the softmax. The `SMax` implements the softmax operation in the numerically stable way, as described in Chapter 3. `SMax` takes an finite-state machine (FSM) to break the computation of the softmax into multiple cycles. For the mask states, which occur after the softmax operation is computed, element-wise multiplication is performed with the mask, which is loaded in the auxiliary buffer.

It is possible for $\mathbf{M} = \mathbf{0}_{n\times m}$. In that case, the softmax computation is unnecessary, since the mask will immediately zero out the outputs. Thus, EdgeBERT skips computation in this case and simply outputs zeros.

9.1.3 Pruning

EdgeBERT takes advantage of sparse data structures to decrease its memory footprint. In this section, we discuss a desirable pruning method that is used to train the ALBERT model. Then, we discuss the hardware implications of supporting sparse data structures and computation in EdgeBERT.

Hardware Implications

As discussed in Chapter 6, we can use a bitmask-encoding to compactly represent sparse data structures. EdgeBERT uses this bitmask-encoding for its inputs, where each data input is accompanied by a mask input. The data input contains all non-zero inputs. The mask input contains a 0 for a null input at the given index and a 1 to indicate a non-null input at the given index. This addresses the issue of representation.

Now, we discuss the implications on program execution. In the datapath, prior to computation, we have a decoding phase, where the bitmask-encoding is used to decode the sparse matrix. Then, the uncompressed matrices are passed to the arithmetic units for computation. Once computation is finished, the output is then encoded back into the same data and mask split, which can then be written in main memory. These phases can be seen in the system diagram shown in the next section.

Furthermore, the VMACs are blocked from usage when one of the input vectors is entirely null to further improve on latency and energy consumption. In particular, if one of the input vectors is entirely null, then the output is 0, regardless of the other input. Therefore, we can stop the VMAC from computing the dot-product. This allows the computation to be completed in fewer clock cycles. Furthermore, less energy is consumed since the transistors in the VMACs do not need to change state.

Movement Pruning [SWR20]

In order to take advantages of EdgeBERT's sparse representation and computation capabilities, we need to find a suitable pruning method for the weights that will be used in the model. As we have discussed in Chapter 6, magnitude-based pruning methods are a natural and intuitive way to prune weights. However, magnitude-based pruning methods are typically not as good in transfer learning settings. As we discussed in Chapter 4, for downstream language tasks, we typically take a pre-trained language model and then append some classification layers on top of the model. Then, we "fine-tune" the model by training it for a shorter time on the downstream task. In this work, our downstream task will focus on sentiment analysis. Magnitude-based pruning methods typically fail in the transfer learning setting because the weights are usually pre-determined by the pre-trained model. While the fine-tuning does change the weights slightly, weights tend to stay close to their initial values, since the training period is typically quite short. Therefore, the "saliency" of the score is dictated more by the initial pre-trained values, which may not be an accurate representation of the weights importance in downstream tasks. Therefore, we are generally more interested in how the weights are changing during fine-tuning. If the weight is moving closer to

a magnitude of 0, this indicates that the weight is less important, since the fine-tuning is pushing its importance down. On the other hand, if the weight is increasing in magnitude, then it is likely that the weight has more importance in the downstream task. Thus, instead of using magnitude as an indication of saliency, [SWR20] presents movement pruning as an alternative for models that are derived from this transfer learning setting.

Now, we formalize these details mathematically. Recall, that the pruned weights are $\mathbf{w} \odot \mathbf{m}$, for weights $\mathbf{w}$ and mask $\mathbf{m} = \mathbf{T} \circ \mathbf{S}(\mathbf{w})$ under a naive sparsity scheme. The initial idea of [SWR20] is to also include the saliency as trainable parameters. Using this initial proposition, we derive a saliency score that has a nice interpretation. First, note that $\mathbf{T}$ is not differentiable everywhere due to its piecewise nature and has gradient 0 where it is differentiable. To get around this, we assume that the gradient of $\mathbf{T}$ is 1 everywhere. This is called the straight-through estimator in the literature, and it is commonly used for non-differentiable functions. Then, suppose that we have a layer with weights $\mathbf{W}$ and input $\mathbf{x}$. Then, assuming a matrix-multiplication operation, we have that the output is $\left(\mathbf{W} \odot \mathbf{T}\left(\tilde{\mathbf{S}}\right)\right)\mathbf{x}$, where $\tilde{\mathbf{S}}$ is of the same size of $\mathbf{W}$ and are the trainable saliency scores. Then, we have that

$$-\frac{\partial \mathcal{L}}{\partial \mathbf{S}_{i,j}} = -\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{i,j}}\mathbf{W}_{i,j},$$

assuming $\mathbf{W}_{i,j}$ is not masked. Recall that $-\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{i,j}}$ has the same sign as the update amount of $\mathbf{W}_{i,j}$. When both $-\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{i,j}}$ and $\mathbf{W}_{i,j}$ share the same sign, the weight is either becoming more positive or more negative. Thus, the magnitude is increasing and the saliency should increase. However, if they have opposite signs, then the weight is being pushed towards 0. In that case, the saliency should decrease. This is nicely reflected in the above equation. Therefore, after T updates and assuming an initial saliency of 0 and learning rate η, we have that

$$\mathbf{S}_{i,j}^{(T)} = -\eta \sum_{t=1}^{T} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{i,j}}\right)^{(t)} \mathbf{W}_{i,j}^{(t)}.$$

Thus, we can see that the saliency score is simply the sum of the movement of the associated weight over the number of update epochs. Using this as a metric for saliency as opposed to magnitude-based methods, [SWR20] shows an improvement on downstream language tasks.

[SWR20] refers to this pruning method as "hard" movement pruning. [SWR20] also proposes a "soft" variant of the method. Furthermore, [SWR20] shows that movement pruning is intimately connected with the approximate method for minimizing the L_0 regularization loss presented in [LWK18] and discussed briefly in Chapter 6. Finally, [SWR20] proves the convergence behavior of this method during training. For all of these details, we refer the reader to the original paper.

As discussed in Chapter 6, it should be noted that all of this pruning can be done offline prior to loading the weights into EdgeBERT. Therefore, the computation and its associated costs are not a main concern for the pruning methodology.

9.1.4 AdaptivFloat Quantization

EdgeBERT employs floating-point quantization and computation for both its activation and weights to a bitwidth of 4 or 8. EdgeBERT uses a rounding-to-nearest AdaptivFloat quantization scheme, as discussed in Chapter 8. The use of floating point quantization, and specifically AdaptivFloat, is motivated by the broad and non-uniform distribution of weights in modern LLMs, which EdgeBERT is targeting.

9.1.5 Hardware Implications

As promised in Chapter 8, we now discuss the details about AdaptivFloat dot product computation more carefully, and compare the arithmetic unit in EdgeBERT to a more standard integer dot product unit. First, we consider a standard integer dot product unit with a scaling factor. Before we do so, we make the following observation. Suppose that we have a unsigned n-bit number a and a signed m-bit number b. Then, we claim that ab can be represented in $n+m$ bits or less. The proof is straightforward. We have $-2^{n-1} \leq a \leq 2^{n-1}-1$ and $-2^{m-1} \leq b \leq 2^{m-1}-1$. Thus, we have that $-2^{n+m-1} \leq ab < 2^{n+m-1}-1$. This is also true for the signed case by a very similar argument. We have $0 \leq a \leq 2^n - 1$ and $0 \leq b \leq 2^m - 1$. Thus, we have that $0 \leq ab < 2^{n+m-1} - 1$.

The setup is as follows. Suppose we have a symmetric quantization scheme. Then, let $\mathbf{w}, \mathbf{a} \in \{-2^{n-1}, \ldots, 2^{n-1}-1\}^d$. In particular, they contain n-bit integers. Then, we have that $\mathbf{w}_i \times \mathbf{a}_i$ can be represented as a $2n$-bit integer. Then, we have that $\sum_{i=1}^d \mathbf{w}_i \times \mathbf{a}_i$ can be represented as a $2n + \lfloor \log_2(d) \rfloor + 1$-bit integer. This follows from the fact that $-|d| \cdot 2^{2n-1} \leq \sum_{i=1}^d \mathbf{w}_i \times \mathbf{a}_i \leq |d| \cdot (2^{2n-1}-1)$. Then, we dequantize the dot-product by the scaling factor c. Let c be a fixed-point number with S total bits and f fractional bits. Then, we have that $2^f c \cdot \left(\sum_{i=1}^d \mathbf{w}_i \times \mathbf{a}_i\right) \cdot 2^{-f}$. We have that $2^f c$ is an integer of S bits. Thus, we have that $2^f c \cdot \left(\sum_{i=1}^d \mathbf{w}_i \times \mathbf{a}_i\right)$ needs $2n + \lfloor \log_2(d) \rfloor + 1 + S$ bits. Finally, we can right shift by f bits to account for the 2^{-f}. Then, we can truncate back to n bits. This is the structure required for the computation of a dot product on integers. Note that we could extend this same structure to fixed point computation. Suppose we have two n-bit fixed point numbers $a.b$ and $c.d$ with d decimal bits. Then, we have that $a.b \times c.d = \left(ab \times 2^{-d}\right) \cdot \left(cd \times 2^{-d}\right) = (ab \times cd) \cdot 2^{-2d}$. Thus, we can just use integer multiplication and then apply a shift of $2d$ bits to the right. This is all illustrated in the diagram below.

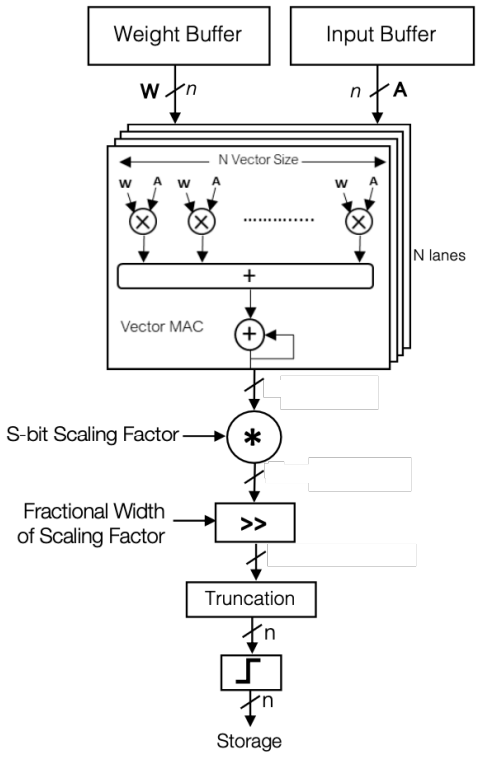

Figure 9.2: Integer multiplication in hardware. [TYW+20]

Now, we discuss the implementation used in EdgeBERT for AdaptivFloat computation. First, recall that AdaptivFloat omits denormals. In particular, all mantissa are of the form $1.M$, where M is a string of bits with length b_m. Thus, suppose that we have two multiply two AdaptivFloat$\langle n_e, n_m \rangle$ numbers $(-1)^{s_a} \times 1.m_a \times 2^{e_a}$ and $(-1)^{s_b} \times 1.m_b \times 2^{e_b}$, with exponent bias b_a and b_b respectively. Then, we have the following.

$$\begin{aligned}
&\left((-1)^{s_a} \times 1.m_a \times 2^{e_a+b_a}\right) \cdot \left((-1)^{s_b} \times 1.m_b \times 2^{e_b+b_b}\right) \\
&\quad = (-1)^{s_a+s_b} \times \left(\left(1m_a \times 2^{-n_m}\right) \cdot \left(1m_b \times 2^{-n_m}\right)\right) \times 2^{e_a+e_b+b_a+b_b} \\
&\quad = (-1)^{s_a+s_b} \times \left(\left(\left(1m_a \times 2^{-n_m}\right) \cdot \left(1m_b \times 2^{-n_m}\right)\right) \times 2^{e_a+e_b}\right) \cdot 2^{b_a+b_b} \\
&\quad = (-1)^{s_a+s_b} \times \left((1m_a \cdot 1m_b) \times 2^{e_a+e_b-2n_m}\right) \cdot 2^{b_a+b_b}
\end{aligned}$$

Therefore, we can implement multiplication with integer addition and multiplication. We have that $1 \leq 1m_a \times 2^{-n_m}, 1m_b \times 2^{-n_m} \leq 2$. Thus, we have that

$$1 \leq \left(1m_a \times 2^{-n_m}\right) \cdot \left(1m_b \times 2^{-n_m}\right) \leq 4.$$

Therefore, we have that $(1m_a \times 2^{-n_m}) \cdot (1m_b \times 2^{-n_m})$ needs at most $2n_m + 2$ bits. Further-

more, we can set $b_a + b_b$ to be the new shared exponent bias. Then, we have that

$$\sum_{i=1}^{n} \mathbf{w}_i \times \mathbf{a}_i$$

can be represented as a $2n_m + \lfloor \log_2(d) \rfloor + 2$-bit fixed-point value. We can then truncate the value to n bits. Then, we can convert it back to AdaptivFloat using the new activation exponent bias. Again, we can see this pictorially with the diagram below.

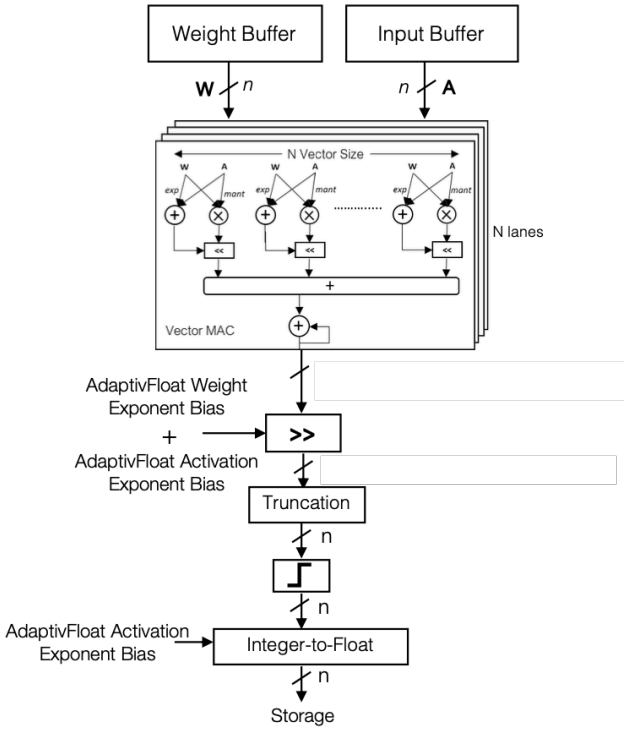

Figure 9.3: Adaptive float multiplication in hardware. [TYW+20]

9.2 EdgeBERT Implementation

First, we present the systems diagram for EdgeBERT to guide our discussion on the implementation of EdgeBERT. EdgeBERT is implemented in SystemC with the help of open-source libraries like MatchLib and HLSLibs. The SystemC code is designed as follows. We create modules for each of the parts of the system. Each module contains a `Run()` function, as well as input and output channels defined using the NVHLS Connections library. We use `SC_THREAD` to create a thread-like process for `Run()` that can be yielded via `wait()`. Furthermore, we can use `sensitive << clk.pos();` to schedule the thread again on the positive clock edge.

The structure of most modules will be as follows. We will have a `while(1)` loop. At the end of each loop, there will be a `wait()` statement. Within the loop, the thread will typically make a `PopNB` or `Pop` call to retrieve new signals. `PopNB` is a non-blocking pop operation provided by the NVHLS Connections library. It allows EdgeBERT to read signals from a configuration in a non-blocking fashion. It is non-blocking, since if the data is not yet in the input channel. On the other hand, `Pop` is a blocking call. It will wait for the data to arrive in the input channel before proceeding. `Push` is the counterpart of `Pop`, and it pushes data to an output channel. The movement of data between channels is typically orchestrated by some higher-level module. Then, after gathering the appropriate inputs, the module performs the appropriate computation.

Now, we discuss the main components of the system diagram. Above, we have already discuss the main hardware optimizations. Thus, in this overview, we simply show the hierarchy of the different modules implemented in SystemC.

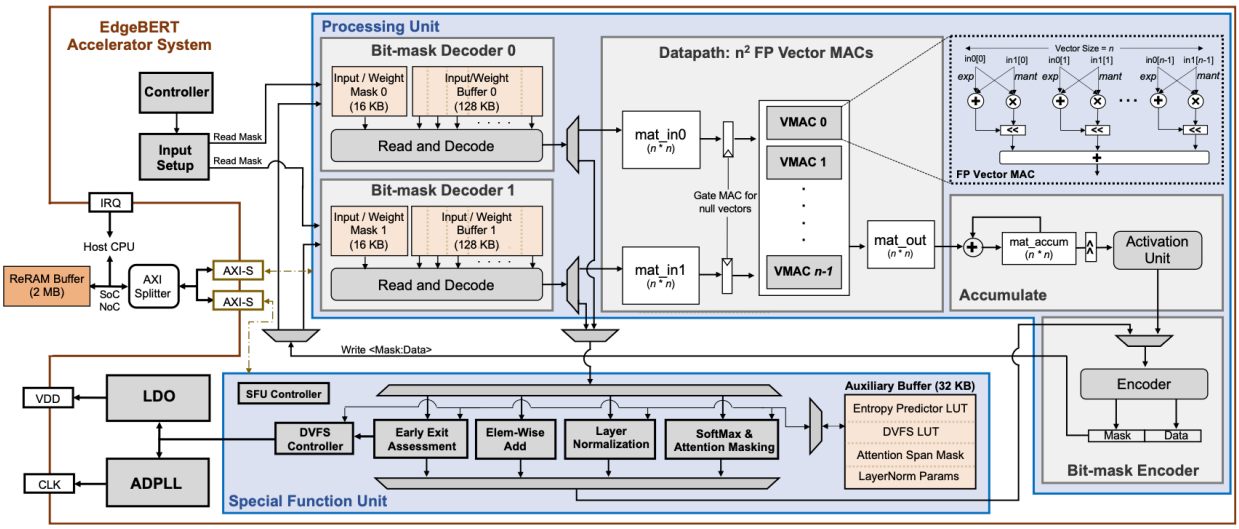

Figure 9.4: EdgeBERT system diagram. [THP+21]

9.2.1 TopAccel

`TopAccel` is the top-level SystemC module that integrates all of the underlying modules and coordinates data movement. It has an `interrupt` output port that notifies the `Control` module when a request is complete or an error has occurred. It also has external output ports for the LDO, `edgebert_ldo_res_sel`, ADPLL, `edgebert_dco_cc_sel`, and interrupt requests, `interrupt`, which are all connected to the control unit. It also defines all the AXI interfaces (`if`). `if_axi_rd` and `if_axi_wr` are configuration ports that are used to interface with the external controller. `if_data_rd` and `if_data_wr` are the shared memory

interfaces that the AXI arbiter uses to communicate with external memory. Finally, there are module-specific ports for the mask, input, and auxiliary data transactions. These are all funneled to the shared memory interface via the AXI arbiter. We will discuss all of this more in depth shortly. `Control` also defines all of the signals that are used to connect the various submodules. Finally, the `Control` module binds all of the signals to the associated submodule.

9.2.2 Control

`Control` is the control unit of the accelerator that orchestrates the various operations on the accelerator. `Control` features two threads: `ControlRun` and `InterruptRun`.

`ControlRun` checks if there is an incoming AXI read or write transaction. `Control` uses a simplified AXI protocol that uses only the ready and valid states. Then, it either writes the incoming value to `config_regs`, an array of 32 32-bit wide registers, or returns the value through an output channel. Then, it handles signals coming from other modules. For example, it checks whether `dco_sel_out` or `ldo_res_sel` have been written to by the DVFS module before updating `edgebert_dco_sel_out` and `edgebert_ldo_res_sel`. Then, it processes the updates made to `config_regs`. Finally, it processes the "command register", which triggers the execution of the command. For reads and writes, it pushes a trigger structure to an AXI module that will trigger an AXI transaction. We will discuss the details about the AXI modules soon. For starting the `PUModule`, it pushes a configuration to the output `mat_config`. For all other computations, a value is pushed to `start`, which is connected to the `GBModule`.

`InterruptRun` checks if interrupt have been raised from any of the modules that indicate the end of a read/write or computation. If an interrupt is received, it asserts an interrupt, which will raise an interrupt through the interrupt controller of the SoC.

9.2.3 `DecodeTop`

The `PUModule` defines two `DecodeTop` modules, `decode_inst0` and `decode_inst1`, for the two bitmask decoders. `DecodeTop` contains `Decode` and `DecMem` modules. At a high level, `Decode` issues memory requests that are handled by `DecMem` through accessing the scratchpad.

`Decode` instantiates the following submodules, which are all variations on a similar structure. It takes in `base_input`, `base_offset`, and `mat_config`. Then, it decodes a bitmask encoded tensor in `RunInputAddrGen`. In particular, for a bank n, it maps `req_reg.addr[i]` to a valid address if `in_mask_reg.data[0][i] == 1`. Otherwise, `req_reg.valids[i] = 0`.

It does this for each bank. Then, it outputs `req_reg` for each bank. Below, we detail each of the variations.

- `Decode_N0` and `Decode_N1`: These are used for loading in matrices for matrix multiplication. Suppose we are multiplying $\mathbf{A} \in \mathbb{R}^{N_0 \times M}$ and $\mathbf{B} \in \mathbb{R}^{M \times N_1}$. Then, we will use `Decode_N0` to decode $\mathbf{A}$ and `Decode_N1` to decode $\mathbf{B}$.
- `Decode_LayerNorm`: This module performs the decoding for a single matrix for the layer normalization operation.
- `Decode_SMax`: This module performs the decoding for the softmax operation. It executes three response requests for each element, since it will be consumed three times in the FSM.
- `Decode_Enpy`: This module performs the decoding for the entropy of a single output vector of length `num_vector`.
- `Decode_Eadd`: This module performs the decoding for an elementwise-addition operation between two matrices of dimension `num_vector` by `num_timestep`. Unlike `Decode_N0` and `Decode_N1`, since the dimensions of the two matrices are the same, `Decode_Eadd` loads in both matrices via the same module.

Each of the submodules outputs `out_req`. These requests are passed to `DecMem` via `Decode`. `DecMem` has two threads `MergeRspRun_Input` and `MergeRspRun_Mask`, which will use an instantiation of `DecMemCore` to return responses to the input and mask requests from `Decode`.

9.2.4 `PUModule`

The processing unit module (`PUModule`) is the heart of the matrix multiplication part of the EdgeBERT system. As mentioned above, `PUModule` instantiates two `DecodeTop` modules. It also instantiates a `Encode` module for encoding after computation. It instantiates a `InputSetup`, `Datapath_Top`, and `Accum` module for computation. The `PUModule` features the following eight threads each of them triggers some computation by writing to the configurations that are connected to each of the modules. They can generally be characterized as propagating the configurations to the submodules below. It also instantiates the one-to-two multiplexers used in the system diagram depending on the configuration settings.

9.2.5 `GBModule`

The `GBModule` module handles the implementation of the special function unit shown in the system diagram. It instantiates a `AuxMem` module for an auxiliary buffer, as well as

the following computational units: `LayerNorm`, `SMax`, `ElemAdd`, `Enpy`, and `Dvfs`. `AuxMem` is very similar in structure to `DecMem`. It instantiates a `AuxMemCore` module that initializes the scratchpad used for the special function unit. The computational modules follow a similar implementation strategy to that outlined in Chapter 4. The main function of `GBModule` is very similar to that of `PUModule`, where it instantiates submodules and propagates information between the submodules.

9.2.6 AXI

We have already briefly described the Advanced eXtensible Interface (AXI) protocol used in EdgeBERT. In particular, we have an AXI arbiter that connects the accelerator to SoC memory via the NoC. Then, the arbiter splits the responses between the `MaskAxi`, `InputAxi`, and `AuxAxi` modules. `MaskAxi` is for obtaining the mask values that are used for the bitmask encoding. `InputAxi` is for obtaining the input values that are used in the computations. `AuxAxi` is for obtaining data for the auxiliary buffer. This can include the attention span mask, lookup tables, and the layer normalization parameters. Each of the modules features three threads.

- `MasterRead`: This triggers a read operation from external memory. The reading is done in bursts with up to 256 words per burst. The output is then issued as a write request to the decoder.
- `MasterWriteReq`: This process generates the read requests from the decoder that are necessary to write to external memory.
- `MasterWrite`: This triggers the write operation to external memory. The write is done in bursts, like `MasterRead`, and takes input from the `DecodeTop` module.

9.3 Hardware Design Considerations

9.3.1 eNVM

[THP+21] proposes a final memory-related optimization. However, we will not use this optimization in our profiling work, so we opt for a briefer discussion. Recall from Chapter 4 that embeddings are a necessary component of many language models. In particular, each token maps to some embedding $\mathbf{z} \in \mathbb{R}^{d_{\text{emb}}}$. These embeddings typically stay constant regardless of the choice of downstream tasks. Therefore, EdgeBERT proposes that these embeddings be stored in embedded non-volatile memory (eNVM), so that they need not be reloaded given the absence of power. In particular, EdgeBERT proposes that the embeddings

be stored in a similar bit-mask encoded fashion where the mask is stored in a single-level resistive RAM (ReRAM) cell, while the values are stored in multi-level cells. This distinction is due to the fact that multi-level cells have reliability concerns. Therefore, a GoldenEye-like functional simulator was used to profile the effect of single bitflip perturbations on end-to-end accuracy.

9.3.2 Simulations

Now, we discuss some of the design considerations for the implementation of EdgeBERT. In particular, we focus on the metrics discussed in Chapter 2. Per [THP+21], the SystemC code for EdgeBERT is transformed into Verilog RTL using the Catapult HLS tool, per Chapter 2. The memories for the scratchpad for the bit-mask decoders and the auxiliary buffers were chosen from foundry-provided options. Finally, the initial simulations results were collected via Catapult. Below, we include a die photo of EdgeBERT.

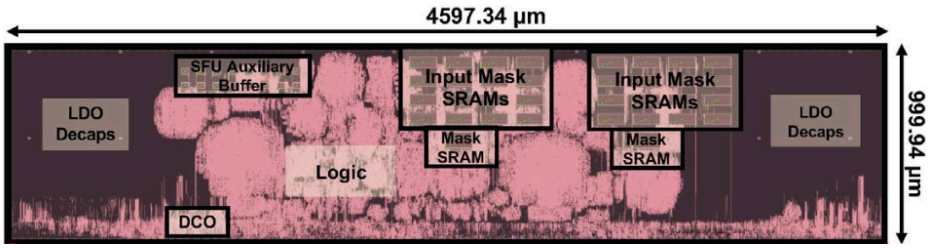

Figure 9.5: EdgeBERT die photo. [TZH+23]

9.4 EPOCHS SoC [DSJZ+24]

EdgeBERT was integrated into the EPOCHS SoC [DSJZ+24]. While this is not the main focus of this chapter, we include an overview of the EPOCHS SoC for context. The EPOCHS SoC is a heterogeneous SoC that features a total of 14 accelerators in addition to four Ariane RISC-V CPUs. Below, we include an annotated die shot of EPOCHS.

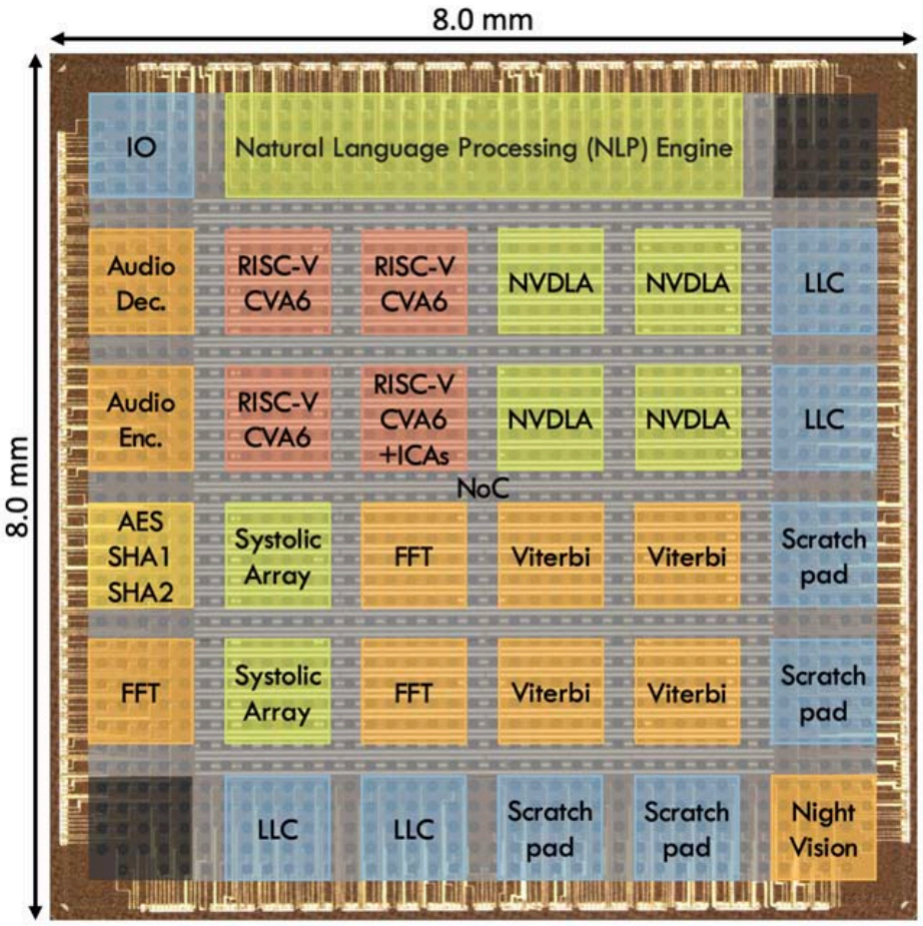

Figure 9.6: EPOCHS die photo. [DSJZ+24]

Now, we will discuss the pertinent parts of the EPOCHS SoC with relation to EdgeBERT. First, the "Natural Language Processing (NLP) Engine" is the tile that contains the EdgeBERT accelerator presented above. We will use a single Ariane RISC-V core to profile baseline results, as well as drive the execution of the workload. We will also use a single systolic array tile to provide an additional baseline. The IO tile contains Universal Asynchronous Receiver/Transmitter (UART) and Ethernet interfaces. This allows us to ssh directly into the device, load in the binary, and read log files. Finally, not pictured in the above image, the chip is connected to FPGA DRAM memory.

POCHS was built atop the ESP [MGDG+20] platform, an open-source platform for designing SoCs. The ESP platform is centered around the "tile" abstraction, where hardware developers can choose the mixture of tile types for each tile in the SoC grid. Below, we give a high-level overview of the different tile types supported in ESP. We will mainly focus on the accelerator tile type.

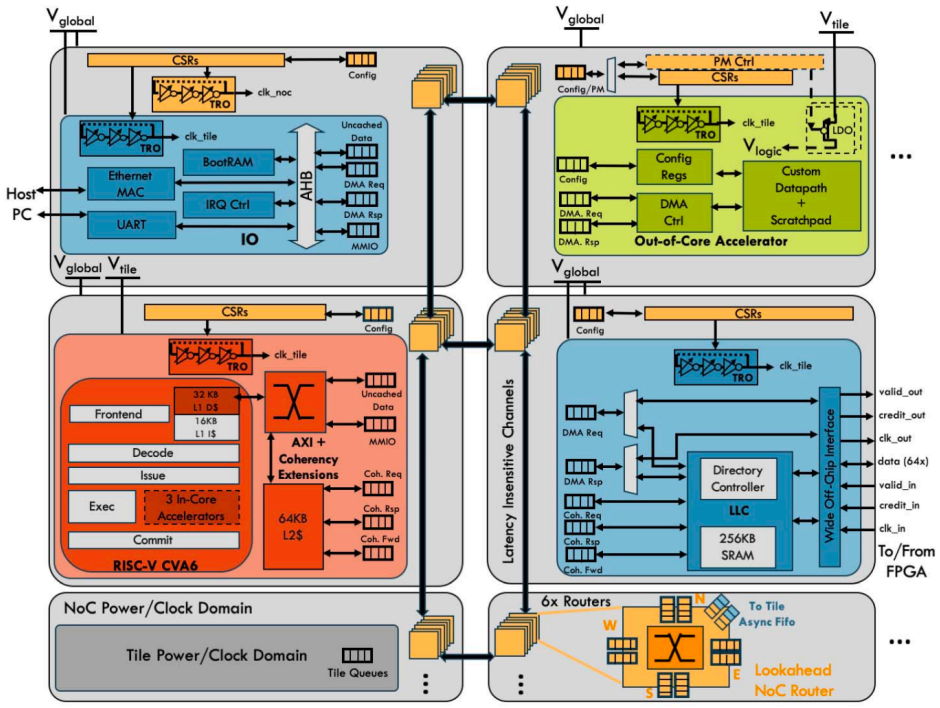

Figure 9.7: Tile types in ESP platform. [MGDG+20]

As discussed for EdgeBERT, each accelerator tile features its own direct memory access (DMA) controller, which uses the AXI protocol to interface with off-chip memory. The ESP platform supports a few options for the coherency model for the DMAs. In particular, accelerators can access off-chip memory directly in a non-coherent fashion, use the last-level cache (LLC), or maintain its own private local cache. ESP handles the initialization of all such coherency models. Furthermore, the ESP platform mediates all interrupt-related requests between the CPU and the accelerator through its multi-plane NoC. It allows the accelerator to issue interrupt requests to the Ariane RISC-V CPU and for the CPU to acknowledge and handle such requests. These interrupts are crucial for our profiling purposes, since they allow the accelerator to indicate that an operation is complete. Finally, ESP supports memory-mapped configuration registers for its accelerators, allowing the CPU to write directly to these registers. These write operations trigger the various operations on EdgeBERT and identify the necessary parameters for each operation. We leave a detailed discussion of EPOCHS and ESP to [DSJZ+24] and [MGDG+20] respectively.

9.5 Compiler

Once the hardware has been constructed, we would like to build a compiler to run high-level language models atop the accelerator. This is our main contribution to this work. In

this section, we describe the general compiler code structure. Then, we present the results by "compiling" an A Lite BERT (ALBERT) model to run on the accelerator in the next section. Our goal is to abstract out the commonly used primitives, like matrix multiplication, softmax, element-wise addition, and layer normalization, and optimize them individually. Then, we can combine these primitives to construct optimized modules for attention head computation, post-processing layers, and feed-forward networks. Our goal is to balance both flexibility and efficiency.

9.5.1 Primitives

First, we discuss the implementation of primitive components of most language models, as discussed in Chapter 4. These primitives are used in a variety of architectures, so providing support for these is critical. We mainly target implementations of these primitives for EdgeBERT. However, we also include implementations for systolic array and CPU to serve as a baseline for our results. EdgeBERT is capable of executing matrix multiplication, softmax, element-wise addition, and layer normalization operations. Systolic arrays are only capable of executing matrix multiplication operations. All other operations must be done on the CPU.

Below, we detail the implementation of each primitive in more detail. Note that our compiler attempts to be as general as possible, but there are some limitations. We will discuss these limitations below and present some solutions in later sections.

9.5.2 Helper Functions

First, we define some useful helper functions that write the appropriate EdgeBERT registers. In particular, we would like to abstract the read and writes to the mask, input, and auxiliary buffers. Therefore, for reads, we have `master_mask_read`, `master_input_read`, and `master_aux_read` to read from the external memory to the accelerator mask, input, and auxiliary buffers respectively. We have `master_mask_write`, `master_input_write`, and `master_aux_write` to write to the external memory from the accelerator mask, input, and auxiliary buffers respectively.

9.5.3 Core Functions

Furthermore, we also write helper functions to trigger primitive operations where all the inputs can fit into the scratchpad. `EdgeBert_mat_mul`, `EdgeBert_atten_softmax`, `EdgeBert_elem_add`, and `EdgeBert_layer_norm` define the operations for matrix multiplication, softmax, and layer normalization respectively. We should note that `EdgeBert_mat_mul`

and `EdgeBert_elem_add` have a `write` argument to indicate whether or not the output should be written out from the scratchpad buffer back to the external memory. This is so that we can use "kernel fusion" and apply the softmax or layer normalization operation immediately after. For `EdgeBert_atten_softmax` or `EdgeBert_layer_norm`, if the input pointer is null, then it assumes that the correct data is already stored on the scratchpad, and it immediately executes the operation without loading in a matrix.

9.5.4 General Function

Now, we discuss the general functions that use the above core functions and helper functions.

Matrix Multiplication

First, we discuss a general matrix multiplication strategy for multiplying $\mathbf{A} \in \mathbb{R}^{m \times n}$ by $\mathbf{B} \in \mathbb{R}^{n \times p}$. This is implemented in `general_mat_mul`. It is required that $M \mid m$ and $M \mid p$, where $M = 16$. We will take a tiling approach as described in Chapter 2. However, we will assume that only row-wise tiling occurs for $\mathbf{A}$ and only column-wise tiling occurs for $\mathbf{B}$. In particular, we assume that 16 rows of $\mathbf{A}$ fit in decoder 0, and 16 columns of $\mathbf{B}$ fit in decoder 1. While this is not the most general implementation, we find that this is generally true for most workloads. We assume this constraint, so that we do not need to utilize element wise addition operations. In particular, we have the following. Let

$$\mathbf{A} = \left[\begin{array}{c} \mathbf{A}^{(1)} \\ \hline \mathbf{A}^{(2)} \\ \hline \vdots \\ \hline \mathbf{A}^{\left(\frac{m}{M}\right)} \end{array}\right] \text{ and } \mathbf{B} = \left[\begin{array}{c|c|c|c} \mathbf{B}^{(1)} & \mathbf{B}^{(2)} & \cdots & \mathbf{B}^{\left(\frac{m}{M}\right)} \end{array}\right],$$

assuming that $\mathbf{A}^{(i)}$ is a collection of M rows and $\mathbf{B}^{(i)}$ is a collection of M columns. Then, we have that

$$\mathbf{AB} = \begin{bmatrix} \mathbf{A}^{(1)}\mathbf{B}^{(1)} & \mathbf{A}^{(1)}\mathbf{B}^{(2)} & \cdots & \mathbf{A}^{(1)}\mathbf{B}^{\left(\frac{m}{M}\right)} \\ \mathbf{A}^{(2)}\mathbf{B}^{(1)} & \mathbf{A}^{(2)}\mathbf{B}^{(2)} & \cdots & \mathbf{A}^{(2)}\mathbf{B}^{\left(\frac{m}{M}\right)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}^{\left(\frac{m}{M}\right)}\mathbf{B}^{(1)} & \mathbf{A}^{\left(\frac{m}{M}\right)}\mathbf{B}^{(2)} & \cdots & \mathbf{A}^{\left(\frac{m}{M}\right)}\mathbf{B}^{\left(\frac{m}{M}\right)} \end{bmatrix}.$$

Then, no element-wise addition is necessary. However, to execute such tiled matrix multiplication, we need to create a temporary buffer to store $\mathbf{A}^{(i)}$ and $\mathbf{B}^{(j)}$. Furthermore, we need to move the data in and out from that buffer. This is particularly challenging for columns $\mathbf{B}^{(j)}$,

since matrices are stored in row-major order. Thus, this constitutes most of the memory accesses, and a large portion of the latency as we will soon see.

9.5.5 Softmax

`general_softmax` performs the softmax operation on $\mathbf{A} \in \mathbb{R}^{m \times n}$. It assumes that an arbitrary row $\mathbf{a}_i \in \mathbb{R}^n$ can be loaded into the scratchpad. Again, this is an assumption that limits the expressivity of the compiler. However, it is a reasonable assumption that allows us to avoid more sophisticated algorithms like FlashAttention [DFE+22].

9.5.6 Element-wise Addition

`general_element_add` performs the element-wise addition operation on $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$. In particular, we have the following. Let

$$\mathbf{A} = \left[\begin{array}{c} \mathbf{A}^{(1)} \\ \hline \mathbf{A}^{(2)} \\ \hline \vdots \\ \hline \mathbf{A}^{\left(\frac{m}{M}\right)} \end{array}\right] \text{ and } \mathbf{B} = \left[\begin{array}{c} \mathbf{B}^{(1)} \\ \hline \mathbf{B}^{(2)} \\ \hline \vdots \\ \hline \mathbf{B}^{\left(\frac{m}{M}\right)} \end{array}\right],$$

assuming that $\mathbf{A}^{(i)}$ is a collection of M rows and $\mathbf{B}^{(i)}$ is a collection of M rows. Then, we have that

$$\mathbf{A} + \mathbf{B} = \left[\begin{array}{c} \mathbf{A}^{(1)} + \mathbf{B}^{(1)} \\ \hline \mathbf{A}^{(2)} + \mathbf{B}^{(2)} \\ \hline \vdots \\ \hline \mathbf{A}^{\left(\frac{m}{M}\right)} + \mathbf{B}^{\left(\frac{m}{M}\right)} \end{array}\right].$$

9.5.7 Layer Normalization

Finally, `general_layer_norm` implements the layer normalization operation as discussed in Chapter 4. Like the preceding operations, it is necessary that an entire row fits in the scratchpad, else we cannot compute the sample mean and variance of the row. The implementation of `general_layer_norm` follows that of `general_softmax` very closely, since the operations are both performed row-wise.

9.5.8 Modules

Attention Head

As discussed in Chapter 4, attention heads are a very common paradigm in current language models. Thus, providing compiler support for attention heads is well-motivated. Here, we focus on the computation needed for a single attention head, but we can easily support multi-headed attention by using multiple attention heads and the post-processing, which we will discuss in the next section. `CPU_EdgeBert_attention` implements the attention head for CPU, `sysarray_attention` implements the attention head for systolic array, and `EdgeBert_attention` implements the attention head for EdgeBERT. They use the corresponding primitives and default to the CPU implementation if the operation is not supported. For softmax and layer normalization operations, we simply omit these from the CPU and systolic array. Thus, our results are a conservative underestimate of the true speedup, since EdgeBERT does all of the components of computation, while the two baselines do some subset of it.

Post-Processing

Post-processing refers to the linear layer, the residual connection, and the layer normalization operation that typically follows multi-headed attention modules. Again, we saw this frequently used in Chapter 4. `CPU_EdgeBert_processing` implements this for the CPU, `sysarray_processing` for the systolic array, and `EdgeBert_processing` for EdgeBERT.

Feed-Forward Networks (FFN)

Feed-forward network is a collection of two linear layers, a residual skip connection, and a layer normalization operation. This feed-forward network typically operates on the attention output. Again, we use the primitives defined earlier. `CPU_EdgeBert_feed_forward` implements this for the CPU, `sysarray_feed_forward` for the systolic array, and finally `EdgeBert_feed_forward` for EdgeBERT.

9.6 Results

In this section, we present our profiling results collected by running workloads on true silicon. We define a static inline function `get_counter` that returns a `uint64_t` that designates the number of clock cycles since reset. The `mcycle` register, a control and status register (CSR) provided in the RISC-V architecture, is used to retrieve the value. The function

uses embedded assembly to load the value of `mcycle` into a temporary register `t0` before returning it from the function.

9.6.1 Profiling Matrix Multiplication

While writing the compiler code, we discovered a hardware bug at the SoC-level that limited DMAs to matrices of size $16 \times n$, with $n \leq 112$ for EdgeBERT. Although this does not affect computation, it limits the amount of data that could be read from scratchpad memory or main memory. Thus, we needed to project the "read" and "write" measurements of the end-to-end latency results.

In order for the compiler code to run successfully, we fix M, the number of words read and written, to 16. Then, we apply the following modification. Suppose we are loading in a matrix with dimensions $m \times n$. Then, we have that the scaling factor is

$$\frac{\frac{mn}{8} + \frac{mn}{16}}{2M},$$

since $\frac{mn}{8} + \frac{mn}{16}$ is the true number of words that should be loaded, and $2M$ is the number of words that are actually loaded (M for the mask and M for the data). We want to apply this scaling factor to the "read" and "write" times.

Below, we verify that there is a linear relationship between the read and write clock cycles as the dimension increases. We perform a matrix multiplication between a matrix of dimensions $16 \times M$ and a matrix of dimensions $M \times 16$, where $M \in \{16, 32, 48\}$. The linear relationship shows that scaling the read and write times linearly is a reasonable approximation.

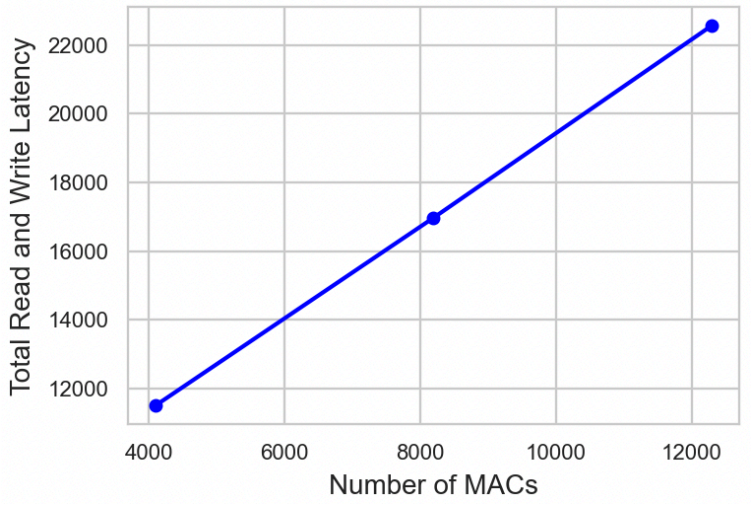

Figure 9.8: Scaling of read and write clock cycles on number of MACs.

9.6.2 Profiling End-to-End Workloads

To profile EdgeBERT's end-to-end performance, we will use the ALBERT (A Lite BERT) [LCG+20] model to measure latency. We will compare EdgeBERT's latency to that of a Ariane RISC-V CPU core and a systolic array. ALBERT is based on the BERT model discussed in Chapter 4. ALBERT is a modification of the BERT architecture that shares weights between each of the layers. This greatly reduces the memory footprint and makes it friendly for memory and energy constrained environments, as illustrated in the diagram below.

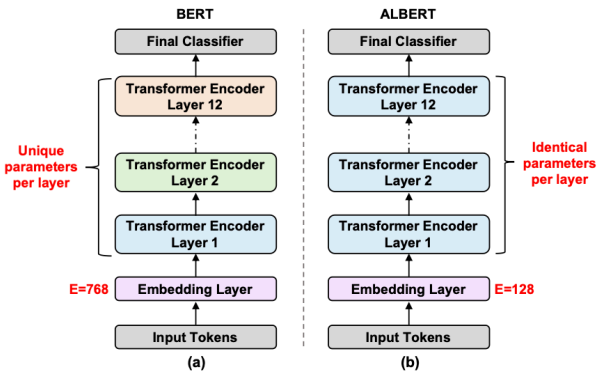

Figure 9.9: BERT and ALBERT comparison. [THP+21]

Below, we show the full computational diagram of an ALBERT workload. We will use the modules we defined earlier to execute each of the components.

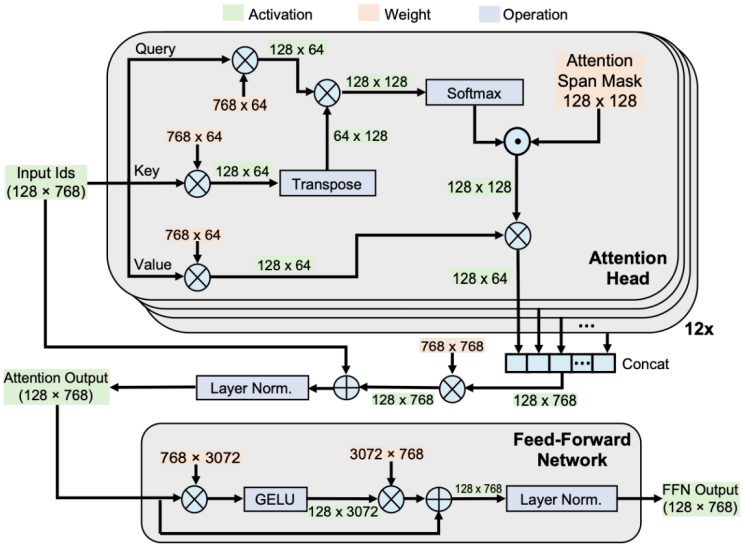

Figure 9.10: ALBERT computational diagram. [THP+21]

End-to-End Performance

Now, we present the end-to-end performance metrics for ALBERT on EdgeBERT. First, we show a high-level comparison between total clock cycles on CPU, systolic array, and accelerator times. Compared to a baseline CPU implementation, we see a 64.1× speedup for the accelerator. Furthermore, we see a roughly 2× speedup from the systolic array and the accelerator.

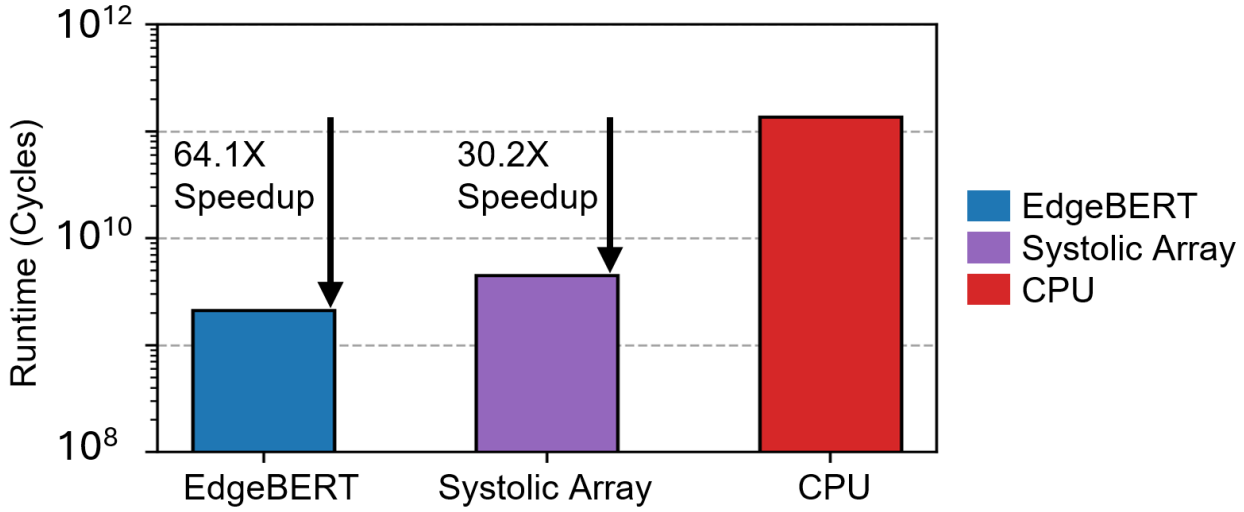

Figure 9.11: End-to-end comparison between EdgeBERT, systolic array, and CPU.

Compared to the CPU, it is expected that the accelerator outperform it substantially. In particular, the accelerator has optimized hardware designed to accelerate and parallelize key operations in the language model workload. Furthermore, EdgeBERT beats out the systolic array because it supports matrix multiplications between larger matrices. Therefore, EdgeBERT avoids unnecessary element-wise operations during tiling. On the other hand, the systolic array needs to use the CPU to perform the element-wise addition.

Now, we provide a more granular breakdown of the measurements to the three modules as described in the computational diagram. We see that the FFN is typically the main bottleneck due to the matrix multiplications in the FFN containing the most MACs. In particular, there are two large matrix multiplications, which require more memory accesses and computation time. Here, the "Attention" column includes the runtime for all twelve attention heads. Although the attention head has the most operations, it should be noted that it has the smallest runtime. This is likely due to the fact that it has the smallest number of MACs and data movement due to the smaller sizes of the matrices.

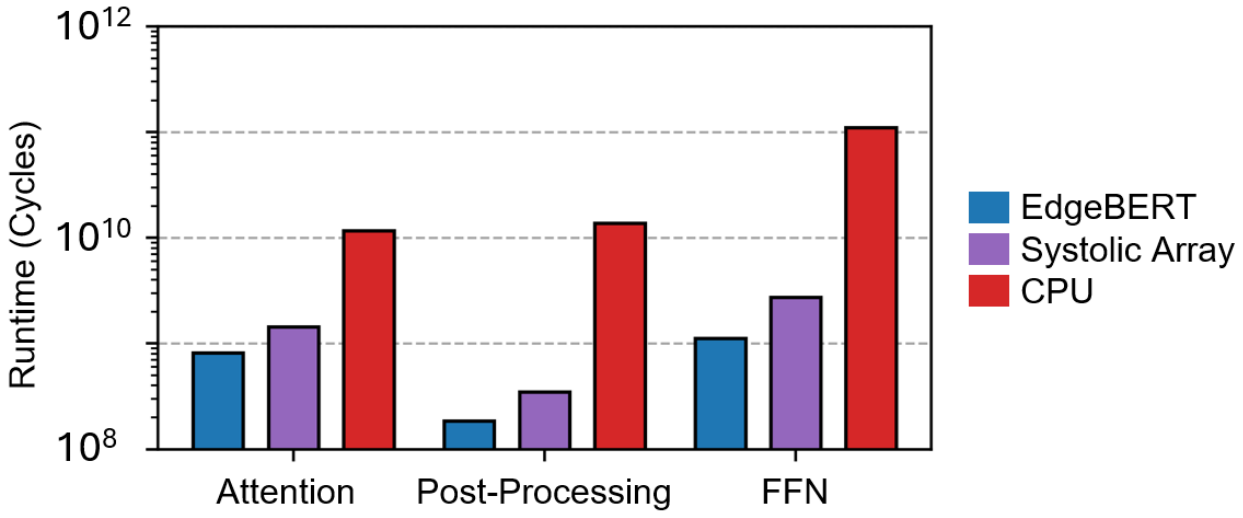

Figure 9.12: Granular comparison between EdgeBERT, systolic array, and CPU.

Finally, we show a fine-grained breakdown for each of the components of each operation for the systolic array and EdgeBERT. We show this to illuminate the speedup that EdgeBERT provides over the systolic array. "Register" denotes the time to write for the CPU to write to the systolic array or EdgeBERT registers prior to executing different operations. "Read" and "Write" refer to times required to read/write data on and off the accelerators. "Compute" measures the time for computation on the accelerators. "Malloc" represents the time required to allocate the intermediate matrices in main memory. "Memset" refers to data movement done by the CPU in main memory for efficient accelerator reading and writing. Finally, "Transpose" refers to the time taken in the attention head for transposing the matrix. Note that this is done on the CPU, since the accelerators do not have a transpose operation. First, we present the fine grained breakdown for the ALBERT workload on a systolic array.

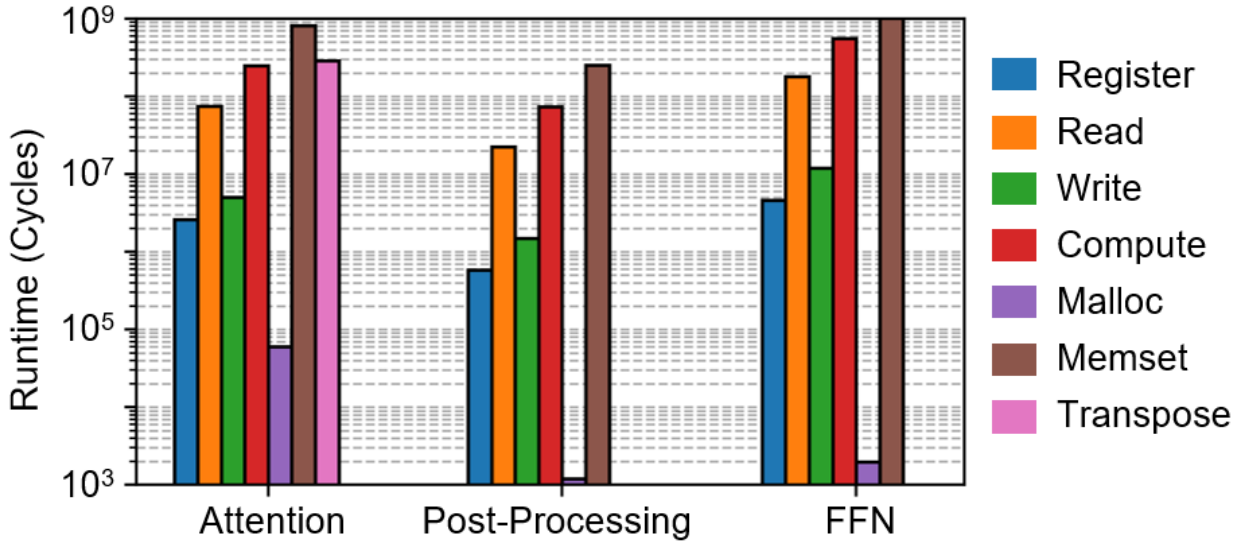

Figure 9.13: Fine grained breakdown on systolic array.

Now, we present the fine grained breakdown for the ALBERT workload on EdgeBERT.

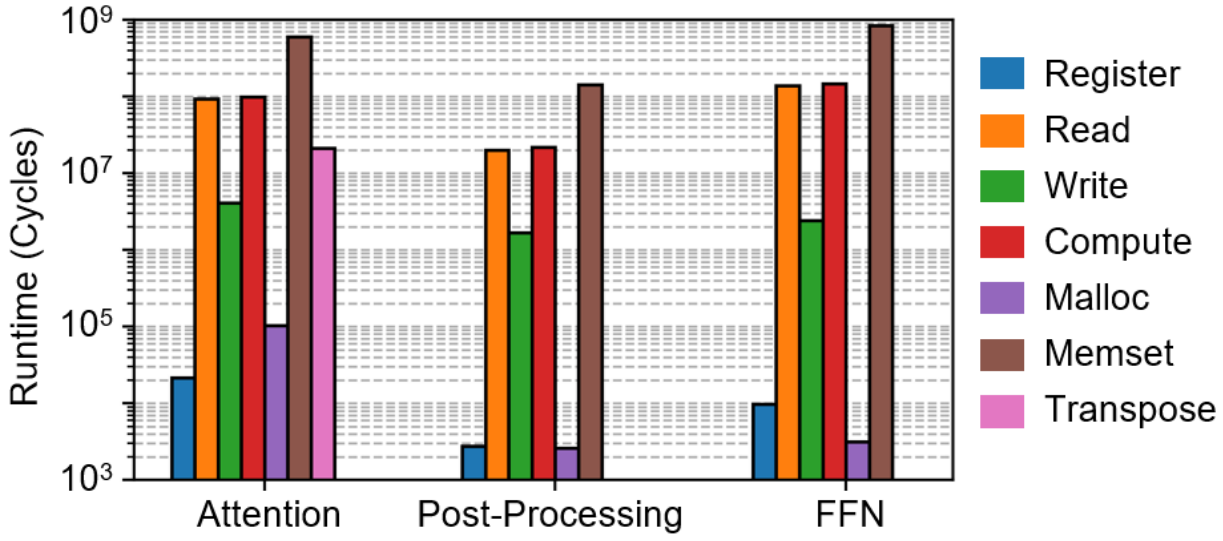

Figure 9.14: Fine grained breakdown on EdgeBERT.

Now, we make the following key observations. First, we see that data movements cost tend to dominate the total number of clock cycles. However, these costs are necessary, since we need the data to be stored in the correct format before loading it onto the accelerators. Since the systolic array and EdgeBERT typically cannot read an entire matrix at once, matrix multiplications need to be done in a piecemeal fashion. Thus, we need to only copy

specific rows and columns into the accelerators, which requires creating temporary matrices and copying over the values. Next, we observe that compute costs tend to be smaller for EdgeBERT, as opposed to the systolic array. This can likely be attributed to the fact that the maximum matrix size supported by the systolic array is quite small. Therefore, the tiling structure requires repeated element-wise addition, which needs to be done on CPU. Finally, we note that the number of writes to registers are abnormally high for the systolic array. This is again because the matrix multiplication is limited to two arrays of size 32×32. Therefore, there are orders of magnitude more calls to the systolic array, as opposed to EdgeBERT. Together, this explains the $2\times$ speedup that we see for EdgeBERT compared to the systolic array implementation.

9.7 Future Work

First, we may want to extend the primitives presented earlier to more general settings. In particular, we want to consider the cases where entire rows may not fit on the accelerator. In such cases, we need to use the CPU to help work around this limitation, such as the softmax optimization presented in FlashAttention [DFE+22]. Such extensions would allow us to support a wider array of models, especially those that have larger memory footprints.

Second, we should note that the compiler we have built is fairly general as is though, and future work can extend to many other language model architectures to better profile EdgeBERT's performance. In particular, we may be interested in running a decoder-only model, as opposed to an encoder-only model like ALBERT. The upshot of a decoder-only model is that we can run it over many timesteps, generating a token at each timestep. Then, we can also take advantage of optimizations like KV caching, as presented in Chapter 4. Profiling the accelerator for such workloads will show its versatility and better illuminate its impact.

Finally, our main focus in this work was on profiling end-to-end performance. We were interested in understanding the baseline speedup of EdgeBERT on true silicon. Thus, we omitted profiling the EE and DVFS optimizations featured in EdgeBERT. However, these are key aspects of EdgeBERT's design that should also be profiled extensively. In future work, we hope to take latency and power measurements with EE and DVFS optimizations being fully applied.

10

Sequential Quantization and Sparsity

Up until now, we have discussed two methods for model compression: quantization (Chapter 5) and sparsity (Chapter 6). In the literature, these two methods are generally treated orthogonally. In particular, there is an assumption that quantization and sparsity methods can be applied sequentially to get even better model compression. In this section, we will try to better understand the interaction of quantization and sparsity when they are both applied to machine learning models. Note that we will focus on the setting where no retraining is done. This allows us to make stronger mathematical claims about the error behavior.

First, we try to understand how these problems are related, and if insights from one can be used for the other. Then, we answer the question whether the order of operations is important or if these operations are commutative. Here, there has been excellent preliminary work in [HCK+25]. However, in this section, we try to extend their analysis. We will extend their analysis in the following ways. First, [HCK+25] makes assumption about the quantization and sparsity strategies. We try to relax some of these assumptions. Second, [HCK+25] characterizes the error mathematically at the tensor and dot-product level. However, in this section, we extend this analysis to the model level. Finally, we propose a quantization-aware sparsity method that improves on current model compression strategies based on our results and those from [HCK+25].

10.1 Naive Max-Scaled Block-Wise Quantization

[HCK+25] focuses their mathematical on max-scaled block-wise quantization. Thus, we spend this section rigorously defining this idea. Recall from Chapter 5 that a quantization scheme is defined as $\{Q_i, D_i\}_{i=1}^d$ for weights $\mathbf{w} \in \mathbb{R}^d$. Now, suppose we divide the weights into blocks (groups) $\{\mathbf{w}_{(1)}, \ldots, \mathbf{w}_{(B)}\}$, defined by a partition $\{P_1, \ldots, P_B\}$ of $\{1, \ldots, d\}$. Typically, blocks are collections of rows. Then, we will enforce a per-block granularity. That is, for all $Q_i = Q_j$ and $D_i = D_j$ for all $i, j \in \mathcal{B}_k$ for $k \in \{1, \ldots, B\}$. This addresses the "block-wise" part of the name. Then, the "max-scaled" part refers to the fact that the quantization grid is scaled by the maximum magnitude for an element in a block. In particular, the quantization grid $\mathcal{Q}_b$ is a function of the maximum magnitude. For convenience, we will define

$$s_b = \max_{i \in P_b} |\mathbf{w}_i|$$

for a block b. We will discuss a few concrete examples below.

We will define a general scheme before giving specific examples. Consider a block $\mathbf{w}_{(b)}$. Then, consider the scale s_b. Then, we will define

$$Q_i(\mathbf{w}) \triangleq \left\lfloor \frac{\mathbf{w}_i}{s} \right\rceil,$$

where s is some function of s_b. We also define

$$D_i(j) \triangleq s \cdot j,$$

where again s is some function of s_b. Although not explicitly stated in [HCK+25], an implicit assumption is that the quantization scheme that is employed is naive, once a scale is determined. That is, for a weight $\mathbf{w}_i \in \mathbf{w}_{(b)}$, Q_i is only a function of $\mathbf{w}_i$ and scale s_b. Furthermore, it must be rounded to the nearest point in the quantization. As discussed in Chapter 5, recent quantization schemes have begun to challenge this notion. In our extension of this work, we will relax these assumptions.

Now, we illuminate a key property of naive max-scaled block-wise quantization. Let $\mathbf{w}_i, \mathbf{w}_j \in \mathbf{w}_{(b)}$ for some block b. Then, if $\mathbf{w}_i \leq \mathbf{w}_j$, we then claim that $D_i(Q_i(\mathbf{w}_i)) \leq D_j(Q_j(\mathbf{w}_j))$. First, note that $D_i = D_j$ and $Q_i = Q_j$, since they are in the same block. Thus, it suffices to show that $D_i(Q_i(\mathbf{w}_i)) \leq D_i(Q_i(\mathbf{w}_j))$. Since it is a rounding-to-nearest scheme, we have that

$$D_i(Q_i(\mathbf{w}_i)) = \arg\min_{a \in \mathcal{Q}} |\mathbf{w}_i - a|.$$

Suppose towards a contradiction that $D_i\left(Q_i\left(\mathbf{w}_j\right)\right) < D_i\left(Q_i\left(\mathbf{w}_i\right)\right)$. Then, we have that

$$\arg\min_{a \in \mathcal{Q}} |\mathbf{w}_j - a| < \arg\min_{a \in \mathcal{Q}} |\mathbf{w}_i - a|.$$

Let $a_i = \arg\min_{a \in \mathcal{Q}} |\mathbf{w}_i - a|$ and $a_j = \arg\min_{a \in \mathcal{Q}} |\mathbf{w}_j - a|$.

We claim that

$$|\mathbf{w}_j - a_j| \geq |\mathbf{w}_j - a_i|.$$

We have that $a_i < a_j$, $\mathbf{w}_i < \mathbf{w}_j$, and $|\mathbf{w}_i - a_i| \leq |\mathbf{w}_i - a_j|$. We have the following.

$$\begin{aligned}
|\mathbf{w}_i - a_j| &\geq |\mathbf{w}_i - a_i| \\
\mathbf{w}_i &\geq \frac{a_i + a_j}{2} \\
\mathbf{w}_j &\geq \frac{a_i + a_j}{2} && (\mathbf{w}_i < \mathbf{w}_j) \\
|\mathbf{w}_j - a_j| &\geq |\mathbf{w}_j - a_i|
\end{aligned}$$

Thus, we have that

$$\mathbf{w}_j - a_j > \mathbf{w}_j - a_i \geq 0,$$

which is a contradiction, since we assumed $a_j = \arg\min_{a \in \mathcal{Q}} |\mathbf{w}_j - a|$. We will call this property the ordering-preserving property. Intuitively, since we are rounding to the nearest, we have two cases. If they lie in different parts of the quantization grid, then the rounding options of $\mathbf{w}_j$ will be greater than or equal to that of $\mathbf{w}_i$. Thus, it is sufficient to consider the case where they lie in the same segment of the quantization grid. In that case, if $\mathbf{w}_i$ rounds up, then we know that $\mathbf{w}_j$ must round up, since $\mathbf{w}_j$ will be closer to the upper point. Similarly, if $\mathbf{w}_j$ rounds down, then $\mathbf{w}_i$ must also round down, since $\mathbf{w}_i$ is closer to the lower point. It is possible that $\mathbf{w}_i$ rounds down and $\mathbf{w}_j$ rounds up. However, it is impossible that $\mathbf{w}_i$ rounds up and $\mathbf{w}_j$ rounds down.

10.2 Non-Naive Block-Wise Quantization Schemes

Working with max-scaled block-wise quantization is nice because of the rounding-to-nearest assumptions. It allows [HCK+25] to prove mathematically sound results that we will cover in the next section. However, as we have already seen in Chapter 5, this rounding-to-nearest assumption may not always hold. AdaRound, which we introduced in Chapter 5, clearly violates the order-preserving property. In particular, since we decide whether to round up or round down on each weight, it is possible that $\mathbf{w}_i$ rounds up and $\mathbf{w}_j$ rounds

down, even if $\mathbf{w}_i < \mathbf{w}_j$. In this section, we introduce two additional quantization algorithms that break this assumption.

10.2.1 Optimal Brain Compression (OBC) [FSA23]

First, we discuss OBC, which takes great inspiration from OBS, which we discussed in Chapter 6. In particular, it uses idea from sparsity and then applies it to quantization. This section will serve as the basis of the motivation for the next section, which will look at the intersection of solutions to the quantization and sparsity problem.

Optimal Brain Surgeon Optimizations

OBS is theoretically sound, but implementing it in practice is quite difficult. Even with the optimization using the iterative inversion technique discussed in Chapter 6, the number of weights in a modern machine learning model has significantly grown since the publication of [HS92]. In order to get around this, [FSA23] takes a layer-wise approach to this problem. Instead of applying OBS at the model level, they view each layer as its own "model". Recall from Chapter 1 that we can model a layer with weights $\mathbf{W}^{(\ell)} \in \mathbb{R}^{d_{\text{row}} \times d_{\text{col}}}$ and inputs $\mathbf{X} \in \mathbb{R}^{n \times d_{\text{row}}}$. Then, we apply some sparsity scheme to the trained $\hat{\mathbf{W}}^{(\ell)}$ which will yield $\tilde{\mathbf{W}}^{(\ell)}$. Since we are assuming a squared error loss, we will try to minimize

$$\|\mathbf{X}\hat{\mathbf{W}}^{(\ell)} - \mathbf{X}\tilde{\mathbf{W}}^{(\ell)}\|_2^2 = \sum_{i=1}^{d_{\text{col}}} \|\mathbf{X}\hat{\mathbf{W}}_{:,i}^{(\ell)} - \mathbf{X}\tilde{\mathbf{W}}_{:,i}^{(\ell)}\|_2^2.$$

We see that there is no second-order interaction between weights in different columns, since they affect different terms in the sum. Thus, when computing $\mathbf{H}_L^{\left(\mathbf{W}^{(\ell)}\right)}$, it is sufficient to consider $\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)} \in \mathbb{R}^{d_{\text{row}} \times d_{\text{row}}}$ for all column i. Then, we have that

$$\mathbf{H}_L^{\left(\mathbf{W}^{(\ell)}\right)} = \begin{bmatrix} \mathbf{H}_L^{\left(\mathbf{W}_{:,1}^{(\ell)}\right)} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{H}_L^{\left(\mathbf{W}_{:,2}^{(\ell)}\right)} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{H}_L^{\left(\mathbf{W}_{:,d_{\text{col}}}^{(\ell)}\right)} \end{bmatrix}.$$

Recall from Chapter 6, using the FIM-like approximation, we were left with

$$\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)} = \frac{\partial^2 L}{\partial \left(\mathbf{W}_{:,i}^{(\ell)}\right)^2} = \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{j=1}^{m} \frac{\partial \hat{\mathbf{y}}_j^{[i]}}{\partial \left(\mathbf{W}_{:,i}^{(\ell)}\right)} \frac{\partial \hat{\mathbf{y}}_j^{[i]}}{\partial \left(\mathbf{W}_{:,i}^{(\ell)}\right)}^{\mathsf{T}} \right) = \frac{2}{N} \sum_{i=1}^{N} \mathbf{X}^{\mathsf{T}}\mathbf{X} = 2\mathbf{X}^{\mathsf{T}}\mathbf{X}.$$

Importantly, under these assumptions, we have that $\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}$ does not depend on $\mathbf{W}_{:,i}^{(\ell)}$. Thus, changing the weights does not affect the Hessian. Note that this is not generally true, but this is true for this sub-problem. Suppose we remove weight $\mathbf{W}_{j,i}^{(\ell)}$. Then, we want to find $\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}$ for this new model. We will notate this as $\left(\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}\right)_{-j}$. Then, we have that

$$\left(\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}\right)_{-j} = \left(\mathbf{X}_{-j}\right)^\intercal \mathbf{X}_{-j},$$

where $\mathbf{X}_{-j}$ is $\mathbf{X}$ with the jth column removed. This is equivalent to removing the jth row and column of $\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}$.

Thus, the only problem that remains is to compute $\left(\left(\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}\right)_{-j}\right)^{-1}$. For this, we will use block matrices. Without loss of generality, we can suppose that $j = d_{\text{row}-1}$, since we can just permute the rows and columns of $\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}$. For notational simplicity, let

$$\mathbf{A} = \mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}.$$

Consider the block matrix

$$\mathbf{A} = \left[\begin{array}{c|c} \mathbf{A}_{-j} & \mathbf{A}_{1:d_{\text{row}}-1,j} \\ \hline \mathbf{A}_{j,1:d_{\text{row}}-1} & \mathbf{A}_{j,j} \end{array}\right].$$

Suppose that $\mathbf{A}$ invertible. Then, let

$$\left(\mathbf{A}\right)^{-1} = \left[\begin{array}{c|c} \mathbf{B} & \mathbf{u} \\ \hline \mathbf{v} & c \end{array}\right],$$

where $\mathbf{B} \in \mathbb{R}^{(d_{\text{col}}-1)\times(d_{\text{col}}-1)}$, $\mathbf{u} \in \mathbb{R}^{(d_{\text{col}}-1)\times 1}$, $\mathbf{v} \in \mathbb{R}^{1\times(d_{\text{col}}-1)}$, $c \in \mathbb{R}^{1\times 1}$. We have that

$$\mathbf{A}\mathbf{A}^{-1} = \left[\begin{array}{c|c} \mathbf{A}_{-j}\mathbf{B} + \mathbf{A}_{1:d_{\text{row}}-1,j}\mathbf{v} & \mathbf{A}_{-j}\mathbf{u} + \mathbf{A}_{1:d_{\text{row}}-1,j}c \\ \hline \mathbf{A}_{j,1:d_{\text{row}}-1}\mathbf{B} + \mathbf{A}_{j,j}\mathbf{v} & \mathbf{A}_{j,1:d_{\text{row}}-1}\mathbf{u} + \mathbf{A}_{j,j}c \end{array}\right] = \mathbf{I}.$$

Then, the claim is that

$$\mathbf{A}_{-j}^{-1} = \left(\mathbf{A}^{-1} - \frac{1}{\left(\mathbf{A}^{-1}\right)_{jj}}\left(\mathbf{A}^{-1}\right)_{:,j}\left(\mathbf{A}^{-1}\right)_{j,:}\right)_{-j} = \mathbf{B} - \frac{\mathbf{u}\mathbf{v}}{c}.$$

We can verify this with the following.

$$\begin{aligned}\mathbf{A}_{-j}\left(\mathbf{B}-\frac{\mathbf{uv}}{c}\right) &= \mathbf{A}_{-j}\mathbf{B}-\frac{1}{c}\mathbf{A}_{-j}\mathbf{uv} \\ &= \mathbf{I}_{(d_{\text{row}}-1)\times(d_{\text{row}}-1)}-\mathbf{A}_{1:d_{\text{row}}-1,j}\mathbf{v}-\frac{1}{c}\left(-\mathbf{A}_{1:d_{\text{row}}-1,j}c\right)\mathbf{v} \\ &= \mathbf{I}_{(d_{\text{row}}-1)\times(d_{\text{row}}-1)}\end{aligned}$$

Thus, we have the following.

$$\left(\left(\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}\right)_{-j}\right)^{-1}=\left(\left(\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}\right)^{-1}-\frac{1}{\left(\left(\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}\right)^{-1}\right)_{jj}}\left(\left(\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}\right)^{-1}\right)_{:,j}\left(\left(\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}\right)^{-1}\right)_{j,:}\right)_{-j}$$

The key insight in this simplification is that the entire Hessian need not be recomputed after pruning a weight. Instead, we can reuse the initial Hessian and just remove some rows and columns and recompute according to the expression above. Thus, the cost to recompute the Hessian becomes $O\left(d_{\text{row}}^2\right)$, since we simply need to subtract off values from $\left(\mathbf{H}_L^{\left(\mathbf{W}_{:,i}^{(\ell)}\right)}\right)^{-1}$.

Up until now, we have considered a single column. However, we want to consider all columns jointly. The key observation is that the Hessian is only affected when elements are pruned from the same column. Thus, we can consider a column at a time. Then, for each of the weights, find the change in loss. Then, we can aggregate all of the changes in losses and prune the smallest $p\%$ changes in losses. This is only possible because the Hessian between weights in different columns is 0. Thus, even if a weight in another column is pruned, it will not affect the change in loss of pruning a weight in another column. This allows us to avoid having to store each column's Hessian in memory simultaneously. Instead, we can store them one at a time. For the update step, we can use a "group" update step, where we minimize the constraint assuming that all of the weights in some subset are equal to 0. The math follows very similarly from that in Chapter 6. We exclude these details, but one can also refer to [KCN+22] for full details.

Optimal Brain Quantizer (OBQ)

[FSA23] extends their OBS optimizations to create a new quantization algorithm. In particular, they adopt an iterative quantization, where at each time step, a single weight is quantized, and the remaining unquantzied weights are updated by some $\Delta\mathbf{w}$. Again, we assume the quantization grid $\mathcal{Q}$ has already been set. To choose the weight to quantize, they

use a score function of

$$S_i(\mathbf{w}) \triangleq \frac{\left(\mathbf{w}_i - \overline{Q}(\mathbf{w}_i)\right)^2}{\left(\left(\mathbf{H}_L^{\hat{\mathbf{w}}}\right)^{-1}\right)_{ii}}$$

and qunatize the weight with the lowest score. Here, $\overline{Q} : \mathbb{R} \to \mathcal{Q}$ is some rounding function. This is exactly the same as OBS, except w_i^2 is replaced by $\left(\mathbf{w}_i - \overline{Q}(\mathbf{w}_i)\right)^2$. We can imagine if $\overline{Q}$ is the zero function, then we retrieve the original OBS form. Then, they use the same update formula. Since they will quantize every weight and not adjust it after it has been quantized, eventually all weights will be mapped to the quantization grid.

Note that this quantization does not respect the order-preserving property of naive quantization methods even if $\overline{Q}$ is a rounding to nearest scheme. This is because the weights are updated after each iteration. In particular, this is because we apply an update $\Delta\mathbf{w}$ after each iteration. Thus, $\mathbf{w}_i < \mathbf{w}_j$, but $\mathbf{w}_i + \Delta\mathbf{w} \geq \mathbf{w}_j$. Therefore, it is not applicable to [HCK+25] discussion.

[FAHA23] introduces GPTQ, which adds some additional optimizations atop the OBQ framework. These optimizations include batching updates to different columns and using the Cholesky decomposition of $\mathbf{H}_L^{(\hat{\mathbf{w}})}$ for more numerical stability in the inversion process. They are critical for allowing even better performance on particularly large models. We leave the specifics out of this thesis for sake of brevity, but we refer the interested reader to [FAHA23] for more concrete details.

10.2.2 Activation-aware Weight Quantization (AWQ) [LTT+24]

An interesting observation that is illuminated by sparsity methods is that not all weights are equally "important". Thus, we should try to preserve the "important" weights during quantization. [LTT+24] confirms this hypothesis by showing that skipping quantization for a small subset of weights can preserve almost all accuracy. However, [LTT+24] observes maintaining the weights of largest magnitude does not preserve much accuracy. Instead, they find that preserving the row (or "channel") of weights that corresponds to high activations is a better of metric of saliency. We will try to better understand this discrepancy in a future section.

Now, we present AWQ more formally. We will focus on matrix-multiplication based computations. Consider some activation $y = \mathbf{w}^\mathsf{T}\mathbf{x}$, with $\mathbf{w}, \mathbf{x} \in \mathbb{R}^d$. Then, we will assume a max-scaled block-wise rounding-to-nearest scheme with bitwidth N. Let $\overline{\mathbf{Q}} = \mathbf{D} \circ \mathbf{Q}$ for notational convenience. Suppose that $\mathbf{w}$ is a subset of a block $\tilde{\mathbf{w}}$. Thus, we have that

$$\left(\overline{\mathbf{Q}}(\mathbf{w})\right)^\mathsf{T}\mathbf{x} = s \cdot \left(\left\lfloor \frac{\mathbf{w}}{s} \right\rceil\right)^\mathsf{T}\mathbf{x},$$

where $s = \frac{\max_{\tilde{\mathbf{w}}_j \in \tilde{\mathbf{w}}} |\tilde{\mathbf{w}}_j|}{2^{N-1}}$. Then, we want to apply some scaling factor r to all the elements in $\mathbf{w}$, if y is "salient". Thus, consider

$$\left(\overline{\mathbf{Q}}\left(r\mathbf{w}\right)\right)^{\mathsf{T}} \left(\frac{1}{r}\mathbf{x}\right) = s' \cdot \frac{1}{r} \cdot \left(\left\lfloor \frac{r\mathbf{w}}{s'} \right\rceil\right)^{\mathsf{T}} \cdot \mathbf{x},$$

where $s' = \frac{\max_{\tilde{\mathbf{w}}_j \in \tilde{\mathbf{w}}'} |\tilde{\mathbf{w}}_j|}{2^{N-1}}$, where $\tilde{\mathbf{w}}'$ contains the scaled weight of $\mathbf{w}$.

Then, [LTT+24] makes the following two observations. First, they claim that

$$\frac{\mathbf{w}}{s} - \left\lfloor \frac{\mathbf{w}}{s} \right\rceil \approx \frac{r\mathbf{w}}{s'} - \left\lfloor \frac{r\mathbf{w}}{s'} \right\rceil.$$

This observation comes from the fact that the quantization grid does not change. Thus, the expected error is still the same. They verify this fact empirically. Second, they claim that

$$s \approx s',$$

since the group size is sufficiently large such that scaling a single row does not significantly change the largest element.

Note that the associated error term for the unscaled and scaled versions is

$$\mathbf{w}^{\mathsf{T}}\mathbf{x} - \left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)^{\mathsf{T}} \mathbf{x} = s \cdot \left(\frac{\mathbf{w}}{s} - \left\lfloor \frac{\mathbf{w}}{s} \right\rceil\right)^{\mathsf{T}} \mathbf{x}$$

and

$$\mathbf{w}^{\mathsf{T}}\mathbf{x} - \left(\overline{\mathbf{Q}}\left(r\mathbf{w}\right)\right)^{\mathsf{T}} \left(\frac{1}{r}\mathbf{x}\right) = s' \cdot \frac{1}{r} \cdot \left(\frac{r\mathbf{w}}{s'} - \left\lfloor \frac{r\mathbf{w}}{s'} \right\rceil\right)^{\mathsf{T}} \mathbf{x},$$

respectively. Then, under the above assumptions, the ratio of errors for the unscaled to the scaled version is approximately r. In particular, we have the following.

$$\begin{aligned} s \cdot \left(\frac{\mathbf{w}}{s} - \left\lfloor \frac{\mathbf{w}}{s} \right\rceil\right)^{\mathsf{T}} \mathbf{x} &\approx s \cdot \left(\frac{r\mathbf{w}}{s'} - \left\lfloor \frac{r\mathbf{w}}{s'} \right\rceil\right)^{\mathsf{T}} \mathbf{x} && \left(\tfrac{\mathbf{w}}{s} - \left\lfloor \tfrac{\mathbf{w}}{s} \right\rceil \approx \tfrac{r\mathbf{w}}{s'} - \left\lfloor \tfrac{r\mathbf{w}}{s'} \right\rceil\right) \\ &\approx s' \cdot \left(\frac{r\mathbf{w}}{s'} - \left\lfloor \frac{r\mathbf{w}}{s'} \right\rceil\right)^{\mathsf{T}} \mathbf{x} && (s = s') \end{aligned}$$

Therefore, if we choose $r > 1$, then we have effectively decreased the error term for the output activation by a factor of r. Therefore, by choosing the salient activations, we can protect their output by scaling the corresponding weights. However, we should note that we cannot increase r arbitrarily large. In particular, as r increases the assumptions above begin to break down. In particular, we have that the approximation that $s \approx s'$ begins to break down, since the scaled weights will begin to dominate the group. Therefore, it suffices to choose r for each row of a weight matrix $\mathbf{W}$. In practice, it is observed that the activations

that tend to be the largest in magnitude also have the largest scale factors, since we want to preserve those.

Similar to other methods, AWQ takes a layer-wise approach to choosing the optimal r. Consider a layer defined by inputs $\mathbf{X}$ and weights $\mathbf{W}$. Then, we want to minimize the function

$$R(\mathbf{r}) = \left\| \left((\text{diag}(\mathbf{r}))^{-1} \mathbf{X} \right) \overline{\mathbf{Q}} (\mathbf{W} \text{diag}(\mathbf{r})) - \mathbf{X}\mathbf{W} \right\|_2,$$

where $\mathbf{r}$ is the scaling factor for each activation. In order to decrease the search space, AWQ proposes that the initial average activation magnitude be used as an initial guess, $\mathbf{r}_0$. Then, we choose hyperparameter α such that $R(\mathbf{r}^\alpha)$ is minimized. This stems from the observation that the scaling factor is positively correlated with the activation magnitude. Thus, activations that are greater in magnitude will have larger scaling factors.

AWQ's quantization strategy is unique in that there are two different scaling factors. A group of weights is still on the same quantization grid, but each of the rows will have different scaling factors applied to it. Therefore, even if rounding-to-nearest is used for the original quantization grid, the additional scaling factor may break the order-preserving property. In particular, let $\mathbf{w}_i < \mathbf{w}_j$. Suppose that these weights are in the same group, but in different rows with scaling factors r_i and r_j respectively. Then, let

$$\left\lfloor \frac{r_i \mathbf{w}_i}{s} \right\rceil = \left\lfloor \frac{r_j \mathbf{w}_j}{s} \right\rceil,$$

where s is the scaling factor for the group. Then, it is possible that

$$\frac{s}{r_i} \left\lfloor \frac{r_i \mathbf{w}_i}{s} \right\rceil > \frac{s}{r_j} \left\lfloor \frac{r_j \mathbf{w}_j}{s} \right\rceil.$$

In particular, let $r_i = 2$, $\mathbf{w}_i = 0.4$, $r_j = 1$, $\mathbf{w}_j = 0.45$, and $s = 1$. Then, we have

$$\frac{s}{r_i} \left\lfloor \frac{r_i \mathbf{w}_i}{s} \right\rceil = \frac{1}{2}$$

and

$$\frac{s}{r_j} \left\lfloor \frac{r_j \mathbf{w}_j}{s} \right\rceil = 0.$$

We will also validate this empirically in the next section.

10.3 Order of Sparsity and Quantization [HCK+25]

The order of operations is a natural question that has only been recently explored in the field. In this section, we explore the most recent work in this area from [HCK+25]. This work

attempts to better understand the interaction of quantization and sparsity, and whether the order matters. For the remainder of the section we will use $\mathbf{Q} \to \mathbf{S}$ to refer to quantization being applied first followed by sparsity and $\mathbf{S} \to \mathbf{Q}$ to refer to sparsity being applied first followed by quantization.

[HCK+25] analyzes this question from both a mathematical and empirical angle. In this section, we mainly focus on the mathematical underpinnings of their analysis. They make two key assumptions in their analysis. First, they assumed that the quantization method being used is max-scaled block-wise quantization. Second, they assume that magnitude-based pruning methods are used. As we have seen in Chapter 5, Chapter 6, and the above sections these only refer to a subset of the current methodology that exists in this field. Thus, in future sections, we hope to extend their analysis to non-max-scaled block-wise quantization and non-magnitude-based sparsity method. However, for this section, we give a brief overview of their work.

10.3.1 Tensor Level Analysis

Let $\mathbf{w} \in \mathbb{R}^d$. Then, suppose that $\overline{\mathbf{Q}} : \mathbb{R}^d \to \mathbb{R}^d$ is the composition of the dequantization and quantization functions applied elementwise for a naive block-wise quantization schemes. That is the quantization, $Q : \mathbb{R} \to \mathcal{B}$, and dequantization, $D : \mathcal{B} \to \mathbb{R}$ functions are the same for each element of $\mathbf{x}$. Then, $\overline{Q} = D \circ Q$, and $\overline{\mathbf{Q}}$ is $\overline{Q}$ applied elementwise. We will also assume that $\overline{Q}(0) = 0$. This assumes that $0 \in \mathcal{Q}$. Furthermore, suppose we have a magnitude-based N:M sparsity scheme $\overline{\mathbf{S}} : \mathbb{R}^d \to \mathbb{R}^d$ that is the composition of the "quantization" and "dequantization" functions. Refer to Chapters 5 and 6 for the notation. Then, [HCK+25] shows the following.

First, they claim that

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}(\mathbf{w})\right)\right\|_p \leq \left\|\mathbf{w} - \overline{\mathbf{S}}(\mathbf{w})\right\|_p + \left\|\mathbf{w} - \overline{\mathbf{Q}}(\mathbf{w})\right\|_p$$

for any $p \geq 1$. That is the combined error of $\mathbf{S} \to \mathbf{Q}$ is less than the sum of the individual errors. The proof is as follows.

Let n be the number of weights that are pruned by $\mathbf{S}$. We will assume that $d \leq M$. Thus, we have that $n \leq N$. Then, without loss of generality, we can reorder $\mathbf{w}$ such that the pruned weights are the last n elements. This follows from the fact that norms are equal

under arbitrary ordering. Then, we have the following.

$$
\begin{aligned}
\left\|\mathbf{w}-\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_p &= \left\|\mathbf{w}-\overline{\mathbf{Q}}\left(\begin{bmatrix}\mathbf{w}_1\\ \vdots\\ \mathbf{w}_{d-n}\\ 0\\ \vdots\\ 0\end{bmatrix}\right)\right\|_p \\
&= \left\|\mathbf{w}-\begin{bmatrix}\overline{Q}\left(\mathbf{w}_1\right)\\ \vdots\\ \overline{Q}\left(\mathbf{w}_{d-n}\right)\\ 0\\ \vdots\\ 0\end{bmatrix}\right\|_p && (0\in\mathcal{Q}) \\
&= \left\|\begin{bmatrix}0\\ \vdots\\ 0\\ \mathbf{w}_{d-n+1}\\ \vdots\\ \mathbf{w}_d\end{bmatrix}+\begin{bmatrix}\mathbf{w}_1-\overline{Q}\left(\mathbf{w}_1\right)\\ \vdots\\ \mathbf{w}_{d-n}-\overline{Q}\left(\mathbf{w}_{d-n}\right)\\ 0\\ \vdots\\ 0\end{bmatrix}\right\|_p \\
&\leq \left\|\begin{bmatrix}0\\ \vdots\\ 0\\ \mathbf{w}_{d-n+1}\\ \vdots\\ \mathbf{w}_d\end{bmatrix}\right\|_p+\left\|\begin{bmatrix}\mathbf{w}_1-\overline{Q}\left(\mathbf{w}_1\right)\\ \vdots\\ \mathbf{w}_{d-n}-\overline{Q}\left(\mathbf{w}_{d-n}\right)\\ 0\\ \vdots\\ 0\end{bmatrix}\right\|_p && \text{(Triangle Inequality)} \\
&= \left\|\mathbf{w}-\overline{\mathbf{S}}\left(\mathbf{w}\right)\right\|_p+\left\|\begin{bmatrix}\mathbf{w}_1-\overline{Q}\left(\mathbf{w}_1\right)\\ \vdots\\ \mathbf{w}_{d-n}-\overline{Q}\left(\mathbf{w}_{d-n}\right)\\ 0\\ \vdots\\ 0\end{bmatrix}\right\|_p
\end{aligned}
$$

$$
\begin{aligned}
&\leq \left\|\mathbf{w}-\overline{\mathbf{S}}\left(\mathbf{w}\right)\right\|_p + \left\|\begin{bmatrix} \mathbf{w}_1-\overline{Q}\left(\mathbf{w}_1\right) \\ \vdots \\ \mathbf{w}_{d-n}-\overline{Q}\left(\mathbf{w}_{d-n}\right) \\ \mathbf{w}_{d-n+1}-\overline{Q}\left(\mathbf{w}_{d-n+1}\right) \\ \vdots \\ \mathbf{w}_{d}-\overline{Q}\left(\mathbf{w}_{d}\right) \end{bmatrix}\right\|_p \\
&= \left\|\mathbf{w}-\overline{\mathbf{S}}\left(\mathbf{w}\right)\right\|_p + \left\|\mathbf{w}-\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right\|_p
\end{aligned}
$$

The equality can be obtained. In particular, imagine $\mathbf{w}=\mathbf{0}$. Then, $\overline{\mathbf{S}}\left(\mathbf{w}\right)=\mathbf{0}$. Furthermore, we have that $\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)=\overline{\mathbf{S}}\left(\mathbf{w}\right)=\mathbf{0}$. Thus, we have that the equality is obtained. Here, there were two implicit assumptions. First, the assumption was that the quantization scale does not change after the elements are pruned. Second, it used the fact that quantization scheme was naive to map the zero weights to zero.

[HCK+25] shows that it is not necessarily true that

$$
\left\|\mathbf{w}-\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_p \leq \left\|\mathbf{w}-\overline{\mathbf{S}}\left(\mathbf{w}\right)\right\|_p + \left\|\mathbf{w}-\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right\|_p .
$$

In particular, consider the following counterexample presented in the paper. Assume an rounding-to-nearest integer quantization grid with 1:2 magnitude sparsity. Let

$$
\mathbf{w}=\begin{bmatrix} 3.9 \\ 4.0 \end{bmatrix}.
$$

Then, we have that

$$
\overline{\mathbf{S}}\left(\mathbf{w}\right)=\begin{bmatrix} 0 \\ 4.0 \end{bmatrix}, \overline{\mathbf{Q}}\left(\mathbf{w}\right)=\begin{bmatrix} 4 \\ 4 \end{bmatrix},
$$

and

$$
\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)=\begin{bmatrix} 4 \\ 0 \end{bmatrix}.
$$

Thus, we have that

$$
\left\|\mathbf{w}-\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_p = \left\|\begin{bmatrix} -0.1 \\ 4 \end{bmatrix}\right\|_p,
$$

$$
\left\|\mathbf{w}-\overline{\mathbf{S}}\left(\mathbf{w}\right)\right\|_p = \left\|\begin{bmatrix} 3.9 \\ 0.0 \end{bmatrix}\right\|_p = 3.9 \text{ and } \left\|\mathbf{w}-\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right\|_p = \left\|\begin{bmatrix} -0.1 \\ 0.0 \end{bmatrix}\right\|_p = 0.1.
$$

Therefore, we see that

$$\left\|\mathbf{w} - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_p > \left\|\mathbf{w} - \overline{\mathbf{S}}\left(\mathbf{w}\right)\right\|_p + \left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\mathbf{w}\right)\right\|_p$$

for any $p \geq 1$. As investigated in the previous section, the key issues with the counterexample is that quantization creates a "collision" of the two weights. Therefore, the weights that were previously distinguishable are no longer distinguishable. We will return to this idea again shortly.

Finally, [HCK+25] proves that

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_1 \leq \left\|\mathbf{w} - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_1,$$

which affirms that $\mathbf{S} \to \mathbf{Q}$ is likely preferred to $\mathbf{Q} \to \mathbf{S}$. In this work, we prove the much more general statement that

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_p \leq \left\|\mathbf{w} - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_p$$

for $p > 1$. This statement is much stronger than the L_1 case, and we will explain the ramifications after the proof. The proof is as follows.*

We have that

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_p = \left(\sum_{i=1}^{d} \left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)_i\right|^p\right)^{\frac{1}{p}}$$

and

$$\left\|\mathbf{w} - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_1 = \left(\sum_{i=1}^{d} \left|\mathbf{w}_i - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)_i\right|^p\right)^{\frac{1}{p}}.$$

For $\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)_i$, we have that

$$\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)_i = \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i$$

if $\mathbf{w}_i$ is not pruned and 0 otherwise. Thus, we have that

$$\left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)_i\right| = \left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|$$

if $\mathbf{w}_i$ is not pruned and

$$\left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)_i\right| = \left|\mathbf{w}_i - \overline{\mathbf{S}}\left(\mathbf{w}\right)_i\right|$$

otherwise. This follows from the fact that the scale does not change after pruning.

*Many thanks to William Shi, who this proof was done in collaboration with!

Now, consider $\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)_i$. We will prove the statement

$$\sum_{i\in T}\left|\mathbf{w}_i-\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)_i\right|^p\leq\sum_{i\in T}\left|\mathbf{w}_i-\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)_i\right|^p$$

inductively on the size of set $T\subseteq\{1,\ldots,d\}$. Let $n=|T|$. This argument is more formal than that presented in [HCK+25].

First, we consider the base case where $n=0$. Then, we have that the statement is trivially true. Now, we show the inductive step. Suppose that the statement is true for some $n=k$. Then, we will show that the statement is true for $n=k+1$. Consider some set T with cardinality $k+1$.

First, consider the case where different elements are pruned before and after quantization. This can only occur if there exists $i,j\in T$ such that $|\mathbf{w}_i|<|\mathbf{w}_j|$, but $\overline{Q}\left(\mathbf{w}_i\right)=\overline{Q}\left(\mathbf{w}_j\right)$. Furthermore, suppose the jth element is pruned after quantization but not pruned before quantization, while the ith element is pruned before quantization and not after. Then, we have that

$$\left|\mathbf{w}_i-\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)_i\right|=\left|\mathbf{w}_i-\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|$$

and

$$\left|\mathbf{w}_j-\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)_j\right|=\left|\mathbf{w}_j\right|.$$

Our goal will be to show that

$$\left|\mathbf{w}_i-\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)_i\right|^p+\left|\mathbf{w}_j-\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)_j\right|^p\leq\left|\mathbf{w}_i-\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)_i\right|^p+\left|\mathbf{w}_j-\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)_j\right|^p.$$

Then, by the inductive hypothesis, we will have that

$$\sum_{i\in T\setminus\{i,j\}}\left|\mathbf{w}_i-\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)_i\right|^p\leq\sum_{i\in T\setminus\{i,j\}}\left|\mathbf{w}_i-\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)_i\right|^p.$$

Thus, we will have that

$$\sum_{i\in T}\left|\mathbf{w}_i-\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)_i\right|^p\leq\sum_{i\in T}\left|\mathbf{w}_i-\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)_i\right|^p.$$

We assume that $\mathbf{w}_i$ was pruned before quantization and not after. Thus, we have the desired inequality is

$$|\mathbf{w}_i|^p+\left|\mathbf{w}_j-\overline{\mathbf{Q}}\left(\mathbf{w}\right)_j\right|^p\leq\left|\mathbf{w}_i-\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p+|\mathbf{w}_j|^p.$$

Furthermore, we assumed that $\overline{Q}(\mathbf{w}_i) = \overline{Q}(\mathbf{w}_j)$. Thus, we have that the inequality is equivalent to

$$|\mathbf{w}_i|^p + \left|\mathbf{w}_j - \overline{\mathbf{Q}}(\mathbf{w})_i\right|^p \leq \left|\mathbf{w}_i - \overline{\mathbf{Q}}(\mathbf{w})_i\right|^p + |\mathbf{w}_j|^p .$$

First, suppose that $\mathbf{w}_i$ and $\mathbf{w}_j$ are opposite signs. Then, we must have that $\overline{\mathbf{Q}}(\mathbf{w})_i = 0$. Without loss of generality, one can assume that $\mathbf{w}_i \leq 0$ and $\mathbf{w}_j \geq 0$. Then, we must have that $\mathbf{w}_i$ rounds up and $\mathbf{w}_j$ rounds down. Assuming $0 \in \mathcal{Q}$, we have that $\overline{\mathbf{Q}}(\mathbf{w})_i = \overline{\mathbf{Q}}(\mathbf{w})_j$ cannot be less than 0 or greater than 0. This follows from the fact that

$$|\mathbf{w}_i| < |c - \mathbf{w}_i|$$

for all $c > 0$ and

$$|\mathbf{w}_j| < |\mathbf{w}_j - c|$$

for all $c < 0$. Then, we have that the inequality trivially holds, since

$$|\mathbf{w}_i|^p + |\mathbf{w}_j|^p = |\mathbf{w}_i|^p + |\mathbf{w}_j|^p .$$

Now, assume that $\mathbf{w}_i$ and $\mathbf{w}_j$ are the same sign. Thus, we also have that $\overline{\mathbf{Q}}(\mathbf{w})_i$ will have the same sign, by a similar argument to the one above. In particular, the distance to 0 is less than the distance to any number with the opposite sign. Then, we consider the following three cases: $|\mathbf{w}_i| \leq \left|\overline{\mathbf{Q}}(\mathbf{w})_i\right| \leq |\mathbf{w}_j|$, $|\mathbf{w}_i| \leq |\mathbf{w}_j| \leq \left|\overline{\mathbf{Q}}(\mathbf{w})_i\right|$, and $\left|\overline{\mathbf{Q}}(\mathbf{w})_i\right| \leq |\mathbf{w}_i| \leq |\mathbf{w}_j|$. Without loss of generality, we can assume that $\mathbf{w}_i, \mathbf{w}_j, \overline{\mathbf{Q}}(\mathbf{w})_i \geq 0$. This follows from the fact that negating all the quantities will yield the same result under the absolute value.

First, consider $|\mathbf{w}_i| \leq \left|\overline{\mathbf{Q}}(\mathbf{w})_i\right| \leq |\mathbf{w}_j|$. We want to show that

$$|\mathbf{w}_i|^p + \left|\mathbf{w}_j - \overline{\mathbf{Q}}(\mathbf{w})_i\right|^p \leq \left|\mathbf{w}_i - \overline{\mathbf{Q}}(\mathbf{w})_i\right|^p + |\mathbf{w}_j|^p .$$

We claim that $|\mathbf{w}_i|^p + \left|\mathbf{w}_j - \overline{\mathbf{Q}}(\mathbf{w})_i\right|^p$ is decreasing on $\overline{\mathbf{Q}}(\mathbf{w})_i$ assuming that $\mathbf{w}_i \leq \overline{\mathbf{Q}}(\mathbf{w})_i \leq \mathbf{w}_j$. We have that $\left|\mathbf{w}_j - \overline{\mathbf{Q}}(\mathbf{w})_i\right| = \mathbf{w}_j - \overline{\mathbf{Q}}(\mathbf{w})_i$. Thus, we have that

$$|\mathbf{w}_i|^p + \left|\mathbf{w}_j - \overline{\mathbf{Q}}(\mathbf{w})_i\right|^p = |\mathbf{w}_i|^p + \left(\mathbf{w}_j - \overline{\mathbf{Q}}(\mathbf{w})_i\right)^p ,$$

which is clearly decreasing on $\overline{\mathbf{Q}}(\mathbf{w})_i$. Thus, we have that

$$|\mathbf{w}_i|^p + \left|\mathbf{w}_j - \overline{\mathbf{Q}}(\mathbf{w})_i\right|^p \leq |\mathbf{w}_i|^p + |\mathbf{w}_j - \mathbf{w}_i|^p .$$

Then, we claim that

$$|\mathbf{w}_i|^p + |\mathbf{w}_j - \mathbf{w}_i|^p \leq |\mathbf{w}_j|^p .$$

We will show that

$$|\mathbf{w}_i|^p \leq |\mathbf{w}_j|^p - |\mathbf{w}_j - \mathbf{w}_i|^p .$$

We have that

$$|\mathbf{w}_i|^p = \mathbf{w}_i^p = \int_0^{\mathbf{w}_i} p x^{p-1} dx$$

and

$$|\mathbf{w}_j|^p - |\mathbf{w}_j - \mathbf{w}_i|^p = \int_0^{\mathbf{w}_i} p \left(\mathbf{w}_j - \mathbf{w}_i + x\right)^{p-1} dx.$$

We have that $\mathbf{w}_i < \mathbf{w}_j$, so $px^{p-1} < p\left(\mathbf{w}_j - \mathbf{w}_i + x\right)^{p-1}$ for $x \in [0, \mathbf{w}_i]$. Thus, we have that

$$\int_0^{\mathbf{w}_i} p x^{p-1} dx \leq \int_0^{\mathbf{w}_i} p \left(\mathbf{w}_j - \mathbf{w}_i + x\right)^{p-1} dx$$

and

$$|\mathbf{w}_i|^p \leq |\mathbf{w}_j|^p - |\mathbf{w}_j - \mathbf{w}_i|^p .$$

Thus, we have that

$$|\mathbf{w}_i|^p + \left|\mathbf{w}_j - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p \leq |\mathbf{w}_i|^p + |\mathbf{w}_j - \mathbf{w}_i|^p \leq |\mathbf{w}_j|^p .$$

Finally, we have that

$$|\mathbf{w}_j|^p \leq |\mathbf{w}_j|^p + \left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p ,$$

since $\left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right| \geq 0$. Therefore, we have proved the desired inequality.

Now, consider the case where $|\mathbf{w}_i| \leq |\mathbf{w}_j| \leq \left|\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|$. We want to show that

$$|\mathbf{w}_i|^p + \left|\mathbf{w}_j - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p \leq \left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p + |\mathbf{w}_j|^p .$$

Rearranging, we have that

$$\left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p - \left|\mathbf{w}_j - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p \geq |\mathbf{w}_i|^p - |\mathbf{w}_j|^p .$$

First, we claim that $\left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p - \left|\mathbf{w}_j - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p$ is increasing in $\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i$. Since $|\mathbf{w}_i| \leq |\mathbf{w}_j| \leq \left|\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|$ and we are assuming that $\mathbf{w}_i, \mathbf{w}_j, \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i \geq 0$, then we have that

$$\left|\mathbf{w}_j - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right| = \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i - \mathbf{w}_j$$

and

$$\left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right| = \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i - \mathbf{w}_i.$$

Thus, it suffices to show that $\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i - \mathbf{w}_i\right)^p - \left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i - \mathbf{w}_j\right)^p$ is increasing in $\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i$. Then, we have that

$$\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i - \mathbf{w}_i\right)^p - \left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i - \mathbf{w}_j\right)^p = \int_{-\mathbf{w}_j}^{-\mathbf{w}_i} p\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i + x\right)^{p-1} dx.$$

We have that $p\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i + x\right)^{p-1}$ is increasing in $\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i$, and $-\mathbf{w}_j < -\mathbf{w}_i$ by assumption. Therefore, $\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i - \mathbf{w}_i\right)^p - \left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i - \mathbf{w}_j\right)^p$ is increasing in $\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i$. Therefore, we have that

$$\left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p - \left|\mathbf{w}_j - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p \geq \left|\mathbf{w}_i - \mathbf{w}_j\right|^p - \left|\mathbf{w}_j - \mathbf{w}_j\right|^p = \left|\mathbf{w}_i - \mathbf{w}_j\right|^p > 0.$$

Furthermore, we have that $|\mathbf{w}_i|^p - |\mathbf{w}_j|^p < 0$, since $0 \leq \mathbf{w}_i < \mathbf{w}_j$. Therefore, the desired inequality is shown.

Finally, consider the case where $\left|\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right| \leq |\mathbf{w}_i| \leq |\mathbf{w}_j|$. Again, we want to show that

$$|\mathbf{w}_i|^p + \left|\mathbf{w}_j - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p \leq \left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p + |\mathbf{w}_j|^p.$$

Rearranging, we have that

$$|\mathbf{w}_i|^p - \left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p \leq |\mathbf{w}_j|^p - \left|\mathbf{w}_j - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p.$$

Then, we have that

$$|\mathbf{w}_i|^p - \left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p = \int_{-\overline{\mathbf{Q}}(\mathbf{w})_i}^{0} p\left(\mathbf{w}_j + x\right)^{p-1} dx$$

and

$$|\mathbf{w}_j|^p - \left|\mathbf{w}_j - \overline{\mathbf{Q}}\left(\mathbf{w}\right)_i\right|^p = \int_{-\overline{\mathbf{Q}}(\mathbf{w})_i}^{0} p\left(\mathbf{w}_i + x\right)^{p-1} dx.$$

We have that $-\overline{\mathbf{Q}}\left(\mathbf{w}\right)_i < 0$ and $\mathbf{w}_i < \mathbf{w}_j$. Thus,

$$\int_{-\overline{\mathbf{Q}}(\mathbf{w})_i}^{0} p\left(\mathbf{w}_j + x\right)^{p-1} dx < \int_{-\overline{\mathbf{Q}}(\mathbf{w})_i}^{0} p\left(\mathbf{w}_i + x\right)^{p-1} dx.$$

Therefore, we have the inequality as desired.

Now, consider the other case. In particular, consider the case where the same elements are pruned before and after quantization. Then, the elements that $\overline{S}$ prunes remains the same regardless if it occurs before and after quantization. Furthermore, since the scale did

not change, we have that

$$\left|\mathbf{w}_i - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)_i\right| = \left|\mathbf{w}_i - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)_i\right|$$

for all $i \in \{1, \ldots, d\}$. Therefore, we have that

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_p = \left\|\mathbf{w} - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_p.$$

Thus, this concludes the proof for

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_p \leq \left\|\mathbf{w} - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_p.$$

This result extends the result of [HCK+25] to the more general setting of L_p norms. This gives further credence to the claim that $\mathbf{S} \to \mathbf{Q}$ is preferred to $\mathbf{Q} \to \mathbf{S}$. As we have already mentioned, this proof illuminates the key reason that $\mathbf{S} \to \mathbf{Q}$ is preferred. Applying quantization first leads to these "collisions", where previously distinguishable weights become indistinguishable. Thus, sparsity has a hard time deciding the correct weights to prune.

As mentioned above, this result is much stronger than that presented in [HCK+25]. In particular, we now have that

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_p \leq \left\|\mathbf{w} - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_p$$

for $p > 1$. The L_1 norm is an arbitrary metric, but we have showed that for all L_p error metrics, $\mathbf{Q} \to \mathbf{S}$ is preferred. For example, we have that

$$\left\|\mathbf{x}\right\|_\infty = \lim_{p \to \infty} \mathbf{x}_p.$$

Therefore, it can be shown that

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_\infty \leq \left\|\mathbf{w} - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_\infty,$$

which tells us the worst-case error inside the tensor is better for $\mathbf{Q} \to \mathbf{S}$ compared to $\mathbf{S} \to \mathbf{Q}$.

Furthermore, for any function f that is Lipschitz continuous, we have that

$$\left\|f\left(\mathbf{a}\right) - f\left(\mathbf{b}\right)\right\|_p \leq L_p \left\|\mathbf{a} - \mathbf{b}\right\|_p,$$

for $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$ and $L_p \in \mathbb{R}_{>0}$. Therefore, we can have a better error bound on functions of

$\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)$ compared to $\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)$. Although this does not guarantee, that

$$\left\|f\left(\mathbf{w}\right)-f\left(\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right)\right\|_p \leq \left\|f\left(\mathbf{w}\right)-f\left(\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right)\right\|_p$$

for all $p > 1$, it gives more reason to believe that the $\mathbf{Q} \to \mathbf{S}$ is preferred, since we have a better upper bound for $\left\|f\left(\mathbf{w}\right)-f\left(\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right)\right\|_p$ compared to $\left\|f\left(\mathbf{w}\right)-f\left(\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right)\right\|_p$. While this is true for the tensor level, we will show that these results do not really translate to the model level case.

Above, we showed that there exists $\mathbf{w} \in \mathbb{R}^d$ such that

$$\left\|\mathbf{w}-\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_p \leq \left\|\mathbf{w}-\overline{\mathbf{S}}\left(\mathbf{w}\right)\right\|_p + \left\|\mathbf{w}-\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right\|_p.$$

[HCK+25] also proves that $\left\|\mathbf{w}-\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_p$ is upper-bounded by some quantity, but we omit the proof since it is not used for the future analysis.

10.3.2 Dot Product Level Analysis

[HCK+25] shows that at the dot-product level, quantization and sparsity are not orthogonal. In particular, they claim that there exists inputs $\mathbf{x} \in \mathbb{R}^d$ and weights $\mathbf{w} \in \mathbb{R}^d$ such that

$$\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right| \geq \left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{S}}\left(\mathbf{w}\right)\right| + \left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right|.$$

Similarly, there exists inputs $\mathbf{x} \in \mathbb{R}^d$ and weights $\mathbf{w} \in \mathbb{R}^d$ such that

$$\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right| \geq \left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{S}}\left(\mathbf{w}\right)\right| + \left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right|.$$

In particular, consider the following counterexample presented in the paper. Assume an rounding-to-nearest integer quantization grid with 1:2 magnitude sparsity. Let

$$\mathbf{w} = \begin{bmatrix}0.6\\1.3\end{bmatrix} \text{ and } \mathbf{x} = \begin{bmatrix}1.0\\1.0\end{bmatrix}.$$

Then, we have that

$$\overline{\mathbf{S}}\left(\mathbf{w}\right) = \begin{bmatrix}0\\1.3\end{bmatrix}, \overline{\mathbf{Q}}\left(\mathbf{w}\right) = \begin{bmatrix}1\\1\end{bmatrix},$$

$$\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right) = \begin{bmatrix}0\\1\end{bmatrix} \text{ and } \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right) = \begin{bmatrix}0\\1\end{bmatrix}$$

Thus, we have that

$$\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right|=1.9-1=0.9 \text{ and } \left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right|=1.9-1=0.9,$$

but

$$\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{S}}\left(\mathbf{w}\right)\right|=1.9-1.3=0.6 \text{ and } \left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right|=|1.9-2.0|=0.1.$$

Therefore, we have that

$$\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right|\geq\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{S}}\left(\mathbf{w}\right)\right|+\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right|$$

and

$$\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right|\geq\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{S}}\left(\mathbf{w}\right)\right|+\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right|.$$

[HCK+25] also shows that they are not orthogonal, if we also apply quantization to the activations. However, as our focus is on weight-only quantization and sparsity, we omit the counterexample. Furthermore, [HCK+25] upper bounds $\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right|$ and $\left|\mathbf{x}\cdot\mathbf{w}-\mathbf{x}\cdot\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right|$. However, we again leave this analysis to the original paper.

10.4 Relation Between Quantization and Sparsity

In this section, we will try to answer the fundamental question of the extent to which quantization and sparsity are related. In Chapters 5 and 6, we framed quantization and sparsity in similar ways. In particular, we viewed both as small perturbations to optimal weights $\hat{\mathbf{w}}$. However, in this section, we will show that this is where the similarities end. The two compression schemes have two different objectives. Quantization attempts to spread "importance" across the weights, while sparsity tries to push "importance" to a few weights. We will provide motivation and verify this claim empirically.

10.4.1 Motivation

OBS and OBQ showed that there is a natural connection between sparsity and quantization by viewing them through the lens of the second-order approximation. In particular, OBQ views quantization as an iterative process, where weights are quantized one-by-one, just as they are pruned one-by-one in OBS. This connection is further illuminated by our mathematical formulations in Chapters 5 and 6, where we showed that sparsity can be viewed as a form of "quantization" and "dequantization". In particular, finding an appropriate mask

$\mathbf{m}_i$ is equivalent to finding some quantization function $Q_i : \mathbb{R}^d \to \{0, 1\}$.

In this section, we want to answer whether the objectives between quantization and sparsity are aligned. We have shown that they are closely related. However, we claim that the objectives are not aligned. Therefore, using a quantization algorithm to find an approximate mask, as proposed above, is not a suitable sparsity method.

10.4.2 Quantization-Induced Sparsity

To show that the objectives are misaligned, we propose the following experiment. First, suppose we have weights $\hat{\mathbf{w}}$. Then, we quantize them to $\hat{\mathbf{w}}_\mathbf{Q}$. We want to show that the importance of the weights has been distributed over the weights in the quantized form. As we mentioned in Chapter 6, magnitude is a natural proxy for saliency. Thus, we can use the quantized weights $\hat{\mathbf{w}}_\mathbf{Q}$ to give a sparsity mask $\mathbf{m}$ for a $p\%$ magnitude-based sparsity scheme. Then, we can apply that sparsity mask $\mathbf{m}$ to the original weights $\hat{\mathbf{w}}$ and evaluate the performance. We expect that the performance will degrade seriously under this scheme as opposed to applying magnitude-based sparsity directly on the weights $\hat{\mathbf{w}}_\mathbf{Q}$. We attribute this degradation in performance to the fact that the quantized version of the weights has "spread out" importance over the weights. However, for optimal pruning, we would like the importance to be concentrated in fewer weights, so that unnecessary weights can be pruned.

We show the results of our empirical analysis below on OPT-125M. We use AWQ with a group size of 128. Then, we use the quantized weights to get a $p\%$ sparsity mask. Then, we apply the mask and compute the perplexity.

Table 10.1: Quantization-Induced Sparsity

PPL↓				**OPT**	
Precision	**Quantization Method**	**Sparsity**	**Sparsity Method**	**125M**	**Threshold**
FP16	N/A	0%	N/A	27.656	0.000000
FP16	N/A	10%	Magnitude	29.119	0.0036
FP16	N/A	25%	Magnitude	35.645	0.0095
INT8	AWQ	10%	Induced	99.466	0.000745
INT4	AWQ	10%	Induced	28.464*	0.000000
INT3	AWQ	10%	Induced	31.398*	0.000000
INT8	AWQ	25%	Induced	8119.545	0.005478
INT4	AWQ	25%	Induced	5278.241	0.004288
INT3	AWQ	25%	Induced	115.140	0.000892

We can see that for the results without the asterisk, the induced method of sparsity fares much worse than ordinary magnitude-based sparsity. These results affirm that the objectives

of the two compression methods are very different, even though they have similar setups. In particular, quantization may decrease the weight of previously "important" weights to try to spread out "importance" more evenly among the weights. Then, the induced mask will mask out these previously important weights. Now, we address the results with the asterisk. We note that these have lower perplexity than the corresponding magnitude-based pruning methods. We argue that both quantization and sparsity treat weights that have very low magnitude the same. In particular, if a weight has very low magnitude, it is likely to get pruned in magnitude based sparsity. Similarly, it is likely to be quantized to the zero point. Therefore, for these measurements, the threshold for the quantized weights, $\hat{\mathbf{w}}_{\mathbf{Q}}$ was 0. Thus, the weights that are pruned in the original $\hat{\mathbf{w}}$ will be those that were already very close to zero. However, once the sparsity is increased, we begin to see differing behaviors.

OBS and OBQ finds the correct relationship between the two compression algorithms through the second-order approximation. However, the similarities end there. For example, when OBS updates the unpruned weights after pruning a single weight, it may try to push "importance" of the pruned weight to a smaller subset of the remaining weights. Alternatively, OBQ may try to spread the importance of the quantized weight to a larger subset of the weights, so that the effect of quantization on other weights is dampened.

10.5 Extending [HCK+25] to Non-Naive Quantization Schemes

In this section, we will extend some of the mathematical insights of [HCK+25] to a scenario where a non-naive quantization scheme is used. While mathematical analysis in this scenario is difficult since we do not have many guarantees on the rounding scheme, we try to make generalizations at the model level. [HCK+25] focuses their mathematical analysis on the tensor and dot-product cases. However, we attempt to extend this to the model case, since we are ultimately interested in the change in end-to-end performance of the given machine learning model. However, we will try to re-examine the analysis presented in [HCK+25] under this more general setting.

10.5.1 Tensor Level Analysis

First, as we noted in the previous section, the result no longer hold. In particular, it is not necessarily true that

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_p \leq \left\|\mathbf{w} - \overline{\mathbf{S}}\left(\mathbf{w}\right)\right\|_p + \left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\mathbf{w}\right)\right\|_p.$$

We provide a counterexample. Let

$$\mathbf{w} = \begin{bmatrix} 0.1 \\ 0.9 \end{bmatrix}.$$

Then, suppose we have a integer quantization grid that employs stochastic rounding. Furthermore, suppose we are doing 1:2 magnitude pruning. Thus, let

$$\overline{\mathbf{Q}}\left(\mathbf{w}\right) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Furthermore, let

$$\overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right) = \overline{\mathbf{Q}}\left(\begin{bmatrix} 0 \\ 0.9 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Then, we have that

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_1 = \left\|\begin{bmatrix} 0.1 \\ 0.9 \end{bmatrix}\right\|_1 = 1.$$

However, we have that

$$\left\|\mathbf{w} - \overline{\mathbf{S}}\left(\mathbf{w}\right)\right\|_p + \left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\mathbf{w}\right)\right\|_p = \left\|\begin{bmatrix} 0.1 \\ 0 \end{bmatrix}\right\|_1 + \left\|\begin{bmatrix} 0 \\ 0.1 \end{bmatrix}\right\|_1 = 0.2.$$

Thus, we have that

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_1 \leq \left\|\mathbf{w} - \overline{\mathbf{S}}\left(\mathbf{w}\right)\right\|_1 + \left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\mathbf{w}\right)\right\|_1.$$

We also have that

$$\left\|\mathbf{w} - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_1 = \left\|\begin{bmatrix} 0.1 \\ -0.1 \end{bmatrix}\right\|_1 = 0.2,$$

since

$$\overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Therefore, it is also not necessarily true that

$$\left\|\mathbf{w} - \overline{\mathbf{Q}}\left(\overline{\mathbf{S}}\left(\mathbf{w}\right)\right)\right\|_1 \geq \left\|\mathbf{w} - \overline{\mathbf{S}}\left(\overline{\mathbf{Q}}\left(\mathbf{w}\right)\right)\right\|_1.$$

These results should not be surprising. Intuitively, we have lost some guarantees about our quantization scheme by removing the rounding-to-nearest assumption. The hope is that while we remove some guarantees at the tensor-level, it is more optimized at the model-level.

Thus, we should focus on a model-level analysis.

10.5.2 Dot Product Level Analysis

The dot-product analysis from [HCK+25] still extends to this more general setting, since the counterexample is still valid.The result simply shows that quantization and sparsity are not orthogonal. Thus, these results still serve as motivation that we should study such interactions at a model-level.

10.5.3 Model Level Analysis

As mentioned above, [HCK+25] does not analyze the change in empirical loss from a model level. Thus, in this work, we try to generalize the ideas from [HCK+25] to the model level. Working at the model level is optimal, since most of the quantization and sparsity methods that we introduced in Chapters 5 and 6 operate at the model level. In particular, they try to minimize the change in empirical loss of the entire model under the perturbation of the weights. Recall that we had the following from chapters 5 and 6 under the assumption that the empirical loss function met some regularity conditions and the distribution of the training data matched the true data generating process.

$$L\left(\tilde{\mathbf{w}}+\Delta\mathbf{w}\right)-L\left(\tilde{\mathbf{w}}\right)\approx\Delta\mathbf{w}^{\mathsf{T}}\nabla_{\mathbf{w}}L\Big|_{\mathbf{w}=\tilde{\mathbf{w}}}+\frac{1}{2}\Delta\mathbf{w}^{\mathsf{T}}\mathbf{H}_{L}^{(\tilde{\mathbf{w}})}\Delta\mathbf{w}$$

Then, suppose we have a quantization scheme that introduces a $\Delta\mathbf{w}_{\mathbf{Q}}$ change to the weights. Then, we had the following, assuming that $\hat{\mathbf{w}}$ was a local optimum.

$$L\left(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}}\right)-L\left(\hat{\mathbf{w}}\right)\approx\Delta\mathbf{w}_{\mathbf{Q}}^{\mathsf{T}}\nabla_{\mathbf{w}}L\Big|_{\mathbf{w}=\hat{\mathbf{w}}}+\frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\mathsf{T}}\mathbf{H}_{L}^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}}=\frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\mathsf{T}}\mathbf{H}_{L}^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}}$$

Therefore, we tried to minimize $\frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\mathsf{T}}\mathbf{H}_{L}^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}}$ under some conditions.

Similarly, we assume that we have a sparsity scheme that introduces $\Delta\mathbf{w}_{\mathbf{S}}$ change to the weights. Then, we had the following, assuming that $\hat{\mathbf{w}}$ was a local optimum.

$$L\left(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{S}}\right)-L\left(\hat{\mathbf{w}}\right)\approx\Delta\mathbf{w}_{\mathbf{S}}^{\mathsf{T}}\nabla_{\mathbf{w}}L\Big|_{\mathbf{w}=\hat{\mathbf{w}}}+\frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\mathsf{T}}\mathbf{H}_{L}^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}}=\frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\mathsf{T}}\mathbf{H}_{L}^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}}$$

Again, this led us to choose sparsity schemes that minimizes the value of $\frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\mathsf{T}}\mathbf{H}_{L}^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}}$ under some conditions.

However, we now illuminate the problem with blindly applying quantization and sparsity algorithms sequentially. In particular, suppose we have applied some quantization algorithm

and the weights are $\hat{\mathbf{w}} + \Delta\mathbf{w}_\mathbf{Q}$. Then, suppose we apply some sparsity algorithm atop these quantized weights with change $\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$. Then, we have the following for the change in empirical loss.

$$L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_\mathbf{Q} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\right) - L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_\mathbf{Q}\right) \approx \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^\mathsf{T} \nabla_\mathbf{w} L\Big|_{\mathbf{w}=\hat{\mathbf{w}}+\Delta\mathbf{w}_\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^\mathsf{T} \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_\mathbf{Q})} \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$$

It is important to now recognize that $\nabla_\mathbf{w} L\Big|_{\mathbf{w}=\hat{\mathbf{w}}+\Delta\mathbf{w}_\mathbf{Q}} \neq \mathbf{0}$ because the weights are now at $\hat{\mathbf{w}} + \Delta\mathbf{w}_\mathbf{Q}$, which is not a local optimum on the empirical loss function. However, the issue that typically people apply sparsity algorithms that only optimize for the second-order $\frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^\mathsf{T} \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_\mathbf{Q})} \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$ term, neglecting the effects of the $\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^\mathsf{T} \nabla_\mathbf{w} L\Big|_{\mathbf{w}=\hat{\mathbf{w}}+\Delta\mathbf{w}_\mathbf{Q}}$ term. While it may be assumed that it has a negligible effect, in low-bit quantization settings, $\hat{\mathbf{w}} + \Delta\mathbf{w}_\mathbf{Q}$ may be far from the true local optimum. This introduces the primary problem with applying quantization and sparsity sequentially. The objectives are no longer aligned.

We see a similar notion when applying sparsity first with a change of $\Delta\mathbf{w}_\mathbf{S}$, and then quantization after with a change of $\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}$.

$$L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_\mathbf{S} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\right) - L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_\mathbf{S}\right) \approx \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^\mathsf{T} \nabla_\mathbf{w} L\Big|_{\mathbf{w}=\hat{\mathbf{w}}+\Delta\mathbf{w}_\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^\mathsf{T} \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_\mathbf{S})} \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}$$

Again, when applying quantization algorithms, we ignore the $\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^\mathsf{T} \nabla_\mathbf{w} L\Big|_{\mathbf{w}=\hat{\mathbf{w}}+\Delta\mathbf{w}_\mathbf{S}}$ term. However, this has non-negligible effects on the change in loss.

In the previous sections, we saw that magnitude-based pruning applied after pruning sees serious accuracy degradations. However, the key issue is that of "collisions". Thus, even though magnitude-based pruning may minimize the second-order term decently well, the first-order term now is non-negligible after quantization has occurred. Thus, we want to apply magnitude-based pruning while conditioning on the fact that quantization has occurred. This motivates the idea that we should design quantization and sparsity algorithms, keeping in mind if another has occurred before it. This will serve as the primary motivation of the next section.

Now, note that we can approximate the first-order term, using another Taylor expansion. In particular, we get the following.

$$\nabla_\mathbf{w} L\Big|_{\mathbf{w}=\tilde{\mathbf{w}}+\Delta\mathbf{w}} - \nabla_\mathbf{w} L\Big|_{\mathbf{w}=\tilde{\mathbf{w}}} \approx \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta\mathbf{w}$$

Substituting this in, we get the following for the change in empirical losses in the $\mathbf{Q} \to \mathbf{S}$ case.

$$L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\right) - L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{Q}}\right) \approx \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}})}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$$

We get the following for the $\mathbf{S} \to \mathbf{Q}$ case.

$$L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\right) - L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{S}}\right) \approx \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{S}})}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}$$

Above, we see the emergence of cross terms. In particular, we see $\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}}$ and $\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}}$. This highlights the conclusion that when choosing to apply quantization and sparsity sequentially, we should also focus on their interactions. In particular, we do not want to just use the best pruning and quantization strategy without regard for the other operation. Then, aggregating the two compounded changes in loss, we get the following for the $\mathbf{Q} \to \mathbf{S}$ case.

$$\begin{aligned}
&L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\right) - L\left(\hat{\mathbf{w}}\right) \\
&\quad = \left(L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\right) - L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{Q}}\right)\right) + \left(L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{Q}}\right) - L\left(\hat{\mathbf{w}}\right)\right) \\
&\quad \approx \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}})}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}
\end{aligned}$$

Similarly, we get the following for the $\mathbf{S} \to \mathbf{Q}$ case.

$$\begin{aligned}
&L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\right) - L\left(\hat{\mathbf{w}}\right) \\
&\quad = \left(L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\right) - L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{S}}\right)\right) + \left(L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{S}}\right) - L\left(\hat{\mathbf{w}}\right)\right) \\
&\quad \approx \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{S}})}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}
\end{aligned}$$

Now, we add the following assumption.

$$\mathbf{H}_L^{(\hat{\mathbf{w}})} \approx \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}})} \approx \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{S}})}$$

This assumes that the Hessian matrix is relatively constant under perturbations of $\Delta\mathbf{w}_{\mathbf{Q}}$ and $\Delta\mathbf{w}_{\mathbf{S}}$. This is a relatively big assumption. In fact, we can see that under this assumption, we just get a Taylor expansion about $\hat{\mathbf{w}}$ with perturbations $\Delta\mathbf{w}_{\mathbf{Q}}+\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$ and $\Delta\mathbf{w}_{\mathbf{S}}+\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}$. However, it leads to a nice expression that we can then use to compare the change in empirical loss of $\mathbf{Q} \to \mathbf{S}$ and $\mathbf{S} \to \mathbf{Q}$.

$$\begin{aligned}
&L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\right) - L\left(\hat{\mathbf{w}}\right) \\
&\quad \approx \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}})}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}} && (\mathbf{H}_L^{(\hat{\mathbf{w}})} \approx \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}})})\\
&= \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\\
&= \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}}\right)^{\intercal} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\\
& && (\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} \text{ is a scalar})\\
&= \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\intercal}\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{\intercal}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\\
&= \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\\
& && (\mathbf{H}_L^{(\hat{\mathbf{w}})} \text{ is symmetric})\\
&= \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\right)^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\right)
\end{aligned}
$$

Below, we show the same derivation for the $\mathbf{S} \to \mathbf{Q}$ case.

$$
\begin{aligned}
&L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\right) - L\left(\hat{\mathbf{w}}\right)\\
&\approx \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{S}})}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\\
&= \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}} && (\mathbf{H}_L^{(\hat{\mathbf{w}})} \approx \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{S}})})\\
&= \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\\
&= \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}}\right)^{\intercal} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\\
& && (\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} \text{ is a scalar})\\
&= \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\intercal}\left(\mathbf{H}_L^{(\hat{\mathbf{w}})}\right)^{\intercal}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\\
&= \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\\
& && (\mathbf{H}_L^{(\hat{\mathbf{w}})} \text{ is symmetric})\\
&= \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\right)^{\intercal}\mathbf{H}_L^{(\hat{\mathbf{w}})}\left(\Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\right)
\end{aligned}
$$

Note that up until this point $\mathbf{Q}$ and $\mathbf{S}$ are indistinguishable. In fact, we could switch the two arbitrarily and get the same results. In particular, we can have not made any assumptions about the $\mathbf{Q}$ and $\mathbf{S}$ algorithms themselves. Now, we make the following assumptions that distinguish the two algorithms. The goal is to show that under these assumption, it is likely to see that $\mathbf{S} \to \mathbf{Q}$ is preferred over $\mathbf{Q} \to \mathbf{S}$ considering the change in model loss. First, let $\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}} = \Delta\mathbf{w}_{\mathbf{S}} + \boldsymbol{\varepsilon}$. $\boldsymbol{\varepsilon}$ represents the difference between sparsity applied before quantization and after quantization. We want to focus on $\boldsymbol{\varepsilon}$ because we have seen previously that the sparsity after quantization tends to lead to suboptimal results. Thus, we want to focus on the effect of $\boldsymbol{\varepsilon}$. We will assume that $\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}} \approx \Delta\mathbf{w}_{\mathbf{Q}}$. That is quantization changes are not very different before and after pruning. This is also a large assumption, but we think it

is reasonable. In particular, sparsity tends to remove smaller, unimportant weights. Thus, the range of the weights remains the same and the precision likely remains constant. Under these assumptions, we get the following for the change in loss under $\mathbf{Q} \to \mathbf{S}$.

$$
\begin{aligned}
&L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\right) - L\left(\hat{\mathbf{w}}\right) \\
&\quad\approx \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}} + \boldsymbol{\varepsilon}\right)^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}} + \boldsymbol{\varepsilon}\right) \\
&\quad= \frac{1}{2}\left(\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \boldsymbol{\varepsilon}\right)^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \boldsymbol{\varepsilon}\right) \\
&\quad= \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right)^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right)^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \boldsymbol{\varepsilon} + \frac{1}{2}\boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \frac{1}{2}\boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \boldsymbol{\varepsilon} \\
&\quad= \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right)^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \frac{1}{2}\left(\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right)^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \boldsymbol{\varepsilon}\right)^{\mathsf{T}} + \frac{1}{2}\boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \frac{1}{2}\boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \boldsymbol{\varepsilon} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \left(\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right)^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \boldsymbol{\varepsilon} \text{ is a scalar}\right) \\
&\quad= \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right)^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \frac{1}{2}\boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \boldsymbol{\varepsilon}
\end{aligned}
$$

We have the following for $\mathbf{S} \to \mathbf{Q}$ under the assumption that $\Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}} \approx \Delta\mathbf{w}_{\mathbf{Q}}$.

$$
L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\right) - L\left(\hat{\mathbf{w}}\right) \approx \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{Q}}\right)^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{Q}}\right)
$$

Now, we take the difference between the change in expected loss for $\mathbf{Q} \to \mathbf{S}$ and $\mathbf{S} \to \mathbf{Q}$. We want to argue that this difference is positive, which indicates that the change in loss from $\mathbf{Q} \to \mathbf{S}$ tends to be larger than $\mathbf{S} \to \mathbf{Q}$.

$$
\begin{aligned}
&\left(L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\right) - L\left(\hat{\mathbf{w}}\right)\right) - \left(L\left(\hat{\mathbf{w}} + \Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\right) - L\left(\hat{\mathbf{w}}\right)\right) \\
&\quad\approx \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right)^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \frac{1}{2}\boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \boldsymbol{\varepsilon} \\
&\qquad - \frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{Q}}\right)^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{S}} + \Delta\mathbf{w}_{\mathbf{Q}}\right) \\
&\quad= \boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \frac{1}{2}\boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \boldsymbol{\varepsilon}
\end{aligned}
$$

First, we have that $\frac{1}{2}\boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \boldsymbol{\varepsilon} \geq 0$, since $\mathbf{H}_L^{(\hat{\mathbf{w}})}$ is a symmetric matrix, and $\hat{\mathbf{w}}$ is assumed to be a local minimum, so it is positive semidefinite. Therefore, it suffices to consider the sign of $\boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right)$. We claim that the sign of this term is not fixed. Therefore, we cannot definitively say that $\mathbf{Q} \to \mathbf{S}$ is preferred to $\mathbf{S} \to \mathbf{Q}$. However, we should note that we claim that the magnitude of $\boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \left(\Delta\mathbf{w}_{\mathbf{Q}} + \Delta\mathbf{w}_{\mathbf{S}}\right) + \frac{1}{2}\boldsymbol{\varepsilon}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \boldsymbol{\varepsilon}$ is likely negligible. We expect that $\Delta\mathbf{w}_{\mathbf{S}}$ and $\Delta\mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$ to prune most of the same weights. Thus, $\boldsymbol{\varepsilon}$ should be zero for many entries where $\Delta\mathbf{w}_{\mathbf{S}}$ is non-zero. This result is quite shocking. In particular, [HCK+25] empirically showed that $\mathbf{S} \to \mathbf{Q}$ is preferred over $\mathbf{Q} \to \mathbf{S}$. However, the claim is that the difference negligible.

We should make a few notes about the analysis we have done above. We compared the

change in empirical loss between $\mathbf{Q} \to \mathbf{S}$ and $\mathbf{S} \to \mathbf{Q}$ schemes in a very general setting. In particular, we made very few assumptions about $\Delta \mathbf{w}_{\mathbf{Q}}$, $\Delta \mathbf{w}_{\mathbf{S}}$, $\Delta \mathbf{w}_{\mathbf{Q} \to \mathbf{S}}$, and $\Delta \mathbf{w}_{\mathbf{S} \to \mathbf{Q}}$. We did not really enforce that any of these were "reasonable". Instead, we made some assumptions, like $\Delta \mathbf{w}_{\mathbf{Q}} \approx \Delta \mathbf{w}_{\mathbf{Q} \to \mathbf{S}}$. We took this approach because as seen in Chapters 5 and 6 many quantization and sparsity schemes exist. We want to show that in a general setting, under some weak assumptions about the quantization and sparsity schemes, we can generally show that $\mathbf{S} \to \mathbf{Q}$ leads to similar loss degradations compared to $\mathbf{Q} \to \mathbf{S}$.

The mathematical analysis does not really tell us how to choose optimal $\Delta \mathbf{w}_{\mathbf{Q}}$, $\Delta \mathbf{w}_{\mathbf{S}}$, $\Delta \mathbf{w}_{\mathbf{Q} \to \mathbf{S}}$, and $\Delta \mathbf{w}_{\mathbf{S} \to \mathbf{Q}}$. We feel that this is an important problem that is not addressed in the literature. In particular, when people apply sequential quantization and sparsity algorithms, they choose each without regard for the other. This is why [HCK+25] sees such serious accuracy degradations for $\mathbf{Q} \to \mathbf{S}$ as opposed to $\mathbf{S} \to \mathbf{Q}$. We will explore this in the sections below.

10.5.4 Empirical Analysis and Results

In this section, we will empirically show that $\mathbf{S} \to \mathbf{Q}$ is preferred over $\mathbf{Q} \to \mathbf{S}$ under the approach presented by [HCK+25]. We should note that this analysis is slightly deceiving, and we will unpackage that in the next section. We will use AWQ as our quantization method and magnitude-based pruning as our sparsity method.

First, [HCK+25] limits their empirical analysis to naive max-scaled block-wise quantization strategies. They do also include some analysis for GPTQ, although they note that GPTQ does not fall under the conditions of their mathematical analysis. Thus, we want to extend their empirics to other non-naive methods like AWQ. Now, we perform a similar analysis to [HCK+25], but on AWQ and magnitude-based pruning at varying bitwidths and sparsity levels.

Table 10.2: Quantization and Sparsity Order

PPL↓					OPT
Precision	**Quantization Method**	**Sparsity**	**Sparsity Method**	**Order**	**125M**
FP16	N/A	0%	N/A	N/A	27.656
INT8	AWQ	10%	Magnitude	$\mathbf{S} \to \mathbf{Q}$	28.574
INT4	AWQ	10%	Magnitude	$\mathbf{S} \to \mathbf{Q}$	29.997
INT3	AWQ	10%	Magnitude	$\mathbf{S} \to \mathbf{Q}$	36.447
INT8	AWQ	25%	Magnitude	$\mathbf{S} \to \mathbf{Q}$	35.666
INT4	AWQ	25%	Magnitude	$\mathbf{S} \to \mathbf{Q}$	37.603
INT3	AWQ	25%	Magnitude	$\mathbf{S} \to \mathbf{Q}$	47.778
INT8	AWQ	10%	Magnitude	$\mathbf{Q} \to \mathbf{S}$	98.724
INT4	AWQ	10%	Magnitude	$\mathbf{Q} \to \mathbf{S}$	29.119*
INT3	AWQ	10%	Magnitude	$\mathbf{Q} \to \mathbf{S}$	35.752*
INT8	AWQ	25%	Magnitude	$\mathbf{Q} \to \mathbf{S}$	8104.065
INT4	AWQ	25%	Magnitude	$\mathbf{Q} \to \mathbf{S}$	6830.779
INT3	AWQ	25%	Magnitude	$\mathbf{Q} \to \mathbf{S}$	179.810

We see that under this implementation, $\mathbf{S} \to \mathbf{Q}$ outperforms $\mathbf{Q} \to \mathbf{S}$ rather significantly for the results without the asterisk. Thus, this validates the empirical results seen in [HCK+25]. For the results with the asterisk, like in the previous section, the threshold following quantization was already 0. Thus, $\Delta \mathbf{w}_{\mathbf{Q} \to \mathbf{S}} = \mathbf{0}$, and this was equivalent to just performing quantization. Therefore, we see that in these cases, $\mathbf{Q} \to \mathbf{S}$ outperforms $\mathbf{S} \to \mathbf{Q}$. However, we disregard these cases, as we are more interested when $\Delta \mathbf{w}_{\mathbf{Q}}$, $\Delta \mathbf{w}_{\mathbf{S}}$, $\Delta \mathbf{w}_{\mathbf{Q} \to \mathbf{S}}$, and $\Delta \mathbf{w}_{\mathbf{S} \to \mathbf{Q}}$ are non-zero.

10.6 Quantization-Aware Magnitude-Based Sparsity

In this section, using the mathematical analysis we have provided above, we propose a new magnitude-based sparsity scheme that is "quantization-aware". In particular, the method is best applied when weights have already been quantized. The motivation is as follows.

10.6.1 Motivation

Recall from our mathematical derivation above that $L(\hat{\mathbf{w}} + \Delta \mathbf{w}_{\mathbf{Q}} + \Delta \mathbf{w}_{\mathbf{Q} \to \mathbf{S}}) - L(\hat{\mathbf{w}})$ has a cross-term. In particular, we had the following.

$$L(\hat{\mathbf{w}} + \Delta \mathbf{w}_{\mathbf{Q}} + \Delta \mathbf{w}_{\mathbf{Q} \to \mathbf{S}}) - L(\hat{\mathbf{w}})$$

$$\approx \frac{1}{2}\Delta \mathbf{w}_{\mathbf{Q}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{Q}} + \Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{Q}} + \frac{1}{2}\Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta \mathbf{w}_{\mathbf{Q}})} \Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$$

The $\frac{1}{2}\Delta \mathbf{w}_{\mathbf{Q}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{Q}}$ and $\frac{1}{2}\Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta \mathbf{w}_{\mathbf{Q}})} \Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$ terms are expected. They are the second order terms from $\Delta \mathbf{w}_{\mathbf{Q}}$ and $\Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$, respectively. However, we have that the $\Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{Q}}$ term represents the cross effects. When choosing a pair of quantization and sparsity schemes, where the sparsity scheme is applied after the quantization scheme, we want to minimize $L\left(\hat{\mathbf{w}} + \Delta \mathbf{w}_{\mathbf{Q}} + \Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}\right) - L\left(\hat{\mathbf{w}}\right)$. The $\Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{Q}}$ term suggests that when choosing a sparsity scheme, we should also consider the $\Delta \mathbf{w}_{\mathbf{Q}}$ that occurred after quantization.

The same can be said for the other direction, $\mathbf{S} \to \mathbf{Q}$. We had the following for the change in loss $L\left(\hat{\mathbf{w}} + \Delta \mathbf{w}_{\mathbf{S}} + \Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\right) - L\left(\hat{\mathbf{w}}\right)$.

$$\begin{aligned} &L\left(\hat{\mathbf{w}} + \Delta \mathbf{w}_{\mathbf{S}} + \Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}\right) - L\left(\hat{\mathbf{w}}\right) \\ &\quad \approx \frac{1}{2}\Delta \mathbf{w}_{\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{S}} + \Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{S}} + \frac{1}{2}\Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta \mathbf{w}_{\mathbf{S}})} \Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}} \end{aligned}$$

We have the expected terms of $\frac{1}{2}\Delta \mathbf{w}_{\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{S}}$ and $\frac{1}{2}\Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta \mathbf{w}_{\mathbf{S}})} \Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}$. However, we also have the cross-term $\Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{S}}$. Below, we discuss the main problem with current sequential sparsity and quantization schemes.

Currently, when sparsity and quantization algorithms are applied sequentially, they are chosen independently, irrespective of the other compression scheme. For example, [HCK+25] and our empirical analysis above should be a bit dubious. We choose some optimal quantization scheme like AWQ or GPTQ, then apply some optimal sparsity scheme like magnitude-based pruning. However, these are only "optimal" with respect to the optimal weights $\hat{\mathbf{w}}$.

First, consider the $\mathbf{Q} \to \mathbf{S}$ case. We will first choose $\Delta \mathbf{w}_{\mathbf{Q}}$ that tries to minimize the second-order term $\frac{1}{2}\Delta \mathbf{w}_{\mathbf{Q}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{Q}}$. Then, we will choose $\Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$ that minimizes the second-order term $\frac{1}{2}\Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta \mathbf{w}_{\mathbf{Q}})} \Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$. However, the mathematical analysis shows us that we completely ignore the cross-term $\Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{Q}}$.

Similarly, consider the $\mathbf{S} \to \mathbf{Q}$ case. We will first choose $\Delta \mathbf{w}_{\mathbf{S}}$ that tries to minimize $\frac{1}{2}\Delta \mathbf{w}_{\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{S}}$. Then, we will choose $\Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}$ that minimizes the second-order term $\frac{1}{2}\Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta \mathbf{w}_{\mathbf{S}})} \Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}$. Again, we completely ignore the cross-term $\Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w}_{\mathbf{S}}$.

Thus, choosing these algorithms in isolation does not necessarily achieve the smallest change in loss. Thus, while our empirical analysis and that from [HCK+25] show that $\mathbf{S} \to \mathbf{Q}$ is preferred, that is under the assumption that the algorithms were chosen independently. Our mathematical analysis does not suffer from this limitation though, since it does not make any strong assumptions about the choice of $\Delta \mathbf{w}_{\mathbf{Q}}$, $\Delta \mathbf{w}_{\mathbf{S}}$, $\Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$, and $\Delta \mathbf{w}_{\mathbf{S}\to\mathbf{Q}}$. The

difference is

$$\boldsymbol{\varepsilon}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\left(\Delta\mathbf{w}_{\mathbf{Q}}+\Delta\mathbf{w}_{\mathbf{S}}\right)+\frac{1}{2}\boldsymbol{\varepsilon}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\boldsymbol{\varepsilon},$$

which we claim is likely negligible. In particular, we claim that ε likely has many zero entires making the above quantity quite small. However, in our analysis and that from [HCK+25], we see serious accuracy degradations from $\mathbf{Q}\rightarrow\mathbf{S}$. Therefore, this motivates the idea that we should try to construct sparsity algorithms that are aware that quantization has occurred before. We will show that under more cohesive algorithms, the difference between the changes in losses is truly negligible.

It is still fine to choose $\mathbf{w}_{\mathbf{Q}}$ that minimizes $\frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}}$. However, when we choose $\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}$, we should try to minimize the sum

$$\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}}+\frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}})}\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}},$$

instead of just $\frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}})}\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}$. Thus, in this section, we will try to propose a sparsity algorithm that is quantization-aware.

10.6.2 Using the Magnitude-Based Sparsity Mask From $\hat{\mathbf{w}}$

As detailed above, when choosing a sparsity scheme, we want to minimize the the sum

$$\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\Delta\mathbf{w}_{\mathbf{Q}}+\frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}})}\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}.$$

As we have discussed in previous chapters, $\mathbf{Q}\rightarrow\mathbf{S}$ is generally undesirable because of the chance of "collisions". Assuming a magnitude-based sparsity scheme, previously distinguishable weights become indistinguishable. The problem is that naive magnitude-based sparsity after quantization is good at minimizing only the $\frac{1}{2}\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}})}\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}$ term. However, it completely ignores the cross term. The naive magnitude only considers $\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}}$.

To consider the cross-term, we consider the simplification that we proposed in our mathematical analysis. In particular, we had

$$L\left(\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}}+\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}\right)-L\left(\hat{\mathbf{w}}\right)\approx\frac{1}{2}\left(\Delta\mathbf{w}_{\mathbf{Q}}+\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}\right)^{\mathsf{T}}\mathbf{H}_L^{(\hat{\mathbf{w}})}\left(\Delta\mathbf{w}_{\mathbf{Q}}+\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}\right).$$

Using this rewriting, we want to consider $\Delta\mathbf{w}_{\mathbf{Q}}+\Delta\mathbf{w}_{\mathbf{Q}\rightarrow\mathbf{S}}$ as one single change applied to the originally trained weights $\hat{\mathbf{w}}$. Thus, we propose the following scheme. Instead of using the perturbed weights $\hat{\mathbf{w}}+\Delta\mathbf{w}_{\mathbf{Q}}$ to find the mask $\mathbf{m}$, use the original $\hat{\mathbf{w}}$ instead. This will yield a mask $\mathbf{m}'$, which can then be applied to the quantized model instead of the original mask $\mathbf{m}$. This idea of "transferring" the mask is similar to the one that we discussed when considering

the relationship between quantization and sparsity. However, this is in the opposite direction. This one finds the sparsity mask from weights that are more distinguishable, then uses that to apply a mask to indistinguishable weights.

We can verify this idea empirically on OPT-125M. We first take pre-trained weights $\hat{\mathbf{w}}$. Then, we quantize it with some quantization algorithm of choice. Finally, we apply $p\%$ magnitude-based sparsity on $\hat{\mathbf{w}}$ to get a mask $\mathbf{m}'$. Then, we apply $(\hat{\mathbf{w}} + \Delta\mathbf{w}_\mathbf{Q}) \odot \mathbf{m}'$ to get the final weights. We show the results in the table below.

Table 10.3: Inverse Induced

PPL↓					**OPT**
Precision	**Quantization Method**	**Sparsity**	**Sparsity Method**	**Order**	**125M**
FP16	N/A	0%	N/A	N/A	27.656
INT8	AWQ	10%	Inverse Induced	$\mathbf{Q} \to \mathbf{S}$	28.557
INT4	AWQ	10%	Inverse Induced	$\mathbf{Q} \to \mathbf{S}$	29.882
INT3	AWQ	10%	Inverse Induced	$\mathbf{Q} \to \mathbf{S}$	36.143
INT8	AWQ	25%	Inverse Induced	$\mathbf{Q} \to \mathbf{S}$	35.641
INT4	AWQ	25%	Inverse Induced	$\mathbf{Q} \to \mathbf{S}$	37.600
INT3	AWQ	25%	Inverse Induced	$\mathbf{Q} \to \mathbf{S}$	48.934

The results strongly affirm our hypothesis and our theoretical results. Comparing Tables 10.2 and 10.3, we see that $\mathbf{Q} \to \mathbf{S}$ has the ability to outperform $\mathbf{S} \to \mathbf{Q}$ given an appropriate choice of sparsity algorithm. Furthermore, we see that the sign of

$$\boldsymbol{\varepsilon}^\top \mathbf{H}_L^{(\hat{\mathbf{w}})} (\Delta\mathbf{w}_\mathbf{Q} + \Delta\mathbf{w}_\mathbf{S}) + \frac{1}{2}\boldsymbol{\varepsilon}^\top \mathbf{H}_L^{(\hat{\mathbf{w}})} \boldsymbol{\varepsilon}$$

may not be fixed, since we see that neither $\mathbf{Q} \to \mathbf{S}$ or $\mathbf{S} \to \mathbf{Q}$ strictly dominates the other. Therefore, this gives strong evidence to the idea that sequential algorithms should be chosen with much care for the interactions between them.

10.6.3 Removing the Dependency on $\Delta\mathbf{w}_\mathbf{Q}$

There does exist one limitation with the above method. In particular, we need to keep a set of "shadow" weights $\hat{\mathbf{w}}$ even after quantization. This is less desirable, especially as models become larger. As the size of models grow, it is likely that they will be distributed in quantized fashion. Therefore, sometimes the "shadow" weights $\hat{\mathbf{w}}$ may be unattainable. Ideally, we would like the sparsity scheme to operate without $\hat{\mathbf{w}}$. This is equivalent to having $\Delta\mathbf{w}_\mathbf{Q}$, we can retrieve $\hat{\mathbf{w}}$ by subtracting $\Delta\mathbf{w}_\mathbf{Q}$ from the quantized weights $\hat{\mathbf{w}}_\mathbf{Q}$. In general,

we do not want the sparsity scheme to be based on $\Delta \mathbf{w_Q}$, since this limits the practicality of the algorithm. Therefore, we cannot minimize

$$\Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}})} \Delta \mathbf{w_Q} + \frac{1}{2} \Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}^{\mathsf{T}} \mathbf{H}_L^{(\hat{\mathbf{w}}+\Delta \mathbf{w_Q})} \Delta \mathbf{w}_{\mathbf{Q}\to\mathbf{S}}$$

directly, since it involves $\Delta \mathbf{w_Q}$.

Thus, we want to extend the method described above to this new setting where we do not have access to the original weights $\hat{\mathbf{w}}$. For some quantized weight $(\hat{\mathbf{w}}_\mathbf{Q})_i$, we want to decide whether the original version of the weight was $(\hat{\mathbf{w}}_\mathbf{Q})_i - (\Delta \mathbf{w_Q})_i$ pruned. A naive way to find this is to re-train the quantized model to the original optimal $\hat{\mathbf{w}}$. However, this would be time-consuming and undesirable. Instead, we make the following key insight. We are really only concerned about the "collisions", where $\hat{\mathbf{w}}_i < \hat{\mathbf{w}}_i$, but $(\hat{\mathbf{w}}_\mathbf{Q})_i = (\hat{\mathbf{w}}_\mathbf{Q})_j$. Therefore, we just need to separate these quantized weights. We observe that this typically takes much less training iterations than training the model to full convergence. In particular, the update $\hat{\mathbf{w}}_\mathbf{Q} - \eta \nabla_\mathbf{w} L$ will tend to move the weights towards the optimal weights. The direction of the update "undos" the rounding performed by quantization. In addition, the magnitude of the update for an individual weight $(\hat{\mathbf{w}}_\mathbf{Q})_i$ is proportional to the initial change $(\Delta \mathbf{w_Q})_i$. We can add a hyperparameter for the sparsity scheme T that indicates the number of update epochs.

10.7 Iterative Methods

10.7.1 Motivation

Inspired by the iterative idea of OBS, discussed in Chapter 6, one might try to consider an iterative approach to applying quantization and sparsity. The idea of OBS is that at each time step a single weight is pruned and the other weights are updated. Thus, at each step we incrementally prune, so that after many time steps we have pruned a reasonable amount, while still maintaining some accuracy.

Naively, one can imagine that we can do the same by alternating between pruning and quantizing. At each time step, we can prune a small amount and quantize to a number format that has smaller bitwidth. The hope is that by incrementally doing this procedure we do not have major degradations in accuracy in any given timestep. Thus, we slowly lower the model to a smaller footprint, avoiding sharp changes to the model weights. This idea takes inspiration from compiler optimizations, where a set of techniques like constant folding, constant propagation, dead code elimination (DCE), among other techniques are applied iteratively. At each step, new optimizations may arise that previously may have

been unreachable. While model compression comes from a vastly different domain, this serves as a intuitive motivation.

However, given the analysis above, it should be evident that this is unlikely to work without any fine-tuning between time steps. In particular, suppose the number of iterations $N > 1$. Then, there will be at least one sequence of $\mathbf{Q} \to \mathbf{S}$, which we have already shown to lead to significant accuracy degradations. Thus, applying this repeatedly will only lead to a worse model than applying $\mathbf{S} \to \mathbf{Q}$ exactly once. We will verify this experimentally below.

10.7.2 Results

Below, we run iterative quantization and sparsity and show that it leads to serious accuracy degradation. Again, we focus on the setting where no finetuning is done, since this allows us to accurately compare this method to other methods above.

Table 10.4: Iterative Quantization and Sparsity

PPL↓					**OPT**
Precision	**Quantization Method**	**Sparsity**	**Sparsity Method**	**Order**	**125M**
FP16	N/A	0%	N/A	N/A	27.656
[**INT8**, **INT4**]	AWQ	[10%,25%]	Magnitude	$\mathbf{S} \to \mathbf{Q} \to \mathbf{S} \to \mathbf{Q}$	4253.609
[**INT8**, **INT4**]	AWQ	[10%,25%]	Magnitude	$\mathbf{Q} \to \mathbf{S} \to \mathbf{Q} \to \mathbf{S}$	2576.735

Similar to the results in Figure 10.2, we see that iterative methods seriously degrade the performance of a language model. This confirms our hypothesis that iterative methods with fine-tuning will lead to worse outcomes than simply setting $N = 1$.

10.7.3 Future Work

As [HCK+25] notes, fine-tuning is typically necessary even after only applying magnitude-based sparsity a single time. However, if this were done iteratively, then fine-tuning at each time step may be necessary. Thus, an interesting direction of work that we propose for future work is the idea of iterative quantization and sparsity with intermediate fine-tuning. Deciding an appropriate fine-tuning schedule is likely the key question that needs to be answered. Fine-tuning should likely not occur after each iteration, but it should occur regularly. While fine-tuning is generally expensive, if the model is close enough to the optimum, then it might not be as costly. Thus, an iterative methodology may actually be desirable, since at each time step the perturbation will not be as much, since the standards required are much more lenient. However, one needs to be careful that the total fine-tuning

time does not grow too large, else it might just be better to apply quantization and sparsity in a single instance.

Part IV

Conclusion

In this thesis, we presented problems in quantization, sparsity, and reliability with respect to machine learning models. We gave mathematical formulations for each, as well as understood the current state-of-the-art solutions in each field. Then, using these insights, we presented research on better understanding the algorithms in tandem.

Through GoldenEye, we saw that reliability is dependent on the underlying quantization scheme and the choice of $\mathcal{B}$. Furthermore, we proposed a new reliability-aware quantization drawing inspiration from HAWQ and information theory. Then, in Chapter 10, we presented a mathematical analysis of the interaction between quantization and sparsity algorithms. In particular, we proposed quantization-aware sparsity, where the sparsity works well for quantized weights. Finally, through EdgeBERT, we saw a practical application where all three topics were combined, leading to dramatic speedups compared to CPU and systolic array baselines.

We believe that there is still a lot to explore at the intersection of these three areas. In particular, we have already presented future research directions for each of the contributions. Now, we take a more holistic view and propose other interesting research directions.

First, while the quantization-aware sparsity framework that was presented in Chapter 10 does much better than ordinary magnitude-based sparsity, it was still quite simple. Therefore, we believe that we can further reduce accuracy degradation by continuing to iterate on this idea. Similarly, we did not really consider sparsity-aware quantization. Sparsity before quantization already sees little accuracy degradation, so naively applying an optimal quantization algorithm after sparsity may be sufficient. However, we envision that there are likely better methods that can further protect accuracy.

Second, we mainly focused on CNNs and LLMs, but the machine learning literature has since introduced many other model variations that present their own challenges. For example, we should test the performance of quantization and sparsity algorithms on multi-modal models or diffusion models and make optimizations based on those observations. Similarly, we also may want to consider other compression schemes like low-rank approximation (LoRA) and its interactions with quantization and sparsity algorithms.

Finally, we can continue to design better hardware that supports these quickly evolving fields, similar to EdgeBERT. As Moore's Law begins to slow, specialized hardware is in high demand. Thus, we can continue to incorporate quantization and sparsity algorithms, along with other optimizations to further improve inference latency. We may also want to consider how these algorithms impact training and design more optimal hardware targeted at training time.

References

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning, 2016.

[AZL+23] Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, Veronica Melesse Vergara, Thomas Beck, Michael Brim, Reuben Budiardja, Sunita Chandrasekaran, Markus Eisenbach, Thomas Evans, Matthew Ezell, Nicholas Frontiere, Antigoni Georgiadou, Joe Glenski, Philipp Grete, Steven Hamilton, John Holmen, Axel Huebl, Daniel Jacobson, Wayne Joubert, Kim Mcmahon, Elia Merzari, Stan Moore, Andrew Myers, Stephen Nichols, Sarp Oral, Thomas Papatheodore, Danny Perez, David M. Rogers, Evan Schneider, Jean-Luc Vay, and P. K. Yeung. Frontier: Exploring exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate, 2016.

[BH19] Joseph Blitzstein and Jessica Hwang. *Introduction to Probability*. Taylor & Francis Group, 2019.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[CAG+14] Franck Cappello, Geist Al, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward Exascale Resilience: 2014 Update. *Supercomput. Front. Innov.: Int. J.*, 1(1):5–28, April 2014.

[CNF+20] Zitao Chen, Niranjhana Narayanan, Bo Fang, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications, 2020.

[CWV+14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning, 2014.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[DFE+22] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, 2022.

[DGY+74] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[DSJZ+24] Maico Cassel Dos Santos, Tianyu Jia, Joseph Zuckerman, Martin Cochet, Davide Giri, Erik Jens Loscalzo, Karthik Swaminathan, Thierry Tambe, Jeff Jun Zhang, Alper Buyuktosunoglu, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani, Luca Piccolboni, Gabriele Tombesi, David Trilla, John-David Wellman, En-Yu Yang, Aporva Amarnath, Ying Jing, Bakshree Mishra, Joshua Park, Vignesh Suresh, Sarita Adve, Pradip Bose, David Brooks, Luca P. Carloni, Kenneth L. Shepard, and Gu-Yeon Wei. 14.5 A 12nm Linux-SMP-Capable RISC-V SoC with 14 Accelerator Types, Distributed Hardware Power Management and Flexible NoC-Based Data Orchestration. In *2024 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 67, pages 262–264, 2024.

[DYG+19] Zhen Dong, Zhewei Yao, Amir Gholami, Michael Mahoney, and Kurt Keutzer. HAWQ: Hessian AWare Quantization of Neural Networks with Mixed-Precision, 2019.

[FAHA23] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers, 2023.

[FJL18] Roy Frostig, Matthew Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. 2018.

[Fly66] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[FSA23] Elias Frantar, Sidak Pal Singh, and Dan Alistarh. Optimal Brain Compression: A Framework for Accurate Post-Training Quantization and Pruning, 2023.

[Gag94] Philip Gage. A new algorithm for data compression. *C Users J.*, 12(2):23–38, February 1994.

[GAGN15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision, 2015.

[GJY11] Dongdong Ge, Xiaoye Jiang, and Yinyu Ye. A note on the complexity of L_p minimization. *Mathematical Programming*, 129(2):285–299, 2011.

[HABN+21] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.*, 22(1), January 2021.

[HCK+25] Simla Burcu Harma, Ayan Chakraborty, Elizaveta Kostenok, Danila Mishin, Dongho Ha, Babak Falsafi, Martin Jaggi, Ming Liu, Yunho Oh, Suvinay Subramanian, and Amir Yazdanbakhsh. Effective Interplay between Sparsity and Quantization: From Theory to Practice, 2025.

[HH12] David Harris and Sarah Harris. *Digital Design and Computer Architecture, Second Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2012.

[HS92] Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal Brain Surgeon. In S. Hanson, J. Cowan, and C. Giles, editors, *Advances in Neural Information Processing Systems*, volume 5. Morgan-Kaufmann, 1992.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[HvC93] Geoffrey E. Hinton and Drew van Camp. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the Sixth Annual Conference on Computational Learning Theory*, COLT '93, pages 5–13, New York, NY, USA, 1993. Association for Computing Machinery.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, 2015.

[IS15] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015.

[JYP+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit, 2017.

[KCN+22] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. The Optimal BERT Surgeon: Scalable and Accurate Second-Order Pruning for Large Language Models, 2022.

[KHG+24] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. SqueezeLLM: Dense-and-Sparse Quantization, 2024.

[KNH+14] Alex Krizhevsky, Vinod Nair, Geoffrey Hinton, et al. The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55(5):2, 2014.

[KR18] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing, 2018.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[Kun82] H. T. Kung. Why systolic architectures? *Computer*, 15:37–46, 1982.

[LBBH98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[LBOM12] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus Robert Müller. *Efficient BackProp*, pages 9–48. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer Verlag, 2012. Copyright: Copyright 2021 Elsevier B.V., All rights reserved.

[LCG+20] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations, 2020.

[LDS89] Yann LeCun, John Denker, and Sara Solla. Optimal Brain Damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989.

[LGT+21] Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. BRECQ: Pushing the Limit of Post-Training Quantization by Block Reconstruction, 2021.

[LTT+24] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration, 2024.

[LWK18] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning Sparse Neural Networks through L_0 Regularization, 2018.

[Mah20] A. Mahmoud. Towards scalable and specialized application error analysis, 2020.

[MAN+20] Abdulrahman Mahmoud, Neeraj Aggarwal, Alex Nobbe, Jose Rodrigo Sanchez Vicarte, Sarita V. Adve, Christopher W. Fletcher, Iuri Frosio, and Siva Kumar Sastry Hari. PyTorchFI: A Runtime Perturbation Tool for DNNs. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 25–31, 2020.

[MGDG+20] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. Agile soc development with open esp. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, pages 1–9. ACM, November 2020.

[MHL14] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast Training of Convolutional Networks through FFTs, 2014.

[Mil94] George A. Miller. WordNet: A lexical database for English. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*, 1994.

[Moo65] G.E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics Magazine*, 38:114–117, 1965.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[MSHF+21] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Christopher W. Fletcher, Sarita V. Adve, Charbel Sakr, Naresh Shanbhag, Pavlo Molchanov, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. Optimizing Selective Protection for CNN Resilience. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 127–138, 2021.

[MSM93] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[MTA+22] A. Mahmoud, T. Tambe, T. Aloui, D. Brooks, and G. Wei. GoldenEye: A Platform for Evaluating Emerging Data Formats in DNN Accelerators. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022.

[MXBS16] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[NAvB+20] Markus Nagel, Rana Ali Amjad, Mart van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or Down? Adaptive Rounding for Post-Training Quantization, 2020.

[NCB+20] Yury Nahshan, Brian Chmiel, Chaim Baskin, Evgenii Zheltonozhskii, Ron Banner, Alex M. Bronstein, and Avi Mendelson. Loss Aware Post-training Quantization, 2020.

[Pea94] Barak A. Pearlmutter. Fast Exact Multiplication by the Hessian. *Neural Computation*, 6(1):147–160, 01 1994.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019.

[Pow64] M. J. D. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The Computer Journal*, 7(2):155–162, 01 1964.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[RGP+18] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. Ares: A framework for quantifying the resilience of deep neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[Ris78] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[Ros58] Frank Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65:386–408, 1958.

[RSR+23] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, 2023.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI*, 2019. Accessed: 2024-11-15.

[SGBJ19] Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive Attention Span in Transformers, 2019.

[Sha48] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[SHB16] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural Machine Translation of Rare Words with Subword Units, 2016.

[SHW57] D. B. Sumner, Hirschmann, and Widder. The Convolution Transform. 1957.

[SK16] Tim Salimans and Diederik P. Kingma. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks, 2016.

[SLP+23] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced Transformer with Rotary Position Embedding, 2023.

[SN12] Mike Schuster and Kaisuke Nakajima. Japanese and Korean Voice Search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152, 2012.

[SSS+21] Xinying Song, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou. Fast WordPiece Tokenization, 2021.

[SWR20] Victor Sanh, Thomas Wolf, and Alexander M. Rush. Movement Pruning: Adaptive Sparsity by Fine-Tuning, 2020.

[SZ15] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2015.

[TARA+16] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A Python framework for fast computation of mathematical expressions, 2016.

[TFF08] Antonio Torralba, Rob Fergus, and William T. Freeman. 80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1958–1970, 2008.

[THP+21] Thierry Tambe, Coleman Hooper, Lillian Pentecost, Tianyu Jia, En-Yu Yang, Marco Donato, Victor Sanh, Paul N. Whatmough, Alexander M. Rush, David Brooks, and Gu-Yeon Wei. EdgeBERT: Sentence-Level Energy Optimizations for Latency-Aware Multi-Task NLP Inference. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 2021.

[TYW+20] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. AdaptivFloat: A Floating-point based Data Type for Resilient Deep Learning Inference, 2020.

[TZH+23] Thierry Tambe, Jeff Zhang, Coleman Hooper, Tianyu Jia, Paul N. Whatmough, Joseph Zuckerman, Maico Cassel Dos Santos, Erik Jens Loscalzo, Davide Giri, Kenneth Shepard, Luca Carloni, Alexander Rush, David Brooks, and Gu-Yeon Wei. A 12nm 18.1TFLOPs/W Sparse Transformer Processor with Entropy-Based Early Exit, Mixed-Precision Predication and Fine-Grained Power Management. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 342–344, 2023.

[VSP+23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, 2023.

[Wor23] BigScience Workshop. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model, 2023.

[WSM+19] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding, 2019.

[XTL+20] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference, 2020.

[YES24] Yifan Yang, Joel S. Emer, and Daniel Sanchez. Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 931–945, 2024.

[YGKM20] Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael Mahoney. PyHessian: Neural Networks Through the Lens of the Hessian, 2020.

[ZRG+22] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models, 2022.

This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. The above illustration, "Science Experiment 02", was created by Ben Schlitter and released under CC BY-NC-ND 3.0. A template that can be used to format a PhD thesis with this look and feel has been released under the permissive MIT (XII) license, and can be found online at github.com/suchow/Dissertate or from its author, Jordan Suchow, at suchow@post.harvard.edu.