Learning to learn in quantized-aware training

Gil Denekamp

Learning to learn in quantized-aware training

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Gil Denekamp

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tamuz 5785 Haifa July 2025

This research was carried out under the supervision of Prof. Daniel Soudry, in the Faculty of Electrical and Computer Engineering.

Acknowledgements

I would like to thank my advisor, my parents, my friends, etc. etc.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

In recent years, Deep Neural Networks (DNNs) transformed countless fields, among them computer vision, natural language processing, and autonomous systems. Both the training and the deployment of DNNs require substantial computational resources to deliver accurate and general results. This resource intensity poses a significant challenge when deploying these models on edge devices with limited capabilities. One important technique to alleviate these constraints is Neural network quantization (or simply Quantization). These approaches aim to reduce the numerical precision of the model parameters from high-precision floating point (typically using 32-bit storage) to lower bit integer formats, represented by down to 1 bit per parameter. These can significantly enhance the computational efficiency, memory utilization as well as power consumption of DNN applications while maintaining great performance.

An important approach for quantization is Quantized Aware Training, which simulates the reduced precision during training, allowing the model to adapt to quantization errors. This proved to be a very effective way to improve the accuracy of quantized models in cases where there is access to the training data and computational resources. The major inherent problem for QAT is the non-differentiability of the quantization function, which is a staircase function with discrete levels. This does not integrate well with gradient-based optimization, typically used for learning, which will see the gradient through the quantization layer as mostly zero. This is traditionally mitigated by using a surrogate gradient estimator for the quantization layer, like the simple and widely used Straight-Through-Estimator (STE). In many cases, the standard STE has shown empirical success, but for extremely low-bit quantization scenarios, it suffers from a ”gradient mismatch” problem that negatively affects the optimization performance.

This thesis addresses this fundamental limitation by developing a framework that dynamically learns an optimal STE function within the QAT process itself. We draw inspiration from meta-learning techniques and propose a family of parameterized surrogate gradients, which we term general STE (GSTE). We optimize their parameters through gradient-based optimization. Several novel GSTE parameterization methods are introduced and analyzed: Scaling GSTE (SGSTE), Piecewise Linear GSTE (PWLGSTE), and Dual PWLGSTE (DPWLGSTE). We will show that the problem of learning the GSTE has unique complications that differ from previous metalearning works on learning parameters of the optimizer. We derive multiple analytic and heuristic optimization strategies that overcome the challenges. Additionally, the thesis explores the theoretical foundations of the STE. We show that the classical STE is a special case of centred finite differences, providing insights into its nature.

Experimental results show improvement in model accuracy in various cases, particularly in ultra-low precision quantization. The outcome shows the potential of adaptive, learned gradient estimators in reducing optimization error, but it comes with some computational overhead.

Chapter 1

Introduction

Deep Neural Networks

Deep neural networks (DNNs) have achieved remarkable success in a variety of revolutionary technologies, such as image recognition, natural language processing, and autonomous systems [1, 2, 3]. These advances have been driven by increasingly complex models trained on large datasets that require substantial computational power and memory. The high resource requirements of DNNs present significant challenges, especially for deployment on edge devices with limited capabilities [4]. Some applications, like smartphones, home appliances, and drones, may place a strict limit on power consumption, which is essential for extending battery life. Other applications like self-driving cars and robots, require a fast response time for operation in real time [5] [6].

Studies [7, 8] have shown that DNNs exhibit a high degree of redundancy in their structure, opening opportunities for optimization. Various methods have been proposed to address these inefficiencies during the training and inference stages. Some prominent techniques include model pruning, Knowledge distillation and Neural Network Quantization. Model pruning is concerned with identifying and removing less important weights whose contribution to the output is negligible, yielding lighter models without retraining from scratch [8]. Knowledge distillation transfers knowledge from a large, complex “teacher” model to a smaller, simpler “student”, inheriting much of the teacher’s accuracy at a fraction of its size [9]. Among these, quantization, which aims to represent network parameters and activations with a lower bit width representation, stands out as a powerful method for reducing resource demands while maintaining model accuracy. Shrinking the word length by a factor of x cuts the memory footprint linearly ($\times 1/x$) and can reduce the matrix multiplication latency and energy by approximately x^2 [6]. Quantization also benefits from decades of research before machine learning in digital signal processing, and it can be used in conjunction with other methods to improve performance even further.

Neural Network Quantization

The main goal of *quantization* is to reduce the numerical precision used to represent the weights and activations in a neural network, typically replacing 32-bit floating-point (FP32) values with lower-bit integer formats. Because most computations in neural networks involve matrix multiplications that combine these weights and activations, lowering the bit-width delivers performance gains in three complementary ways. First, the most obvious improvement is reducing the amount of memory needed for storing the network. This has a less obvious positive effect on execution time by reducing the amount of data that needs to be fetched from memory. This is often a significant bottleneck, more important than the arithmetic itself [4, 10]. Second, lower-bit multiplication involves fewer arithmetic operations at the hardware level, with less chip area and energy needed. Theoretically, multiplication cost scales roughly with the square of the bit width, which was also observed empirically [11]. Third, floating-point multiplication involves exponent and mantissa calculations, which are more power-intensive than simple integer multiplications. We can take advantage of quantization to convert to a more efficient format [6, 12]. Taken together, these effects mean that on modern specialized hardware such as TPUs and GPUs, quantization will present a significant memory, energy, and time boost.

Quantization methods generally fall into two categories: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). PTQ converts a pre-trained high-precision FP32 neural network directly into a low-bit integer format. Without using a full training pipeline and often with only a handful of unlabeled calibration data or none at all. There are usually two key steps in PTQ: calibration and rounding [6]. The calibration step determines the appropriate quantization parameters, such as scaling factors and clipping thresholds. A naive approach would be to pick the representable range to be between the maximal and minimal values in the tensor. A more sophisticated data-driven approach is to optimize according to a mean-squared-error (MSE) criterion that minimizes the MSE between the original and the quantized tensor on a small dataset without updating the weights themselves [13]. The second step in PTQ is to round the network parameters according to the quantization hyperparameters found in calibration. The naive standard approach is to round to the closest integer. While "Round-to-nearest" is the default, there are often better alternatives. For example, Stochastic rounding injects randomness that preserves unbiasedness [12].

Many PTQ frameworks exist that try to push accuracy much further. ADAROUND [14] learns a continuous relaxation of the rounding mask applied in the rounding steps using a small calibration set and gives impressive results, often recovering FP32 accuracy at 4 bits. The more recent ADAQUANT [15] method improves ADAROUND by learning the quantization hyperparameters for the calibration step via a slightly improved optimization method. This method enables quantization up to 2-3 bit widths while adding activation quantization. A different approach is PTQ for cases where there is no available calibration data, for example, applications with sensitive or proprietary training data. The ZeroQ [16] approach showed very impressive results by synthesizing a “distilled” dataset that matches batch-normalization statistics.

PTQ offers several practical advantages over QAT. First, it uses a pre-trained FP32 network, skipping the expensive retraining involving a costly back-propagation loop entirely. This allows quantizing a model in much less time and computational effort. Second, it does not require a large dataset, which is often unavailable, particularly in privacy-sensitive applications. Third, the PTQ simple calibration plus rounding workflow introduces few hyperparameters. Allowing PTQ to be used as a simple black box solution via an API, making quantization more accessible. Despite all this, PTQ still has a large drawback by not changing the weights themselves. This leads to lower top-1 accuracy compared to QAT, especially in ultra-low precision quantization. Consequently, PTQ can be viewed as an efficient solution for scenarios where retraining is infeasible or where a modest accuracy drop is acceptable.

Quantization Aware Training

Quantization Aware Training (QAT) is a method that integrates quantization into the training process of a neural network. Unlike Post-Training Quantization (PTQ), which applies quantization after the model has been trained, QAT simulates quantization during training itself. Essentially mimicking the behavior of quantized inference during training, this allows the network to adapt its weights and activations to the quantization noise, resulting in a model that is inherently robust to the reduced precision.

The way QAT simulates quantization during training is by introducing “quantization layers” which are layers that apply fake quantization in the sense that they round the values to discrete levels while keeping the underlying variables in full precision. For example, a quantization layer can map floating-point values to 256 levels for 8-bit quantization, but still store them with 32 bits. In **??**, we see where the quantization layer is applied in cases where we only quantize the weights. The Quantization layer is always some type of staircase function applied during the forward pass to the values of weights and activations before further calculations (such as matrix multiplication), therefore simulating the quantized form for the calculation of the loss.

The quantization operation is inherently not differentiable, this presents a major challenge for QAT methods. As a result, usually during the backward pass, gradients are calculated in a way that bypasses the quantization layer using various techniques that we will later discuss to update the network parameters. The gradients themselves usually are not quantized, and the update is done to a full-precision weight.

An early example that showed the promise of QAT for ultra-low-quantization is Binarize Neural Networks (BNN). In the BNN paper, they achieved competitive accuracy with both weights and activations constrained to $-1, +1$[17]. Later, DoReFa-Net [18] suggested to additionally quantize the gradients, accelerating the training speed.

Subsequent works focused on learning the quantizer itself. This highlights a benefit of QAT, which enables dynamic quantization that learns quantization parameters on the fly from the data. Contrary to PTQ methods, which either set the parameters manually or use a small calibration set. Learned Step Quantization (LSQ) [19] treats the scale factor, which determines the size of the quantization bin, as a trainable parameter and achieves high performance on 3-4 bits. PACT [20] is an example that optimises the activation-clipping threshold similarly.

Empirical studies [21] still show that once the target precision drops below roughly four bits, QAT almost always surpasses post-training quantization in top-1 accuracy, and it remains the only successful way to get sub-2-bit and fully binary models. Because it is commonly the case that the machine and data that trained the baseline model are available to whoever wants to quantize it, the extra compute required for a QAT fine-tuning run is often a worthwhile trade-off for the accuracy it recovers.

This thesis focuses on the QAT paradigm and particularly on its most dominant problem: how to pass reliable gradients through the non-differentiable quantization layer? This is key for bridging the accuracy gap between full-precision models and ultra-low-precision networks.

Straight Through Estimator

One of the central challenges in QAT is that the quantization layer is a non-differentiable Staircase function. Therefore, their gradient is zero almost everywhere and undefined at the transition points. This makes them incompatible with standard gradient-based backpropagation since zero-valued gradients prevent any updates to the network parameters.

To overcome this obstacle, the chain rule is typically modified to use a surrogate hand-crafted gradient, which will no longer represent the true mathematical gradient of the function but a “pseudo gradient”. The earliest, simplest, and still widely used approach is the Straight-Through Estimator (STE) introduced by Bengio et al. [22]. Under this ’standard STE’, the forward pass applies a hard round quantifier, while the backward pass propagates the gradient unchanged as if the quantization layer did not exist. This "pseudo gradient" approximates the staircase function to be a linear identity mapping, which results in a constant gradient with the value 1. Some versions of the standard STE include clipping of the gradient between the quantization clipping bounds so that weights that were clipped in the forward pass will not receive a gradient. This is visualized in Figure 2.1b, where we see the STE pseudo gradient function in the backward operation.

Standard STE became a common solution because of its surprising empirical effectiveness despite its simplicity, in some cases getting close to the full precision models' performance [17, 18]. But its extreme simplicity and heuristic origin come with some problems. Several studies argued that although STE gives good enough results in cases of moderate quantization (up to 4 bits), for lower precision, where the non-differentiability becomes more significant, we have a "gradient mismatch" problem [23, 24]. This means the surrogate hand crafted pseudo gradient design is no longer a good approximation and does not represent faithfully the quantization layer, making the wrong updates to the network parameters.

Ideally, we would want the resulting network after quantization to give equivalent, if not superior, results to the full precision model. In our work, we distinguish between two main sources of error in QAT that prevent us from achieving the ideal: *approximation error* is the intrinsic error caused when we try to represent a continuous value with a discrete amount of steps. This will always be the case when we try to reduce the amount of bits that represent a variable, we can only address that by choosing wisely the bit precision we quantize to, to avoid misrepresenting in a way that destroys the results. *Optimization error* stems from the inability to converge to the correct value. For example, if a weight was set to 1, but a 0 value would give a smaller overall loss. This error is what we aim to minimize as much as possible in QAT using backpropagation to update the weight values. Using inaccurate gradient updates, as we do with STE, substantially increases this error. A relevant example presented in [24] is that given full precision values 0.51 and 1.49, after the quantization layer, both will be mapped to 1. Therefore, when the gradient is calculated using STE, both will get exactly the same update with no regard to how the quantization itself affected the original value. This is clearly the wrong way to treat them since a small increment to 1.49 will change its discrete value to 2 whereas for 0.5 it will not. This shows how STE induces substantial optimization error, suggesting it is the single biggest obstacle to achieving the ideal of FP32 performance with low-bit quantized models.

Several notable works tried to improve the standard STE. One approach suggests using a soft differentiable approximation to the step function to derive a more appropriate surrogate gradient. For example, Differentiable Soft Quantization (DSQ) [25] replaces the hard steps with a smooth function such as sigmoid or hyperbolic tangent controlled by a temperature parameter. This temperature parameter is increased during training to make the approximation less soft as the training process progresses. Such methods provide a more accurate gradient, but it is still decided heuristically. Another interesting approach demonstrated in [26] is adding a controlled amount of noise (typically Gaussian) to the quantization process. This is intended to mimic the uncertain effects of quantization and smooth the loss landscape. The amount of noise is gradually reduced, making it more robust and similar to the hard quantization towards the end. One promising approach involved updating the STE approximation based on the data. In EWGS [24], they used a surrogate gradient based on the STE with a scaling factor that is based on the distance between the latent weight (original weight) and its nearest quantized level. The motivation is that when a weight is far from a threshold, the "error" induced by quantization is greater, so the gradient should reflect that. This approach proved to be quite effective in ultra-low quantization scenarios.

Many past works in the field rely on heuristic solutions that try to minimize the quantization error as much as possible and require very delicate hyperparameter tuning. One interesting work [27] suggests that all these heuristic gradient replacements are Equivalent to the Straight through estimator, and all of their success can be achieved with correct weight initialization and learning rate. This indicates that the popular approaches do not actually help us toward lowering the quantization error compared to the standard STE. Importantly, this paper does not rule out methods such as EWGS that adjust dynamically based on the data itself. This is a clear motivation for our optimization-driven approach that dynamically learns the STE replacement.

Finite Differences and STE motivation

A less explored approach for estimating the gradient in the context of QAT is the Finite Differences (FD) numerical technique. FD is used to approximate the derivative of a function directly using its values rather than its analytical form. The basic idea is that to find the derivative of a function $f(x)$ at the point x, we can calculate the ratio between the change in the function value to the change in the input x. So that for an increment h, we get:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{1.1}$$

This is one of several possible formulations. It is often used in applications such as optimization [28, 29], numerical discretization of differential equations [30], and computational fluid dynamics [31], where finding the gradient for a complex or unknown function is required.

Motivation for the STE, Originally introduced as a purely heuristic workaround [32], the STE has since prompted attempts at theoretical justification. In [33], they showed that on a small, simple network, the“coarse gradients” generated by STE correlate well with the true population gradient. Lee et al. [24] quantified the bias and optimization errors that arise from the STE, while [25] interpreted the STE as the infinite temperature limit of a smooth tanh.

In section **??** we derive a new equivalence between the STE and a centered finite Differences estimator, proving that, for a suitable perturbation step h, the STE is mathematically identical to the FD gradient. This gives a new intuition for the STE behavior and opens a new direction for understanding the limitations and improving the STE through a well-researched tool like FD. This also motivates the GSTE parameterizations we introduce later in the thesis.

Meta Learning

The problems we described with both the STE and newer surrogates motivate us to look for less heuristic options, bringing us to the main idea behind our approach. Since the STE takes the backpropagated loss gradient at the quantizer's output and turns it into a different pseudo gradient with respect to the full-precision weight, it is simply a gradient transformation. Just as an SGD optimizer rescales the genuine weight gradient by a global learning rate, the STE reshapes the incoming gradient according to its surrogate rule. In both cases, you're choosing how to transform a loss gradient into a better update, so picking or tuning your surrogate is directly analogous to selecting optimizer hyperparameters, which is a rich field of study. Several Optimization-based methods for finding those hyperparameters were suggested under the concept of Meta learning. Meta learning can be simply described as “Learning to learn”, meaning that the goal is to find algorithms that can automatically improve their own learning process. In our work, we want to find a procedure that adapts the STE and learns the optimal shape for it based on the data as part of the regular training using backpropagation.

One of the early and significant works in meta learning is [34], which suggested learning the process of optimization. They train the optimizer as a separate LSTM model that is updated using gradient descent. Essentially, back-propagating from the loss through the update rule, finding a step size that is data-dependent. Modeling the optimizer as a recurrent neural network lets it adapt based on gradients from the past, discovering strategies such as an adaptive learning rate.

A subsequent important work is [35], which rediscovered a method from [36], where they update the parameters of the optimizer using standard gradient descent. They use the term “hyper-gradient” to refer to derivatives taken with respect to a hyperparameter. They showed that one can compute and differentiate the loss with respect to the learning rate at every iteration, updating the hyperparameter online, without using any inner optimization with multiple iterations as done before. There they lay the mathematical foundations we use in our work, which we will describe later with more detail **??**. Chandra et al. [37] advanced the idea further, noting that a method based on manually calculating a gradient for every type of optimizer and each of its parameters is tedious and error-prone. Also, crucially, gradient-based hypergradient calculation introduces a new hyperparameter that chooses a step size for the update of the original hyperparameter (e.g., a learning rate for the learning rate). Their solution is using automatic differentiation, which eliminates the need for hand-derived hyper-gradients. They also observe that with automatic differentiation, they can optimize the newly introduced hyper-hyper-parameters, the hyper-hyper-hyper-parameters, and so on. Building such optimization towers theoretically removes the need for hyperparameter tuning. They find that as the optimization tower grows, it becomes more robust to a poor choice of hyper-parameters.

Going back to QAT, we ask whether an STE can be parameterized and meta-learned in the same fashion. A similar attempt has been made in MetaQuant [38], where they replaced the STE with a neural network whose weights are dynamically learned by differentiation through the update rule. Despite claiming otherwise, this method still relies heavily on the STE for the updates of the neural network intended to replace it, therefore inheriting a similar bias and gradient mismatch.

Our Work

In our work, we dive into the problem of learning the optimal STE replacement using gradient descent jointly with the network weights. We suggest several ways to create a learnable surrogate gradient by inserting a parameter into popular methods for estimation. We term this family of functions *Learnable Straight Through Estimator* (LSTE). We first suggest *Scaling GSTE* (SLSTE), its simplicity lets us derive an exact analytical update rule later on. Another approximation we explore uses a smoother approximation for the quantization function named *Piecewise-linear LSTE* (PLSTE). We extend it with *Dual-parameter LSTE* (DLSTE) that illustrates how LSTE can handle multiple learnable parameters. Along the way, we provide a new intuition for the STE by showing its equivalence with the centered finite-differences method for specific cases.

We develop several methods to optimize the LSTEs as part of the network training. We discover that meta learning the optimal gradient approximation is more challenging than tuning conventional optimizer hyperparameters. This is because we modify the weight gradient itself and not just the update rule treatment of it. We develop a full analytical solution for the update rule of SLSTE. We then propose several heuristic optimization techniques for more general cases that do not require derivation by hand, such as *Future-Aware Gradient Update* (FAGU), *Final Loss Gradient Update* (FLGU), and *Delayed Updates* (DU). We suggest several tricks for implementing these methods in a simpler and more generalized way.

We benchmark every suggested GSTE–optimizer combination against the vanilla STE on 1–3-bit weight quantization. The learned surrogates surpass the standard STE at 2 and 3 bits and yield the largest gains for 1-bit quantization.

The contributions of this thesis are:

- **Theoretical** - A proof for the equivalence between STE and FD for specific perturbation h, providing context and insight to the capabilities of the STE.
- **Learnable Surrogates for the STE** - We propose SGSTE, PWL GSTE and DPWL GSTE.
- **Optimization** - Several optimization methods for the GSTEs: Analytical, ATT, LGATT and DU. As well as implementation descriptions and tricks.
- **Empirical results** - Demonstrate that GSTE variants improve accuracy over standard STE on multiple settings.

Chapter 2

Preliminaries

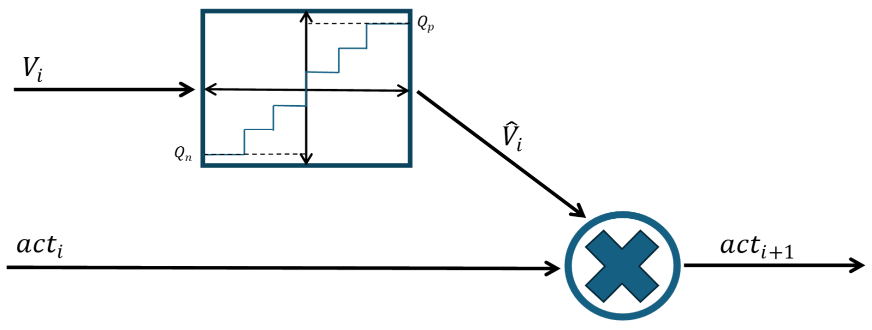
Before introducing our GSTE framework, we should first explain the basic notation and definitions needed. Section **??** explains and formalises symmetric uniform weight quantization for QAT, establishing the notation we will use later. Section **??** then introduces gradient-based hyperparameter learning, showing how hypergradients are typically computed. This technique is the basis for our method to adapt surrogate gradients.

2.1 Formalizing quantization

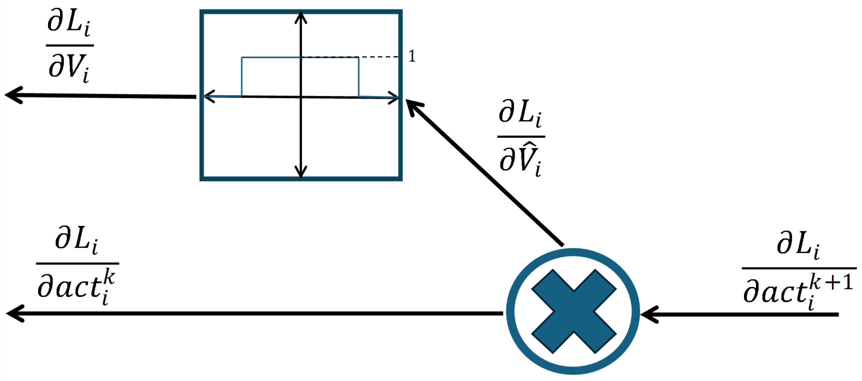
Our goal in quantization is to have a DNN that operates at inference time using low-bit precision weights. We are using the QAT approach, which requires a quantization layer. The way we define this quantization layer is the key to lowering the quantization error of the network. Therefore, we will start by discussing the main design choices and then describe the formula itself.

In our work, we focus on the quantization of the weights as opposed to schemes that apply quantization on the activations as well. We work with uniform quantization, which means that every quantization bin has an equal size. It is the most common scheme because it allows a simple and efficient implementation for floating-point arithmetic. We use symmetric signed quantization, which means the quantization function is centered around 0. Asymmetric quantization can be more expressive, but it also introduces significant computational overhead [6]. The final design choice for the quantization layer is using per-tensor quantization, which means every weight gets its own quantization hyperparameters.

Let $v \in \mathbb{R}$ denote a full-precision weight and $\hat{v}$ its quantized counterpart. For a b-bit



(a)


$\frac{\partial L_i}{\partial V_i}$
1
$\frac{\partial L_i}{\partial \hat{V}_i}$
$\frac{\partial L_i}{\partial act_i^k}$
$\frac{\partial L_i}{\partial act_i^{k+1}}$

(b)

Figure 2.1: Quantization-aware training visualized in some layer at step i: (a) fake-quantization in the forward pass, and (b) STE in the backward pass.

signed representation we have:

$$Q_p = 2^{b-1} - 1, \qquad Q_n = 2^{b-1}$$

Denoting the number of positive and negative quantization levels, giving 2^b total levels that are symmetric around zero. We introduce a step size (or scale) $s > 0$ that can be learned using the LSQ method [19]. We define

$$\hat{v} = \lfloor \text{clip}(\frac{v}{s}, -Q_n, Q_p) \rceil \times s \tag{2.1}$$

where $\text{clip}(x, l, h) = \min\{\max\{x, l\}, h\}$ and The operation $\lfloor x \rceil$ returns the nearest integer to x. Using a learned s value, we can balance between errors coming from the clipping and rounding operations. Increasing s widens the representable range and reduces clipping, but also enlarges the rounding step. This is the reason we use LSQ in later experiments when we quantize to 2-bit and above.

During back-propagation we need to use some custom STE to pass through the quantization function. The common STE can be formalized as follows:

$$\frac{\partial \hat{v}}{\partial v} = \begin{cases} 1 & \text{if } -Q_n < \frac{v}{s} \leq Q_p \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

This means the gradients pass unchanged when the forward quantization does not clip, otherwise, they will not pass. This STE formulation (visualized in Figure **??**)is the most common one. It evolved from the version used by Bengio et al. [32] to have clipping, as already used in Hubara et al. [17]. The main intuition for STE is that it is the gradient of a linear clipped function (visualize in Figure**??**), which is supposed to be a simple differentiable approximation for the staircase function. This approximation gets worse as we use fewer steps in the quantization function, which explains why the STE becomes a major source of error for low bit widths. This is where we later come in with GSTE in the following chapters.

Now that we have the basic quantization tools, we will turn to discussing the method for learning hyperparameters of the optimizer, which is the building block for our optimization.

2.2 Learning the hyperparameters

In this section, we present the framework from [39, 37] that enables optimizing any scalar hyperparameter used in the weight update rule. We will use the SGD optimizer in our derivation for simplicity. Let v_i denote the network weights at the beginning of step i, $f(v_i)$ the training loss calculated using weight v_i, and μ the learning rate. The standard SGD update rule is:

$$v_{i+1} = v_i - \mu \frac{\partial f(v_i)}{\partial v_i} \tag{2.3}$$

We can take a small detour to understand the relation of learning the learning rate here to the STE. If we open further this equation with notation we described in the previous section we get:

$$v_{i+1} = v_i - \mu \frac{\partial f(v_i)}{\partial \hat{v}_i} \frac{\partial \hat{v}_i}{\partial v_i}$$

where $\frac{\partial \hat{v}_i}{\partial v_i}$ is the STE, so we get that it multiplies the expression $\frac{\partial f(v_i)}{\partial \hat{v}_i}$ similarly to the learning rate. Therefore both the STE and the learning rate have similar effects on the update rule.

With this motivation at hand, we will look for a way to update μ according to the loss. We will define μ_i as the learning rate at the beginning of step i. We will perform an SGD step on the learning rate, followed by an update on the weights:

$$\mu_{i+1} = \mu_i - \kappa \frac{\partial f(v_i)}{\partial \mu_i}, \tag{2.4a}$$

$$v_{i+1} = v_i - \mu_{i+1} \frac{\partial f(v_i)}{\partial v_i}, \tag{2.4b}$$

where $\kappa > 0$ is a second order learning rate that controls how fast μ is changing.

What remains to address is how to compute the loss gradient of μ_i. A manual approach to calculating this was developed by Baydin et al. [39] and used by Chandra et al. [37]:

$$\begin{aligned} \frac{\partial f(v_i)}{\partial \mu_i} &= \frac{\partial f(v_i)}{\partial v_i} \frac{\partial v_i}{\partial \mu_i} \\ &= \frac{\partial f(v_i)}{\partial v_i} \frac{\partial}{\partial \mu_i} \Big(v_{i-1} - \mu_i \frac{\partial f(v_{i-1})}{\partial v_{i-1}} \Big) \\ &= - \frac{\partial f(v_i)}{\partial v_i} \frac{\partial f(v_{i-1})}{\partial v_{i-1}} \end{aligned} \tag{2.5}$$

The first equality is using the chain rule, the second equality substitutes 2.4b, and the third observes that v_{i-1} and $f(v_{i-1})$ do not depend on μ_i. The gradient we get is quite intuitive since it appears to increase when two successive gradients agree in direction and shrink when they oppose each other.

We were able to easily derive this gradient because we used the simple SGD optimizer, but if we had used a more complicated one (e.g. Adam), the derivation would not be this simple. This manual approach quickly becomes tedious and error-prone, so Chandra et al. [37] proposed a general approach that can automatically find the gradients for arbitrarily complicated optimizers with several hyperparameters. Modern deep learning frameworks already record the entire computation graph, so we can treat μ_i as a variable and obtain $\partial f / \partial \mu_i$ for free via reverse-mode Auto Differentiation. This is done via a small change to the optimizer, allowing gradients to be tracked through the update rule using the *detach* operation in PyTorch. In a later chapter **??**, we will elaborate more and explain how this is used in our context.

Chapter 3

GSTE parameterizations

Now we turn to the central task of optimizing a parameterized STE during backpropagation. This problem naturally splits into two parts. The first, which we tackle in this chapter, is to define a parameterized STE replacement whose parameters will later be learned. The second, which we discuss in the next chapter **??**, is to find the specific update rule to update those parameters. We will see that both problems are subtle, requiring novel solutions and managing several tradeoffs. Additionally, at the end of this chapter, we will look into the relation between Finite Differences and STE in section **??**.

For the task of designing a good parameterized STE replacement, which we term *General Straight Through Estimator* (GSTE), we identify several guiding principles. First, we want the GSTE to initialize from some standard STE. This will help us compare whether we can learn from the same initial condition a better approximation. Second, the parameterization should be general enough to describe a wide spectrum of GSTE shapes so that learning can discover the best suited form for the data. Thirdly, the possible shapes for the GSTE should be logical in terms of being a good approximation of the gradient of the staircase function. This will help the learning avoid extreme overfitting. Last, we want it to be simple with few parameters to minimize the computational overhead introduced. Simplicity is also important for the development of the analytical optimization method from chapter**??** that is derived manually.

3.1 Scaling GSTE

We begin with a simple parameterization that stays close to the standard STE. It is handy for deriving an analytical solution later on and serves as a straightforward comparison to the standard STE. Let a be a learnable scalar initialized to 1. We define the gradient of the quantization layer to be:

$$\frac{\partial \hat{v}}{\partial v} = \begin{cases} a & \text{if } \frac{-Q_n}{a} < \frac{v}{s} \leq \frac{Q_p}{a} \\ 0 & \text{otherwise} \end{cases}$$

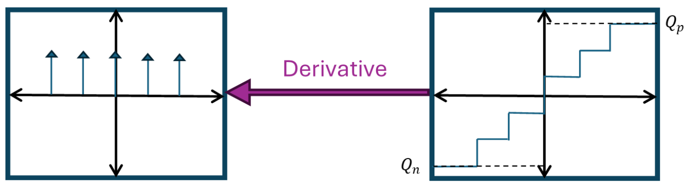
It is designed so that $a = 1$ recovers the standard STE. When $0 < a < 1$, the non-clipped region between $-Q_n/a$ and $-Q_p/a$ enlarges, passing gradients that were previously suppressed. Additionally, every gradient will be multiplied by a factor of a, which results in smaller updates. The opposite occurs when $a > 1$, the non-clipped band narrows, and the gradients are amplified. This is visualized in figure **??** . A key design choice is to keep the integral of this function equal to $Q_n + Q_p$ for every a. This is supposed to limit the learning range to logical approximations of the quantization function. We know that it estimates a staircase between $-Q_n$ and Q_p, so any other integral would be inconsistent. This assumption about the integral is common, often implicitly, in many STE replacements [25, 24].

One element of this STE parameterization that distinguishes it from previous replacements is separating the clipping range of the backward pass from that of the forward pass. Our method can theoretically learn the appropriate clipping range for the gradients without changing the forward quantization as well, avoiding discarding useful information near the quantization thresholds. This is a known issue with the STE, which can destabilize training and decrease performance [**?**, **?**]. An interesting observation about the a parameter is its similarity to the learning rate. Both parameters scale the same gradient. However, a has additional advantages, such as controlling the backward clipping range.

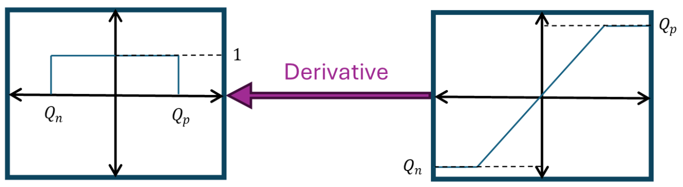
This idea is related to the EWGS [24] method, where the gradient of the quantization layer is multiplied by a dynamic scalar. This scalar is calculated from the mismatch between real and quantized weights with a scaling factor decided via a separate Hessian-based optimization loop. Despite the simple transformation that is done on the gradient, the method achieves impressive results. Scaling GSTE similarly scales the gradient but allows for the multiplier to be directly learned from the loss, avoiding extra heuristics or approximations, so its scaling can potentially be better than EWGS.

3.2 Piecewise Linear

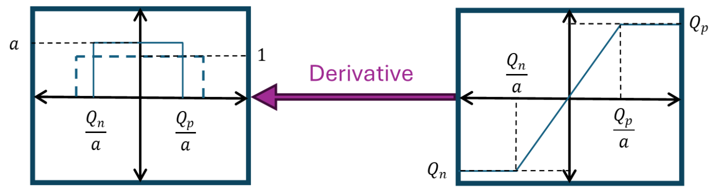
The Scaling GSTE has many benefits, including its simplicity, but it approximates the quantization operation to a linear function. We introduce a *Piecewise Linear GSTE* (PWL GSTE) that allows learning a more interesting shape that can get closer to an ideal step gradient, reducing gradient mismatch. Earlier studies have shown that



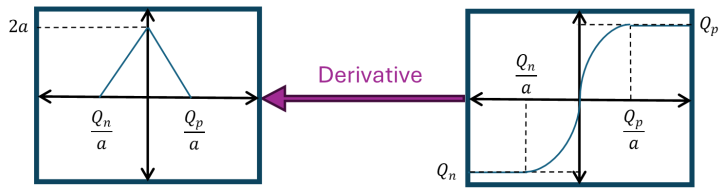
(a) No STE



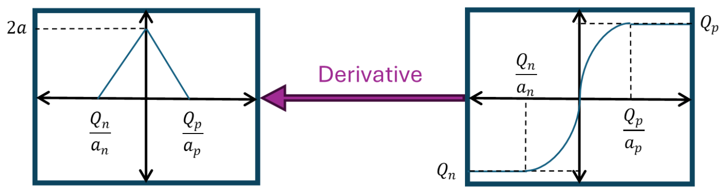
(b) Regular STE



(c) Scaling GSTE



(d) Piecewise-Linear STE



(e) Dual PWLSTE

Figure 3.1: Comparing different surrogate function variants for the quantization layer and how well they approximate it. Each subfigure shows (left) the surrogate function and (right) how it approximates the quantization layer.

replacing the STE with a triangular function can help alleviate mismatch and stabilize learning in ultra-low bit quantization [?]. The formulation we present here mostly fits binary quantization, but can also work with ultra-low bit widths.

Single parameter form

The first approximation, which we call PWL GSTE, uses a single parameter a, which is a trainable scalar initialized to 1. The derivative of the quantization is replaced by:

$$\frac{\partial \hat{v}}{\partial v} = \begin{cases} 2a\Big(1 + \frac{a}{Q_n}\,\frac{v}{s}\Big) & -\frac{Q_n}{a} \le \frac{v}{s} \le 0 \\ 2a\Big(1 - \frac{a}{Q_p}\,\frac{v}{s}\Big) & 0 < \frac{v}{s} \le \frac{Q_p}{a} \\ 0 & \text{otherwise} \end{cases}$$

When $a = 1$, we will have a rough triangle approximation with the full range passed backwards, but as we keep training a can shrink, sharpening the triangle and pushing the derivative closer to the ideal impulse. This type of regime, starting with a coarse approximation and refining it through training is common in similar works like [25] where they set it heuristically.

Two parameter form

An approach to making the parameterized STE more expressive is to add more learnable parameters, allowing more flexibility. Because the positive and negative halves of the step should not necessarily be symmetric in practice, we can generalize further by using each side its own scale. We refer to the following method as *Dual PWL GSTE* (DPWL GSTE). Let a_n and a_p be learnable parameters initialized to 1. The gradient replacement would be:

$$\frac{\partial \hat{v}}{\partial v} = \begin{cases} 2\,a_n\Big(1 + \dfrac{a_n}{Q_n}\,\frac{v}{s}\Big), & -\dfrac{Q_n}{a_n} \le \frac{v}{s} \le 0, \\ 2\,a_p\Big(1 - \dfrac{a_p}{Q_p}\,\frac{v}{s}\Big), & 0 < \frac{v}{s} \le \dfrac{Q_p}{a_p}, \\ 0, & \text{otherwise.} \end{cases}$$

We get a more general form that can adapt independently on each side according to the loss and data statistics. This parameterization is important, serving as an example for learning multiple parameters simultaneously. This opens opportunities for more general methods, such as full neural networks, to replace the STE. However, adding more parameters makes the optimization more complicated and adds computational overhead, increasing the time and resources required for training.

Notice that both PWL variants presented maintain the integral to be $Q_n + Q_p$, ensuring they remain logical approximations of the forward quantization operation. They also maintain the feature of separating forward and backward clipping ranges, which can be advantageous, as we described in the previous section.

3.3 STE derived from finite differences

The STE is a heuristic solution for the non-differentiable gradient issue of the quantization layer. It differs from the mathematical gradient, making it non-trivial that it still gives a good direction for update. Therefore, the following questions have been a topic of discussion 'Why is it a successful replacement for the gradient of the quantization layer?' and 'Why does going in its negative direction minimize the loss?' We will provide a new way to answer those questions through the Finite Differences numerical technique. We will show how the STE can be derived as an approximation of the gradient of the quantization function using finite differences. This provides an intuitive explanation for the effectiveness of the STE and a justification for its use.

We use the Heaviside step function defined as:

$$H(t) = \begin{cases} 1 & t > 0 \\ 0 & t \leq 0 \end{cases}$$

We will start by addressing the binary quantization case. We use the following general formulation for binarized quantization schemes.

$$Q_b(x) = c_- + (c_+ - c_-)H(x - \tau) \tag{3.1}$$

where c_- and c_+ are the two low and high quantization levels ($c_- < c_+$), and τ is the threshold. In many cases, we set $c_- = -1, c_+ = 1$, but we solve for the more general case for later use. We work with the centered finite differences approach with window size $2h$ as defined in 1.1. We would like to find the gradient of $Q_b(x)$ with this approach:

$$\frac{\partial Q_b(x)}{\partial x} \overset{\text{FD}}{\approx} \frac{Q_b(x+h) - Q_b(x-h)}{2h} = \frac{c_+ - c_-}{2h}\Big(H(x + h - \tau) - H(x - h - \tau)\Big) \tag{3.2}$$

The first equality uses the definition in 1.1 and the second substitutes 3.1. The relation between the Heaviside function and an indicator is:

$$H(x+h-\tau) - H(x-h-\tau) = \mathbf{1}_{(\tau-h,\tau+h)}(x)$$

Since only between $\tau - h$ and $\tau + h$ the two Heavisides don't cancel each other. So overall, the gradient approximation we got is:

$$\frac{\partial Q_b(x)}{\partial x} \overset{\text{FD}}{\approx} \frac{c_+ - c_-}{2h} \mathbf{1}_{(\tau-h,\tau+h)}(x) \tag{3.3}$$

An important private case is $\{-1, +1\}$ for which we get:

$$\frac{\partial\, Q_b(x)}{\partial x} \;\approx\; \frac{1}{h} \mathbf{1}_{(-h,\, h)}(x). \tag{3.4}$$

Another important case we will later use is $\{i, i+1\}$:

$$\frac{\partial Q_b(x)}{\partial x} \;\approx\; \frac{1}{2h} \mathbf{1}_{(\tau-h,\, \tau+h)}(x). \tag{3.5}$$

What we get is a generalization of several popular STE's. The original STE from [22] can be described as $h \to \infty$ where there is no clipping, but with FD, we get that the value of the gradient approaches zero. The opposite idea is $h \to 0$, approximating the gradient as a delta function, which approaches the real gradient of the step function. In between these two extremes, we have the most common STE with clipping used in BNN, DoReFa, LSQ and many more [17, 18, 19], which set exactly $h = 1$. An interesting observation is that STE's look for the balance between the misrepresentation of the quantization caused by a large perturbation step in FD, while avoiding the unstable gradient with vanishing and exploding gradients associated with a small perturbation step on nondifferentiable functions. In other words, lower h implies a closer approximation to the true gradient but with the risk of vanishing gradient, higher h stabilizes the gradients but introduces more bias to the weight updates. This motivates later methods that try to learn or dynamically adjust the STE shape, which essentially try to find this balance. Another insight into the STE we get from the FD comparison is understanding the mass preservation that many STE's implicitly use. This refers to the integral of the STE being constant and equal to 1 (for binary quantization) for most STE approximations.

We will now look at the case of multi-bit quantization. We can describe the quantization function for b-bit quantization using Heaviside:

$$Q(x) = -Q_n + \sum_{j=0}^{Q_n+Q_p-1} H\Big(u - (-Q_n + j + \tfrac{1}{2})\Big).$$

Where Q_p and Q_n are the number of positive and negative quantization steps. To find the gradient of $Q(x)$ we can use the linearity of the derivative and separate the sum into the derivatives of its elements:

$$\frac{\partial Q(x)}{\partial x} = \sum_{j=0}^{Q_n+Q_p-1} \frac{\partial H\Big(x - (-Q_n + j + \frac{1}{2})\Big)}{\partial x} = \sum_{j=0}^{Q_n+Q_p-1} \frac{1}{2h}\mathbf{1}_{(-Q_n+j+\frac{1}{2}-h,\ -Q_n+j+\frac{1}{2}+h)}(x)$$

In the second equality, we use 3.5, which is the derivative of the standard Heaviside with threshold τ. This derivative is a superposition of rectangular pulses centered at each transition point. Similarly to the binary case we get that $h \to \infty$ avoids clipping but vanishes the gradient, and $h \to 0$ approaches the real gradient of the staircase function. At exactly $h = 1/2$ we get the standard STE with clipping, where all of the pulses connect to form a constant gradient value between the Q_p and Q_n. We once again see the phenomenon the constant support keeping the gradient equal to $Q_p + Q_n$. This h value is similar to some works that introduced a dynamic temperature parameter that makes the gradient closer or further away from the real gradient, like in DSQ [25].

Overall, the FD approach deepens our understanding of STE, explaining its tradeoffs and key attributes, while giving it a mathematical origin. It also motivates works that insert temperature parameters, or learn the optimal balance point for the STE. Our GSTE approach is built upon this idea, and the SGSTE focuses on finding this balance using an FD-inspired parameterization.

Chapter 4

GSTE optimization methods

In section **??**, we showed how to learn the hyperparameters of the optimizer. Later, in chapter **??** we explored different parameterizations of the STE that introduced learnable parameters. We now turn to learning those parameters of the GSTE in a similar approach. Unlike the optimizer's hyperparameters, the GSTE's parameters change the gradient itself, which introduces an additional layer of complexity. We will start following the standard approach in section **??**, where we encounter a problem that we analyzed in **??**, followed by an analytical solution presented in **??**. Afterwards, we describe more heuristic and general methods for optimization: **??**, All times together **??**, less greedy all times together **??**. We conclude the chapter by presenting schemes for efficient and general implementation **??**.

4.1 Manual hypergradient derivation

Our goal in this section is to derive an update rule for the SGSTE parameters a_i by treating it analogously to an optimizer hyperparameter. We choose to focus here on the SGSTE because of its simplicity, similarity to the STE, and to the learning rate as described in **??**. We use the familiar notation we described in the section **??** where v_i denotes the full precision weight at step i, $\hat{v}_i$ is the quantized version, and L_i is the loss at the end of step i. We use μ_1 for the learning rate used to update v, and μ_2 for that of a. Also, s_i is the quantization scale, while Q_n and Q_p are the negative and positive clipping levels in the forward quantization layer. We use the following definition of an indicator function:

$$I_{x\in S} = \begin{cases} 1 & x \in S \\ 0 & \text{otherwise} \end{cases}$$

Recall the formula for the simple SGSTE:

$$\frac{\partial \hat{v}_i}{\partial v_i} = \begin{cases} a_{i+1} & \frac{-Q_n}{a_{i+1}} < \frac{v_i}{s_i} < \frac{Q_p}{a_{i+1}} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

We assume $a_{i+1} > 0$ since the other case approximates the quantizer with an infinite or negative slope, which is illogical. For convenience, we will rewrite the SGSTE expression using indicators as follows:

$$A_1 = \left\{ a \,|\, a < Q_p \frac{s_i}{v_i} \right\}, \quad A_2 = \left\{ a \,|\, a < -Q_n \frac{s_i}{v_i} \right\}$$

$$\frac{\partial \hat{v}_i}{\partial v_i} = \begin{cases} a_{i+1} I_{a_{i+1} \in A_1} & \frac{v_i}{s_i} > 0 \\ a_{i+1} I_{a_{i+1} \in A_2} & \frac{v_i}{s_i} < 0 \end{cases} \tag{4.2}$$

For this derivation, we assume the optimizer being used is SGD for simplicity. We define the following process where we start with an SGD update to a followed by an SGD update to v:

$$a_{i+1} = a_i - \mu_1 \frac{\partial L_i}{\partial a_i} \tag{4.3}$$

$$v_{i+1} = v_i - \mu_2 \frac{\partial L_i}{\partial v_i} = v_i - \mu_2 \frac{\partial L_i}{\partial \hat{v}_i} \frac{\partial \hat{v}_i}{\partial v_i} \tag{4.4}$$

The value of $\frac{\partial L_i}{\partial v_i}$ required for 4.4 can be obtained using regular backpropagation with the SGSTE we defined. To complete the process, what is left is finding the value of $\frac{\partial L_i}{\partial a_i}$ used in 4.3. We start by opening it using the chain rule:

$$\frac{\partial L_i}{\partial a_i} = \frac{\partial L_i}{\partial v_i} \frac{\partial v_i}{\partial a_i} = \frac{\partial L_i}{\partial v_i} \frac{\partial \big(v_{i-1} - \mu_2 \frac{\partial L_{i-1}}{\partial \hat{v}_{i-1}} \frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}}\big)}{\partial a_i}$$

$$= -\mu_2 \frac{\partial L_i}{\partial v_i} \frac{\partial L_{i-1}}{\partial \hat{v}_{i-1}} \frac{\partial \big(\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}}\big)}{\partial a_i} \tag{4.5}$$

The second equality is derived by substituting 4.4 into v_i, and the third equality observes that v_{i-1} and L_{i-1} do not depend on a_i. Taking a look at the expression we got, the value $\frac{\partial L_{(i-1)}}{\partial \hat{v_{(i-1)}}}$ can be obtained via regular backpropagation, but the two other gradients require more attention. The term $\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}}$ involves differentiating the SGSTE with respect to a. Since the GSTE we use is not differentiable, we will now find an approximation for its gradient. Note that this can be avoided by picking a GSTE that is smooth. We write:

$$\frac{\partial\left(\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}}\right)}{\partial a_i} = \begin{cases} \frac{\partial\left(a_i I_{a_i \in A_1}\right)}{\partial a_i} & \frac{v_{i-1}}{s_{i-1}} > 0 \\ \frac{\partial\left(a_i I_{a_i \in A_2}\right)}{\partial a_i} & \frac{v_{i-1}}{s_{i-1}} < 0 \end{cases} = \begin{cases} I_{a_i \in A_1} + a_i \frac{\partial\left(I_{a_i \in A_1}\right)}{\partial a_i} & \frac{v_{i-1}}{s_{i-1}} > 0 \\ I_{a_i \in A_2} + a_i \frac{\partial\left(I_{a_i \in A_2}\right)}{\partial a_i} & \frac{v_{i-1}}{s_{i-1}} < 0 \end{cases} \tag{4.6}$$

The first equality substitutes 4.2 because we want to derive with respect to a_i, the indicator formulation is intended to remove the dependency on a_i in the condition. This allows derivation according to the known rules for products in the second equality. We face a familiar issue, calculating the step functions $\frac{\partial\left(I_{a_i \in A_1}\right)}{\partial a_i}$ and $\frac{\partial\left(I_{a_i \in A_2}\right)}{\partial a_i}$. This is equivalent to the problem of deriving the quantization function, which requires deriving a step function. The most obvious solution is to use the STE, but our GSTE approach can work here as well. For instance, we can suggest another version of SGSTE with parameter b:

$$B_p = \{a \,|\, Q_p \frac{s_{i-1}}{v_{i-1}} - b < a < Q_p \frac{s_{i-1}}{v_{i-1}} + b\}, \quad B_n = \{a \,|\, -Q_n \frac{s_{i-1}}{v_{i-1}} - b < a < -Q_n \frac{s_{i-1}}{v_{i-1}} + b\}$$

$$\frac{\partial\left(I_{a_i \in A_1}\right)}{\partial a_i} = \frac{1}{2b} I_{a_i \in B_p}, \quad \frac{\partial\left(I_{a_i \in A_2}\right)}{\partial a_i} = \frac{1}{2b} I_{a_i \in B_n}$$

Finding the correct b parameter is similar to finding the a parameter. We can either set b manually or try to learn it as well by deriving the appropriate SGD step, which might introduce another parameter. This sort of situation was addressed by Chandra et al. [37], where they constructed similar ”optimization towers” to learn the learning rate. Importantly, they discovered that as the towers grow taller, the initial choice of hyperparameters becomes less significant. Indicating that even if we choose to stop the optimization tower at the first level by setting b manually, it would already be better than standard STE. In our implementation, we work with $b = \frac{1}{2}$, which results in the standard STE:

$$B_p = \{a \,|\, Q_p \frac{s_{i-1}}{v_{i-1}} - \frac{1}{2} < a < Q_p \frac{s_{i-1}}{v_{i-1}} + \frac{1}{2}\}, \quad B_n = \{a \,|\, -Q_n \frac{s_{i-1}}{v_{i-1}} - \frac{1}{2} < a < -Q_n \frac{s_{i-1}}{v_{i-1}} + \frac{1}{2}\}$$

$$\frac{\partial\left(I_{a_i \in A_1}\right)}{\partial a_i} = I_{a_i \in B_p}, \quad \frac{\partial\left(I_{a_i \in A_2}\right)}{\partial a_i} = I_{a_i \in B_n}$$

Overall, we can substitute this gradient into 4.6 and get an expression for the required gradient:

$$\frac{\partial\left(\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}}\right)}{\partial a_i} = \begin{cases} I_{a_i \in A_1} + a_i \frac{\partial\left(I_{a_i \in A_1}\right)}{\partial a_i} & \frac{v_{i-1}}{s_{i-1}} > 0 \\ I_{a_i \in A_2} + a_i \frac{\partial\left(I_{a_i \in A_2}\right)}{\partial a_i} & \frac{v_{i-1}}{s_{i-1}} < 0 \end{cases} = \begin{cases} I_{a_i \in A_1} + a_i I_{a_i \in B_p} & \frac{v_{i-1}}{s_{i-1}} > 0 \\ I_{a_i \in A_2} + a_i I_{a_i \in B_n} & \frac{v_{i-1}}{s_{i-1}} < 0 \end{cases}$$

$$\frac{\partial\left(\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}}\right)}{\partial a_i} = \begin{cases} 1, & \frac{v_{i-1}}{s_{i-1}} > 0 \ \wedge \ a_i \in A_1 \setminus B_p, \\ 1 + a_i & \frac{v_{i-1}}{s_{i-1}} > 0 \ \wedge \ a_i \in A_1 \cap B_p, \\ a_i & \frac{v_{i-1}}{s_{i-1}} > 0 \ \wedge \ a_i \in B_p \setminus A_1, \\ 1 & \frac{v_{i-1}}{s_{i-1}} < 0 \ \wedge \ a_i \in A_2 \setminus B_n, \\ 1 + a_i & \frac{v_{i-1}}{s_{i-1}} < 0 \ \wedge \ a_i \in A_2 \cap B_n, \\ a_i & \frac{v_{i-1}}{s_{i-1}} < 0 \ \wedge \ a_i \in B_n \setminus A_2, \\ 0 & \text{otherwise} \end{cases}$$

This derivation is an example of the manual calculation of an LGSTE gradient. we later describe how it can be calculated automatically and does not necessarily require this calculation for every LGSTE variant. We now have the first two pieces needed to calculate 4.5 and only finding $\frac{\partial L_i}{\partial v_i}$ is left. We will open it up using the chain rule:

$$\frac{\partial L_i}{\partial v_i} = \frac{\partial L_i}{\partial \hat{v}_i}\frac{\partial \hat{v}_i}{\partial v_i} = \frac{\partial L_i}{\partial \hat{v}_i} LSTE(a_{i+1}) \tag{4.7}$$

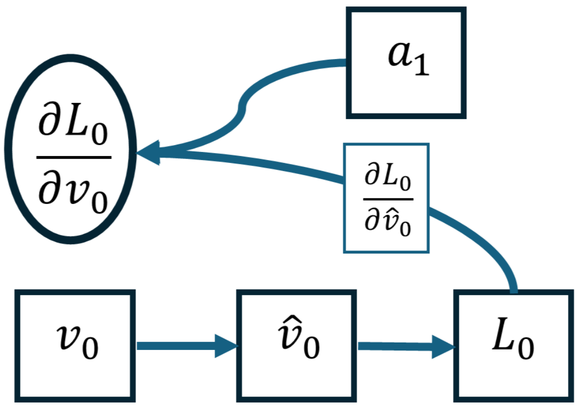
Where $SGSTE(a_{i+1})$ denotes the the SGSTE gradient estimator from 4.1 which depends on a_{i+1}. Since 4.7 is used in 4.5 to calculate $\frac{\partial L_i}{\partial a_i}$, which is then used in the update rule 4.3 to calculate a_{i+1}, we get a dependency on an unknown future value. In other words, to find a_{i+1} we first need to know the value of a_{i+1}.

Thus, the method that worked for learning the hyperparameters of the optimizer breaks down for the parameters of the GSTE, no longer giving a simple, direct update rule. We later show how we overcome this problem by extracting an analytical solution in section **??**, but we first need to understand the issue and how it arises.

4.2 Understanding the problem

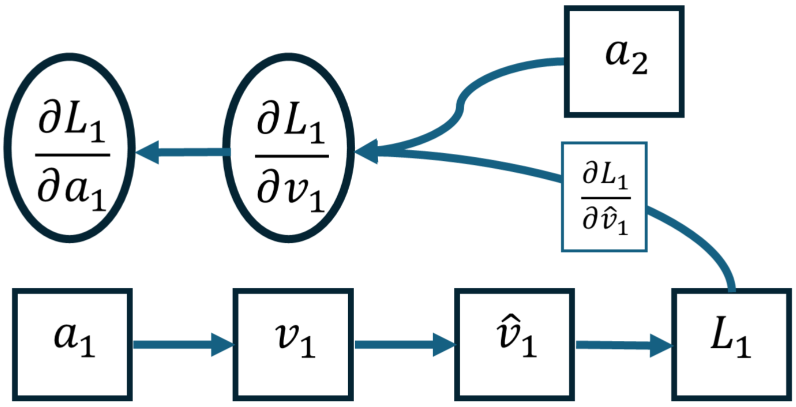
A useful way to understand the problem is to iterate through the process and see where exactly it appears and why. We will visualize each important value as a block and draw lines representing that a value is used to calculate the following one. This is similar to the computational graph created in Automatic Differentiation (AD).

Automatic differentiation refers to methods that automatically compute the derivative of a function in a computer program. Most commonly, AD methods use a computational graph, which is a directed acyclic graph (DAG) where the leaves are weights, the internal nodes represent intermediate computations, and the root of the graph is the



a_1
$\frac{\partial L_0}{\partial v_0}$
$\frac{\partial L_0}{\partial \hat{v}_0}$
v_0
$\hat{v}_0$
L_0

(a) First iteration



a_2
$\frac{\partial L_1}{\partial a_1}$
$\frac{\partial L_1}{\partial v_1}$
$\frac{\partial L_1}{\partial \hat{v}_1}$
a_1
v_1
$\hat{v}_1$
L_1

(b) Second iteration

Figure 4.1: Visualizing the emergence of the problem using a simplified computational graph. (a) shows the first forward and backward passes (b) shows the second forward and backward passes.

final loss. The graph is created dynamically every time a computation is made, and it enables backpropagating through the computations and applying the chain rule easily. The backpropagation process begins at the root and calculates the gradient for every step until it reaches the weight in the leaf, which is then updated in the optimizer, and the process starts again. We will go over a simplified version of the full computational graph, where we don't see irrelevant nodes, and the connection between lines means there is a directed path in the graph going between those two nodes. We don't assume any optimizer or specific GSTE parameterization to show that the problem is inherent to the process and not to our design choices.

We begin in Figure 4.1a from the first node v_0 at step 0, which represents some weight in the network. When applying the forward process, we get a connection from v_0 to its quantized counterpart $\hat{v}_0$, which is then connected to the loss L_0. At this stage, we begin the backpropagation process, which in a standard manner can calculate the gradient $\frac{L_0}{\hat{v}_0}$ to the quantized node. To find the loss gradient of v_0 we use the GSTE, which has some learnable parameter a_1. The next step is applying the optimizer update using the loss gradient, we make sure not to detach a_1 from the graph while detaching the gradient from the loss, this is a crucial step to keep the gradient tracked all the way to a_1 so this parameter would be optimized according to the loss L_1.

Now we begin the second iteration visualized in Figure 4.1b, where the connection between a_1 and v_1 is because it was used to find the loss gradient, which is used in the update rule to calculate the new weight value. Similarly, we have a connection from v_1 to L_1 followed by a backpropagation step that easily finds $\frac{\partial L_1}{\partial \hat{v}_1}$, while we use a_2 for the calculation of $\frac{\partial L_1}{\partial v_1}$. The backpropagation keeps going to find the loss gradient of a_1, but this gradient depends on a_2.

Here we encounter the problem, we see that the a parameters are a part of the calculation of the gradient of v_1 and directly affect the calculation of the loss gradient of the weight, which in our process is the basis for finding the loss gradient of a. This is different from learning the hyperparameters of the optimizer, which do not change the gradient directly but only how it is used in the following update rule. We can deduce that the only way for this process to work is if we can know the a_2 parameter before applying the update to a_1. One heuristic way we suggest doing this is by assuming the values of a_i are known for some period of time which allows us to use a_{i+1} to calculate a_i. Another idea is changing the order of the updates, since we have a rule connecting a_i to a_{i+1}, we can modify it to suggest an update to a_{i+1} given a_i. This avoids the problem by treating the process in Figure 4.1b not as originally intended for the calculation of a_1, but rather for finding a_2.

4.3 Analytical solution

We wish to find an analytical solution to the problem by deriving some non-trivial update rule for a_{i+1} without depending on a_{i+2}. The current update rule can be written as:

$$a_{i+1} = a_i + \mu_1\mu_2 \frac{\partial L_i}{\partial \hat{v}_i} \frac{\partial L_{i-1}}{\partial \hat{v}_{i-1}} \frac{\partial(\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}})}{\partial a_i} LSTE(a_{i+1}) = a_i + \mu_1 c_i LSTE(a_{i+1}) \tag{4.8}$$

The first equality is obtained by substituting 4.7 and 4.5 into 4.3. While c_i is a new notation we use for readability, defined as:

$$c_i = \mu_2 \frac{\partial L_i}{\partial \hat{v}_i} \frac{\partial L_{i-1}}{\partial \hat{v}_{i-1}} \frac{\partial (\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}})}{\partial a_i}$$

In this section, we work with the SGSTE parameterization. We assume $\frac{v_i}{s_i} > 0$ since the other case will provide symmetric results. Recall the indicator form of the SGSTE from 4.2, with the previous assumption it could be written as $SLSTE(a_{i+1}) = a_{i+1} I(a_{i+1} \in A_1)$ when $A_1 = \left\{ a \,|\, a < Q_p \frac{s_i}{v_i} \right\}$. Therefore, the update rule of a can be written as:

$$a_{i+1} = a_i + \mu_1 a_{i+1} I(a_{i+1} \in A_1) c_i$$

We want to obtain an expression for a_{i+1} that depends only on values from steps i and below.

$$a_{i+1} \left(1 - \mu_1 I(a_{i+1} \in A_1) c_i\right) = a_i$$

$$a_{i+1} = \left(1 - \mu_1 I(a_{i+1} \in A_1) c_i\right)^{-1} a_i$$

Which can be written as:

$$a_{i+1} = \begin{cases} a_i & a_{i+1} \notin A_1 \\ (1 - \mu_1 c_i)^{-1} a_i & a_{i+1} \in A_1 \end{cases} = \begin{cases} a_i & a_{i+1} > Q_p \frac{s_i}{v_i} \\ (1 - \mu_1 c_i)^{-1} a_i & a_{i+1} < Q_p \frac{s_i}{v_i} \end{cases} \tag{4.9}$$

Notice that we did not eliminate the problem since the thresholds still depend on a_{i+1}. We will refer to the first condition as the identity condition ($a_{i+1} = a_i$) and the second condition as the scaling condition ($(1 - \mu_1 c)^{-1} a_i < Q_p \frac{s_i}{v_i}$). To remove this dependency, we use the following logic: for which values of a_i will the identity choice be self-consistent? And similarly, we ask for what a_i values will the scaling choice be self-consistent? The following lemmas answer these questions and set us up for the method itself.

Lemma 4.3.1. *The assignment $a_{i+1} = a_i$ satisfies the condition $a_{i+1} > Q_p \frac{s_i}{v_i}$ iff $a_i > Q_p \frac{s_i}{v_i}$ holds*

Proof. For the first direction, we assume that $a_{i+1} > Q_p \frac{s_i}{v_i}$ is satisfied therefore according to the update rule in Equation 4.9 we know $a_{i+1} = a_i$. We can substitute this expression and get that $a_{i+1} = a_i > Q_p \frac{s_i}{v_i}$ is satisfied as well.

For the second direction, we assume $a_i > Q_p \frac{s_i}{v_i}$. We can see that the assignment $a_{i+1} = a_i$ will satisfy $a_{i+1} = a_i > Q_p \frac{s_i}{v_i}$. So, this side holds as well. ■

Lemma 4.3.2. *The assignment $a_{i+1} = (1 - \mu_1 c_i)^{-1} a_i$ satisfies the condition $a_{i+1} < Q_p \frac{s_i}{v_i}$ iff $a_i (1 - \mu_1 c_i)^{-1} < Q_p \frac{s_i}{v_i}$*

Proof. Similarly to the previous proof, we first assume $a_{i+1} < Q_p \frac{s_i}{v_i}$ then directly from the update rule we have $a_{i+1} = a_i \left(1 - \mu_1 c\right)^{-1} < Q_p \frac{s_i}{v_i}$ as well.

For the opposite direction, we assume $a_i \left(1 - \mu_1 c\right)^{-1} < Q_p \frac{s_i}{v_i}$ and we see that the assignment $a_{i+1} = \left(1 - \mu_1 c\right)^{-1} a_i$ gives $a_{i+1} = a_i \left(1 - \mu_1 c\right)^{-1} < Q_p \frac{s_i}{v_i}$. ■

The update we developed is not a normal assignment since we depend on the unknown value a_{i+1}. So, before we can use this in any way, we must first answer the following questions "For a given a_i, which of the two branches may I legally choose?" or "For which values of a_i is each branch self-consistent?". Answering those questions means tracing back through the update and understanding where the a_{i+1} in the condition came from. Our two lemmas answer those exact questions, they show that each of the branches has some a_i values that keep it legal in the sense that, given the update, it will adhere to the condition on a_{i+1}. An intuitive explanation is that Lemma 1 says that if you try to leave a in the next step unchanged (Identity branch), your a_i value must be above the stated threshold. Lemma 2 states that if you try to scale a (taking the scaling branch), the a_i value must adhere to the correct side of the presented threshold.

Now we can partition every a_i value into those who fit only the Identity branch, those who fit only the Scaling branch, those who fit both of them, and those who fit neither. We will later discuss what happens in the overlap and exclusion regions, but for now, we can formulate the following update for a_i:

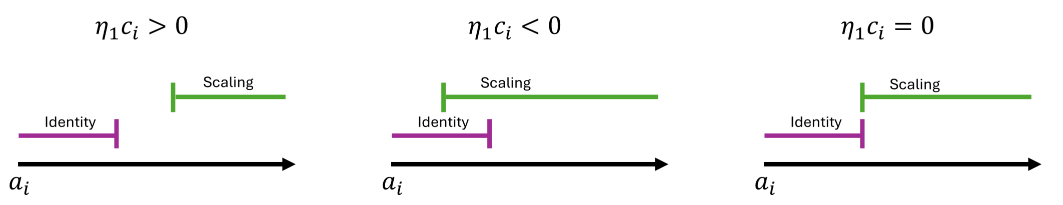
$$a_{i+1} = \begin{cases} a_i & , if \;\; a_i > Q_p \frac{s_i}{v_i} \\ \left(1 - \mu_1 c_i\right)^{-1} a_i & , if \;\; \left(1 - \mu_1 c_i\right)^{-1} a_i < Q_p \frac{s_i}{v_i} \end{cases}$$

And symmetrically if $\frac{v_i}{s_i} < 0$ we get:

$$a_{i+1} = \begin{cases} a_i & , if \;\; a_i > -Q_n \frac{s_i}{v_i} \\ \left(1 - \mu_1 c_i\right)^{-1} a_i & , if \;\; \left(1 - \mu_1 c_i\right)^{-1} a_i < -Q_n \frac{s_i}{v_i} \end{cases}$$

As we can see, the branches of this update rule can be determined from the value $\left(1 - \mu_1 c_i\right)^{-1}$ and specifically on the value of $\mu_1 c_i$. If $\mu_1 c_i > 0$, then the conditions are partial and we need to define an action in the space where it's not defined. If $\mu_1 c_i < 0$, the conditions overlap and we need to choose which one to take, only when $\mu_1 c_i = 0$ we have no problems (illustrated in Figure 4.2).

To deal with this, we suggest a mathematically sound approach, which comes with a small computational overhead. We identify that the hyperparameter μ_1 is the learning rate, which is a hyperparameter we can choose. This means that the value of the expression $\left(1 - \mu_1 c\right)^{-1}$ can be controlled by changing the learning rate used for the update rule, with the right manipulation we can choose μ_1 that removes all ambiguity



(a) Partial conditions case (b) Overlapping condition case (c) Equal threshold case

Figure 4.2: The scaling and identity branch coverage on different a_i values for different $\mu_1 c_i$ values.

and gives exactly one legal update rule. The question is how to find the appropriate value for μ_1? We know that the process involves decreasing the value of the learning rate since, as we decrease it, we make the thresholds for the two branches get closer and closer (see Figure 4.2). Eventually, when we get to $\mu_1 = 0$, the conflict necessarily resolves entirely. We identify that the optimal value would be the largest μ_1 (below the current learning rate). We suggest the following process: check if there is an ambiguity with the current a_i value, if so, halve the learning rate and repeat until the ambiguity is removed. This algorithm ensures finding a learning rate that is at least $\frac{1}{2}$ of the optimal value while being simple and computationally efficient.

This method gives mathematically accurate updates, but adds a small computational overhead for every parameter update. We can suggest a more heuristic methods that add less computational overhead. For example, to use the decision rule $a_i > Q_P \frac{s_i}{v_i}$ (for identity) or $a_i < Q_P \frac{s_i}{v_i}$ (for scaling). This is equivalent to the branches used in the previous method, but without changing the value of the hyperparameter μ_1. Another possibility is always staying on $a_{i+1} = a_i$, which is a passive choice, until we get out of the problematic state (by the change of $\frac{v_i}{s_i}$ and c_i), however this will not necessarily solve the problem, and we might just be stuck.

Overall, we reached a legal update rule for a_i, which we derived analytically and optimized based on the final loss, but the method does come with two disadvantages. Firstly, it introduces computational overhead over the standard STE, which we discuss further when describing the implementation in section **??** Our method takes more time because of the complicated process of obtaining the required gradients, and also more memory since we use gradients that were stored from the previous iteration. A second problem with the method is that it is not general and requires adjustments for every optimization method or STE replacement chosen. This comes from the assumptions in this section, where we must choose an optimizer and an STE to open up and change the order of elements to arithmetically remove the dependence on the future. These problems motivate exploring heuristic methods that are more general or less computationally intensive. We suggest several such methods in the following sections.

[Possibly insert algorithm here]

4.4 All times together

For the next bunch of methods, we try to solve the problem raised in section **??** using the following intuition: "what if we know ahead of time every future value of a_i?". We will therefore suggest a couple of methods that assume we know all of the values ahead of time, making the calculation of correct mathematical gradients relatively easy.

The method nick-named *All Times Together* (ATT) allocates a distinct GSTE hyperparameter a_i to every step i in the optimization of the weights, and uses its value to calculate the gradient using GSTE as we've seen. The a_i values are optimized offline after the entire training of the weights has finished. So the process consists of an inner loop where the weights are trained, which is enveloped by an outer loop that optimizes the GSTE parameters based on the gradient with respect to the loss at each step.

To formalize and understand this method, we first define $\bar{a}^t$:

$\bar{a}^t = (a_1^t, \ldots, a_T^t) \in \mathbb{R}^T$

The numbers below indicate what step this parameter corresponds to in the weight training, so T is the total number of training steps done on the weights. The number above indicates the number of steps done on the a parameters in the outer loop. This vector is initialized with the initialization value corresponding with the specific GSTE, for example, in the SGSTE case $\bar{a} = \mathbf{1}$, so the first inner loop is just the STE baseline training. The definition of the GSTE stays similar:

$$\frac{\partial \hat{v}_i^t}{\partial v_i^t} = LSTE(a_i^t) \tag{4.10}$$

The update rule for the weights is also familiar:

$$v_{i+1}^t = v_i^t - \mu_2 \left(\frac{\partial L_i^t}{\partial \hat{v}_i^t} \right) LSTE(a_i^t) \tag{4.11}$$

This is the regular SGD update rule, but with the t index keeping track of how many steps have been done on a, which is the amount of time we restarted training from scratch. The update rule for the parameters in $\bar{a}^t$ is:

$$a_{i+1}^{t+1} = a_i^t - \mu_1 \left(\frac{\partial L_i^t}{\partial a_i^t} \right) \tag{4.12}$$

This update is applied after a full training of the weights occurred, which accumulates all of the gradients for every i. Only then, we start updating the parameters of the GSTE one by one (which moves us from t to $t+1$). TO use this update rule we need to find the gradient:

$$\frac{\partial L_i^t}{\partial a_i^t} = \frac{\partial L_i^t}{\partial \hat{v}_i^t} LSTE(a_{i+1}^t) \left(-\mu_2 \frac{\partial L_{i-1}^t}{\partial \hat{v}_{i-1}^t} \frac{\partial \left(LSTE(a_{i-1}^t) \right)}{\partial a_i^t} \right) \tag{4.13}$$

This is calculated using the chain rule and the fact that a_i affects the loss only through the LSTE. This brings us to the initial intuition of the method, if we know ahead of training all the a_i values that will be used, we know the value of a_{i+1} for the update of a_i, avoiding the problem of dependence on future values. This solves the problem for every a_i update except for one, which we need to address.

The very last parameter a_T^t, which is used for the last step on the weights, still encounters the problem of depending on future values. This is clear from the gradient calculation at 4.13 where we require the value of a_{T+1}^t to find the gradient of a_T^t. A bad a_T^t could lead to a trickle down of bad gradients all the way to a_1^{t+T} since in every step on $\bar{a}$ the updates depend on each other. We suggest two heuristic approaches that address this. The first solution is to assume that $a_{T+1}^t = 1$, which is equivalent to using the STE for the last step of the training. The impact of using a simple STE here would be quite small since every other a_i^t for $0 < i < T$ (where T is typically quite large) is still updated correctly. We also know that even though standard STE is not optimal, it is still a decent estimator, so it should give decent gradients. Another similar approach is to use $a_T^t = a_{T+1}^t$, so the GSTE parameter for the last step of the training of the weights would be the same as the previous step. This approach lets the a_T^t change for different t which can be beneficial since firstly we don't necessarily expect the parameters to change rapidly between T and $T+1$ anyway and secondly this keeps the schedule of a similar to what was learned so for example if we see that a_i gets smaller as i increases instead of having large value for a_T^t like 1 we will have a smaller value based on the previous parameter.

[Possibly insert algorithm here]

The main benefits of this method are its expressiveness and generality. The method can learn any schedule for the GSTE parameters based on the data, where almost all the updates are calculated analytically, except for the last one.

It also enables easy generalization to different GSTE variants and optimizers, enabling the use of autograd to calculate all of the required gradients automatically, as we later describe in section **??**. Yet the computational requirements of this method are not practical for many real-world applications. It requires repeating the full weight training several times, increasing the training time. It also requires a separate a parameter for each weight in the network and each training step on the weights, adding a significant memory overhead.

The All Times Together method can give an accuracy boost for extreme quantization with any GSTE or optimizer in cases where training compute is not a concern. In this work, it is meant to provide a proof of concept showing that we can learn a GSTE that is better than the STE. It is also the basis for more relaxed versions that allow deployment with less computational power.

Relaxed All Times Together

We suggest two ways to make the All Times Together scalable to large networks and datasets. The first relaxation we employ is using T that is smaller than the training time. This brings about a different process where we focus on training and optimizing the $\bar{a}$ for the first T and when we finish them, we go on to face the next T steps and so on. If the entire training on the weights is kT steps, we look at every T separately and optimize the a parameters for them to minimize the loss at the end of those specific steps. This method mainly improves the memory demand because it uses a much smaller $\bar{a}$ for every weight.

This goes hand in hand with another helpful approach, which is using a small number of steps to train $\bar{a}$. Each step adds another training from scratch of the weights for T steps, even if we use a small T, it still adds up to a full training from scratch, accumulating all of the repetitions. We will later see that using the correct learning rate for the GSTE parameters enables a major improvement with few optimization steps. This approach decreases the training time significantly by training from scratch only a few times. Together the methods allow deploying on any task with any optimizer or LSTE, maintaining a more reasonable computational overhead both in terms of memory and training time.

4.5 Less greedy ATT

The preceding methods update the GSTE parameters using the instantaneous loss. Meaning we use the first loss calculated using a_i, which we notate L_i, to find the update for a_i. This type of optimization can be described as "greedy" in the sense that the optimization rule only looks at getting the best loss right now without looking ahead to see how this choice affects the future loss, which we actually care about. The ATT method is structured in a way that separates the update of GSTE parameters from the update of the weights, essentially performing a step on a every T steps on the weights. We observe that ATT is using a greedy approach, in the worst case, despite having calculated every loss function from step i to step $i+T$ on the weights, the update to a_i is only according to L_i as we see in 4.13. We suggest *"Less Greedy All Times Together"* (LGATT), which builds upon the Relaxed ATT but uses the last loss calcualted. For example when finding the optimial LSTE parameters betweeb steos kT and $(k+1)T$ on the weights we optimize according to the loss $L_{(k+1)T}$ the $\bar{a}$ values. This way we are considering the consequences of a_i on several subsequent weight updates and giving a more accurate update.

To describe the method, we start by defining the update rules for the weights and the GSTE parameters. The weights update rule is familiar:

$$v_{i+1} = v_i - \mu_1 \left(\frac{\partial L_i}{\partial v_i}\right) = v_i - \mu_1 \left(\frac{\partial L_i}{\partial \hat{v}_i}\right)\left(\frac{\partial \hat{v}_i}{\partial v_i}\right) \tag{4.14}$$

For the update of a we first deal with the simple case of $T=2$ steps each time on the weights to understand how the updates work. This approach requires updating a_i and a_{i+1} using the loss L_{i-1} and the previous values a_{i-1} and a_{i-2} in the following way:

$$(a_{i-1}, a_{i-2}) = (a_{i-1}, a_{i-2}) - \mu_2 \left(\frac{\partial L_{i-1}}{\partial a_{i-1}}, \frac{\partial L_{i-1}}{\partial a_{i-2}}\right) \tag{4.15}$$

The first gradient $\frac{\partial L_{i-1}}{\partial a_{i-1}}$ can be easily found using the chain rule:

$$\frac{\partial L_{i-1}}{\partial a_{i-1}} = \frac{\partial L_{i-1}}{\partial v_{i-1}} \frac{\partial v_{i-1}}{\partial a_{i-1}}$$

The first gradient can be obtained from standard backpropagation, and the second via the LSTE definition. The second gradient requires a bit more attention, as we are looking for the effect of a_{i-2} that was used to calculate v_{i-2} on the loss L_{i-1}, which depends on v_{i-1}. We will use the fact that v_{i-2} is used to calculate v_{i-1} and get:

$$\begin{aligned}
\frac{\partial L_{i-1}}{\partial a_{i-2}} &= \frac{\partial L_{i-1}}{\partial v_{i-1}} \frac{\partial v_{i-1}}{\partial v_{i-2}} \frac{\partial v_{i-2}}{\partial a_{i-2}} \\
&= \frac{\partial L_{i-1}}{\partial v_{i-1}} \frac{\partial}{\partial v_{i-2}}\Big(v_{i-2} - \mu_1 \frac{\partial L_{i-2}}{\partial \hat{v}_{i-2}} \frac{\partial \hat{v}_{i-2}}{\partial v_{i-2}}\Big) \frac{\partial}{\partial a_{i-2}}\Big(v_{i-3} - \mu_1 \frac{\partial L_{i-3}}{\partial \hat{v}_{i-3}} \frac{\partial \hat{v}_{i-3}}{\partial v_{i-3}}\Big) \\
&= -\mu_1 \frac{\partial L_{i-1}}{\partial v_{i-1}} \Big(1 - \mu_1 \frac{\partial^2 L_{i-2}}{\partial^2 v_{i-2}}\Big) \frac{\partial}{\partial a_{i-2}} \Big(\frac{\partial L_{i-3}}{\partial v_{i-3}}\Big)
\end{aligned} \tag{4.16}$$

The first equality uses the chain rule, the second substitutes 4.14, and in the third we remove elements that don't depend on the derived parameter. We are not worried about dependence on future values because we work with the ATT framework that enables us to know all the a values ahead. Later in this section, we address how to calculate each of the terms in this expression.

We will now look at the case for an arbitrary number of steps $T = k$ on the weights. The update rule would be:

$$(a_{i+k-1}, ..., a_i) = (a_{i-1}, ..., a_{i-k}) - \mu_2 \left(\frac{\partial L_{i-1}}{\partial a_{i-1}}, ..., \frac{\partial L_{i-1}}{\partial a_{i-k}} \right)$$

We need to provide an expression we can calculate for all of the required gradients. We first use the chain rule:

$$\frac{\partial L_{i-1}}{\partial a_{i-k}} = \frac{\partial L_{i-1}}{\partial v_{i-1}} \frac{\partial v_{i-1}}{\partial v_{i-2}} \frac{\partial v_{i-2}}{\partial v_{i-3}} \cdots \frac{\partial v_{i-k+1}}{\partial v_{i-k}} \frac{\partial v_{i-k}}{\partial a_{i-k}} \tag{4.17}$$

We can develop by substituting the weight update rule for any j, we have:

$$\frac{\partial v_j}{\partial v_{j-1}} = \frac{\partial}{\partial v_{j-1}}(v_{j-1} - \mu_1 \frac{\partial L_{j-1}}{\partial \hat{v}_{j-1}} \frac{\partial \hat{v}_{j-1}}{\partial v_{j-1}}) = \Big(1 - \mu_1 \tfrac{\partial^2 L_{j-1}}{\partial^2 v_{j-1}}\Big) \tag{4.18}$$

$$\frac{\partial v_{i-k}}{\partial a_{i-k}} = \frac{\partial}{\partial a_{i-2}}\Big(v_{i-3} - \mu_1 \frac{\partial L_{i-3}}{\partial \hat{v}_{i-3}} \frac{\partial \hat{v}_{i-3}}{\partial v_{i-3}}\Big) = -\mu_1 \frac{\partial^2 L_{\,i-k-1}}{\partial a_{i-k}\, \partial v_{i-k-1}} \tag{4.19}$$

Now plug it into the gradient:

$$\frac{\partial L_{i-1}}{\partial a_{i-k}} = -\mu_1 \frac{\partial L_{i-1}}{\partial v_{i-1}} \Big(1 - \mu_1 \tfrac{\partial^2 L_{i-2}}{\partial^2 v_{i-2}}\Big) \cdots \Big(1 - \mu_1 \tfrac{\partial^2 L_{i-k}}{\partial^2 v_{i-k}}\Big)$$

$\frac{\partial^2 L_{\,i-k-1}}{\partial a_{i-k}\, \partial v_{i-k-1}}$(4.20)

$$= -\mu_1 \frac{\partial L_{i-1}}{\partial v_{i-1}} \left[\prod_{t=1}^{k-1} \Big(1 - \mu_1 \tfrac{\partial^2 L_{i-t-1}}{\partial v^2_{i-t-1}}\Big) \right] \frac{\partial^2 L_{\,i-k-1}}{\partial a_{i-k}\, \partial v_{i-k-1}} \tag{4.21}$$

What's left is to simplify the last term. For that, we will use the following lemma.

Lemma 4.5.1. *For any $k \geq 1$ and time index i, assuming we use the SGSTE*

$$\frac{\partial}{\partial a_{i-k}} \Big(\tfrac{\partial L_{i-k-1}}{\partial v_{i-k-1}} \Big) = \frac{1}{a_{i-k}} \frac{\partial L_{\,i-k-1}}{\partial v_{\,i-k-1}}$$

Proof. Using the chain rule we have $\frac{\partial L_{i-k-1}}{\partial v_{i-k-1}} = \frac{\partial L_{i-k-1}}{\partial \hat{v}_{i-k-1}} \frac{\partial \hat{v}_{i-k-1}}{\partial v_{i-k-1}}$, And we define the GSTE surrogate gradient $\frac{\partial \hat{v}_{i-k-1}}{\partial v_{i-k-1}} = GSTE(a_{i-k})$ to be either 0 (clipped) or a_{i-k} (unclipped). We will see that the claim is true for both cases.

Case 1 (clipped)- If the gradient was clipped in the GSTE we have $\frac{\partial \hat{v}_{i-k-1}}{\partial v_{i-k-1}} = GSTE(a_{i-k}) = 0$ and therefore $\frac{\partial L_{i-k-1}}{\partial v_{i-k-1}} = \frac{\partial L_{i-k-1}}{\partial \hat{v}_{i-k-1}} \cdot 0 = 0$ The derivative of a constant gives $(\partial / \partial a_{i-k})(0) = 0$, which matches the right-hand side since it consists of multiplication by 0.

Case 2 (unclipped)- When the GSTE is not clipped we have $GSTE(a_{i-k}) = a_{i-k}$ which means $\frac{\partial L_{i-k-1}}{\partial v_{i-k-1}} = \frac{\partial L_{i-k-1}}{\partial \hat{v}_{i-k-1}} a_{i-k}$. Therefore, taking the derivative with respect to

a_{i-k} on the left-hand side yields

$$\frac{\partial}{\partial a_{i-k}}\left(\frac{\partial L_{i-k-1}}{\partial v_{i-k-1}}\right) = \frac{\partial L_{i-k-1}}{\partial \hat{v}_{i-k-1}}$$

And the right-hand side yields $\frac{1}{a_{i-k}}\frac{\partial L_{i-k-1}}{\partial v_{i-k-1}} = \frac{1}{a_{i-k}}\frac{\partial L_{i-k-1}}{\partial \hat{v}_{i-k-1}}\, a_{i-k} = \frac{\partial L_{i-k-1}}{\partial \hat{v}_{i-k-1}}$ So both sides are equivalent as claimed. The two cases agree with the same expression, so the lemma is proven. ■

Using this lemma, we have the following full expression when using the SGSTE:

$$\frac{\partial L_{i-1}}{\partial a_{i-k}} = -\mu_1 \frac{\partial L_{i-1}}{\partial v_{i-1}} \left[\prod_{t=1}^{k-1}\left(1 - \mu_1 \frac{\partial^2 L_{i-t-1}}{\partial v_{i-t-1}^2}\right)\right] \frac{1}{a_{i-k}} \frac{\partial L_{\,i-k-1}}{\partial v_{\,i-k-1}} \tag{4.22}$$

Breaking down the different gradients that are used in this expression, the first and last ones are calculated naturally as part of the backpropagation. Regarding the expression $\prod_{t=1}^{k-1}\left(1 - \mu_2\left(\frac{\partial^2 L_{i-t-1}}{\partial^2 v_{i-t-1}}\right)\right)$ we need to present a process that calculates it efficiently. We suggest to calculate $\left(\frac{\partial^2 L_{i-t-1}}{\partial^2 v_{i-t-1}}\right)$ after every backward operation for the weights of every layer in the network by keeping track of the computation graph during backward then deriving again. From there we can multiply the gradient of each a used up to this point by the expression $1 - \left(\frac{\partial^2 L_{i-t-1}}{\partial^2 v_{i-t-1}}\right)$ obtained in the current step. This will add some overhead at each step, and it spreads the calculation of this expression over time.

Overall, we developed a ”less greedy” approach built upon the ATT or Relaxed ATT methods. It can optimize all of the a values according to the most recent loss instead of the instantaneous one. It adds some computational overhead, requiring the calculation of the second derivative and storing some additional values between iterations. Additionally, although this method can relatively easily be generalized to different GSTEs and optimizers, we did assume SGSTE for the last step of our derivation.

4.6 Efficient training scheme

For the GSTE method, we introduced the learnable parameter a, which should be updated inside the same iteration in which the ordinary weights v are trained. We dealt with the mathematics of finding the SGD update rule in section **??** and arrived at the formula **??**. But we have not yet addressed the algorithmic approach to finding the required gradients. In this section, we strive to give a clear understanding of the approach taken for the implementation, as well as showing how this approach gives good generalization beyond the assumptions we took to calculate the mathematical update rule. This will give a deeper insight into the advantages and disadvantages of using GSTE and similar approaches for learning parameters of the backpropagation.

4.6.1 Two-phase backward pass

When deriving the analytical solution in section **??**, we wanted to give an expression for a_{i+1} using only the values calculated in previous steps (step i or before). To do this, we used several partial derivatives that satisfy this requirement, but we glossed over how they can be found as part of the learning process. We will now break this down and show an important modification to backpropagation that is required. For clarity, we will restate the update rule of the analytical solution, assuming $\frac{s_i}{v_i} > 0$ we have:

$$a_{i+1} = \begin{cases} a_i & , if \;\; a_i > Q_p \frac{s_i}{v_i} \\ (1-\mu_1 c_i)^{-1} a_i & , if \;\; (1-\mu_1 c_i)^{-1} a_i < Q_p \frac{s_i}{v_i} \end{cases}$$

With the definition:

$$c_i = \mu_2 \frac{\partial L_i}{\partial \hat{v}_i} \frac{\partial L_{i-1}}{\partial \hat{v}_{i-1}} \frac{\partial (\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}})}{\partial a_i}$$

We will go over every term used here and explain how it was attained. The values a_i, v_i,s_i,μ_1, and Q_p are either parameters of the model or hyperparameters that are easily accessible. For the gradient $\frac{\partial (\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}})}{\partial a_i}$ we provided a thorough calculation in section **??** giving an expression that only consists of known parameters. We will later show that it can be calculated automatically using autograd without the need for future manual derivation.

The two final gradients $\frac{\partial L_i}{\partial \hat{v}_i}$ and $\frac{\partial L_{i-1}}{\partial \hat{v}_{i-1}}$ are the loss gradients of the quantized value $\hat{v}$, and are calculated as part of the forward process without being parameters themselves. This makes them an intermediate node in the computational graph created by autograd. Therefore, their gradient value is not stored anywhere during a regular backpropagation process. Another issue is that the gradient $\frac{\partial L_i}{\partial \hat{v}_i}$ is only known during the latest backpropagation, which later uses $\frac{\partial \hat{v}}{\partial v} = GSTE(a_{i+1})$ where a_{i+1} is the value we currently want to update. The solution we suggest is splitting the training process into two different backward passes. During the first pass, all network parameters are temporarily frozen while $\hat{v}$ is regarded as a learnable parameter of the network. This forces autograd to store the required gradient, so after the first backpropagation, we have all the necessary values for the update rule and can calculate a_{i+1}. At this stage, we can continue the backpropagation without a problem and find the gradient of the network parameters. In our implementation, we go through the full forward then backward process again instead of continuing the computational graph where we left off. This is where most of the computational overhead of the method comes from, it requires each forward operation to be calculated twice with a corresponding backward pass. But importantly, a more careful implementation is possible with just one forward-backward pass, which will result in a significantly smaller overhead over standard STE. In terms of memory, it does require storing several values between iterations, which inevitably increases the memory consumption.

4.6.2 Utilizing Autograd for simplicity and generalization

In section **??**, we explained the method from [37] to automatically compute hypergradients with a small modification to backpropagation, where we carefully detach to integrate hyperparameters into the computational graph. As we saw in the mathematical derivation in section **??** for the gradient of the a parameter, finding the gradient by hand can be complicated even for simple optimizers and GSTE shapes like the SGSTE. Therefore, we want to utilize the selective detach trick in our implementation to calculate the gradient automatically using autograd. This will enable generalization for different types of GSTEs and optimizers. This approach is useful for the heuristic methods we described, like ATT and DU, but it is not suited for the analytical solution because in equation **??** of the derivation, we assumed the optimizer to be SGD. Any other optimizer would require a new derivation, but a version of this approach can be devised for finding the gradient of the GSTE parameter automatically.

An essential part of standard optimizers is detaching the previous weight from the graph before making the update to get the new weight. This way, we prevent the link between the weight and the parameters that created it in the previous iteration. We suggest detaching everything that was used to calculate v_i except for the parameter a_{i+1} in order to create a link from the GSTE parameter to the loss calculation. If we define the update rule to v_{i+1} with a general optimizer as:

$$v_{i+1} = f(\frac{\partial L}{\partial v_i}, v_i)$$

Where f is some function using known operators that are derivable by autograd and could depend on external hyperparameters. Since we won't detach a_{i+1}, which is used to find the gradient for this update, we will create a connection in the computation graph between a_{i+1} and v_i. Therefore, the relevant branch in the computational graph can be visualized as:

$$a_{i+1} \rightarrow \frac{\partial L}{\partial v_{i-1}} \rightarrow v_i \rightarrow \hat{v}_i \rightarrow ... \rightarrow L_i$$

We want to understand the gradient of a_i obtained with this method:

$$\begin{aligned}
\frac{\partial L_{i-1}}{\partial a_i} &= \frac{\partial L_{i-1}}{\partial v_i} \frac{\partial}{\partial a_i} f\Big(\frac{\partial L_{i-1}}{\partial v_{i-1}}, v_i\Big) \\
&= \frac{\partial L_{i-1}}{\partial v_i} \frac{\partial f}{\partial\big(\frac{\partial L_{i-1}}{\partial v_{i-1}}\big)} \frac{\partial\left(\frac{\partial L_{i-1}}{\partial v_{i-1}}\right)}{\partial a_i} \\
&= \frac{\partial L_{i-1}}{\partial v_i} \frac{\partial f}{\partial\big(\frac{\partial L_{i-1}}{\partial v_{i-1}}\big)} \frac{\partial}{\partial a_i}\Big(\Big(\frac{\partial L_{i-1}}{\partial \hat{v}_{i-1}}\Big)\Big(\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}}\Big)\Big) \\
&= \frac{\partial L_{i-1}}{\partial v_i} \frac{\partial f}{\partial\big(\frac{\partial L_{i-1}}{\partial v_{i-1}}\big)} \frac{\partial L_{i-1}}{\partial \hat{v}_{i-1}} \frac{\partial}{\partial a_i}\Big(\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}}\Big).
\end{aligned} \tag{4.23}$$

The first and third equalities use the chain rule, the second equality uses the fact that in the optimizer function f, only the gradient depends on a_i, and then applies the chain rule. The fourth equality observes that only the GSTE depends on the parameter a_i and applies the chain rule as well. This expression would be calculated using the computational graph, where the value of $\frac{\partial L_{i-1}}{\partial v_i}$ will be known using one of the methods we described for handling the dependence on future values **??**. The The gradient $\frac{\partial f}{\partial\big(\frac{\partial L_{i-1}}{\partial v_{i-1}}\big)}$ that derives through the optimizer's update rule and $\frac{\partial L_{i-1}}{\partial \hat{v}_{i-1}}$ that is the loss gradient of the quantized value can be easily found without standard autograd tools. Regarding the final gradient $\frac{\partial}{\partial a_i}\left(\frac{\partial \hat{v}_{i-1}}{\partial v_{i-1}}\right)$ which we previously calculated by hand, is now found through Automatic Differentiation. It will still get the same expression as we did since the way we handled the discontinuities in section **??** is equivalent to how autograd handles them by default.

Overall, we get a correct general approach that can integrate with our methods from sections **?? ?? ??**. This allows using any optimizer and GSTE easily without requiring any manual derivations, making these methods more useful in practice and significantly easier to implement. It also potentially enables testing GSTE optimization towers as we described in section **??** and similarly to [37] to remove any dependence on hyperparameters for the GSTE gradient.

Chapter 5

Experiments

In this chapter, we evaluate the proposed learnable surrogate gradients and the different optimization methods. We report results on the CIFAR-10 dataset **??** using the ResNet-20 architecture **??** and compare against standard STE-based training under several weight precisions. We first detail the setup, then present the main results and describe them.

5.1 Experimental Setup

All experiments use the CIFAR-10 [?] dataset, which enables checking all the different scenarios and training regimes on a single GPU. Despite its size, it is sensitive to optimization and quantization artifacts, which makes it an interesting test. ResNet-20 [?] is the residual network used for CIFAR-style inputs, it is also lightweight and commonly used in QAT. Unless explicitly noted, we quantize weights only in all layers except the final layer (kept FP32), while activations remain FP32. Keeping the last layer FP32 is standard in low-bit QAT because it preserves class-separation geometry. Quantizing the weights only isolates the optimization error induced by surrogate gradients of the weights, thereby avoiding the effects of activation quantization. For 2–3-bit comparisons, we additionally report configurations where both weights and activations are quantized using LSQ.

The quantization method we use for 2–3 bits is the Learned Step Size Quantization (LSQ) [19], which learns a scale parameter s (per-channel for weights in our runs) jointly with the network and provides stable training under QAT. For the 1-bit case we use the DoReFa forward mapping [?] without gradient quantization. This serves as a simple yet effective binary quantization scenario. The decision of the appropriate quantization function is not important for our comparison, since we are interested in the optimization error, and therefore any quantization function is just our baseline approximation error for the comparisons we make. The optimizer we use is the SGD without momentum. This matches the scope of our analytical derivation (Ch. **??**), avoids interaction effects between momentum and surrogate updates, and is considered a good optimizer for ultra-low bit precisions.

In the following tests we use the backward column to specify which surrogate gradient is used to replace the gradient of the quantizer. We check all of our learnable STEs versus their appropriate STE baseline, so for the SLSTE, this baseline is the standard clipping STE **??** while for the PLSTE and DLSTE we compare to the piece-wise linear STE **??**. The LSTE parameters a are always initialized so the learned surrogate equals STE at iteration 0. This ensures a controlled, and fair comparison with equivalent initialization. The a values are learned by the procedures specified in previous chapters for each method. Two key hyperparameters in the FAGU, P-FAGU and FLAGU approaches are T and K. We denote by T the number of weight epochs between the LSTE updates, and by K the total number of a updates, which is the ammount of times we reapeat the training of the T weight epochs.

Ultra-low-bit quantization narrows activation/weight dynamic ranges and makes training sensitive to BN's running statistics. We therefore perform a calibration sweep: prior to the relevant training phase, we (i) set the model to training mode, (ii) freeze gradients, (iii) reset BN running means/variances, and (iv) pass the full training set once to recompute stable statistics, then resume training. This reduces train/valuation mismatch and mitigates accuracy drop observed without calibration in low-bit QAT (also noted in prior quantization literature [5, **?**, **?**]). We apply this calibration in the binary/2-bit settings and before long PFAGU runs where distribution drift accumulates.

All CIFAR-10 experiments were run on a single NVIDIA GeForce RTX 2080 Ti (11,GB).

we use a unique calibration of the Batch Normalization whihc is required in our case since we work with quantization to very low bit widths, we essentially go over all of the data before an update and adjust the batch notmalization paramters according to it (There is a paper that explain why its neccessary refer to it).

5.2 Analytical SLSTE results

We evaluate the analytical update derived for SLSTE (Ch. **??**) under three precisions. For 2–3 bits we use LSQ for both weights and activations; for the 1-bit case we use DoReFa (weights) with FP32 activations. Our implementation follows the process we discussed in section **??** with the 2-phase backpropagation. We train for 500 epochs with the a learning rate being The baseline we work with is the Standard STE which the SLSTE is built on. We test both methods for 500 epochs of weights, where we apply an update on a after every update for w.

Table 5.1: Testing the analytical optimization method on CIFAR-10 dataset with ResNet-20 architecture for 500 epochs. "Precision (W/A)" denotes weight/activation bit-widths. Forward is the forward quantizer (LSQ/DoReFa). Backward is the surrogate gradient (STE or Analytical LSTE).

Precision (W/A)	**Forward**	**Backward**	**Top-1 Test**
3 bit, 3 bit	LSQ	STE	86.34
		SLSTE	**86.46**
2 bit, 2 bit	LSQ	STE	81.02
		SLSTE	**81.19**
1 bit, 32 bit	DoReFa	STE	78.43
		SLSTE	**78.74**

SLSTE consistently matches or outperforms the standard STE, with the largest gains observed at the 1-bit weight setting. This aligns with our expectation that SLSTE should become more effective as optimization error intensifies at lower bit-widths. In particular, it is specifically designed to handle the 1-bit case, where only two discrete levels exist, since we derived the SLSTE from the finite differences approximation for the binary case. Training curves further indicate improved validation accuracy across learning rates (see Fig. 5.1).

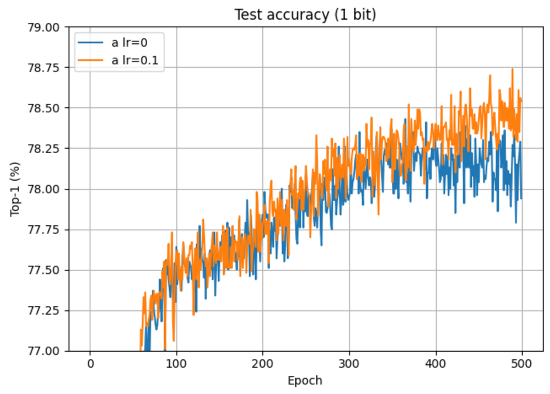


Figure 5.1: Convergence on CIFAR-10 (1-bit DoReFa weights) for analytical SLSTE under multiple learning rates.

5.3 FAGU results

We next evaluate the Future-Aware Gradient Update (FAGU) optimization method. In this setting, we train $T = 1$ weight epochs and perform $K = 50$ updates on the LSTE parameters a. This procedure isolates the optimization of the LSTE parameters from long-term weight training and provides a controlled test for comparing different LSTE parameterizations.

This setup can be interpreted as repeatedly replaying the same first epoch of weight training while optimizing the LSTE, giving better gradients each time. The goal is to examine how much improvement can be achieved by learning the surrogate function itself, independently of further weight updates. It also enables testing parameterizations beyond the scaling-based SLSTE, such as piecewise-linear (PLSTE) and dual-parameter piecewise-linear (DLSTE) variants, under a unified optimization scheme.

In table **??** we see that across all tested precisions, the learned surrogates significantly outperform their non-learned baselines. For 2-bit weights with LSQ forward quantization, FAGU improves SLSTE over the standard STE by more than 2 percentage points in test accuracy. In the binary scenario, FAGU further enhances SLSTE compared to the standard STE. Additionally, we see big improvement for PLSTE/DLSTE compared to their fixed PWL STE baseline with up to 5 percent improvement. These improvements confirm that the additional degrees of freedom in PLSTE and especially DLSTE can be exploited effectively when coupled with FAGU, suggesting that richer surrogate families can capture sharper gradient approximations.

Table 5.2: Testing the FAGU optimization method on CIFAR-10 dataset with ResNet-20. We apply 50 steps on a for $T = 1epoch$ steps on the weights. So the results show how learned LSTE improves the accuracy after 1 epoch of training the weights. "Precision" denotes weight bit-widths while activations are FP32. Forward is the forward quantizer (LSQ/DoReFa). Backward is the surrogate gradient (STE, PWL STE, SLSTE, PLSTE, and DLSTE)

Precision	**Forward**	**Backward**	**Top-1 Test**	**Top-1 Training**
2 bit	LSQ	STE	87.06	93.896
		SLSTE	**89.12**	**95.33**
1 bit	DoReFa	STE	83.71	82.96
		SLSTE	**85.80**	**85.62**
1 bit	DoReFa	PWL STE	82.63	84.02
		PLSTE	**87.54**	85.58
		DLSTE	86.30	**86.14**

To further test scalability, we extend FAGU to longer weight trainings ($T = 5, 10, 15$

epochs). Figure 5.2 shows that FAGU continues to yield substantial gains when we train for more epochs compared to the standard STE. We also compare to a full precision baseline, which is initialized similarly by quantization to 2 bits, but keeps training without quantization. Interestingly, in some cases, FAGU-trained surrogates approach the accuracy of FP32 models trained under this identical initialization, highlighting that much of the performance gap is attributable to gradient mismatch rather than fundamental quantization error. This suggests that, if computational constraints were not a bottleneck, running FAGU for the entire weight training trajectory could yield large accuracy gains.

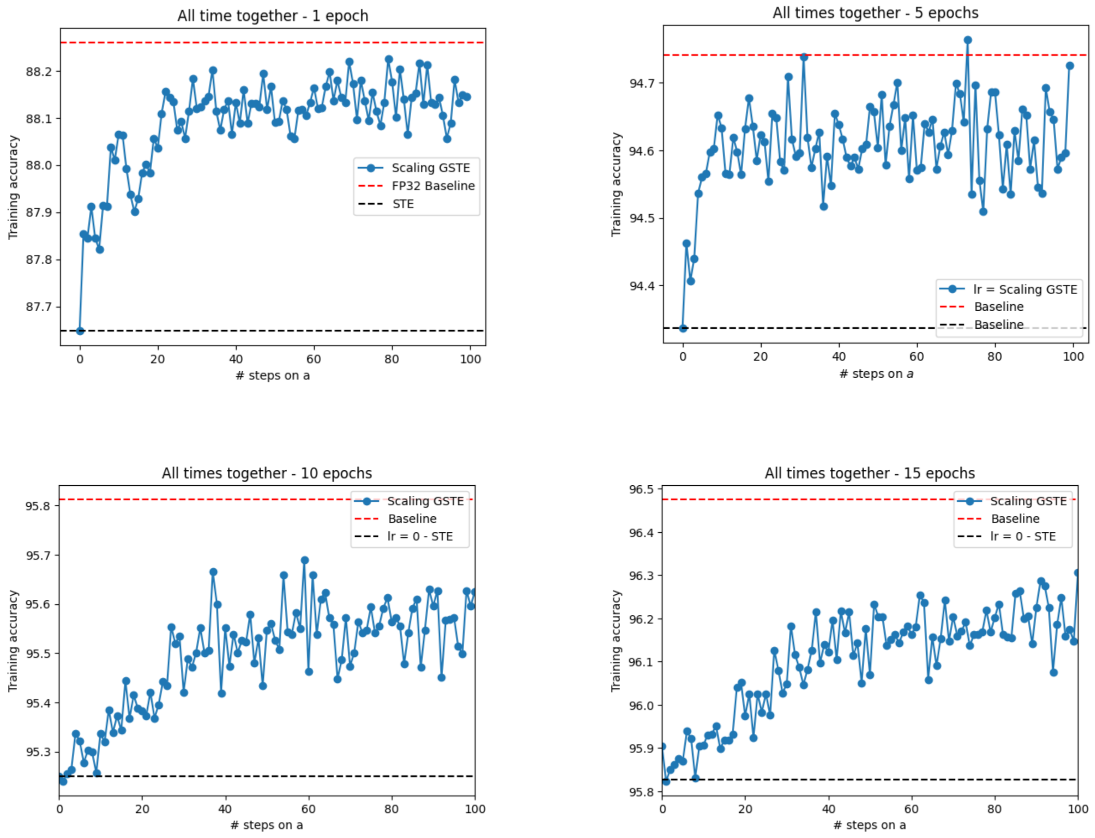


Figure 5.2: Figures showing the training curve of the All Times Together method for different ammount of weight training epochs compared to the STE and FP32 results.

Finally, we compare FAGU with its less-greedy variant FLAGU, which updates all surrogate parameters using only the final loss of the trajectory. We use LSQ quantization to 2 bits as we've seen before for FAGU. As shown in Table **??**, FLAGU achieves nearly identical test accuracy to FAGU, with a slight training advantage. However, FLAGU incurs substantially higher computational overhead due to Hessian calculations. This makes FAGU the better choice for most scenarios and a better building block for the practical P-FAGU method.

Table 5.3: Testing the FLAGU optimization versus FAGU on CIFAR-10 dataset with ResNet-20. We apply 50 steps on a for $T = 1epoch$ steps on the weights. Therefore, the results show how a learned LSTE improves the accuracy after 1 epoch of training the weights. Both methods use weight quantization to 2-bit with FP32 activations. They also use the LSQ quantization function and the SLSTE.

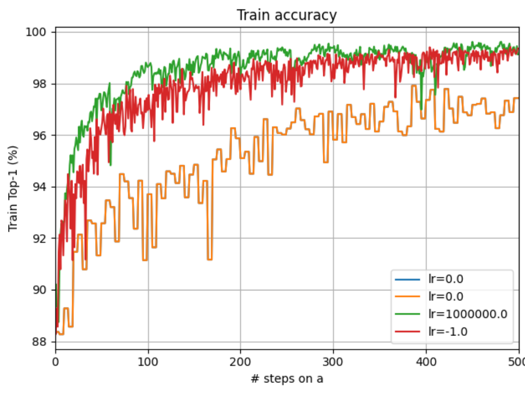
Method	**Top-1 Test**	**Top-1 Training**
Baseline	87.06	93.896
FAGU	**89.12**	95.33
FLAGU	89.07	**95.66**

5.4 P-FAGU results

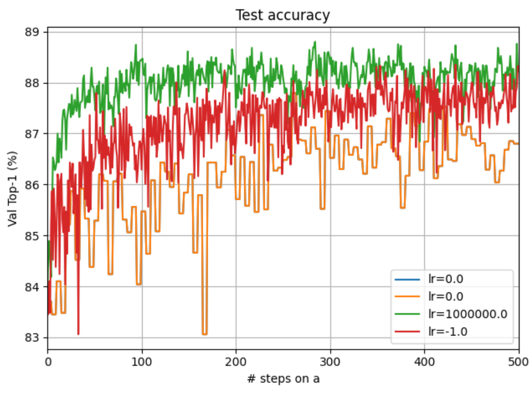
FAGU produces strong early improvements but is limited by memory/compute when extended to long horizons. To overcome this, we adopt a practical variant, P-FAGU, that interleaves a short inner loop that optimizes the LSTE parameter a with standard weight training. This preserves the future-aware approach to the gradient calculation while allowing training to proceed for arbitrarily many epochs. Our P-FAGU training schedule requires that before each weight epoch of length T=1, we perform K=5 gradient steps on the LSTE parameter a while keeping the weights w fixed. We then update w for one epoch with a fixed and continue to the next epoch. We repeat this schedule for E=100 weight epochs (i.e., 500 total a-updates). We adopt the same quantization protocol as in the FAGU experiments, with identical forward and backward configurations across the same bit-widths (see Sec. **??**).

Table 5.4: Testing the PFAGU optimization method on CIFAR-10 dataset with ResNet-20. We optimize 5 steps on a for every $T = 1epoch$ steps on the weights, then repeat for the next T steps for a total of 100 epochs. Essentially, we repeat every epoch 5 times to learn the optimal LSTE parameter for it, then continue to the next epoch. "Precision" denotes weight bit-widths while activations are FP32. Forward is the forward quantizer (LSQ/DoReFa). Backward is the surrogate gradient (STE, PWL STE SLSTE, PLSTE, and DLSTE)

Precision	**Forward**	**Backward**	**Top-1 Test**	**Top-1 Training**
2 bit	LSQ	STE	89.16	97.64
		SLSTE	**90.49**	**99.97**
1 bit	DoReFa	STE	87.79	97.02
		SLSTE	**88.56**	**98.98**
1 bit	DoReFa	PWL STE	88.98	99.36
		PLSTE	89.26	99.65
		DLSTE	**89.54**	**99.81**



(a) Train accuracy



(b) Test accuracy

Figure 5.3: PFAGU convergence: train vs. test accuracy over steps on a.

Table **??** shows that P-FAGU consistently improves over the corresponding non-learned baselines under identical weight-update budgets: The gains appear in both binary and 2-bit regimes, indicating that learning the surrogate reduces optimization error beyond what fixed STE/PWL-STE can provide. We see improvements of between 0.3 to 1.33 percentage points on the test set, with corresponding improvements on the training set accuracy. These results indicate that learning the surrogate reduces optimization error under the same number of weight updates: the network sees exactly the same weight-update budget as the baseline, yet the gradient signal is better aligned with the low-precision landscape and converges to higher test accuracy. We also observe that higher-capacity learned surrogates outperform the fixed piecewise baseline: DLSTE $>$ PLSTE $>$ PWL-STE on the same forward quantizer, implying that additional shape flexibility helps adapt the surrogate to local curvature.

Figure 5.3 illustrates that P-FAGU produces major improvements in the early epochs; the margins decrease later as training approaches the approximation error floor imposed by low precision. Crucially, the learned surrogates maintain a persistent advantage in both train and test accuracy throughout the 100-epoch run. Our primary comparisons hold the number of weight epochs fixed, the K inner updates on a are an additional optimization budget used to improve the quality of the surrogate gradient. To understand whether the gains are merely a byproduct of extra optimization steps, we also consider a stricter baseline that continues training the non-learned surrogate for many more weight epochs (up to the same total step count spent by the inner loop). We see that still P-FAGU reaches comparable or better accuracy substantially earlier and surpasses the extended baseline by the end of training as well. This suggests that investing optimization budget into learning the surrogate is more effective than simply extending weight training with a fixed surrogate.

Across settings, learned surrogates (SLSTE/PLSTE/DLSTE) beat their non-learned counterparts -FAGU is substantially more scalable than long-horizon ATT because it limits the inner optimization to K steps per epoch. Improvements are largest early in training—precisely when gradient mismatch is most detrimental. The inner loop adds compute/memory overhead. That said, reallocating the same optimization budget to simply extend weight training is less effective than learning the surrogate. Overall this gives us a practical method that gives real improvement over the standard non learned STE.

Chapter 6

Discussion

6.1 Summary

This thesis approached the core limitation of quantization-aware training (QAT), the non-differentiability of the quantizer, through a theoretically grounded and empirically validated framework that learns the surrogate gradient. On the theoretical side, we showed that the classical Straight-Through Estimator (STE) is a sub-case of the centered finite-difference (FD) gradient approximation method. This perspective explains both why STE can work and where it fails: larger FD windows stabilize gradients at the cost of bias, while smaller windows approach the true derivative but risk vanishing/unstable behavior around discontinuities. The FD view, therefore, motivates the replacement of the fixed surrogate with a learnable one whose shape is adapted to the data and training phase.

Building on this, we proposed a family of learnable STEs (LSTEs) that keep the forward quantizer unchanged while parameterizing only the backward surrogate. The first version, SLSTE, is motivated directly from the FD approximation, it scales the gradient and decouples backward clipping from the forward range. piecewise-linear variants (PLSTE, DLSTE) increase expressivity while preserving desirable properties such as mass conservation (constant integral) and controllable clipping bands. In practice, this separation lets training retain informative gradients near quantization thresholds without perturbing the forward discretization and provides a spectrum of shapes that can fit local curvature better than a fixed surrogate.

Optimizing a learned surrogate is fundamentally different from tuning conventional optimizer hyperparameters because the surrogate modifies the weight gradient itself. We therefore developed several optimization strategies to handle this case. We derived an analytical update for SLSTE, but it required new hand derivations for every LSTE or optimizer. For the general case, we introduced practical procedures that avoid the future dependency inherent to surrogate gradient learning. Our FAGU scheme assumes future values for the LSTE parameters and optimizes them in an external loop. The FLAGU approach proposes a more accurate update, which comes with additional complexity. The PFAGU approach provides a practical FAGU-based update enabling deployment on real tasks. These techniques provide meaningful and stable parameter updates across tasks and bit-widths.

Across 1–3-bit weight QAT, learned-surrogate (LSTE) variants consistently outperform fixed STE in both peak top-1 and accuracy-per-compute. The analytic SLSTE yields stable gains with only one hyperparameter to tune and small computational overhead. Future-aware methods (FAGU and the lightweight PFAGU) enable training more expressive piecewise-linear surrogates (PLSTE, DLSTE). Added expressivity translates into further improvements, with DLSTE, which allows asymmetric shapes, typically surpassing PLSTE. Analytic updates are mathematically derived and can be relatively cheap. Full FAGU delivers the best accuracy across forward quantizers and bit widths, with benefits appearing early and persisting through training, while PFAGU recovers most of that lift at a fraction of compute/memory. Improvements are largest at the harshest precisions (1-bit) and taper as precision increases. Overall: (i) spending budget to learn the backward surrogate beats spending it on more fixed-STE weight steps; (ii) surrogate expressivity helps when compute allows; and (iii) short-horizon, future-aware updates are a practical, scalable way to realize these gains.

6.2 Future directions

A natural next step is to scale LSTE to larger datasets, architectures, and additional modalities, testing whether the observed gains persist under realistic training dynamics. Similarly, examining how LSTE integrates with modern optimizers and schedulers (momentum variants, Adam, etc.) can allow deployment-ready schemes and reveal potential failures. Building on this, a comprehensive ablation of our design choices would be valuable: for PFAGU, varying the inner loop horizon T (weight epochs per a update) and the number of a updates K per cycle, also more elaborate training schemes should be considered like selecting a by validation accuracy within the inner loop rather than simply using the last computed value. For FAGU, revisiting the last a heuristic rule with alternatives such as resetting to STE, carrying forward the last valid a, or any other solution can yield improvements. Finally, exploring learning rate schedules for a is important to maximize the LSTE learning.

Another promising avenue is testing whether an LSTE learned in one setting carries over to others, such as different datasets, architectures, bit-widths, or forward quantizers.

This could allow several important improvements like stronger fixed-STE initializations, a small library of reusable surrogate priors, or a meta-learned initialization that adapts quickly with minimal compute. Another important avenue can be to extend LSTE to activation quantization. This could be highly impactful, as activations often become the main bottleneck at ultra-low precision. Addressing the connection between activation gradients and many downstream weights may require additional work, but our approaches give a good starting point.

Finally, exploring the surrogate design space has great promise. Our piecewise-linear family demonstrates that added expressivity pays off, motivating highly general and parameterized LSTEs, such as a neural network able to learn any shape. Additionally, smooth LSTEs could reduce analytical non-differentiability while retaining control over mass and clipping, giving more accurate updates and simpler derivations. Overall, following all the suggested research directions can help toward closing the remaining gap between ultra-low-bit QAT and full-precision performance.

Appendix A

Some sort of an appendix

You may want to include appendices of your own volition. Also, if you've developed any computer software, that needs to go in as an appendix as well.

A.1 A section

Some appendix content here. And something nice to finish things off:

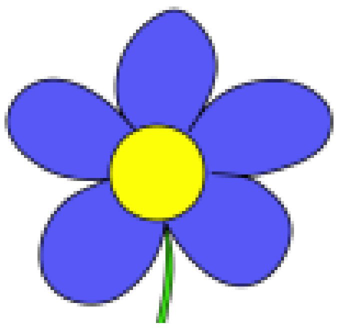


Figure A.1: A Flower.

A.2 Delayed Updates

[This section might not be in the final version because I don't have up-to-date experiments on it. Therefore, this section is not written properly currently]

This is the first heuristic method we present, intended to avoid the dependence on future values from the update rule in **??**. The main idea behind the *Delayed Updates* method is freezing the GSTE parameter for several iterations to avoid the inherent problem of not knowing the next value. This way we get a valid update for some step and apply it with a delay, a couple of steps later.

If we use the SGD optimizer, the update rule for the GSTE parameter could be described as:

$$a_i = \begin{cases} a_{i-1}, & i \bmod k \neq 0, \\ a_{i-k} - \eta_1 \dfrac{\partial L_{i-k+1}}{\partial a_{i-k}}, & i \bmod k = 0, \end{cases} \tag{A.1}$$

Here η_1 is the learning rate and L_{i-k} is the loss at the end of step $i-k$. We initialize the $\tilde{a}_0$ as the standard initialization point of the STE replacement (with our STE $\tilde{a}_0$ initialized to 1). This method introduces the k hyperparameter that sets the delay. We suggest using $k \in 2, 3$, which are responsive enough in terms of applying the gradient when it is still relevant, and they reduce the amount of updates used by keeping the same value of a for more than one iteration. Increasing k further can decrease the computational overhead of the method (which is already quite small), but it yields updates that are less accurate and relevant.

Whats left is to find $\frac{\partial L_{i-k}}{\partial \tilde{a}_{i-k}}$ Using the chain rule, we work with $k = 2$ and get:

$$\frac{\partial L_{i-1}}{\partial a_{i-2}} = \frac{\partial L_{i-1}}{\partial v_{i-1}} \frac{\partial v_{i-1}}{\partial a_{i-2}} \tag{A.2}$$

$$= \frac{\partial L_{i-1}}{\partial \hat{v}_{i-1}} \, GSTE(a_{i-1}) \, \frac{\partial}{\partial a_{i-2}} \Big(v_{i-2} - \mu_2 \frac{\partial L_{i-2}}{\partial \hat{v}_{i-2}} \, GSTE(a_{i-2}) \Big) \tag{A.3}$$

$$= -\mu_2 \frac{\partial L_{i-1}}{\partial \hat{v}_{i-1}} \, E(a_{i-1}) \, \frac{\partial L_{i-2}}{\partial \hat{v}_{i-2}} \, \frac{\partial E(a_{i-2})}{\partial a_{i-2}} \tag{A.4}$$

We see that the dependence on future values is removed because by definition $a_{i-1} = a_{i-2}$ at the time of the update. Therefore we have a well-defined gradient using parameters that have known values. This method relies on the loss landscape being sufficiently smooth, so a one-step delay would add a relatively small error. We can see that the computational overhead this method introduces is quite small compared to the standard STE. It only requires a buffer to store the delayed gradient and to evaluate the gradient of $\tilde{a}_i$ every k steps, which can be implemented using regular backpropagation. One key element of this method is that it is independent of the optimizer type and can be used with minor adjustments with any gradient-based optimizer.

Overall, we see that the Delayed updates method is advantageous in terms of its simplicity and low overhead as well as being generalized for different optimizers and QAT implementations. But it gives less accurate gradients with the k step delay, and it is less suited to rapid loss surface changes, which will make the delayed gradient immediately out of date. It is the simplest and least computationally expensive method we present, making it a middle-ground solution.

[Possibly insert algorithm here]

A.3 Diffrent a learning rates effect on the methods

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2012, pp. 1097–1105.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 5998–6008.

[3] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," arXiv:1604.07316, 2016.

[4] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[5] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 2704–2713.

[6] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, "A white paper on neural network quantization," *arXiv preprint arXiv:2106.08295*, 2021.

[7] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. de Freitas, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2013, pp. 2148–2156.

[8] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," *International Conference on Learning Representations (ICLR)*, 2016, arXiv:1510.00149.

[9] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *Neural Information Processing Systems Deep Learning and Representation Learning Workshop*, 2015, arXiv:1503.02531.

[10] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural networks," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 243–254.

[11] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, 2014, pp. 10–14.

[12] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International conference on machine learning*. PMLR, 2015, pp. 1737–1746.

[13] R. Banner, I. Hubara, T. Sosic, and R. H. Zemel, "Post training 4-bit quantization of convolutional networks for rapid deployment," in *Advances in Neural Information Processing Systems (NeurIPS) Workshop on Energy Efficient Machine Learning and Cognitive Computing*, 2018.

[14] M. Nagel, M. Cisse, T. Blankevoort, and M. Welling, "Up or down? adaptive rounding for post-training quantization," in *International Conference on Machine Learning (ICML)*, 2020, pp. 7197–7206.

[15] I. Hubara, R. Banner, A. Friedman, G. Berdugo, E. Hoffer, and D. Soudry, "AdaQuant: Improved quantization of neural networks via adaptive quantization parameter search," *arXiv:2004.01902*, 2020.

[16] Y. Cai, Z. Yao, Z. Yan, A. Gholami, M. W. Mahoney, and K. Keutzer, "ZeroQ: A novel zero-shot quantization framework," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 13 169–13 178.

[17] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to $\{-1, +1\}$," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2016, pp. 4107–4115.

[18] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," arXiv:1606.06160, 2016.

[19] S. Esser, J. Emer, R. Appeltant, R. Amin, C. Diorio, and D. Modha, "Learned step size quantization," in *International Conference on Learning Representations (ICLR)*, 2020.

[20] Y. Choi, M. El-Khamy, and J. Lee, "PACT: Parameterized clipping activation for quantized neural networks," in *International Conference on Learning Representations (ICLR)*, 2018.

[21] C. Yu, S. Yang, F. Zhang, H. Ma, A. Wang, and E.-P. Li, "Improving quantization-aware training of low-precision network via block replacement on full-precision counterpart," *arXiv preprint arXiv:2412.15846*, 2024.

[22] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[23] D. Kim, J. Lee, and B. Ham, "Distance-aware quantization," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 5271–5280.

[24] D. Lee and J. Jang, "Network quantization with element-wise gradient scaling," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021, p. 342–351.

[25] Y. Gong, X. Liu, Y. Wang, F. Wang, C. Tan, C. Lv, J. Qin, and T. Chen, "Differentiable soft quantization: Bridging full-precision and low-bit neural networks," in *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 4852–4861.

[26] Y. Wang, Z. Liu, Y. Wang, M. Wu, Z. Shen, K.-T. Cheng, and S. Han, "Error-aware quantization through noise tempering," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[27] H. Li, Y. Xu, X. Li, C. Liu, X. Sun, W. Liang, and Y. Jiang, "Custom gradient estimators are straight-through estimators in disguise," *arXiv:2302.01575*, 2023.

[28] J. Kiefer and J. Wolfowitz, "Stochastic estimation of the maximum of a regression function," *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 462–466, 1952.

[29] J. C. Spall, "Multivariate stochastic approximation using a simultaneous perturbation gradient approximation," *IEEE Transactions on Automatic Control*, vol. 37, no. 3, pp. 332–341, 1992.

[30] R. J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems.* Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2007.

[31] D. A. Knoll and D. E. Keyes, "Jacobian-free newton–krylov methods: A survey of approaches and applications," *Journal of Computational Physics*, vol. 193, no. 2, pp. 357–397, 2004.

[32] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv:1308.3432*, 2013.

[33] P. Yin, J. Lyu, S. Zhang, S. Osher, Y. Qi, and J. Xin, "Understanding straight-through estimator in training activation quantized neural nets," *arXiv preprint arXiv:1903.05662*, 2019.

[34] M. Andrychowicz, M. Denil, S. G. Colmenarejo, M. W. Hoffman, D. Pfau, T. Schaul, and N. de Freitas, "Learning to learn by gradient descent by gradient descent," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2016, pp. 3981–3989.

[35] A. G. Baydin, R. Cornish, D. M. Rubio, M. Schmidt, and F. Wood, "Online learning rate adaptation with hypergradient descent," *arXiv preprint arXiv:1703.04782*, 2017.

[36] L. B. Almeida and T. Almeida, "Parameter adaptation in stochastic optimization," *On-line Learning in Neural Networks*, pp. 111–134, 1998.

[37] K. Chandra, A. Xie, J. Ragan-Kelley, and E. Meijer, "Gradient descent: The ultimate optimizer," *Advances in Neural Information Processing Systems*, vol. 35, pp. 8214–8225, 2022.

[38] S. Chen, W. Wang, and S. J. Pan, "Metaquant: Learning to quantize by learning to penetrate non-differentiable quantization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[39] A. G. Baydin, R. Cornish, D. Martinez-Rubio, M. Schmidt, and F. Wood, "Online learning rate adaptation with hypergradient descent," *International Conference on Learning Representations (ICLR)*, 2018, arXiv:1703.04782.

[40] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*, vol. 58, pp. 13–30, 1963.

... - — - , -PDF —
-ii. ” ” , , — .

( ). 1000-2000 .
.

:

- .
- - .
- , , , , , ,
.

- ; -
\textenglish{} - , -PDF
. : . - :
\cite{Hoeffding}, : [40]; , : \textenglish{\cite{Hoeffding}},
[40] ( ).

-

. . .

” ”, -PDF ,
, (,). —
- , - -PDF —
i ...

,   .

.   (   ).   ,

.

,   ,   ,   '   ':

.

.

2025

—

”   2025