POLITECNICO DI TORINO

Master's Degree in Physics of Complex Systems

Master's Degree Thesis

Sparsification of deep neural network via ternary quantization

Supervisors

Prof. Enrico MAGLI

Prof. Giulia FRACASTORO

Prof. Sophie FOSSON

Prof. Andrea MIGLIORATI

Prof. Tiziano BIANCHI

Candidate

Luca DORDONI

December 2023

Abstract

In recent years, deep neural networks (DNNs) have achieved remarkable results in several machine learning tasks, especially in computer vision applications where they can often outperform human performance. Typically, deep models consist of tens of layers and millions of parameters, resulting in high memory consumption and computational overload. Conversely, the demand for smaller models is growing fast with the desire to deploy DNNs in environments with limited resources such as mobile devices. Methods to tackle this crucial challenge and obtain more compact networks while preserving performance rely on quantization or sparsification of the parameters. This thesis explores a combination of the two techniques, i.e. a sparsification method based on the ternarization of network parameters. Our approach is an extension of plain binarization of the parameters by adding a quantization interval centered around zero and of amplitude Δ such that parameters falling inside it are set to zero and removed from the network topology. Specifically, we use a ResNet-20 architecture to tackle the image recognition problem on the CIFAR 10 dataset. We show that increasing Δ as the training proceeds allows for sparsification rates over 90% while also ensuring improvement in classification accuracy over the binary framework. Despite the increased complexity required for implementing the ternarization scheme compared to a binary quantizer, we demonstrate that the remarkable sparsity rates translate to parameter distributions with significantly smaller average entropy (around $0.6 bits/symbol$) and therefore highly compressible sources. Our findings show substantial improvements and have significant implications for the development of more efficient deep neural networks.

Table of Contents

List of Tables

List of Figures

Acronyms

DNN
Deep Neural Network

NN
Neural Network

CNN
Convolutional Neural Network

RNN
Recurrent Neural Network

FFNN
Feedforward Neural Network

ReLU
Rectified Linear Unit

BN
Batch Normalization

ViT
Vision Transformer

SGD
Stochastic Gradient Descent

ADAM
Adaptive Moment Estimation

STE

Straight Through Estimator

BNN

Binarized Neural Network

TWN

Ternary Weight Network

TTQ

Trained Ternary Quantization

FP

Full-Precision

Chapter 1

Introduction

In recent years, significant technological improvements have been developed in machine learning and computer vision especially, where human abilities are being outperformed, and the demand for more sophisticated frameworks keeps rising. These advancements are progressively finding their way into embedded systems and lightweight devices, requiring high efficiency without losing performance. Within the landscape of machine learning, deep neural networks (DNNs) have emerged as a dominant force. Applications ranging from speech and facial recognition heavily lean on these frameworks. As an instance, significant progress has been made in the field of generative architectures, employing advanced neural networks like GPT models for text generation and Stable Diffusion for text-to-image tasks.

Deep neural network models typically have millions of parameters and tens of layers, which leads to high memory usage and computational overload. On the other hand, with the need to use DNNs in settings with constrained resources, like mobile devices, the need for smaller models is increasing quickly. Approaches to this problem rely on parameter quantization or sparsification, several techniques have been explored, with two of the most significant ones being pruning and quantization.

In this context, this thesis aims to deal with this challenge, proposing an approach that relies on a combination of those techniques. We employ the ternarization of the network's parameters during the training, to achieve both quantization and sparsification of the structure. Our approach is an extension of plain binarization of the parameters by adding a sparsification interval centered around zero and of amplitude Δ such that the network topology excludes parameters that fall within it. Tuning Δ, we aim to maximize sparsity without losing accuracy.

We employ a ResNet-20 architecture, to specifically address the image recognition problem on the CIFAR-10 dataset. As the training proceeds the Δ interval is increased, allowing to obtain greater sparsification rates. Our work has focused on finding the best growth regime for Δ, we show that a framework with a steep

increase in the first iterations of the learning process, followed by a rapid decrease yields an accuracy of over 90%. The research shows these results come with classification accuracies that exceed the accuracy of the binary framework.

Implementing a ternary quantizer is more complex than a binary one, but we prove that the significant sparsity results translate into smaller entropy values (up to 0.4 $bits/symbol$) and thus highly compressible models. Our research demonstrates marked improvements, denoting significant implications for the advancement of more efficient deep neural networks.

Throughout this research work, we'll go over the background of neural networks in chapter 2, starting from the origins of these frameworks and moving to their mathematical structure and the training procedures implemented to tune the parameters. Furthermore, in chapter 2, we discuss model compression, the challenge to reduce the size of neural network architectures, introducing the aforementioned pruning and quantization approaches, and introducing the binarization operation that forms the foundation upon which thesis work is built. In chapter 3 we will focus on the proposed method to address the challenge, reporting the ternarization operation and the implementation of an increasing Δ sparsification interval. Experimental findings are reported in chapter 4, along with the discussion on the results and graphs. Finally, chapter 5 reports the conclusions on the research we carried out and a discussion of possible further directions for our investigations on the topic.

Chapter 2

Background on neural networks and model compression

2.1 Neural networks

This section introduces neural networks with an overview followed by a brief subsection on their origins. It subsequently examines the neural network structure and the most common layers, focusing on the convolutional neural network architectures used in computer vision. In conclusion, NN training is explored.

2.1.1 An overview

Neural networks (NNs) represent a versatile machine learning technique with a multitude of applications. These networks are instrumental in addressing numerous tasks, demonstrating their efficacy in fields such as pattern recognition, including speech, and image recognition. Beyond this, they find utility in an array of domains, including natural language processing and predictive analytics, to name a few.

The adaptability of neural networks extends to various learning paradigms, with their utility spanning supervised learning, unsupervised learning, and reinforcement learning. This thesis focuses primarily on their applications in supervised learning. In the domain of supervised learning, neural networks undergo a training process using labeled datasets, which enables them to create mappings between provided inputs and the corresponding desired outputs. This approach allows neural networks to make accurate predictions or classifications. One of the most important applications of supervised learning is image recognition, a field that forms the

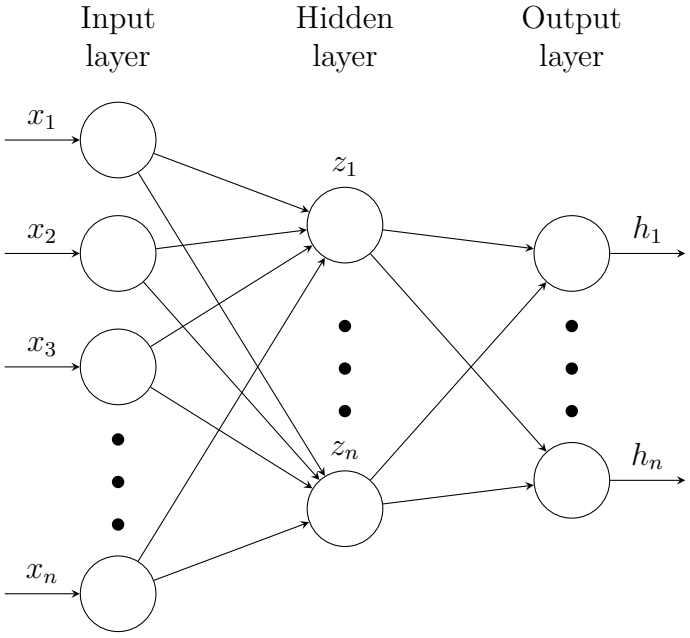


Figure 2.1: Neural network structure with one fully connected hidden layer.

central focus of this thesis. Neural networks have significantly advanced image recognition tasks, innovating the field of computer vision.

Unsupervised learning, on the other hand, focuses on the identification of latent patterns within data. Neural networks designed for unsupervised learning aim to reveal inherent structures or associations in datasets, even without the need for predefined labels. This category of machine learning is employed in data exploration, anomaly detection, and the extraction of information from complex and unstructured data. Furthermore, neural networks operate in the field of reinforcement learning, where autonomous agents seek to improve their performance by interpreting how to take specific actions within an environment governed by a system of cumulative rewards. Reinforcement learning operates on a "trial and error" principle, where the agent iteratively refines its strategies to maximize cumulative rewards over time. Neural networks have a crucial role in this process by enabling the agent to comprehend and respond to various environmental signals, thereby improving its decision-making abilities.

2.1.2 The origins

Neural networks represent a computational framework using interconnected elementary units known as neurons. This concept was originally introduced in 1943 by McCulloch and Pitts [1]. The foundational neuron model outlined by McCulloch

and Pitts featured a set of identical weights, a fixed threshold, binary inputs and outputs, and an inhibitory signal. According to their definition, a neuron would yield an output of "1" only if the inhibitory signal remained inactive and the weighted sum of the inputs exceeded the chosen threshold. Failure to meet either of these conditions would result in the neuron producing an output of "0".

In 1958, Rosenblatt's perceptron [2] refined this model by eliminating the concept of an inhibitory signal. This enhancement allowed for the assignment of different real values to the weights and bias for each distinct input. The advancement lies in providing an algorithm for learning these parameters. The influence of this discovery remains relevant in contemporary neuron design, with the primary distinction being that the activation function, as elaborated in subsequent sections, no longer necessitates a binary threshold.

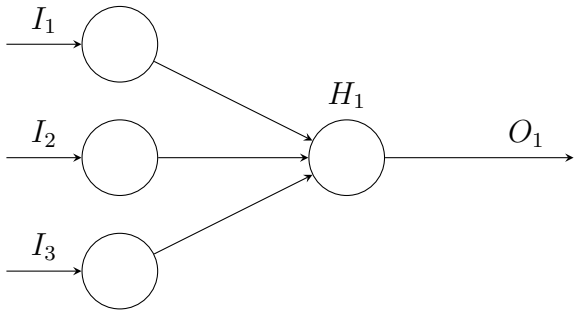


Figure 2.2: Schematic view of Rosenblatt's perceptron.

Nowadays, neural networks can be distinguished in *recurrent* or *feedforward* frameworks. Recurrent Neural Networks (RNNs), are employed for sequential data processing and are commonly used in tasks like natural language understanding and speech recognition. They are characterized by cycles in the structure. On the other hand, feedforward NNs (FFNNs) don't exhibit cycles and are utilized with independent input data.

This thesis will primarily concentrate on FFNNs, precisely on Convolutional Neural Networks (CNNs), an architecture that is mainly used in image and video analysis due to its ability to capture spatial hierarchies and patterns.

CNNs were first introduced in 1980 with the *neocognitron* by Fukushima and Miyake [4], and popularized by Yann LeCun [3] with LeNet-5. The network uses two sets of convolutional and pooling layers, followed by three fully-connected layers. This architecture was primarily applied to the task of classifying hand-written digits. LeNet-5 holds a crucial role in the advancement of deep learning for two main reasons. Firstly, Lecun proved that convolutional neural networks could be effectively trained using the backpropagation algorithm, a significant breakthrough in the field. Secondly, LeNet-5 laid the foundation for most CNN

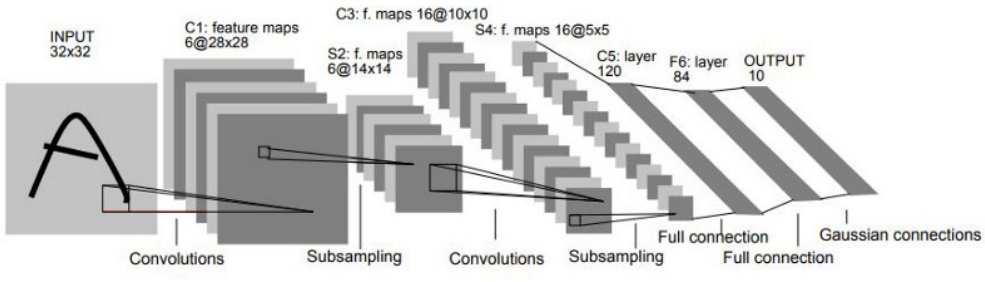


Figure 2.3: LeNet-5 architecture, featured in [3].

architectures, establishing the concept of network depth as the count of these alternating non-linear layers within the structure.

2.1.3 The structure

Artificial neural networks draw inspiration from their biological counterparts [5]. Indeed, the structure of artificial neural networks closely mirrors the interconnected neurons in the human brain. In both cases, information is passed from one neuron to another through a network of connections. Neural networks exhibit an architecture composed of interconnected *layers* (specifically *feedforward* architectures), see Figure 2.1 for reference. These layers include the input layer, one or more hidden layers, and the output layer. Each one is made of *artificial neurons* that process information. Neurons within one layer are connected to those in adjacent layers through weighted connections, and these connections enable the flow of information during the network's computations. The input layer receives data, which is then processed and transformed as it propagates through the hidden layers, ultimately leading to the network's output. The connections between neurons are associated with specific weights, which are adjusted during the training process to enable the network to make accurate predictions or classifications.

Therefore, the working unit in a neural network is the neuron, each one can take multiple inputs and output a single value according to Equation 2.1:

$$y = f\left(b + w^\top x\right) = f\left(b + \sum_{i=1}^{n} w_i x_i\right) \tag{2.1}$$

where y is the output of that single neuron, b and w are parameters of the NN, precisely, b is a bias introducing an intercept term and w represents the weighted connections between the input layer and the neuron, $x \in \mathbb{R}^n$ is the input itself. f is the activation function. In the deep learning landscape, the activation function is generally non-linear. This choice gives the neural network the ability to represent intricate and non-linear functions, it plays a crucial role in determining

the overall performance of the network. In the subsequent section, we present the most prevalent activation functions commonly employed in the literature:

- **Logistic function (sigmoid)**: $f(z) = \frac{1}{1+\exp^{-z}}$,
- **Hyperbolic tangent (tanh)**: $f(z) = \tanh(z)$,
- **Rectified Linear Unit (ReLU)**: $f(z) = \max(0, z)$,
- **Softmax**: $f(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, i = 1, \ldots, K.$

The *sigmoid* activation function produces output values within the range of $[0, 1]$, it had a prevalent position as the primary activation function in the early days of deep learning. Sigmoid is a strictly increasing and differentiable function, displaying nearly linear behavior around zero and saturation toward the extremes. It asymptotically approaches 0 and 1, which often correspond to the target values used in various classification tasks. One issue with this function is the vanishing gradient problem, particularly during backpropagation, which can block the learning process in deep networks.

The hyperbolic tangent [6] maps its input to a range between -1 and 1, providing a symmetric output centered around zero. The tanh function is advantageous because it avoids the saturation issues experienced with the sigmoid function, particularly for extremely deep architectures. One of the main benefits of the tanh function is that it has stronger gradients compared to the sigmoid. This characteristic allows the network to learn faster. However, similar to the sigmoid, the tanh function can still encounter vanishing gradients when used in very deep networks.

The ReLU [7] is widely used and has several advantages:

1. Simplicity: The ReLU function is simple to implement and computationally efficient. It is just a thresholding operation that doesn't involve complex mathematical computations.
2. Mitigation of vanishing gradient: ReLU helps alleviate the vanishing gradient problem. The reason is that ReLU provides a gradient of 1 for all positive input values, ensuring that gradients don't vanish as quickly during backpropagation.
3. Sparsity: ReLU introduces sparsity in the network, as it sets negative values to zero. Sparsity can improve network capacity, as only a subset of neurons is activated at any given time, leading to more efficient memory usage and faster computations.

Conversely, a drawback of employing ReLU activation functions is the potential for suppressing neurons. Additionally, ReLU, as the sigmoid function, is not zero-centered. Various ReLU variants have been developed to address these issues, the most notable are:

- Leaky ReLU:

$$f(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{otherwise,} \end{cases} \tag{2.2}$$

- Exponential Linear Unit(ELU):

$$f(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha \left(e^z - 1\right) & \text{otherwise.} \end{cases} \tag{2.3}$$

The final function to be mentioned is the *Softmax*. It is commonly used as the last activation function, just before the network's output. Softmax is particularly crucial for classification tasks because it serves as a probability distribution, normalizing the network's outputs.

2.1.4 The most common layers

Fully-connected layer

Neurons in NNs are organized in layers, the most basic one is the *fully-connected*, as the hidden layer is in Figure 2.1. It consists of neurons that are linked through weighted connections to all units in the previous layer. Consequently, the amount of parameters is $N_{l-1} \times N_l$, where N_l is the number of neurons in the l-th layer. The fully-connected layers use unique learnable weights for each connection, this means that it analyzes the input as a whole, without spatial awareness of the input structure. Most architectures position the fully-connected segment of the network at the network's end, specifically in CNNs. This arrangement makes sense because spatial correlation information becomes less crucial at the final classification stage, which is located at the network's output. Technically, any number of fully-connected layers could be attached, but the more there are, the more complex the network will be, leading to the learning process being difficult with a high risk of running into overfitting.

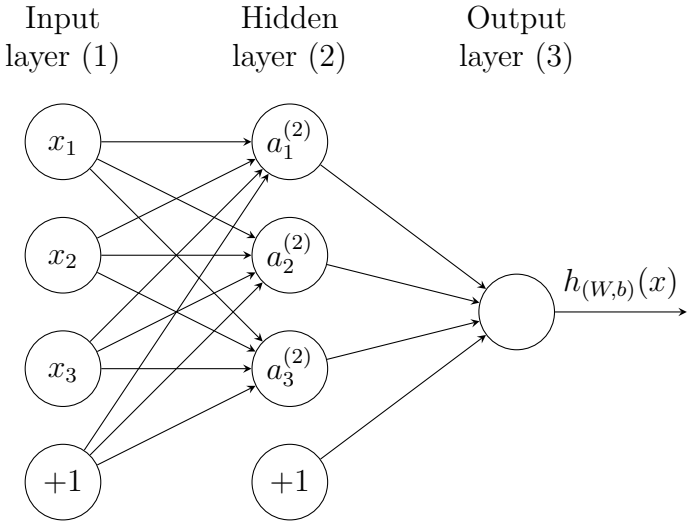


Figure 2.4: Hidden fully-connected layer structure.

Analyzing the structure of a fully-connected layer, following Figure 2.4, every hidden neuron receives signals from all inputs and a bias (always sending "+1"). Then, these are scaled by the network's parameters $(\mathbf{W}, \mathbf{b}) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$, where $W_{ij}^{(l)}$ is the weight on the connection between unit j (in the layer l) and unit i (in layer $l+1$), and $b_i^{(l)}$ is the bias bound to neuron i in layer l. Biases don't have any inward connection, as a matter of fact, they always signal "+1" which is multiplied by $\mathbf{b}$ values. The output of the hidden layer units are $a_i^{(2)}$, they represent the *activations* of the neurons, i.e. what the activation function outputs (input values $\mathbf{x}$ are also represented by $a_i^{(1)}$). The weighted sum of inputs going in unit i of layer 2 (the hidden layer in Figure 2.4), $z_i^{(2)}$, is given by Equation 2.4:

$$z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)} + b_1^{(1)}, \tag{2.4}$$

$z_i^{(2)}$ is the input to the activation function f (arbitrary choice), obtaining the *activation* of the neurons, following Equation 2.5:

$$\begin{aligned} a_1^{(2)} &= f\left(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}\right), \\ a_2^{(2)} &= f\left(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}\right), \\ a_3^{(2)} &= f\left(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}\right). \end{aligned} \tag{2.5}$$

In conclusion, the full Figure 2.4 output $h_{(W,b)}(x)$ is computed with these activations, according to Equation 2.6:

$$h_{(W,b)}(x) = f\left(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)}\right). \tag{2.6}$$

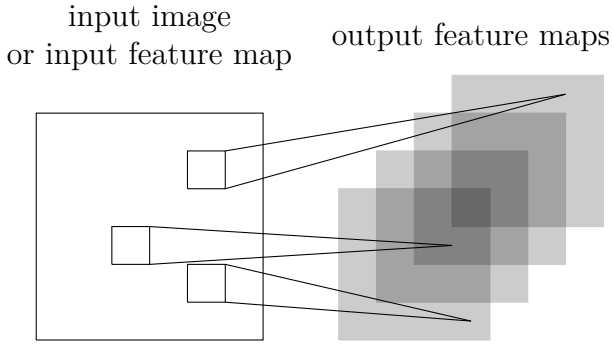


Figure 2.5: Visual representation of a convolutional layer. When layer l takes on the role of a convolutional layer, it processes either the input image (in the case of $l = 1$) or a feature map from the preceding layer by employing diverse filters. This process generates the output feature maps specific to layer l.

Convolutional layer

Convolutional layers are responsible for feature extraction using shared weights and locality principles. In a convolutional layer, inputs and outputs are organized as sequences of 2D maps. Each neuron associated with an output is connected only to a small spatial neighborhood of input maps. These local connections capture spatial information, as often, features meaningful in one part of the image are also meaningful in other parts.

Each output map is linked to a feature map, representing the spatial position where the feature was detected, thus the term "feature map". The number of maps in each sequence is known as channels, much like the channels in a color image. For example, in image recognition, the first convolutional layer connected to input images (known as the first layer) will have three input channels for RGB images and one for grayscale images.

A more detailed examination of shared weights and locality reveals that the convolutional layer's output can be computed by convolving the input with the set of shared weights, which is why it's called a convolutional layer. In this context, each group of local weights is referred to as a *kernel*. It's worth noting that each kernel must have the same number of channels as the input signal. The kernel acts as in Figure 2.6, moving across the input image or feature map. As the filter moves the input, it performs element-wise multiplications and sums to produce a feature map, which highlights areas where certain patterns are detected. Generally, multiple kernels are employed in a single convolutional layer, each one yielding a different output map that will be fed to the next layer. The ongoing mathematical operation for a single-channel image is expressed in Equation 2.7:

$$a(x, y) = f\left(\sum_m \sum_n I(x + m - 1, y + n - 1)W(m, n) + b\right), \tag{2.7}$$

where (W, b) are the parameters, W being the weights of the kernel and b the bias,

$I(x,y)$ represents the value of the pixel in x and y.

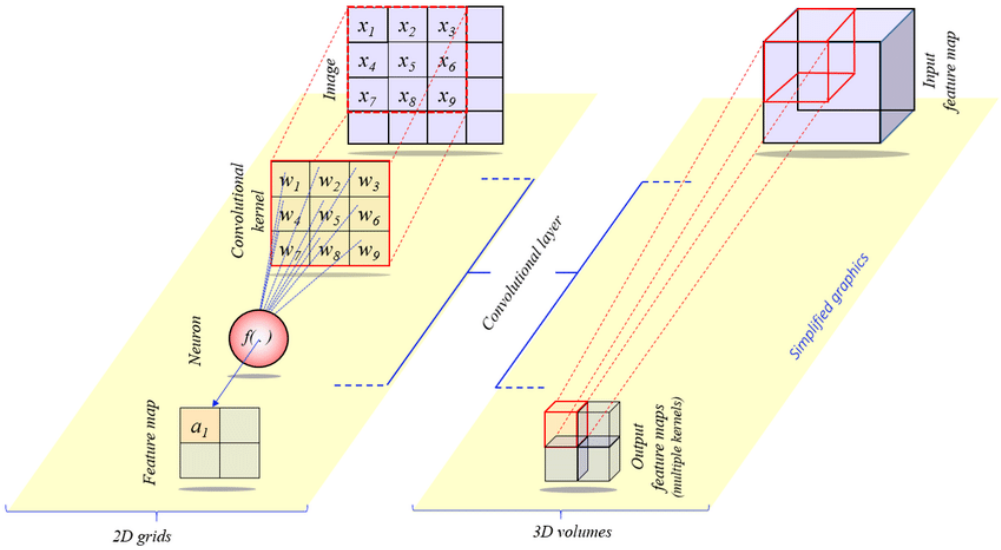


Figure 2.6: Kernel in action. (Left) In 2D grids, a single kernel crosses over the image or input signal. (Right) A volume containing multiple kernels traverses the input volume, producing output volumes.

The convolutional layers can be stacked on top of each other, to some extent, the primary function of the first one is identifying basic patterns, whether the more intricate patterns are analyzed by deeper layers within the network.

Pooling layer

Pooling layers are used to downscale the input data by performing a sample-based discretization. This downsizing reduces the dimensionality of the input. Additionally, pooling offers benefits such as preventing overfitting and reducing computational complexity by decreasing the number of learnable parameters. Pooling is typically performed using a filter, which slides over the input feature map with a predefined stride. As it crosses over the input, this filter groups a set of neighboring values and applies a specific operation, such as maximum or average pooling, to generate a single output value. By controlling the size of the filter and the stride, the degree of downsampling can be adjusted. Importantly, the choice of filter size and stride directly impacts the output size of the pooled feature map. Pooling is an essential component in various CNN architectures. As previously mentioned, two primary approaches can be considered, as depicted in Figure 2.7:

- Max Pooling

- Average Pooling

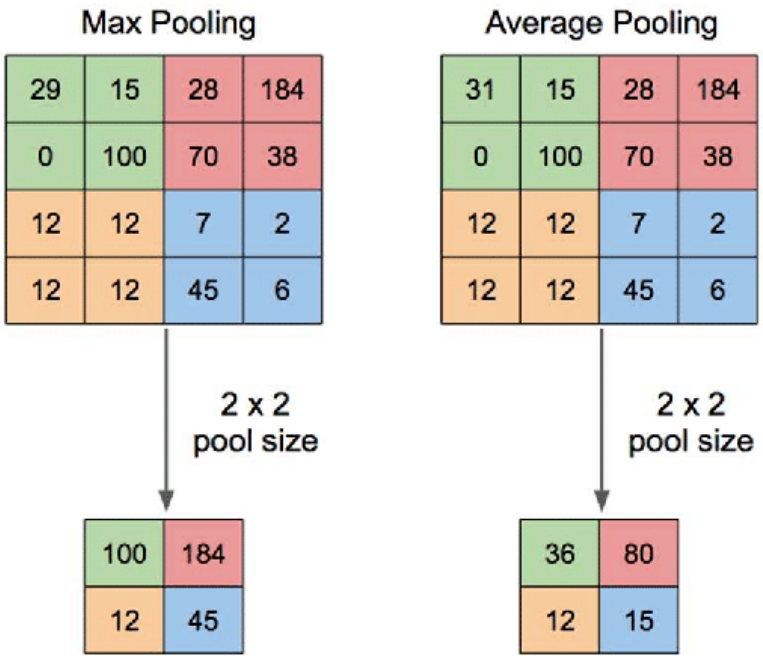


Figure 2.7: Max pooling and average pooling visualization [8, © Muhamad Yani *et al.*].

Max pooling is implemented as a layer positioned between the stacked convolutional layers, and it outputs the maximum value obtained from a small kernel window applied to non-overlapping subregions across the entire input. If the final classification doesn't require spatial information, a final max pooling filter can be employed to achieve complete translation invariance. Indeed, by applying this layer at the end, the focus is placed on the most important features without being concerned about their precise spatial positions. Max pooling only works on the spatial dimensions (x, y), not on the depth of its inputs (i.e. the number of features maps), its action is shown on the left in Figure 2.7.

Average pooling, similarly to max pooling, serves as a layer within the sequence of convolutional layers and functions to downsample the input, consequently reducing its dimensionality. While max pooling emphasizes the most prominent feature within a small kernel window, average pooling, on the other hand, calculates the mean value of the elements in that window. Analogously to max pooling, average pooling operates solely on the spatial dimensions (x, y) of the input, preserving the depth of its inputs. Its functionality can be observed on the right side of Figure 2.7, in which the focus is placed on the average values within the kernel

window, allowing for a less pronounced selection of dominant features.

Batch Normalization

Batch normalization (BN) is a technique for normalizing the inputs to a neural network layer, applied to either the activations of a prior layer or inputs directly. The batch normalization layer normalizes its input data within each mini-batch during training. It was proposed by Sergey Ioffe and Christian Szegedy in 2015 [9]. They discovered that BN significantly accelerated training, in some cases by halving the epochs or better, and provided some regularization, reducing generalization error. From a mathematical point of view, BN acts as follows (Equation 2.8):

$$\widehat{x}^{(l)} = \gamma^{(l)} \left[\frac{x^{(l)} - E\left[x^{(l)}\right]}{\sqrt{\mathrm{Var}\left[x^{(l)}\right]}} \right] + \beta^{(l)}, \tag{2.8}$$

where $x^{(l)}$ is the input to the layer, while $\widehat{x}^{(l)}$ is the normalized output, $E(\cdot)$ and $\mathrm{Var}(\cdot)$ respectively perform expected value and variance. The parameters $\gamma^{(l)}$ and $\beta^{(l)}$ are learnable values that let the output deviate from a unit Gaussian bell. For example, the network during the learning process could find optimal the values of $\gamma^{(l)} = \sqrt{\mathrm{Var}\left[x^{(l)}\right]}$ and $\beta^{(l)} = E\left[x^{(l)}\right]$, effectively recovering the initial inputs and reversing the batch normalization application.

Batch Normalization yields significant advantages:

- **Stability**: Deep neural networks can be unstable during training, BN addresses this issue by normalizing the input activations to each layer, ensuring that the distribution of activations remains consistent throughout training. This stabilizes the training process and allows the network to learn more effectively.
- **Improves gradient flow**: BN helps to improve the gradient flow through the network, which can further enhance stability. By normalizing the activations, BN helps to reduce the impact of outliers and noisy data, allowing the network to focus on the more meaningful patterns in the data.
- **Decreases dependence on the initialization**: Batch normalization can make the learning process less sensitive to weight initialization. This is because BN helps to distribute the activations of each layer more evenly, which can help to prevent the network from becoming overly dependent on the initial values of the weights.

2.1.5 CNN architectures

Convolutional neural networks (CNNs) have become renowned machine learning techniques in the field of computer vision, achieving remarkable performance in

tasks such as image classification, object detection, and image segmentation. This subsection will explore some of the most important CNN architectures that have been developed over the years.

AlexNet [10]

AlexNet by Alex Krizhevsky *et al.* won the 2012 Large Scale Vision Challenge (ILSVRC) [11], with an architecture composed of five convolutional layers and three fully-connected layers. It achieved a top-5 classification error of 15.3%, beating the second-placed and the previous winner by more than ten percentage points. The schematic framework is shown in Figure 2.8.

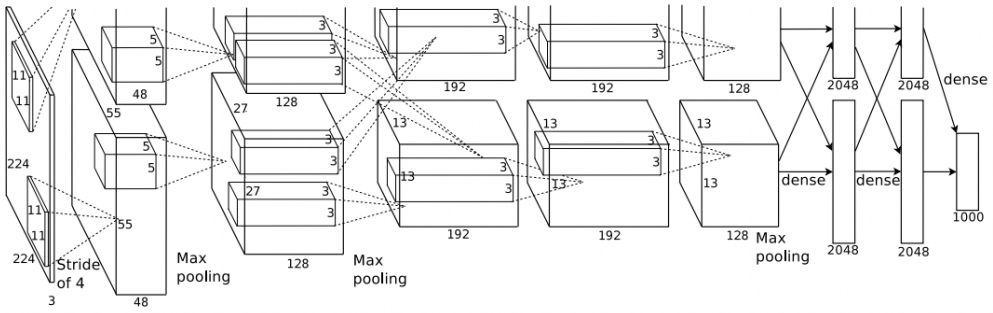


Figure 2.8: A breakdown of AlexNet's structure, originally published in [10]. Two different GPUs work on parallel paths to decrease the computational time.

VGG [12]

In 2014 the ILSVRC competition was won by a new CNN architecture, the VGG Net, introduced by Karen Simonyan and Andrew Zisserman. One of the remarkable differences between VGG Net and its predecessor, AlexNet, is its depth, while the latter introduced the idea of using deep CNNs for image classification, the former took this concept further by employing an even deeper network. The VGG network, particularly the VGG-16 and VGG-19 variants, consists of a significantly larger number of layers compared to its precursor. VGG's deep architecture is characterized by its use of 3x3 convolutional filters and 2x2 max-pooling layers, making it more uniform and easier to understand and implement. This characteristic contributed to the interest in VGG Nets, as it enabled the construction of very deep networks while keeping the architecture straightforward. Despite the increased complexity, VGG Net proved superior performance in various image classification tasks.

GoogleNet [13]

Researchers at Google in 2015 introduced a new fundamental concept, the *Inception* module (Figure 2.9), that was employed in their architecture, GoogleNet, a 22-layer deep CNN.

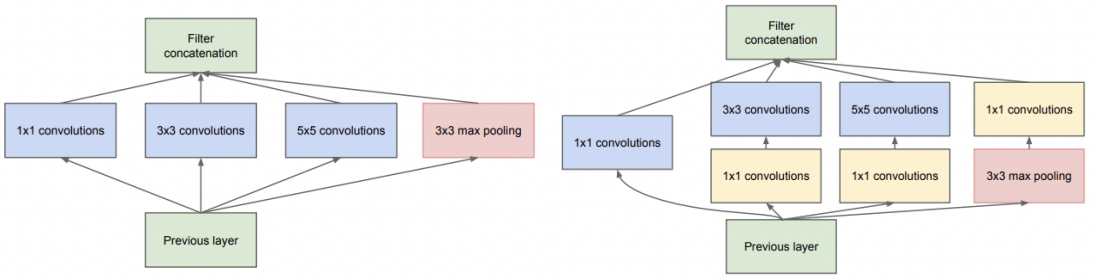


Figure 2.9: Inception modules found in GoogleNet, on the left a so-called *naïve* unit, on the right a more complex module that includes dimensionality reduction with specific convolutional and pooling layers. Originally appeared in [13].

The inception module was introduced to address the challenge of designing deep neural networks while managing computational complexity. These modules consist of a *local* network embedded in the whole NN, they are then stacked on top of each other. The main idea behind the module is to perform parallel convolution operations with different filter sizes and a max-pooling operation, and then concatenate the results along the depth dimension. This parallel processing enables the network to capture features at different levels of abstraction. Additionally, as shown on the right in Figure 2.9, 1x1 convolutions are used to reduce the dimensionality before applying larger filter sizes, which helps reduce the computational cost. GoogleNet managed to achieve state-of-the-art performance on a variety of image classification tasks while still keeping computation efficient.

ResNet [14]

ResNet was developed in 2016 by He *et al.* [14], at present time, it still is one of the most used architectures in the computer vision field, frequently serving as a benchmark. ResNet, short for Residual Networks, instead of trying to directly map inputs to outputs, focuses on learning residual functions, quantifying the difference between the output and the input. This approach has great performance with deep models, reaching up to 152 layers, for instance, it achieved a 3.57% top-5 error on the ImageNet dataset. This architecture employs residual blocks, Figure 2.10, consisting of *skip connections*, that let the gradients flow easily during backpropagation, facilitating the process, and enabling deep networks to learn

accurately.

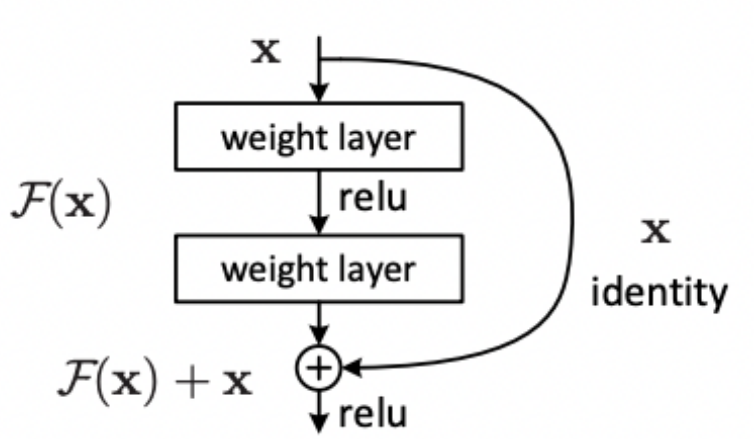


Figure 2.10: Residual block in ResNet architectures.

This research employs a ResNet-20 architecture, used to address the image classification of the CIFAR-10 dataset.

GANs [15]

The previously cited architectures are employed in computer vision networks mainly for their classification capabilities. On the other hand, GANs popularized in 2014 by Goodfellow *et al.* [15], are employed in other fields, notably in the *image generation* one. Their ability to generate realistic pictures is necessary for tasks like image creation, data augmentation, and super-resolution imaging. These networks consist of a generator and a discriminator (neural networks themselves) engaged in an adversarial training process. The generator produces data, while the discriminator has to distinguish real from generated data. They keep improving their performance to the point of equilibrium, where the generator is able to produce a high-quality output.

Transformers [16]

"Attention is all you need" by Vaswani *et al.* (2017)[16] introduced a new concept in the deep learning landscape, *attention*. The authors implemented a new architecture, called the *transformer*, they were initially popularized in natural language processing, but more recently have made an impact on computer vision tasks as well. The key innovation lies indeed in their attention mechanism, which assigns varying weights to different parts of the input sequence, allowing the model to focus on certain elements while considering the whole context of the input.

Applied to computer vision, transformers differ from the conventional convolutional neural network approach by using attention mechanisms. This enables the model to capture contextual information across the entire image, providing a global understanding of the data. Vision Transformer (ViT) [17] is an example of such architectures. The ability to consider global relationships has proven to be effective in tasks such as image classification and object detection, confirming the importance of transformers in computer vision.

2.1.6 Training feedforward neural networks

Training is the process by which the network learns from input data. Through the use of a learning algorithm, the network is exposed to labeled training data in this process, which enables it to modify its internal parameters (in this section, only *supervised* learning will be treated).

Backpropagation is an essential technique in neural network training, which allows the network's parameters to be optimized iteratively depending on the error or loss between the expected and actual outputs. Backpropagation is essential for adjusting the internal weights and biases of the neural network. We will discuss this fundamental algorithm in the next section.

Backpropagation

The backpropagation algorithm is a tool that's used in teaching the network. It tells how much the network's weights (and biases) should be modified to adapt to the labeled input data, to classify them correctly. Specifically, the parameters of the architecture are randomly initialized (this work used the so-called *He* initialization), which means that the classification in the first stages of the learning process will be more or less casual. The NN output will produce a certain *loss* (or *cost*) which is computed using the architecture's hypothesis and the correct labeled data. Backpropagation aims to reduce it and does so by looking for minima in the loss landscape, from a mathematical point of view it searches for the values of the parameters that minimize the cost. Most commonly, *cross-entropy* is the employed loss function in classification tasks, its mathematical formula is displayed in Equation 2.9.

$$L(W,b) = -\frac{1}{m}\sum_{t=1}^{m}\sum_{i=1}^{C}\mathbb{I}\left(i = y^{(t)}\right)\log\left(h_{W,b}(x^{(t)})_i\right), \tag{2.9}$$

where i runs over the possible classes and t over the labeled data samples, m is the number of samples, $y^{(t)}$ is the true label and $h_{W,b}(x^{(t)})_i$ is the network's prediction for the i-th class. $\mathbb{I}$ is the indicator function which tells if the current i index is pointing to the correct class.

The architecture of the network is composed of layers that are connected to the previous ones. Hence, the output of each single one of them can be interpreted as a function of the outputs of the previous layers, ultimately leading to the net's final prediction. By utilizing the derivative chain rule to transmit backward the error gradient computed on the cost function, which gives rise to the technique's name, the backpropagation algorithm takes advantage of the network's layered structure.

Updating parameters

Once the gradient $\nabla_W L$ with respect to a specific parameter (weight or bias) has been computed through backpropagation, the parameter needs to be updated. Various parameter-update algorithms have been developed over the years. This thesis explores the foundation of the most basic algorithm, *Stochastic Gradient Descent* (SGD), and introduces another that has been applied in the conducted simulations, *Adaptive Moment Estimation* (ADAM).

The most popular optimization method for NN training is SGD [18], notably for CNNs. Similar to traditional gradient descent, the algorithm updates the network parameters in the opposite direction to the gradient, it is displayed in Equation 2.10. The only difference is that samples are chosen at random, or shuffled, rather than according to their order in the training set. After training is completed, this improves the network's capacity for generalization. SGD works by taking the current vector of parameters and subtracting the gradient vector, which is weighted by a positive constant λ, known as *learning rate*.

$$W_{ij,t+1}^{(l)} = W_{ij,t}^{(l)} - \lambda \nabla L\left(W, b\right), \tag{2.10}$$

where $W_{ij,t+1}^{(l)}$ is the transformed weight, $W_{ij,t}^{(l)}$ is the weight prior to the SGD update, λ the learning rate and $\nabla L\left(W, b\right)$ the gradient of the loss function with respect to that weight.

ADAM [19] is an algorithm that actively adjusts the learning rate for every parameter. Specifically, ADAM relies on adaptive estimates of lower-order moments and optimizes stochastic objective functions using first-order gradients. The idea behind ADAM is to split the learning rate for a given weight by calculating running averages of the second moments of the gradients and the magnitudes of recent gradients for the same weight. This leads to a method for rescaling gradients that is both parameter and training-data invariant and computationally efficient, making it suitable for large problems.

2.2 Model compression

Since their introduction, the demand for better and more memory-efficient neural networks has increased considerably. For this reason, researchers developed techniques to reduce memory consumption and the size of the newer architectures. This section targets model compression, involving the aforementioned techniques and focusing on binary training, that lays the foundation for our proposed method in chapter 3.

2.2.1 Pruning

Pruning dates back to 1989, with "Optimal brain damage" by LeCun *et al.* [20]. In the last decade, it has been extensively researched and applied in various forms. It is implemented to reduce the number of unnecessary weights in a neural network, it typically works by identifying the parameters with the least impact on the NN performance and removing them. Two are the main procedures: structured and unstructured pruning.

Structured pruning is used to remove whole neurons, layers, convolutional filters, and full blocks for more complex neural networks, resulting in the shrinking of the number of parameters. This method is often preferred to its unstructured counterpart because it's more likely to result in a more systematic reduction of the network.

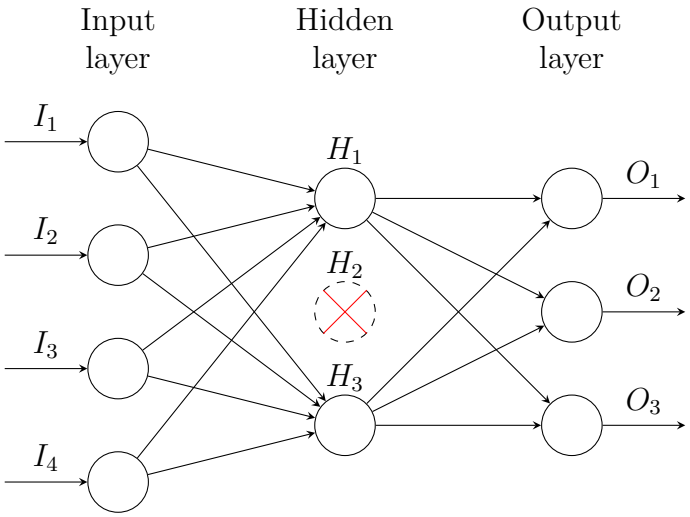


Figure 2.11: Example of a pruned neuron neural network.

On the other hand, unstructured pruning aims to remove individual weights. It is frequently implemented with magnitude pruning, which consists of the removal

of the weights whose value is below a certain threshold. It is the most basic pruning technique. Magnitude pruning creates a sparse network, where the smallest weights are taken away, resulting in a lighter model, and enhancing its memory efficiency.

Some regularization techniques can aid the pruning procedure. For instance, $L1$ and $L2$ regularizations add penalty terms to the loss, gearing the weight values towards 0, encouraging an intrinsic sparsification of the parameters.

Pruning can be applied to the fully trained model, directly acting on the final values of the parameters. Alternatively, it can follow an iterative approach, requiring a fine-tuning stage that occurs after the removal of the unnecessary weights (neurons and layers in the structured procedure). Specifically, the network is trained up to a certain level, then it undergoes a pruning step, which is followed by retraining, and so on iteratively.

Figure 2.12, reported in Han *et al.*'s work [21], shows that applying pruning to the fully trained network without retraining degrades the accuracy much faster compared to when using fine-tuning. Even better results are obtained by employing iterative pruning. Moreover, it has the effect of depleting the number of weights with values near 0, which is slowly repopulated by retraining, as depicted in Figure 2.13. The results are derived from iterative pruning on AlexNet [10].

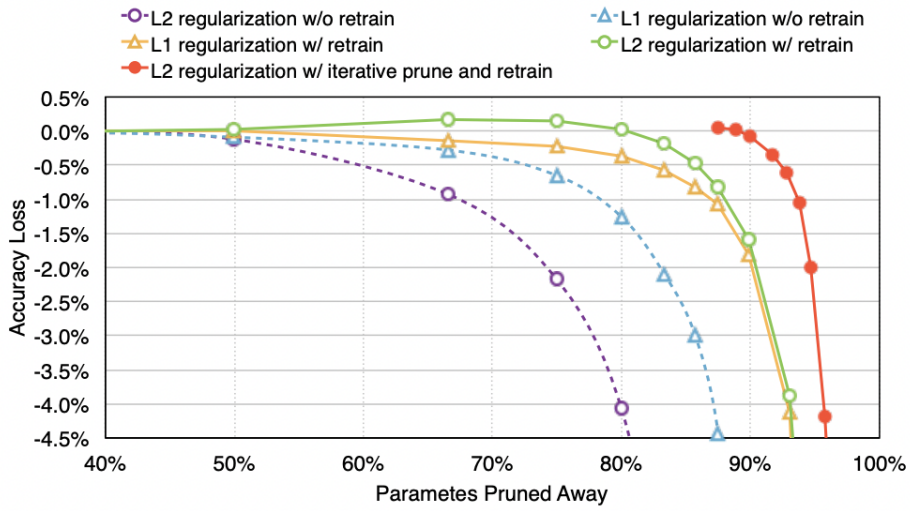


Figure 2.12: The effect of pruning on the top-5 accuracy loss.

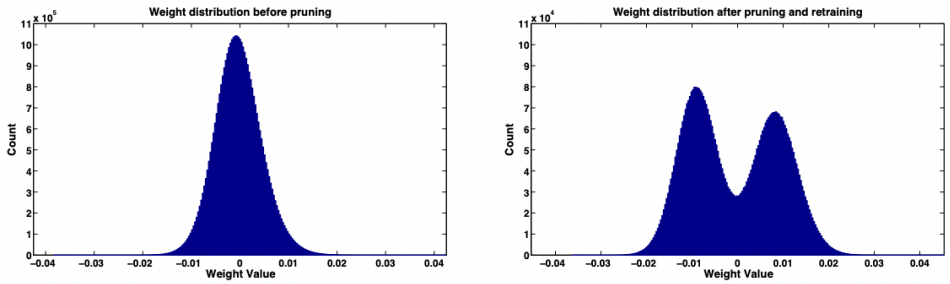


Figure 2.13: Weights distribution, before pruning and after [21].

2.2.2 Quantization

Typically, weights are stored as 32-bit floating point numbers, which provide high precision but consume a significant amount of memory and power. In the context of neural networks, quantization techniques offer a solution to reduce this memory usage while still preserving the model's ability to perform tasks effectively.

Quantization is the process of decreasing the precision of the weights of a neural network (and/or the activations). It converts 32-bit floating point numbers to lower precision types, considerably reducing their memory uptake. Furthermore, it allows for faster computation, indeed, lower-precision parameters require less computational power. However, keeping minimal precision can lead to the degradation of the neural network's performance. A common quantization approach utilizes the following expression [22]:

$$Q(r) = Int(r/S) - Z \tag{2.11}$$

r is the weight or parameter that needs to be quantized (frequently neuron activations are quantized as well), S is the scaling factor, Z is an integer representing the origin point, and $Int(\cdot)$ is generally a rounding function.The scaling factor deserves special attention, it's a parameter that defines the step between quantized levels, and it's usually computed with Equation 2.12:

$$S = \frac{\beta - \alpha}{2^b - 1} \tag{2.12}$$

$[\alpha, \beta]$ represents the clipping range, a predefined interval used to restrict real values, whereas b signifies the bit precision. The clipping range choice (calibration) can lead to symmetric quantization when $|\alpha| = |\beta|$, to asymmetric quantization if $|\alpha| \neq |\beta|$.

Two methods are common for quantizing neural networks:

- Post-Training Quantization (PTQ)

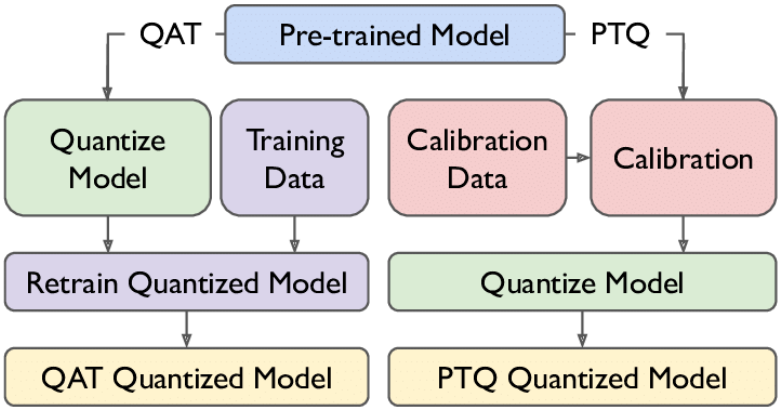


Figure 2.14: Visual comparison of quantization methods. Figure by Olivia Weng [23], licensed under CC BY 4.0.

- Quantization-Aware Training (QAT)

The first involves applying quantization after the whole network has been trained, doing so may lead to losses in the network's performance, which was built around 32-bit numbers and consequently loses information when dropped to lower precision numbers. Calibration is performed on a small dataset to fix the quantization parameters. Occasionally, clustering is utilized to find centroids around which the quantization is performed, Deep Compression [24] introduced by Han *et al.* in 2015 is a renowned example of such a procedure. PTQ is often used when training data is not sufficient to perform QAT.

On the other hand, Quantization-Aware Training is a process that entails retraining a model with quantized parameters. This allows the model to learn and compensate for any quantization bias that may arise due to rounding errors. The standard forward and backward passes are executed on the quantized model in floating point, but the model parameters are quantized following each gradient update. Backpropagation in these cases involves a complex aspect, dealing with the non-differentiable quantization operator. A popular way to tackle this is by using the Straight Through Estimator (STE) [25], which avoids the quantization operation and treats it like an identity function.

Table 2.1: A comparative analysis of the two quantization procedures (reproduced from [23]).

	Accuracy loss	**Training time**	**Minimum achievable precision**
Post-Training Quantization	Moderate	Low	$\geq$ 4 bits
Quantization-Aware Training	Negligible	High	$\geq$ 1 bit

The research we present employs a combination of both pruning and quantization procedures.

Binary quantization

Quantization techniques can be extended to minimal precision reduction, even leading to weights being binary, occupying just $1bit$. In 2015 Courbariaux *et al.* applied *binarization* to the weights of the networks with BinaryConnect [26]. They intended to reduce the memory occupation of the weights and decrease the number of multiplications performed by the model, employing bitwise operations. Therefore, the weights are constrained to $+1$ or -1 values, employing one of two possible binarization techniques, deterministic and stochastic binarization (represented in Equation 2.13 and Equation 2.14, reproduced from [26]).

- Deterministic binarization: involves binarizing the weight according to a simple sign function:

$$W_q = \begin{cases} +1 & \text{if } W >= 0 \\ -1 & \text{if } W < 0, \end{cases} \tag{2.13}$$

  where W is the full-precision weight and W_q is quantized.

- Stochastic binarization: binarizes weights using a probabilistic approach:

$$W_q = \begin{cases} +1 & \text{with probability } p = \sigma(W) \\ -1 & \text{with probability } 1 - p, \end{cases} \tag{2.14}$$

  where $\sigma(W)$ is a probability distribution computed with:

$$\sigma(W) = \max\left(0, \min\left(1, \frac{W+1}{2}\right)\right) \tag{2.15}$$

In 2016 "Binarized Neural Networks" (BNN) [27] popularized binary frameworks. Their major contribution was being able to binarize both weights and neuron activations with their architecture. This thesis expands BNN to ternary parameters, the proposed method is outlined in chapter 3.

2.2.3 Other compression techniques

One of the numerous compression techniques involves knowledge distillation, introduced in the paper "Distilling the Knowledge in a Neural Network" [28]. It offers a solution to the memory consumption challenge by transferring knowledge from large NN models to smaller "distilled models" without sacrificing performance. The increasing trend in Deep Learning of training larger models for superior results has

led to practical deployment issues due to their size. Rather than solely focusing on trained parameters, the authors propose a novel perspective, suggesting that neural network knowledge is represented as a learned mapping between input and output. This approach leverages class probabilities, known as "soft targets", predicted by the more complex model to train simpler networks, while still maintaining accuracy. The introduction of a "softmax with temperature" parameter allows for the creation of a softer probability distribution over classes, which enhances the learning process. Experimental results on the MNIST dataset and in speech recognition tasks demonstrate that training smaller models with soft targets outperforms models trained with actual targets.

Chapter 3

Methodology

In this chapter, we present a brief section on the background of neural network ternarization and follow it with a section on the method we adopted. It focuses on addressing the memory-usage challenge outlined in chapter 1, primarily through the development of a comprehensive approach that centers on both the sparsification and quantization of the neural network architecture.

3.1 The ternarization

In 2016 Liu *et al.* [29] exploited ternarization in their work "Ternary weight networks" (TWNs), trying to achieve a compromise between full-precision (FP) architectures and binary counterparts. In TWNs every parameter is quantized to either $+1$, 0, or -1, therefore occupying $2bit$, highly reducing memory usage. It is compressed around 16 times more than a $32bit$ floating point number.

Weights are quantized via a thresholding operation, according to Equation 3.1:

$$W_q = \begin{cases} +1 & \text{if } W > \Delta \\ 0 & \text{if } |W| \leq \Delta \\ -1 & \text{if } W < -\Delta, \end{cases} \tag{3.1}$$

where W_q is quantized and W is full-precision, Δ represents the positive threshold parameter. The value for Δ is here computed through an optimization procedure, notably, via minimization of the Euclidean distance between W_q and W.

Another crucial contribution was provided by Zhu *et al.* [30], proposing "Trained Ternary Quantization" (TTQ) in 2016. The ternarization approach is based on the weights $\{-W_n^{(l)}, 0, W_p^{(l)}\}$ instead of $\{-1,0,+1\}$, with the superscript (l) standing for the layer number, indeed these quantization values are layer-dependent. Weights

are quantized following Equation 3.2:

$$W_t^{(l)} = \begin{cases} W_p^{(l)} & \text{if } W^{(l)} > \Delta^{(l)} \\ 0 & \text{if } \left|W^{(l)}\right| \leq \Delta^{(l)} \\ W_n^{(l)} & \text{if } W^{(l)} < -\Delta^{(l)}, \end{cases} \tag{3.2}$$

with $\Delta^{(l)}$ being the threshold linked to the layer l. They employ a constant t for all layers, a parameter with which $\Delta^{(l)}$ is computed, according to Equation 3.3.

$$\Delta^{(l)} = t \times \max\left(\left|W^{(l)}\right|\right). \tag{3.3}$$

The TTQ training procedure leverages two different gradients, one has the role of adjusting full-precision weights upon which the quantization is developed, and the other is needed to update the quantized ternary weights $W_p^{(l)}$ and $W_n^{(l)}$, Figure 3.1 displays a simple illustration of the process.

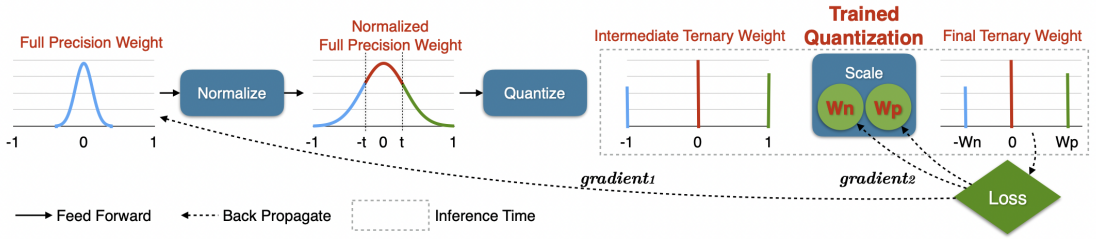


Figure 3.1: Schematic representation of TTQ, originally appeared in [30]. Full-precision weights are normalized to the range $[-1, +1]$ and then quantized. This process entails two gradients, one involved in the update of the weights and the other in updating the quantized weight values.

3.2 Proposed method

In continuation to the exploration of neural network ternarization presented in the preceding section, we turn the focus to the description of our proposed methodology. This method aims to effectively sparsify and quantize network architectures right from the initial stages of the learning process, employing quantization-aware training.

The quantization is performed, similarly to Equation 3.1, with a thresholding operation indicated in Equation 3.4:

$$\theta_q = \begin{cases} +1 & \text{if } \theta \geq \Delta \\ 0 & \text{if } |\theta| < \Delta \\ -1 & \text{if } \theta \leq -\Delta, \end{cases} \tag{3.4}$$

as is customary θ_q is quantized while θ is the full precision parameter. The sparsification interval Δ is not layer-dependent and it influences the whole architecture, moreover, the quantized weights are fixed to $+1$, 0, and -1 and are not tuned as in TTQ. However, similar to TTQ, our model necessitates recording full-precision latent parameter values used during the ternarization stage.

The proposed method applies the ternarization function to the FP weights and activations in the forward pass, effectively sparsifying the network. Namely, all parameters that fall within $(-\Delta, +\Delta)$ are set to zero. This entails simplified computational calculations, primarily dominated by multiplications that can be readily set to 0 when the architecture is sparse. These weights don't contribute to the network's output and consequently, to the calculation of the loss. As discussed in [29], the *zero* values introduced by the ternarization allow the model to be compatible with dedicated deep-learning hardware for even more efficient computation of the mathematical operations.

The activations are quantized via Equation 3.4, while the FP weights are both clipped to $[-1, +1]$ and ternarized. Clipping is a common practice in many binary or ternary frameworks [26], this precaution is taken to prevent their values from drifting away from the $[-1, +1]$ interval, and acquiring large values that would have no impact on the quantization scheme.

The significance of full-precision parameters becomes clear during the backward pass. During the backward phase, it is essential to compute the gradients to determine the descent direction in the loss landscape for error minimization. This cannot be achieved with ternary weights since they are obtained through the non-differentiable ternarization function. Therefore, we compute these derivatives with respect to the full-precision weights. Furthermore, accumulating updates on the quantized weights would barely see a modification, as the quantization step is much bigger than what a simple optimization step could change.

Backpropagation algorithms employed to compute gradients encounter challenges with non-differentiable functions as Equation 3.4. In 2013, Bengio *et al.* proposed a solution known by the name of *straight through estimator* [25]. It involves using the gradient of the identity function during the backward pass, allowing gradients to flow through non-differentiable elements. This strategy enables the incorporation of non-differentiable operations, such as ternarization, into the backpropagation process. Similarly to BNN [27], we use a version of the STE that accounts for the saturation effect, as stated by Equation 3.5:

$$g_r = g_q \times \mathbb{1}_{|r|\leq 1}, \tag{3.5}$$

where g_q is an estimator of the gradient $\frac{\partial L}{\partial q}$, g_r is the gradient of the loss with respect to r, where q and r are related through $q = \text{Ternarize}(r)$. Effectively in the backward pass, the non-differentiable operation is bypassed by an identity function.

The update is calculated on the full-precision parameters, which are then clipped to $[-1, +1]$ as previously mentioned. We employ an Adam optimizer with learning rate (λ) decay.

The overall outline of the method is presented in Algorithm 1 and Algorithm 2.

Algorithm 1 Outline of our ternarization procedure in the forward pass. The Ternarize() function operates the ternarization of activations and weights. BatchNorm() computes the normalization with parameters ϕ. $W^{(k)}$ is the weight of layer k, Δ the threshold for 0 in the ternary setup, and $a^{(k-1)}$ is the input to layer k.

procedure FORWARD PASS
  for k = 1 to N **do**
    $W_q^{(k)} \leftarrow \text{Ternarize}(W^{(k)}, \Delta)$ ▷ Ternarize full-precision weights
    $z^{(k)} \leftarrow a_q^{(k-1)} W_q^{(k)}$ ▷ Compute multiplication operations
    $a^{(k)} \leftarrow \text{BatchNorm}(z^{(k)}, \phi^{(k)})$ ▷ Apply batch normalization
    if k < N **then**
      $a_q^{(k)} \leftarrow \text{Ternarize}(a^{(k)}, \Delta)$
    end if
  end for
end procedure

Our results prove that keeping a fixed Δ entails models with levels of sparsity that depend on the threshold itself, but hardly surpass 40%. Thus, we take a step forward to try to maximize the sparsification induced by the Δ threshold. More detailed information about the results obtained will be provided in chapter 4, elaborating on the impact of fixed and growing Δ ternarization regimes on model outcomes.

$$\Delta_{new} = \Delta_0 + \Delta_0 \times f\left(M \times \text{Epoch}\right) \tag{3.6}$$

The employed method is based on a Δ growth regime, mathematically described in Equation 3.6. Δ_{new} is the current iteration threshold, and we choose an estimate for the 0-interval, Δ_0, which acts as an initial value for the growth regime, whose functional form is defined by f. M is a global hyperparameter selected a priori, a constant multiplier fundamental to adjust the shape of the growth, while "Epoch" represents the index for the current epoch of training. This implies that the current threshold Δ utilized in the ternarization framework is updated to Δ_{new}, contingent on the ongoing training epoch. The selection of the hyperparameter M depends on the selected regime and is employed to influence the Δ, ensuring it acquires specific values at predefined epochs.

In the *standard* fixed-Δ configuration, weights undergo random initialization, and their subsequent updates quickly lead them to surpass the threshold. This

Algorithm 2 The framework involves optimizing the cost function L for a minibatch. The learning rate is denoted by λ, and the number of layers is represented by N. $\circ$ is an element-wise multiplication. Clip() function defines how weights are clipped during the training iterations, while BackBatchNorm() computes the necessary backward pass operations through the BN. Decay() represents the λ schedule. The gradient $g_{a_L} = \frac{\partial L}{\partial a_L}$ is computed knowing output a_L and the targets a^*.

procedure BACKWARD PASS AND PARAMETER UPDATE
  ▷ **Backward pass.**
  for k = N to 1 **do**
    if k < N **then**
      $g_{a^{(k)}} \leftarrow g_{a_q^{(k)}} \circ 1_{|a^{(k)}|\leq 1}$
    end if
    $(g_{z^{(k)}}, g_{\phi^{(k)}}) \leftarrow \text{BackBatchNorm}(g_{a^{(k)}}, z^{(k)}, \phi^{(k)})$
    $g_{a_q^{(k-1)}} \leftarrow g_{z^{(k)}} W_q^{(k)}$
    $g_{W_q^{(k)}} \leftarrow g_{z^{(k)}}^{\top} a_q^{(k-1)}$
  end for
  ▷ **Parameter update.**
  for k = 1 to N **do**
    $\phi^{t+1,(k)} \leftarrow \text{Update}\left(\phi^{(k)}, \lambda, g_{\phi^{(k)}}\right)$ ▷ Update BN parameters
    $W^{t+1,(k)} \leftarrow \text{Clip}(\text{Update}(W^{(k)}, \lambda, g_{W_q^{(k)}}), -1,1)$ ▷ Update FP weights and clip to $[-1, +1]$
    $\lambda^{t+1} \leftarrow \text{Decay}(\lambda)$ ▷ Update learning rate
  end for
end procedure

implies that during the initial training iterations, a significant portion of the parameters is already ternarized to either -1 or $+1$, while their full-precision counterparts fluctuate around these values, extending beyond the 0-interval.

Hence, we introduce a dynamic growth regime for Δ (Equation 3.6) to conform to the swift evolution of parameters. This approach ensures that the ternarization threshold adapts to the nature of the parameter update, facilitating keeping a sparsified model throughout the entirety of the training process.

However, a consideration arises, as an upper bound must be imposed to prevent Δ increases beyond a critical value. Indeed, Δ could reach an absolute value equal to 1, leading to overlapping intervals in Equation 3.4. This scenario would breach the ternarization constraints, as the intervals would no longer remain distinct. Thus, it is essential to restrict Δ's growth before reaching this value. Generally, a higher Δ value entails a greater amount of *zeros*, but if taken too far, it can lead to situations where the model is too sparse. This can result in highly oscillating validation accuracy, and, in more extreme cases, it can result in a complete drop to zero accuracy. Finding the ideal Δ balance is essential to preserving a trade-off between model performance and sparsity.

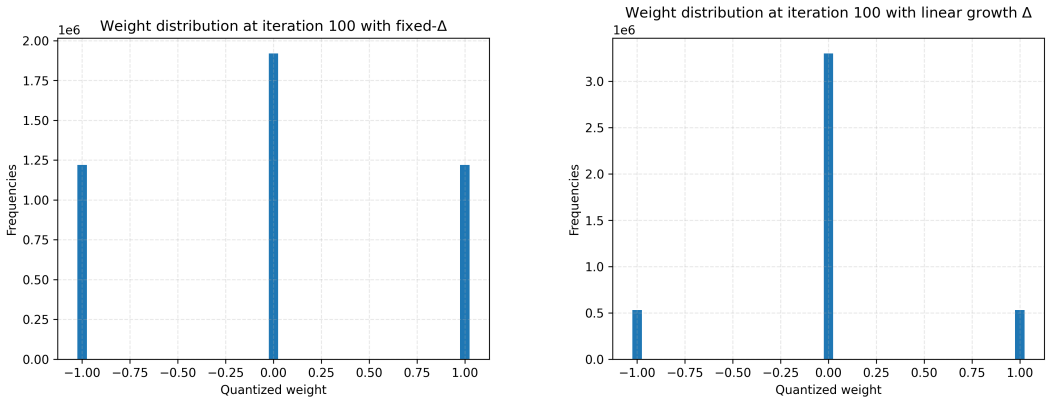


Figure 3.2: A comparison between the distributions of fixed and linear Δ ternarization regimes. (Left) Parameter distribution for a fixed $\Delta = 0.1$ at epoch 100. (Right) Parameter distribution for linear growth (with $\Delta_0 = 0.1$, $M = 0.05$, and f is the identity function) at epoch 100.

In the following chapter 4, we comprehensively assess the efficacy of the proposed method employing the described architecture on the designated CIFAR-10 dataset for image classification. Our goal is to achieve high accuracy even at elevated sparsity rates, providing a robust evaluation of the method's effectiveness. We test various configurations to find the setup that yields the best results.

Chapter 4

Experimental evidence

This chapter focuses on the practical validation of our proposed memory-efficient neural network methodology. Building upon the theoretical framework described in earlier sections, our objective is to evaluate the impact of the employed ternarization on sparsification and quantization during the learning process with empirical results. We emphasize providing an objective analysis of the experimental outcomes. In the initial section, we delve into the experimental design, focusing on the employed architecture and making considerations on the dataset and hyperparameters. The performance metrics for evaluation are described in the subsequent section. The chapter closes with a segment on results and their analysis.

4.1 Experimental design

In this section, we outline the experimental design conducted to evaluate the efficacy of our proposed ternarization method. The experiments were performed on the ResNet-20 architecture, a well-established model for image classification tasks, specifically utilizing the CIFAR-10 dataset. We test the proposed method on a ResNet architecture because it's a widely popular framework, known for its crucial results in image classification tasks. The CIFAR-10 dataset (Figure 4.1) is used as a benchmark for the conducted experiments, it comprises 60000 32×32 color images categorized into 10 classes, with each class containing 6000 images. The dataset is divided into 50000 training images and 10000 test images.

The model processes image batches with a fixed size of 256, and allows for parallel processing, leveraging the capabilities of the GPUs, thereby accelerating the training process. Additionally, it provides sufficiently diverse samples of data to generalize well during the learning process. The learning rate (λ) undergoes a scheduled decay as follows:

- Initial $\rightarrow \lambda = 5\text{e}{-3}$

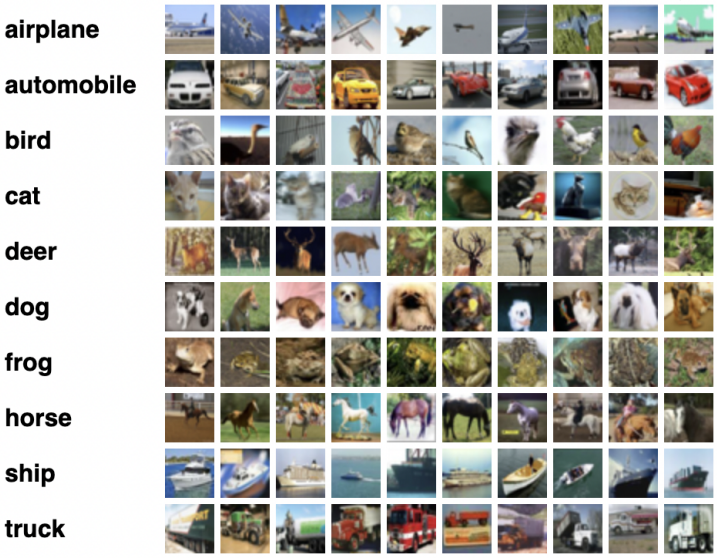


Figure 4.1: CIFAR-10 dataset. The dataset has 10 classes, all images are randomly selected, originally featured in [31].

- Epoch 101 $\rightarrow \lambda = 1e{-}3$
- Epoch 142 $\rightarrow \lambda = 5e{-}4$
- Epoch 184 $\rightarrow \lambda = 1e{-}4$
- Epoch 220 $\rightarrow \lambda = 1e{-}5$

In chapter 3, our proposed ternarization approach introduces a Δ growth regime, featuring a non-learned hyperparameter denoted as M, which requires a predefined value. Our investigation has been dedicated to determining optimal values for this constant. Training is performed for a total of 500 epochs, taking into account the characteristics of the chosen growth regime. Indeed, the growth regime (Equation 3.6) itself influences the learning process, we exploited different functions:

- **Identity (linear)**:

$$\Delta_{new} = \Delta_0 + \Delta_0 \times M \times \text{Epoch}, \tag{4.1}$$

- **Square**:

$$\Delta_{new} = \Delta_0 + \Delta_0 \times (M \times \text{Epoch})^2, \tag{4.2}$$

- **Square root**:

$$\Delta_{new} = \Delta_0 + \Delta_0 \times \sqrt{M \times \text{Epoch}}, \tag{4.3}$$

- **Exponential**:
$$\Delta_{new} = \Delta_0 + \Delta_0 \times e^{(M \times \text{Epoch})}, \tag{4.4}$$

- **Logarithm**:
$$\Delta_{new} = \Delta_0 + \Delta_0 \times \log\left(M \times \text{Epoch}\right). \tag{4.5}$$

The majority of our efforts were dedicated to optimizing linear and logarithmic regimes. We initially explored the simplicity of the linear approach in the early stages of our research, and as our findings progressed, the logarithmic regime emerged as the most effective.

The parameters in the ternarized convolutional layers are initialized according to Equation 4.6:

$$\theta \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n}}\right). \tag{4.6}$$

This initialization strategy involves drawing the parameters from a normal distribution $\mathcal{N}$ with a mean of 0 and a standard deviation of $\sqrt{\frac{2}{n}}$, where n corresponds to the number parameters in the convolution kernel.

4.2 Performance metrics

Within this section, we provide a definition of the metrics used to assess the effectiveness of our proposed method.

- **Top-1 validation accuracy:** henceforth referred to as accuracy, is a fundamental metric in evaluating the performance of a neural network model, particularly in the context of image classification tasks. It represents the percentage of correctly predicted labels out of the total number of validation samples. The *top-1* indicates that we are focusing on the accuracy of the single most likely prediction. This metric provides a measure of the model's ability to correctly classify instances from the validation set.

- **Sparsity:** is the second most important metric in the landscape of the optimization of memory usage, it measures the percentage of *zero* elements among the parameters of a neural network. In the context of our proposed ternarization method, sparsity plays a crucial role as it represents the portion of parameters set to zero among the quantized ones. Higher sparsity is desirable for memory-efficient neural networks as it leads to more compact model representations.

- **Entropy of the parameters' distributions:** for our proposed ternarization method, we compute the entropy of the distribution of the ternary parameter

values (-1, 0, and $+1$) over the whole neural network. A higher entropy indicates a more diverse configuration of parameters, while a lower entropy suggests a more concentrated distribution around a single value. The formula for the entropy is given by:

$$H(X) = \sum_i p(x_i) \log_2(p(x_i)), \tag{4.7}$$

where X is a random variable that can have outcomes x_i, specifically -1, 0 and $+1$, $p(\cdot)$ represents the frequency count of x_i. Therefore, in the ternarization context, $p(0)$ is the proportion of zeroes out of the total number of parameters. Entropy is expressed in bits/symbol and it measures the average number of bits needed to represent the information content of a random variable.

- **Training velocity:** denotes the epoch during training where the highest accuracy is reached. This metric provides details on the duration of the training process needed to achieve the optimal model result.

4.3 Results and analysis

We focus on the course of the experiments we performed, from the early results to the most recent that achieved ternary models that can outperform the binary counterpart by a reasonable margin, both in terms of accuracy and sparsity.

4.3.1 Examining ternarization conditions

The initial test we performed was aimed at understanding whether the sparsification interval had to include the "equals" condition or not. In Equation 4.8 the quantized parameter θ_q is set to 0 if $|\theta| < \Delta$, while in Equation 4.9 the 0 is retrieved through $|\theta| \leq \Delta$.

$$\theta_q = \begin{cases} +1 & \text{if } \theta \geq \Delta \\ 0 & \text{if } |\theta| < \Delta \\ -1 & \text{if } \theta \leq -\Delta, \end{cases} \tag{4.8}$$

$$\theta_q = \begin{cases} +1 & \text{if } \theta > \Delta \\ 0 & \text{if } |\theta| \leq \Delta \\ -1 & \text{if } \theta < -\Delta, \end{cases} \tag{4.9}$$

We estimated that there would be no distinction between the two approaches because there would only be a difference in the unusual cases where the parameter was updated exactly to the value of Δ. The outcomes of the experiments proved us right, in Table 4.1 we report results for five different simulations for both frameworks, the difference in average accuracy is minimal, and it is justified by a

discrepancy in standard deviation, likely caused by random fluctuations. Visual proof is provided in Figure 4.2.

Table 4.1: Comparison between ternarization conditions in five experiments each. The table specifies the best accuracy of the simulations along with their average and standard deviation.

	Accuracy					**Average**	**SD**
$\|\theta\| < \Delta$	90.74	91.37	91.41	91.42	91.46	91.28	0.30
$\|\theta\| \leq \Delta$	91.66	91.56	91.25	90.92	91.67	91.41	0.32

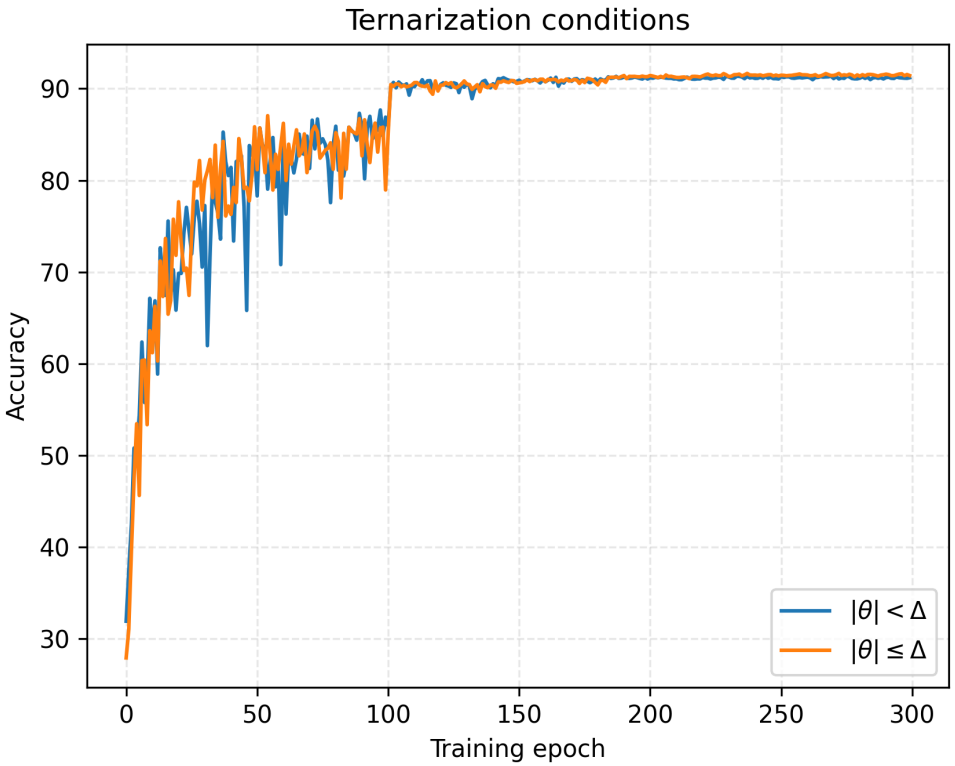


Figure 4.2: Comparing the ternarization conditions with two accuracy graphs chosen at random among the five simulations mentioned in Table 4.1. The experiment undergo a training process of 300 epochs, with fixed $\Delta_0 = 0.01$

Based on the conducted experiments and found no significant difference between the two approaches, we selected Equation 4.8 as the preferred method. All the

following experiments are carried out using this configuration.

4.3.2 Δ_0 impact on model learning

After selecting the ternarization setup described in the previous subsection, we turned our attention to finding a proper initial value to Δ_0.

The starting threshold is a fundamental parameter that needs to be set for both fixed-Δ and increasing-Δ frameworks. In the former, it not only represents the initial value but the whole training process is based on that. Whether in the latter it sets the basis for the growth regime. This subsection's work concentrated on fixed-Δ models. Threshold values of 1e−1, 1e−2 and 1e−3 enable the model to achieve accuracy over 90%, as shown in Figure 4.3.

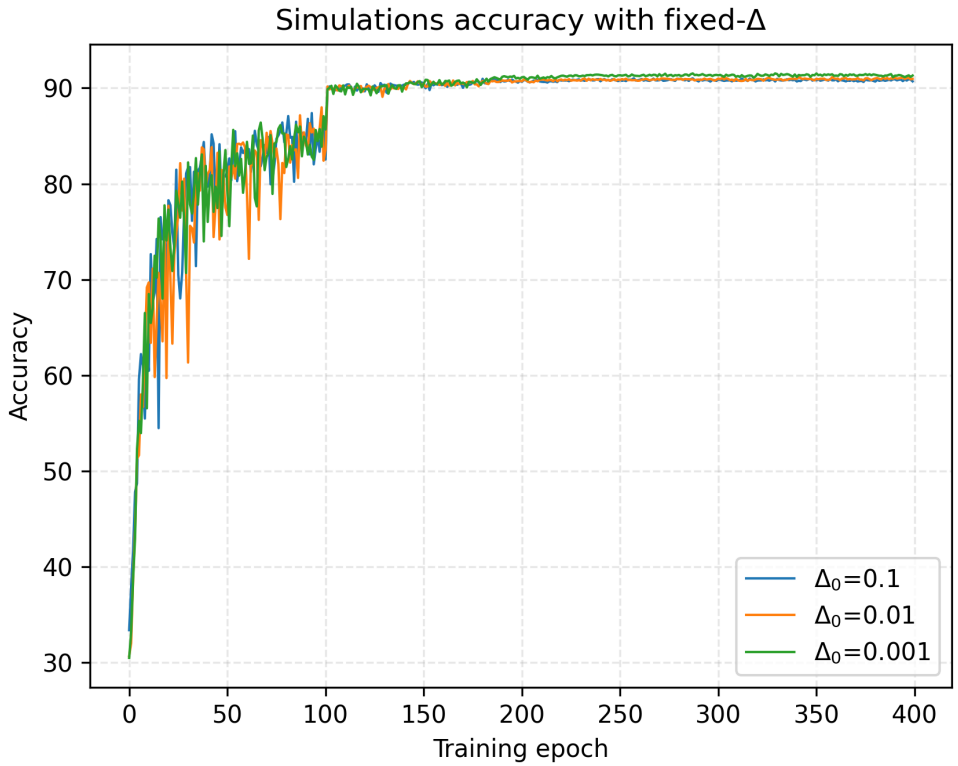


Figure 4.3: Accuracy of simulations with $\Delta_0 = 0.1$, $\Delta_0 = 0.01$ and $\Delta_0 = 0.001$

These values are significant for a fixed threshold configuration, the accuracy rapidly increases in the first iterations and reaches a plateau after epoch 101 where the learning rate drops according to schedule. However, they entail values of sparsity that decrease considerably during training, this trend is depicted in Figure 4.4.

There are substantial differences between the three curves appearing in Figure 4.4,

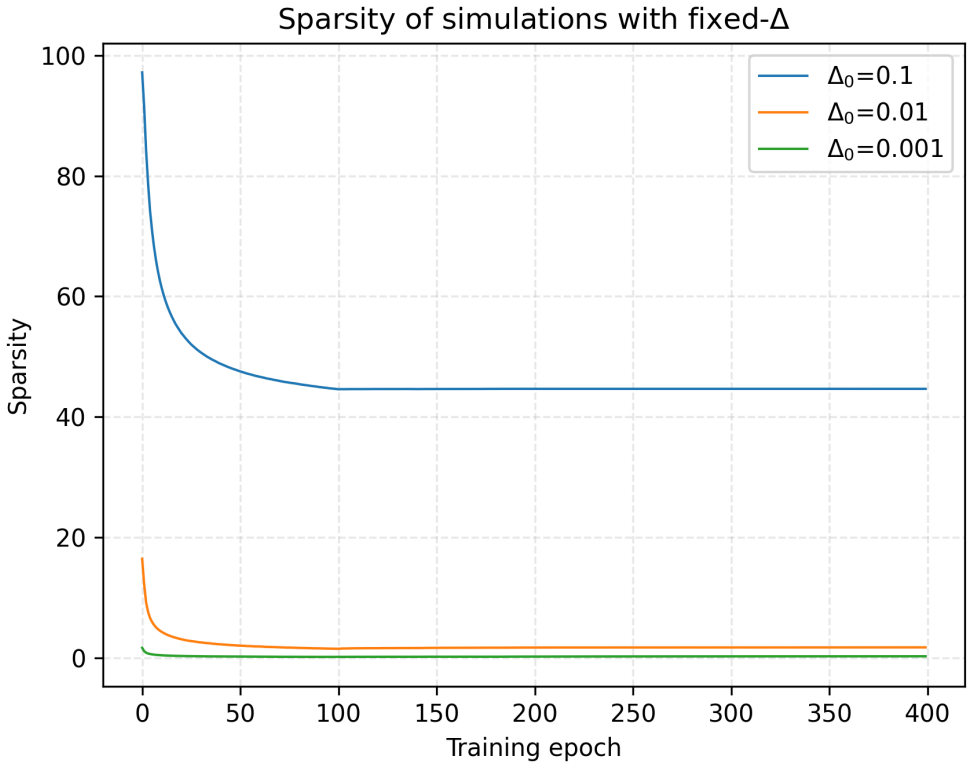


Figure 4.4: Sparsity trends of simulations with various Δ_0 values.

all contingent on the value of Δ_0. An initial threshold of 0.1 eventually leads to much higher sparsities than 0.01 and 0.001.

At the beginning of the learning process, when the gradients of the loss function are stronger and rapidly push the parameters away from the threshold, all trends have a steep decrease. After 101 epochs of training the sparsity reaches a plateau as the accuracy did in Figure 4.3. $\Delta_0 = 0.1$ yields the best sparsity with fixed-Δ configurations without sacrificing accuracy percentages. It's tempting to push the values of Δ_0 beyond the presented setups, but that causes a rupture of the model training process. We set up fixed-Δ models with initial thresholds of 0.2, 0.5, and 0.75, the outcomes are depicted in Figure 4.5.

At first glance, simulations with $\Delta_0 = 0.2$ seem to enable learning, however, upon closer inspection, it is clear that the accuracy values achieved are significantly lower than those with $\Delta_0 \leq 0.1$. Higher thresholds of 0.5 and 0.75 don't even initiate the training, keeping the accuracy to a level of 10% (they are overlapping), which, given the classification task we are solving on CIFAR-10, represents a random

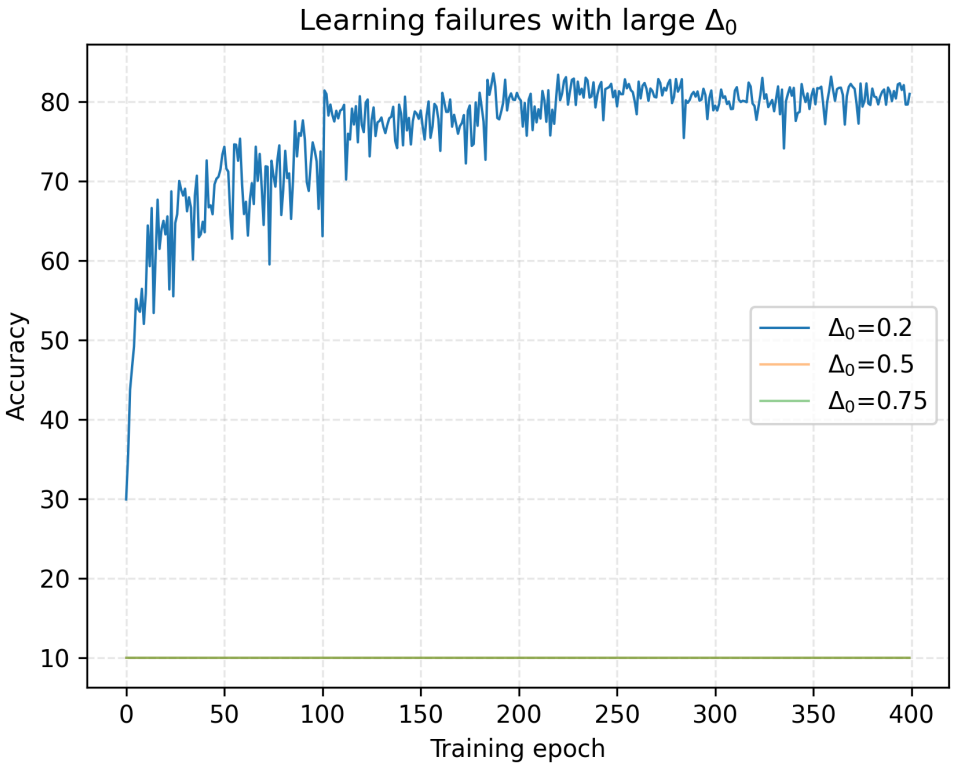


Figure 4.5: Failed simulations with large Δ_0. Over 0.1 thresholds prevent the model from learning.

guess. We inferred the reason is to be found in the number of parameters that are quantized to zero in the first epochs. With elevated threshold levels, sparsification affects the majority of parameters, as they indeed fall within the sparsification interval $(-\Delta, +\Delta)$. When most (if not all) parameters are quantized to 0, the training process becomes ineffective, preventing the update of weights and biases. The histogram illustrated in Figure 4.6 provides an overview of the distribution of full-precision parameters in the first epoch of training, with the majority falling within the sparsification interval, for a setup with $\Delta_0 = 0.2$.

In the next subsection, we investigate potential solutions and adjustments, aiming to enhance the training process and improve overall model performance.

4.3.3 Unbounded Δ growth

The outcomes of the fixed-Δ approach are promising but can be improved in a variety of ways. The sparsity is notably modest, averaging 44.11% across five

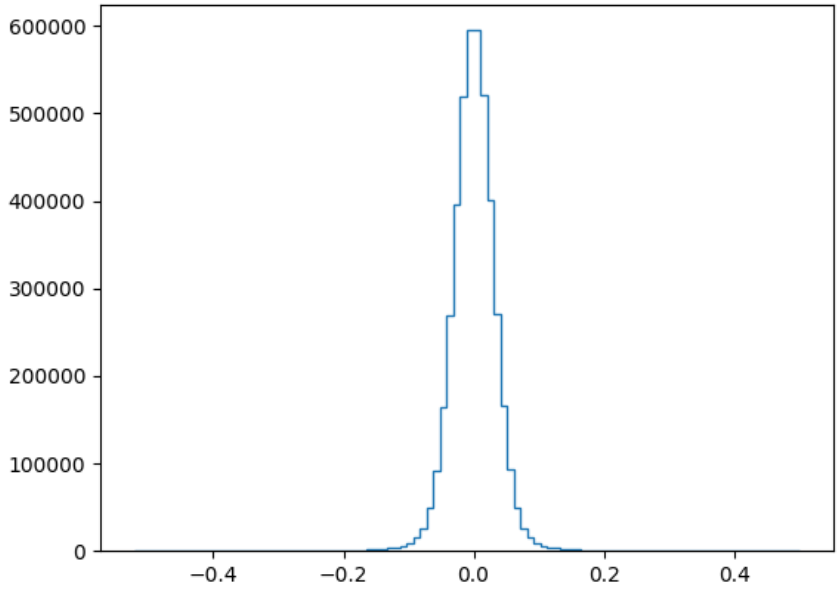


Figure 4.6: Distribution of full-precision parameters at the beginning of training in $\Delta_0 = 0.2$ models. On the y-axis is the number of parameters.

distinct simulations with $\Delta_0 = 0.1$. In Table 4.2, we present the results of these simulations conducted on a model with $\Delta_0 = 0.1$.

Table 4.2: Accuracy and sparsity of five simulations with $\Delta_0 = 0.1$. The accuracy fluctuates around 91.4% and the sparsity hardly surpasses 45%, meaning that the architecture relies on more than half of the parameters.

	Outcomes					**Mean**	**SD**
Accuracy	91.01	91.45	91.47	91.56	91.45	91.39	0.22
Sparsity	44.65	45.01	43.43	44.24	43.21	44.11	0.77

Additionally, the achieved accuracy is not remarkable, aligning closely with that of a binary counterpart implemented using BNN [27]. In Figure 4.7 we depict the accuracy of the model as a function of the training epoch and compare it with that of a binary architecture. Specifically, we plot the chart of the simulation with $\Delta_0 = 0.1$ that yields the best results, paired with the average level of the best accuracy of three binary experiments.

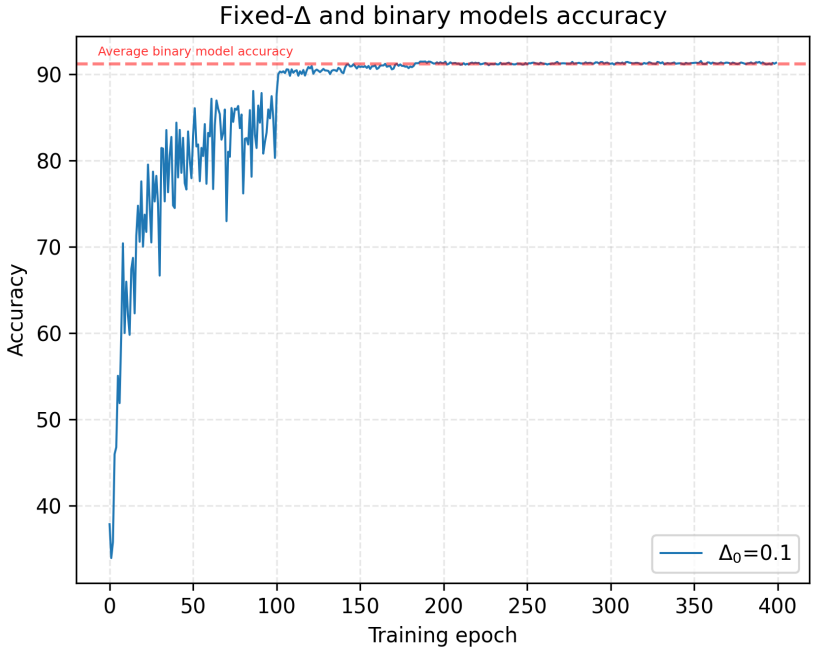


Figure 4.7: Average binary accuracy between three simulations compared to the accuracy curve of the best $\Delta_0 = 0.1$ fixed-Δ simulation.

Fixed-Δ models and their binary counterparts have similar accuracy. Although ternary architectures perform better than BNNs, the difference is not large, suggesting potential for improvement. Therefore, we introduce an approach characterized by a dynamic Δ threshold described by Equation 3.6, especially with identity function f, which essentially produces linear growth. The first growing Δ experiments performed had a limit set to 1 to avoid conflict between the ternarization conditions (henceforth referred to as *unbounded*). As a result, once Δ reached that terminal level, all parameters would be quantized to 0, which is a consequence of the clipping to $[-1, +1]$ of the full-precision coefficients. As previously mentioned, learning is not possible if all parameters are set to zero, when the model has Δ close to 1, we anticipate a decrease in accuracy in the results. Figure 4.8, Figure 4.9 and Figure 4.9 depict the situation for various simulations with $\Delta_0 = 0.1$, $\Delta_0 = 0.01$, and $\Delta_0 = 0.001$.

In Figure 4.8, it is evident that the training process encounters difficulties initiating, due to the excessively large M linear coefficients, Δ increases faster than the parameters' rate of update until most of the parameters are quantized to zero and accuracy drops to random guess (10%). Conversely, Figure 4.9 displays a more varied scenario. Two experiments fail due to the issues explained in Figure 4.8,

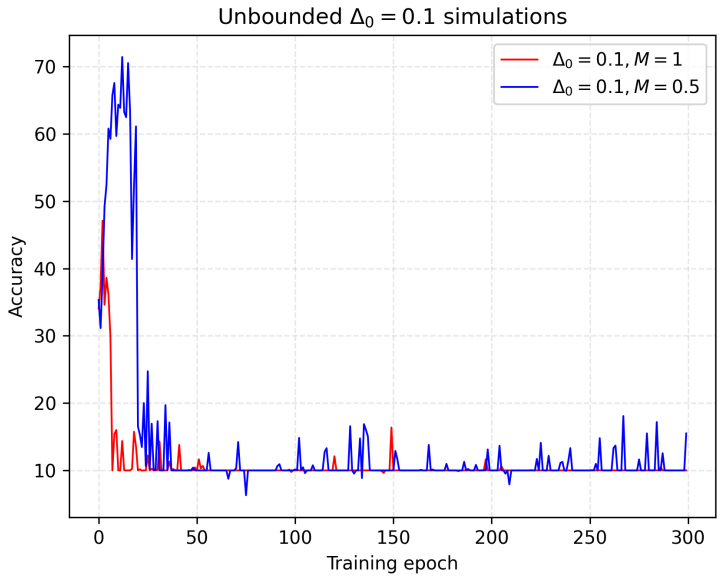


Figure 4.8: Accuracy of unbounded simulations with $\Delta_0 = 0.1$.

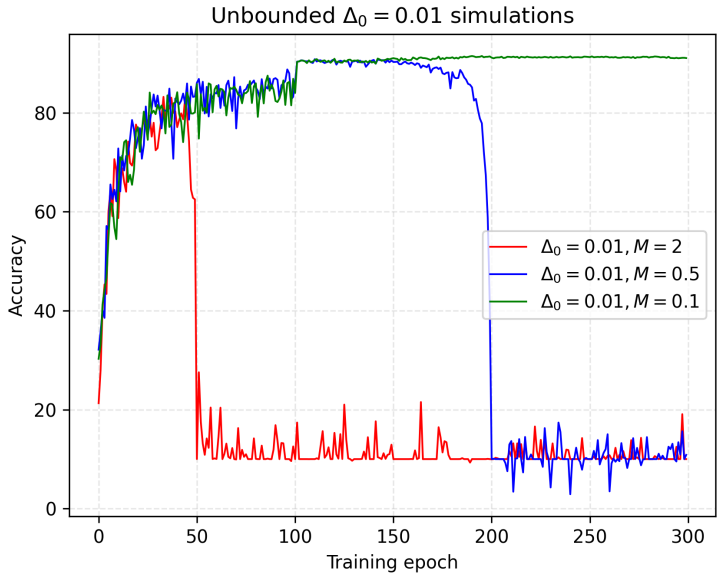


Figure 4.9: Performance evaluation for unbounded simulations with Δ_0 value of 0.01.

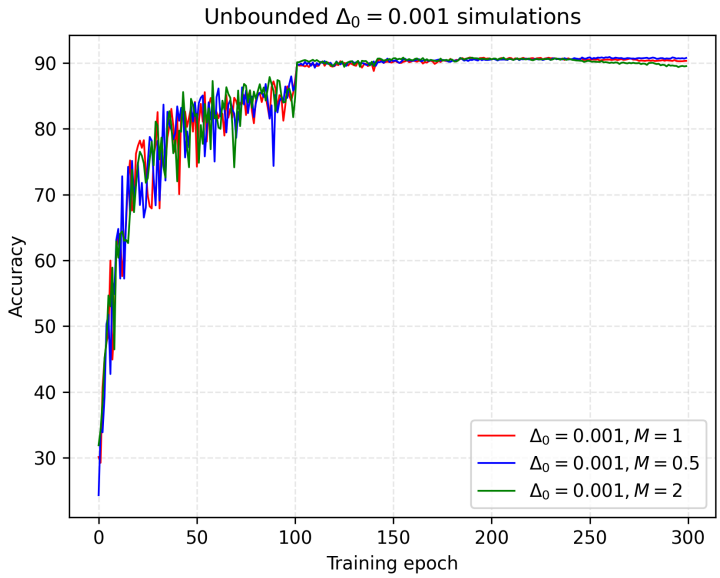


Figure 4.10: Analyzing the accuracy in unbounded simulations with $\Delta_0 = 0.001$.

while another maintains high accuracy throughout the entire training process. The difference in the third curve lies in the M coefficient. This lower coefficient facilitates a more gradual growth of Δ, preventing it from reaching excessively large values until the conclusion of the training. Figure 4.10 depicts three accuracy graphs that perform similarly to the green curve in Figure 4.9. The initial Δ_0 is an order of magnitude smaller, signifying that M, which shares the same order of magnitude as in the previous experiments, adapts to the configuration more effectively.

In Figure 4.8, we presented simulations with $\Delta_0 = 0.1$ that did not yield promising results. However, upon careful consideration, we attempted to employ the same value with a different, lower M value, in the hope of achieving a fully-trained model and obtaining higher sparsity compared to those with lower orders of magnitude of Δ_0. Indeed, in subsection 4.3.2, we highlighted that $\Delta_0 = 0.1$ implied better sparsity values right from the beginning, as evident in Figure 4.4. Therefore, a not excessively large value for M might allow the model to successfully complete training, possibly introducing higher sparsity into the framework.

We tested the concept using simulations based on $M = 0.02$ and $M = 0.01$. The former is adjusted such that Δ would reach 1 at epoch 450 and the latter is half of the first one, to show the result of an even slower increase. The outcomes are displayed in Figure 4.11, Figure 4.12 and Figure 4.13, and they are much more promising than in Figure 4.8, with training reaching acceptable levels of accuracy in

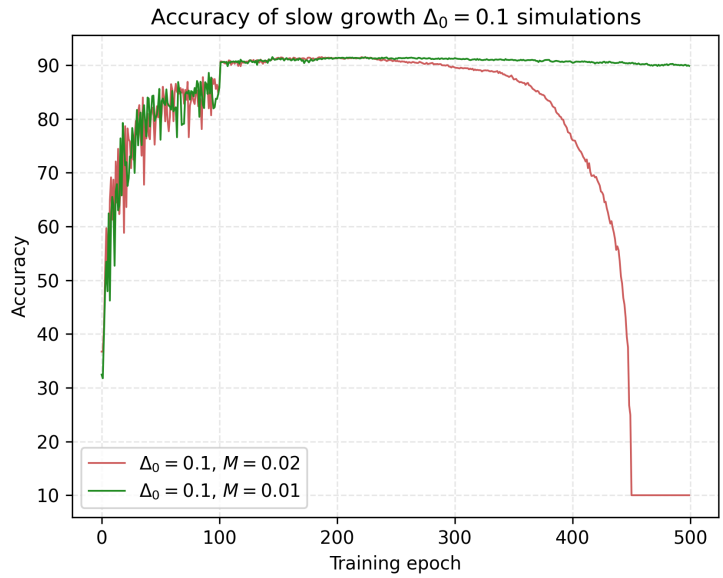


Figure 4.11: Accuracy of experiments conducted with low values of M and $\Delta_0 = 0.1$.

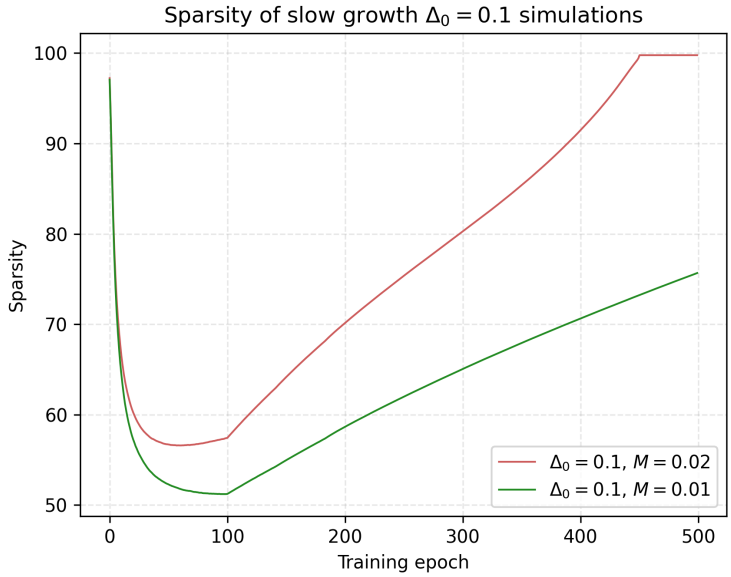


Figure 4.12: Sparsity of experiments conducted with low values of M and $\Delta_0 = 0.1$.

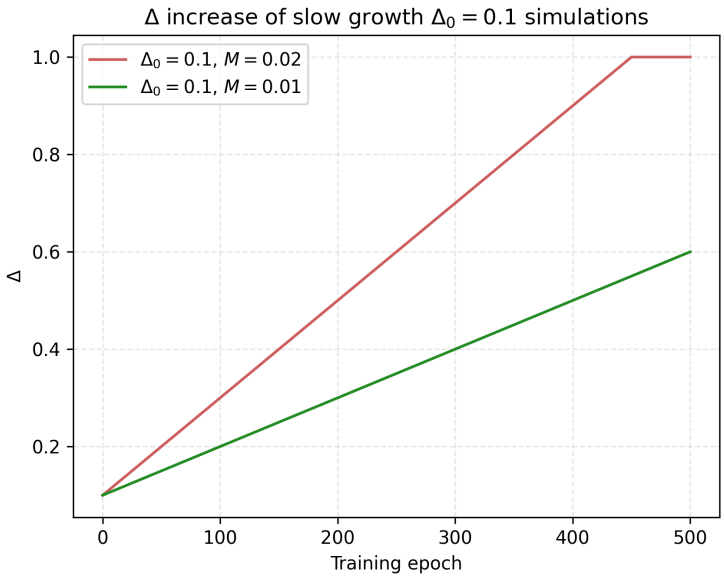


Figure 4.13: Δ increase of experiments conducted with low values of M and $\Delta_0 = 0.1$.

both simulations. While $M = 0.01$ allows the model to maintain high accuracy, the experiment with $M = 0.02$ suffers from the same previously mentioned accuracy dropping to 0 issues. The sparsity observed in these models, as illustrated in Figure 4.12, is noteworthy. Throughout the entire training process, it maintains a level considerably higher than 44.11%, which denotes the average sparsity in fixed-Δ simulations, as depicted in Table 4.2. Because of its larger Δ, illustrated in Figure 4.13, the $M = 0.02$ model is substantially sparser than the $M = 0.01$ framework. In the following subsection, we introduce a constraint on the increment of Δ to further explore these findings.

4.3.4 Constraining the linear increase of Δ

Introducing a linear growth for Δ appears to be the correct approach. Nevertheless, as noted in the preceding subsection, it must be coupled with a constraint on Δ, as the absence of such a cap can potentially disrupt the training process.

The conducted experiments focus on models with $\Delta_0 = 0.1$, as selected upon thorough considerations of its ability to promote higher sparsities, discussed in the previous subsection. In these simulations, we introduce a new hyperparameter Δ_f which denotes the limit to the growth of Δ. The accuracy of the simulations is illustrated in Figure 4.14, the plotted curves differ in Δ_f. A Δ_f equal to 0.9 signifies

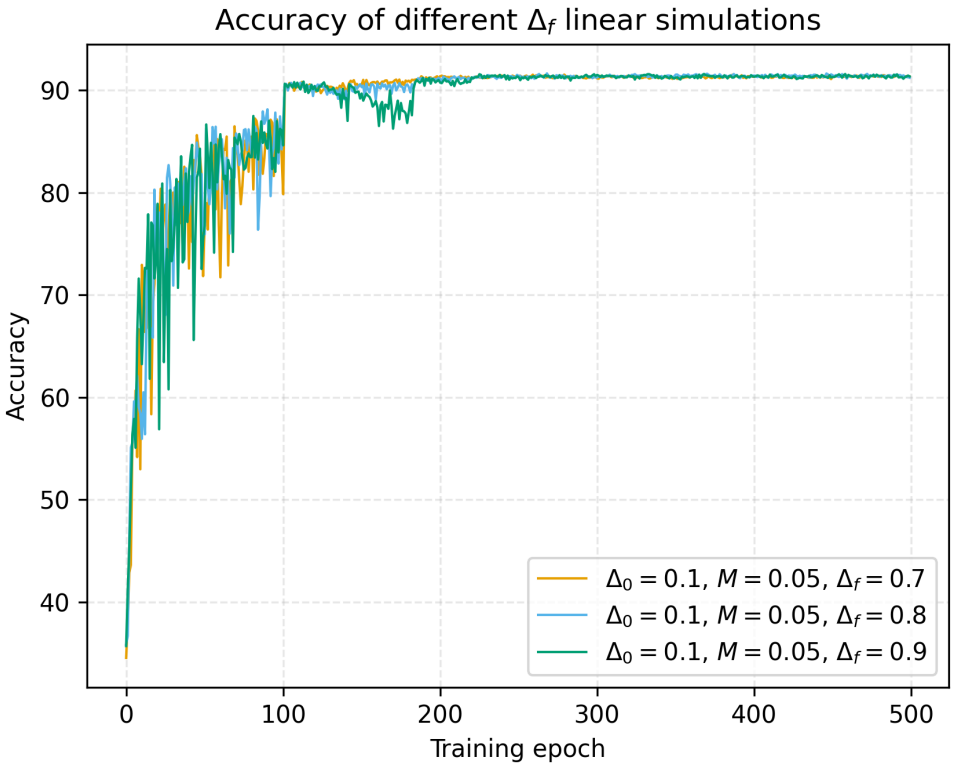


Figure 4.14: Comparison of simulation accuracy varying Δ_f. All experiments yield similar best accuracy, the higher Δ_f is, the more fluctuations the curve has.

that during the training the value of the Δ threshold updates up to 0.9. The observed fluctuations in models with a higher constraint on Δ can be attributed to the quantization of parameters to zero during training. Specifically, a larger cap on the growth of the threshold results in a greater number of parameters being ternarized to 0. This reduction in the number of weights and biases available to the architecture makes it more susceptible to random fluctuations. All experiments show high accuracy that does not experience drops as in some unbounded cases presented in subsection 4.3.3.

The sparsity is remarkable, in Figure 4.15 we display the trends for the same simulations represented in Figure 4.14. Each curve initiates with elevated sparsity values that decrease rapidly due to high gradients in the update algorithm. Then, sparsity increases again until the model reaches $\Delta = \Delta_f$, where it peaks, leading to a subsequent stabilization of the model. Larger values of Δ_f lead to higher sparsities but generate more fluctuating models.

Implementing a constraint on the growth of Δ results in neural network models

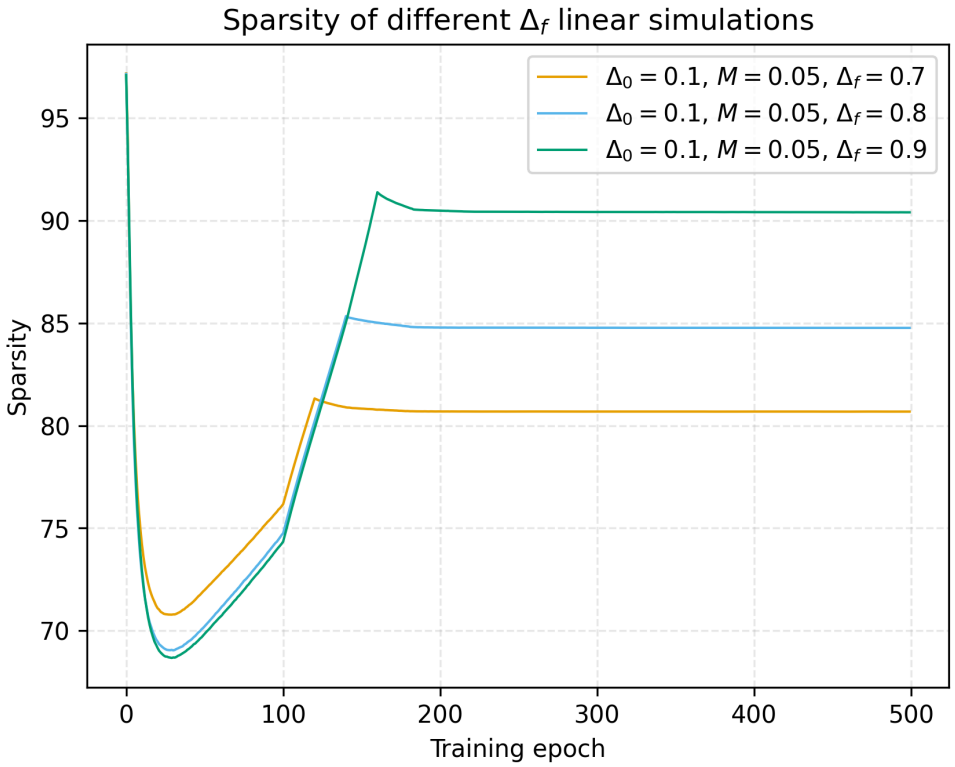


Figure 4.15: Analysis of sparsity levels varying Δ_f. Sparsity provided by linearly growing and constrained Δ increases with the value of Δ_f, it rises to 90%.

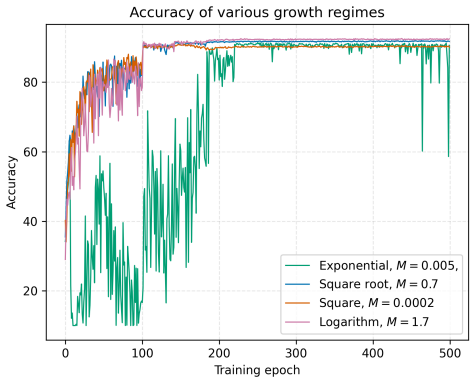
with both significant accuracy and significant sparsity. This subsection focused on linear growth, the next is characterized by the research of the optimal growth regime, namely, finding the f function in Equation 3.6 that yields the best outcomes.

4.3.5 Finding the optimal growth regime

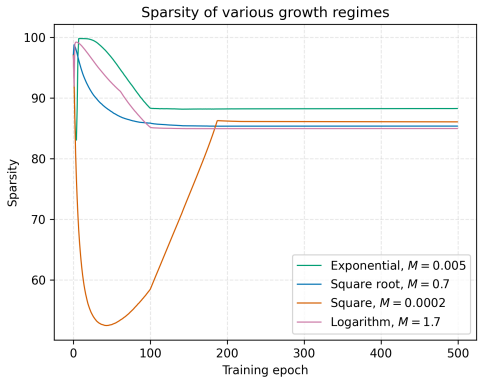
The concept of a growth regime for Δ during training was presented in Equation 3.6. In this subsection, we conduct experiments to identify the optimal regime in terms of accuracy, sparsity, entropy, and training velocity.

In section 4.1, we introduced the employed regimes, distinguished by exponential, square root, square, and logarithmic functions. We display the outcomes of accuracy and sparsity in Figure 4.16, and couple them with the growth trend of the respective model's Δ, moreover, we enter into the details of the accuracy after 200 iterations.

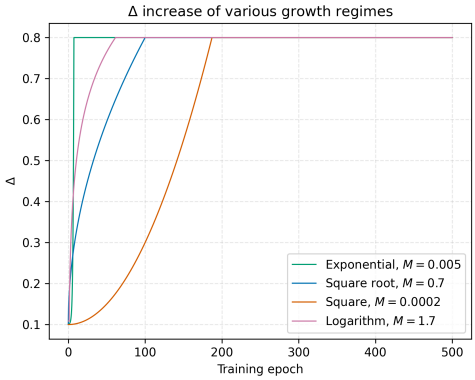
With the exception of the exponential curve, which fluctuates more, the accuracy follows the previously observed trend. This is because the exponential causes Δ to



(a) Accuracy levels for different growth regimes ($\Delta_f = 0.8$).



(b) Sparsity for different growth regimes ($\Delta_f = 0.8$).

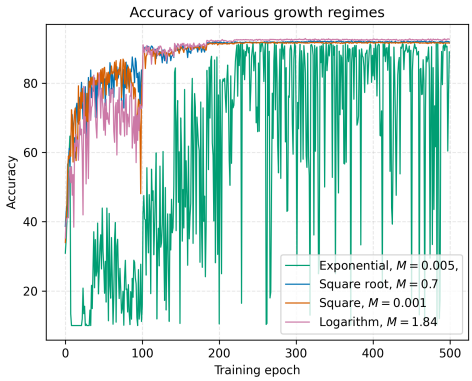


(c) Δ increasing for different growth regimes ($\Delta_f = 0.8$).

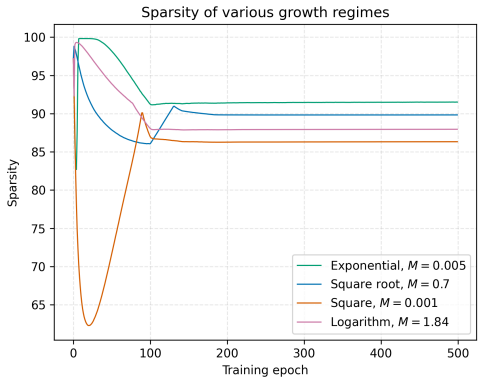
(d) Detail of the accuracy trends over 200 iterations for different growth regimes ($\Delta_f = 0.8$).

Figure 4.16: The figures show different metrics that are used to assess how well the models perform. Despite having the same initial $\Delta_0 = 0.1$ and final $\Delta_f = 0.8$, they illustrate distinct types of growth f.

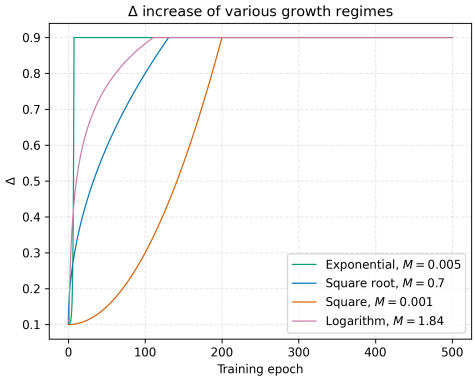
grow too quickly, sparsifying the model too soon and leaving the architecture with an excessively low number of parameters. Sparsity exhibits two distinct trends: the first rises immediately upon training commencement and stabilizes later; the second, typical of square growth, is characterized by a decline, which is followed by an increase and stabilization. This difference results from the square regime's initial lower slope. In Figure 4.16c the detail of the accuracy after 200 epochs is illustrated. This figure is particularly useful to focus on the levels of accuracy these models can achieve. Square regimes performance is underwhelming, and exponential functions result in too many fluctuations. Conversely, both logarithm



(a) Accuracy levels for different growth regimes ($\Delta_f = 0.9$).



(b) Sparsity for different growth regimes ($\Delta_f = 0.9$).



(c) Δ increasing for different growth regimes ($\Delta_f = 0.9$).

(d) Detail of the accuracy trends over 200 iterations for different growth regimes ($\Delta_f = 0.9$).

Figure 4.17: The figures show different metrics utilized to evaluate the performance of models characterized by different growth regimes ($\Delta_f = 0.9$). Compared to curves with $\Delta_f = 0.8$, these experiments lead to higher accuracy and sparsity, at the cost of more fluctuations.

and square root regimes perform better than binary models.

In an effort to enhance sparsity, we experiment with a larger value of constraint to Δ, employing $\Delta_f = 0.9$. The outcomes are represented in Figure 4.17. With accuracy exceeding 92% and sparsity surpassing 85%, the logarithmic growth simulation produces the best stable combination of results among these experiments without exhibiting the fluctuating behavior tested in the exponential regime.

4.3.6 Comparative analysis of growth regimes

To prove that the logarithmic regime performs the best, we tested more experiments with various parameter combinations, differing Δ_f, multipliers M, and regimes, while we maintained $\Delta_0 = 0.1$ across all of them. In order to compare the simulations, we plot the training velocity, entropy, accuracy, and sparsity metrics against each other. Notably, four graphs result from this analysis:

- **Accuracy vs. Sparsity**: a comparison that provides insight into which simulation is more accurate and sparser than the others.
- **Accuracy vs. Entropy**: shows the models' entropy on the x-axis. It specifies which experiment yields the highest accuracy and the greatest compressibility.
- **Sparsity vs. Velocity**: determines which simulations achieve higher sparsity levels most rapidly.
- **Accuracy vs. Velocity**: identifies which models produce their best accuracy results the fastest.

The study's findings are here summarized. In Figure 4.18 point clouds of different models, characterized by specific combinations of parameters and regime, are compared in a graph that tracks accuracy against sparsity. Every combination experiment is conducted three times, giving rise to three points on the graph. This chart is notably significant as it serves as the most influential instrument to compare models and select the optimal regime. The best configurations are represented by both large accuracy and sparsity, namely, they occur in the upper right portion of the graph where logarithmic regime points are located. Exponential simulations, as previously stated, result in higher sparsities but are not practical due to excessive accuracy fluctuations during training. The accuracy achieved by all of the described configurations is greater than that of BNN models, with logarithmic frameworks exceeding it by approximately two percentage points.

Logarithmic simulations reach accuracy levels of nearly 93.5% and have sparsity percentages of nearly 90%, meaning that these frameworks are highly compressible. The results in sparsity rates are correlated with the value of the growth constraint Δ_f; the higher the limit, the better the sparsity can be; however, as was previously observed, if the limit is too large, the accuracy may deteriorate. Higher multiplier M logarithmic simulations typically exhibit better accuracy, this could indicate that Δ growth adapts more readily to parameter updates, allowing for a model that operates effectively with ternarized parameters.

The best accuracy points are plotted against entropy in Figure 4.19, compared to the chart that was previously described, this is inverted. The reason behind this behavior lies in the fact that entropy is closely related to sparsity, indeed

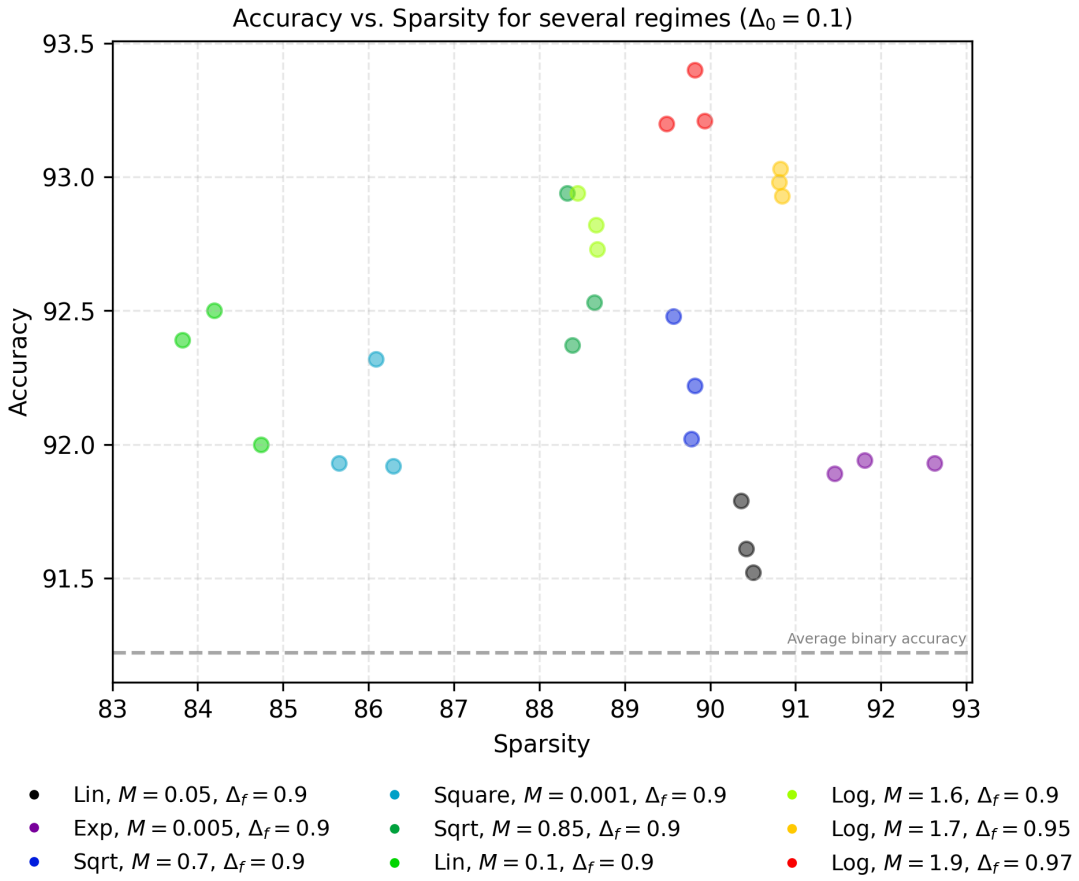


Figure 4.18: Scatter plot of accuracy vs. sparsity in multiple regime setups. Three identically colored points for every set of parameters correspond to three equal experiments. The dotted grey line indicates the level of average accuracy of BNN models.

a highly sparse network results in lower entropy. The values of the logarithmic configurations are approximately 0.6bits/symbol.

An analysis of the relationship between accuracy and training velocity, as well as sparsity and training velocity, is needed to thoroughly understand the performance of various growth regimes. Figure 4.20 and Figure 4.21 depict them. Compared to the previous two charts, the point clouds in these graphs are much more dispersed, suggesting that the best outcomes for each setup can be found at various points during training. Occasionally, random fluctuations will cause the model to find its optimal result earlier than in other instances. Adding up these considerations, it becomes evident that the logarithmic growth regime stands out as the most favorable choice.

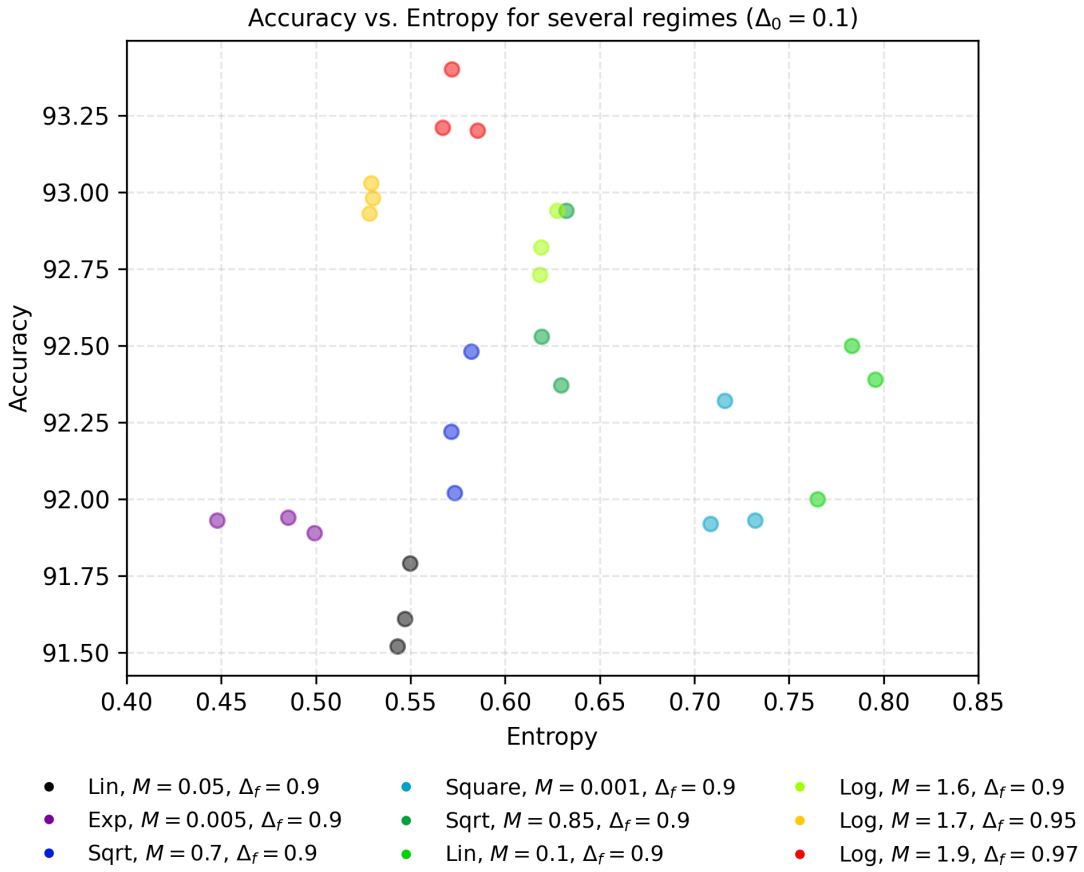


Figure 4.19: Accuracy point cloud for each model against entropy. Each set of parameters is represented by three points sharing the same color, indicating three identical experiments.

In this chapter, we examined the evidence that gave rise to our novel ternarization method, which utilizes a growth regime to produce outcomes that outperform both the fixed-threshold ternarization and binary architectures. After that, we conducted a selection process using a variety of metrics, which enabled us to conclude that the logarithmic regime produces the best results, with an entropy of approximately 0.6bits/symbol and accuracy exceeding 93% and sparsity approximately 90%. We observed that the values of the hyperparameters, specifically Δ_f and M, can be tuned to lean towards more accuracy-based or more sparsity-based models.

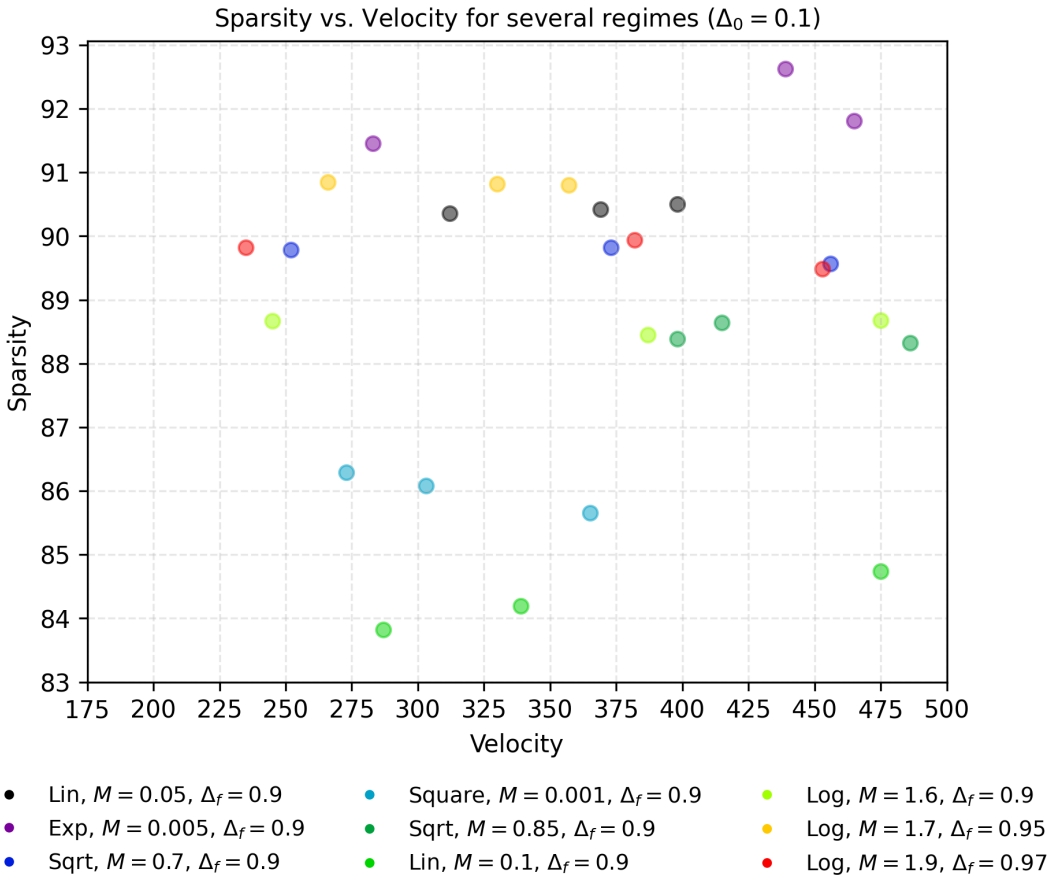


Figure 4.20: Sparsity in relation to velocity for various growth regimes. Three experiments, all identical in terms of parameters, are denoted by three points of matching color.

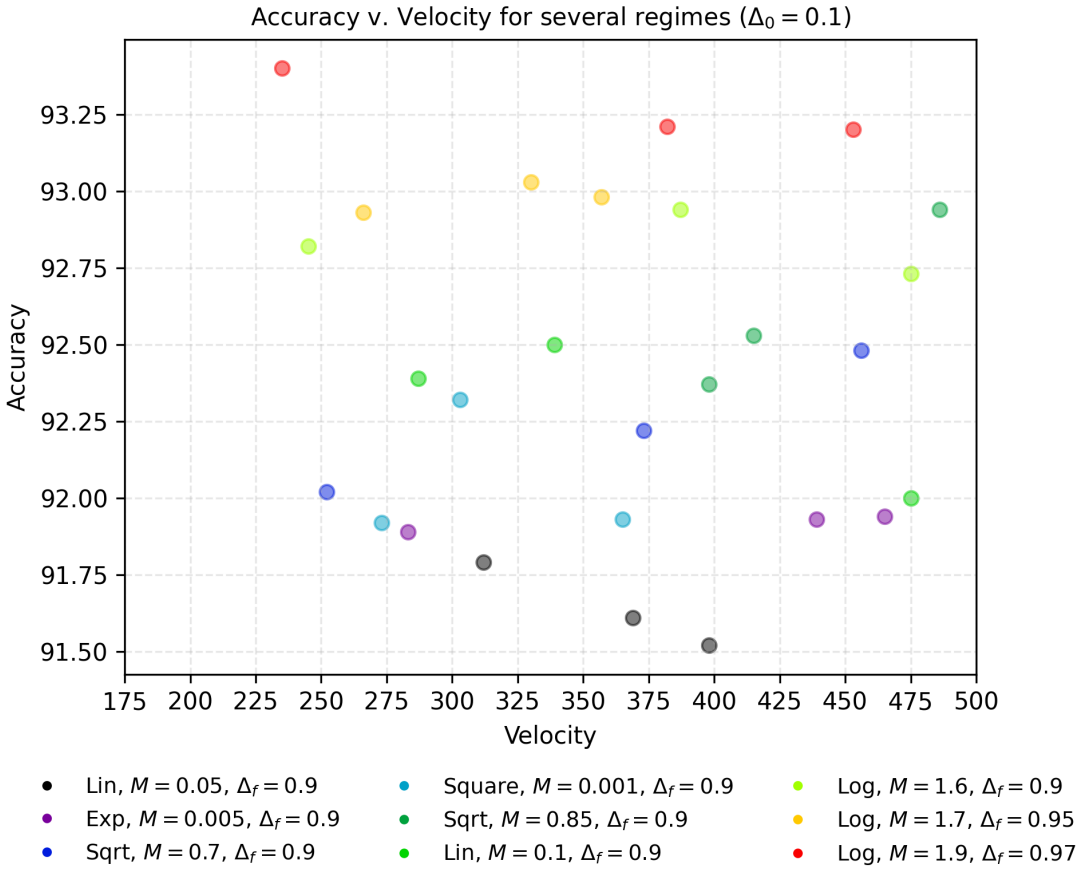


Figure 4.21: Relationship between accuracy and training velocity in various models. For every configuration of parameters, three points with consistent coloring indicate three equivalent experiments.

Chapter 5

Conclusions and future studies

In this thesis, we examined the challenge of improving neural network efficiency from the perspective of memory occupation. We started with an introduction to how neural networks work, explaining their origins and their structure, mentioning some of the most important architectures, and how neural networks are trained. Subsequently, we presented the techniques of model compression currently in use, among which are pruning and quantization, and we proceeded to outline the method employed to address this challenge. The proposed approach is characterized by the ternarization of the neural network, it is a combination of sparsification and quantization procedures that act on parameters and activations of the architecture during training. It sets them to -1, 0, and +1 values, effectively sparsifying the network topology. The experimental evidence is collected working in an image classification landscape, especially employing a ResNet architecture with the CIFAR-10 dataset. Our most significant contribution arises with the introduction of a dynamic threshold Δ for the zero-value quantization. Specifically, this thesis demonstrates that the best overall results for our framework in accuracy and sparsity are obtained with a dynamic logarithmic growth of the threshold. We show that this method allows for neural network frameworks with sparsification rates over 90% and improvements in top-1 validation accuracy with respect to their binary counterparts. Despite the additional complexity of implementing the ternarization, we prove that the remarkable sparsity rates result in parameter distributions with minimal entropy (about 0.6bits/symbol), providing highly compressible architectures.

Our findings are remarkable, yet there remain open questions that require further examination. While the CIFAR-10 dataset has been the focus of our experimental evaluation, our ternarization approach can be extended to other data sources. Furthermore, this dissertation targeted image classification, but the method we

proposed can be evaluated on neural networks trained to perform various tasks beyond the ones explored in this study. Moreover, the employed architecture is a ResNet, but further exploration of a broader range of frameworks would prove the solidity of this method. While the experimental setup explored various combinations of hyperparameters, including the initial threshold Δ_0, threshold limit Δ_f, multiplier M, and regime function, the impact of the learning rate on model performance remained mostly ignored. A thorough examination of this relationship would provide valuable insights and potentially enhance the method's effectiveness.

To conclude, our work has produced significant results toward addressing the memory consumption challenge of deep neural networks and toward developing models that can be deployed on devices with limited resources, more manageable and efficient.

Bibliography

[1] Warren S McCulloch and Walter Pitts. «A logical calculus of the ideas immanent in nervous activity». In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133 (cit. on p. 5).

[2] Frank Rosenblatt. «The perceptron: a probabilistic model for information storage and organization in the brain.» In: *Psychological review* 65.6 (1958), p. 386 (cit. on p. 6).

[3] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on pp. 6, 7).

[4] Kunihiko Fukushima. «Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position». In: *Biological cybernetics* 36.4 (1980), pp. 193–202 (cit. on p. 6).

[5] DE Rumelhart, GE Hinton, and RJ Williams. «Learning representations by back-propagating errors (from Nature 1986)». In: *Spie Milestone Series Ms* 96 (1994), pp. 138–138 (cit. on p. 7).

[6] Xavier Glorot and Yoshua Bengio. «Understanding the difficulty of training deep feedforward neural networks». In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256 (cit. on p. 8).

[7] Vinod Nair and Geoffrey E Hinton. «Rectified linear units improve restricted boltzmann machines». In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814 (cit. on p. 8).

[8] Muhamad Yani, S Irawan, and Casi Setianingsih. «Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail». In: *Journal of Physics: Conference Series* 1201 (May 2019), p. 012052. DOI: 10.1088/1742-6596/1201/1/012052 (cit. on p. 13).

[9] Sergey Ioffe and Christian Szegedy. «Batch normalization: Accelerating deep network training by reducing internal covariate shift». In: *International conference on machine learning*. pmlr. 2015, pp. 448–456 (cit. on p. 14).

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «Imagenet classification with deep convolutional neural networks». In: *Advances in neural information processing systems* 25 (2012) (cit. on pp. 15, 21).

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. «Imagenet: A large-scale hierarchical image database». In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255 (cit. on p. 15).

[12] Karen Simonyan and Andrew Zisserman. «Very deep convolutional networks for large-scale image recognition». In: *arXiv preprint arXiv:1409.1556* (2014) (cit. on p. 15).

[13] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. «Going deeper with convolutions». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9 (cit. on p. 16).

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90 (cit. on p. 16).

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. «Generative adversarial nets». In: *Advances in neural information processing systems* 27 (2014) (cit. on p. 17).

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention is all you need». In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 17).

[17] Alexey Dosovitskiy et al. «An image is worth 16x16 words: Transformers for image recognition at scale». In: *arXiv preprint arXiv:2010.11929* (2020) (cit. on p. 18).

[18] Herbert Robbins and Sutton Monro. «A stochastic approximation method». In: *The annals of mathematical statistics* (1951), pp. 400–407 (cit. on p. 19).

[19] Diederik P Kingma and Jimmy Ba. «Adam: A method for stochastic optimization». In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 19).

[20] Yann LeCun, John Denker, and Sara Solla. «Optimal brain damage». In: *Advances in neural information processing systems* 2 (1989) (cit. on p. 20).

[21] Song Han, Jeff Pool, John Tran, and William Dally. «Learning both weights and connections for efficient neural network». In: *Advances in neural information processing systems* 28 (2015) (cit. on pp. 21, 22).

[22] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. «A survey of quantization methods for efficient neural network inference». In: *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326 (cit. on p. 22).

[23] Olivia Weng. *Neural Network Quantization for Efficient Inference: A Survey*. 2023. arXiv: 2112.06126 [cs.LG] (cit. on p. 23).

[24] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV] (cit. on p. 23).

[25] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. 2013. arXiv: 1308.3432 [cs.LG] (cit. on pp. 23, 29).

[26] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. «Binaryconnect: Training deep neural networks with binary weights during propagations». In: *Advances in neural information processing systems* 28 (2015) (cit. on pp. 24, 29).

[27] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. arXiv: 1602.02830 [cs.LG] (cit. on pp. 24, 29, 42).

[28] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. 2015. arXiv: 1503.02531 [stat.ML] (cit. on p. 24).

[29] Bin Liu, Fengfu Li, Xiaoxing Wang, Bo Zhang, and Junchi Yan. «Ternary Weight Networks». In: *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2023, pp. 1–5. DOI: 10.1109/ICASSP49357.2023.10094626 (cit. on pp. 27, 29).

[30] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. «Trained ternary quantization». In: *arXiv preprint arXiv:1612.01064* (2016) (cit. on pp. 27, 28).

[31] Alex Krizhevsky, Geoffrey Hinton, et al. «Learning multiple layers of features from tiny images». In: (2009) (cit. on p. 35).

Acknowledgements

Questa tesi la dedico a te, Ponchi,
con le tue parole e il tuo sorriso sei stata la mia ispirazione.

Il mio primo ringraziamento va ai professori che mi hanno seguito durante questo percorso di tesi, il professor Magli, la professoressa Fracastoro, la professoressa Fosson, il professor Bianchi e il professor Migliorati, che mi han dato preziose indicazioni per procedere nel modo migliore durante lo sviluppo delle simulazioni e nel processo di scrittura dell'elaborato.

Un pensiero per Ponchi, che oggi non potrà venire, ma che nel mio cuore mi sarà sempre accanto, mi hai dato la forza di concludere questo lavoro, ti abbraccio da qui, ovunque tu sia.

Non sarei qui se non fosse stato per la mia famiglia, Marta, la mia gemellina che sempre mi è stata vicina supportandomi, Mamma e Babbo che mai hanno smesso di ascoltarmi. Manna che mi ha accompagnato nel cammino e zii Paolo e Sergio che dal Piemonte hanno sempre creduto in me. Grazie anche ai nonni che non ci sono più, spero sarebbero fieri di questo traguardo.

L'altro giorno mi hai detto "sono importanti i ricordi", ed è anche merito tuo, Marti, se questo giorno e questi anni me li ricorderò per tutta la vita, sei la luce che mi ha illuminato il cammino, grazie a te per tutto quello che sei e che fai per me.

Ad Ale, Giordi e Maggie, che disturbo da una vita e disturberò per una vita, un ringraziamento speciale, so che potrò contare sempre su di voi, siete il mio punto di riferimento.

E che dire dei miei compagni di viaggio qui a Torino, Auri, Cops, Erny, Fra, Gra, Nick, Zimo, vi ho conosciuti in un momento della mia vita dove mi sentivo solo, voi mi avete accolto e reso parte di un gruppo che mi ha ha accompaganto attraverso tutte le difficoltà e le bellezze di questo percorso, non potrò mai ringraziarvi abbastanza.

Come non parlare dei miei compagni di quando ero ancora ingegnere, Ivan, Lorenzo, Loris e Peppe, ricordo con nostalgia le serate sulle panchine fuori dal poli e le giornate passate a fare gli elettronici, grazie di tutto.

Fondamentali nella mia vita siete stati voi amici di Bovezzo, Anna, Arianna, Bruno, Chiaretta, Giuli, Luca, Man, Marco, Matteo e Ventu, che ormai da anni siete il motivo per cui sono legato così tanto al mio paesino, siete una certezza.

E i miei torinesi preferiti, Gio, Ste e Dani, con cui ho passato giornate e serate bellissime all'insegna del cibo piemontese e della bagna cauda, vi sono profondamente grato per quello che siete per me, è anche merito vostro se sono così

affezionato a questa città.

È il momento di ringraziare due delle persone che ho conosciuto in collegio che più mi sono vicine, Chia la mia amica 98 e Luci la mia compagna di pale, ho passato momenti fantastici con voi che mi avete migliorato le giornate.

Grazie anche alle mie coinquiline Deni e Paola, con cui vivo da relativamente poco tempo ma con cui mi sento già benissimo e a mio agio, grazie per la comprensione che avete.

Ogni anno non vedo l'ora che arrivi l'estate, per passarla a San Felice e stare con voi, i miei amici del lago, Alex, Luchi, Nico, il Baro e tutti gli altri, per le belle giornate che mi permettete di passare, le nuotate e i vari giochi in scatola, vi ringrazio immensamente.

A settembre 2021 mi trasferito a Leuven per l'Erasmus, e ne sono uscito come una persona nuova, ho vissuto momenti fantastici che ancora sono vividi nella mia mente, ho conosciuto amici con cui ho vissuto mille esperienze fantastiche, partite di pallavolo, salti coi trampolini e gite in varie città belga. Ho conosciuto persone con background unici e culture diverse, grazie a voi che avete reso tutto ciò irripetibile.

Ultimi ma non per importanza, i compagni di collegio che ho conosciuto nei miei anni universitari, vivere in residenza mi ha cambiato la vita, sono cresciuto e maturato, senza di voi non sarei la persona che sono ora, tante, tante grazie.