UCLouvain

epl

École polytechnique de Louvain

Improving generalization of quantized activation neural networks using dither

Author: **Baptiste STANDAERT**
Supervisors: **Christophe DE VLEESCHOUWER, Laurent JACQUES**
Readers: **Antoine VANDERSCHUEREN, Michel VERLEYSEN**
Academic year 2020–2021
Master [120] in Electrical Engineering

Abstract

The recent growth in performances of deep neural networks in image classification tasks is coupled with an increase in the complexity of the inference. For embedded applications, it becomes necessary to use compression techniques to simplify the inference cost of these networks.

A compression technique consists in quantizing the activation. However, this compression implies an accuracy loss that can be explained by a suboptimal training due to the quantization.

We introduce a dither during the quantization of activation. The latter is used to control the distribution of the quantization error. We show that this dither allows improving the generalization of quantized activation networks by correcting the training. We also introduce a dithered set-based approach, which takes advantage of the stochastic inference of this method, and improves accuracy.

Keywords *neural networks, compression, quantized activation, dither.*

Acknowledgements

I would like to thank my supervisors who have accompanied me for hours and hours week after week throughout my work. I have learned a lot through this thesis and it is certainly thanks to their advises and remarks.

I would also like to sincerely thank Antoine for his precious framework but also and especially for having supported me throughout the whole research.

Contents

Chapter 1

Introduction

Deep learning has imposed its reputation over the last few years due to its impressive performance in a wide range of applications. The field of computer vision is not sparse. Indeed, it is now common to obtain convolutional networks reaching a top-1 accuracy of more than 90 % on the image classification dataset ImageNet [29]. This recent emergence can largely be explained by technological advances that have allowed increasing computational resources and thus facilitate their deployment.

The increase in performance of these networks is accompanied by a grow in the complexity of these architectures making training and inference more computationally expensive. While the implementation of these networks at the server-GPU scale does not seem to be a problem, their deployment remains very challenging, if not impossible, on an embedded mobile application with limited resources. It is therefore necessary to drastically reduce the inference cost of these networks. This can be achieved through compression techniques.

Quantization of activation is a widespread method of network compression. It permits a reduction in the amount of information passing through the network and also enables hardware optimizations on embedded devices. All this in order to make the inference cost achievable. Obviously, these compressions are often accompanied by a loss of performance.

The objective of this thesis is to study a method to limit this loss of accuracy. Recent studies have investigated the causes of this deterioration. It is now known that it can be explained, at least partly, by a phenomenon called gradient mismatch which makes the training of networks suboptimal. We have been able to demonstrate, by studying the training of networks with quantized activation, that this gradient mismatch is related to the quantization error.

In signal processing, and more widely in computer vision, dithering consists in adding a completely random noise to the signal before the quantization process. This permit, on the one hand, to decorrelate the distortion introduced by the

quantization but also and especially to erase via an averaging process the impact of the quantization. The addition of dither during the quantization process allows, in expectation, to obtain a null quantization error.

The technique introduced and studied for this work consists in applying dither during the quantization of activation within the network. This approach should thus enable the minimization of the quantization error resulting in a correction, at least partial, of the training conducted on quantized activation networks.

The addition of dither during inference also makes the classification process of the network stochastic since it will depend on the dither generation. A set-based approach taking into account the different results obtained for the different inferences of the same sample allows refining the classification rule and consequently, improving the accuracy.

However, the addition of dither raises many questions about how to apply it throughout the network. Hundreds of networks have therefore been trained in order to find the best dither strategy to achieve what has just been explained.

The contributions of this work are the following.

For the first time and at the best of our knowledge, we have addressed the study of the gradient mismatch with a quantization error-oriented approach and have been able to explain how this distortion impacts the training of networks. This also allowed us to demonstrate theoretically how the addition of dither could correct this problem.

A complete study of the dither strategy to be applied in order to maximize the classification performance has then been performed.

Once this dither strategy was implemented, we were able to verify that an improvement of the network performance was empirically observable. We showed that the performance loss between a network simply quantized on its activation and a non-compressed network could be reduced to half by using a dithered training. This supports the previous theoretical conclusions.

Finally, a set-based study showed that it was possible to obtain, with only a few inferences from a quantized and dithered network, almost the same performance as a non-compressed network. A comparative study with other set-based approaches then enabled to establish the pros and cons of this technique, thus positioning it within the state-of-the-art.

Structure of this thesis

This work is divided into eight chapters.

Chapter 2 presents the state-of-the-art related to neural networks by focusing on the equations that drive the training. Chapter 3 presents the background related

to quantization. The dither will be introduced here but also how the training of quantized activation networks is implemented.

Once these different technical concepts have been introduced, the motivations (including demonstrations) concerning the application of dither to quantization will be discussed in Chapter 4, whereas Chapter 5 aims at presenting the related works.

Chapter 6 introduces the general methodology observed for the experiments that will be carried out while Chapter 7 aims at presenting and discussing the different obtained results. Finally, Chapter 8 concludes this work.

Chapter 2

Background on neural networks

Neural networks are computer systems that strive to recognize the underlying relationships in a data set by a process that mimics the functioning of the human brain. *Convolutional neural networks* (CNN), a class of deep neural networks, perform very well in computer vision and more particularly in image classification tasks.

In this master thesis, different concepts related to convolutional networks will be presented. This Chapter presents an intuitive approach to the development of increasingly complex image classification networks: starting from the main base block of neural networks to build up more complex ones. This background Chapter finishes with a section dedicated to the training of those.

In the first section, the percepton is described. It will be the main block constituting these networks. Secondly, multi-layer perceptrons, which is a neural network using a particular arrangement of a perceptron set, will be presented. The type of networks at the heart of this study will then be introduced. Namely, the convolutional neural networks. This Chapter finally ends with a section dedicated to the training procedures and how the learning of such networks are operated.

2.1 Perceptron

Perceptron is an algorithm for *supervised learning* of *binary classifiers* introduced by Rosenblatt [31]. It allows to decide whether or not an input vector belongs to a specific class on the basis of supervised learning. It is considered as the first and simplest feedforward neural network.

A perceptron is composed of a set of *learnable parameters*: the *weights* $\mathbf{w}$ and a *bias* b. The term learnable parameter is used to designate the parameters that will be tuned during the training phase which will be described later. All parameters are assumed to be already fine-tuned.

2.1.1 Forward propagation

Single-layer perceptron takes as input a feature vector $\mathbf{x}$ which it associates via a scalar product with the weights. The bias is then added to this intermediate result to form the *activation* a. The activation is then passed through a non-linear function ϕ called the *activation function*. The output o is a prediction on the class associated to the input vector.

Mathematically, it is defined as:

$$\begin{aligned} a &= \sum_{i=1}^{n} w_i x_i + b \\ &= \sum_{i=0}^{n} w_i x_i \end{aligned} \tag{2.1}$$

$$o = \phi(a) \tag{2.2}$$

In order to compact future equations, the bias is fused into the weights by writing $b := w_0$ and by fixing $x_0 := 1$. A graphical representation of a perceptron is shown in the following Figure:

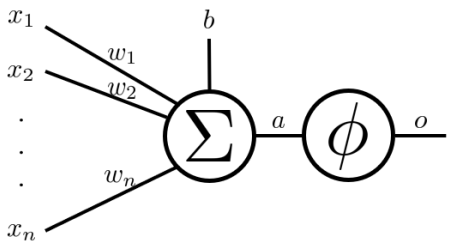


Figure 2.1: Graphical representation of a single-layer perceptron

A. Activation function

Originally, Rosenblatt uses the unit step function as activation function.

The classification rule f therefore becomes:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

Where $\{0, 1\}$ is respectively the negative and positive instances for the determination of the class.

2.1.2 Limitations

Rosenblatt demonstrates the convergence of the neuron when the data is linearly separable. On the other hand, it cannot perfectly classify a set that is not linearly separable. The perceptron is thus a linear classifier. For obvious reasons, this could be a problem for many applications on different datasets. This brings us to the next section.

2.2 Multi-layer perceptron

Multi-layer perceptron (MLP) is a class of *feedforward neural networks* organized by a succession of *layers* connected to each other. A layer is formed by a variable number of perceptrons. Each perceptron of a layer is connected to each other perceptrons of its neighboring layers i.e. each instance of a layer is connected to each of the other instances from the previous and next layers. Those are called *fully connected* (FC) layers.

For sake of representation, those networks are often associated with graphs. Each node of the graph is a perceptron and each of the edges of the graphs is a weighted connection between the perceptrons. This arrangement is somewhat reminiscent of the structure of a brain linking neurons together. A perceptron can be seen as a neuron and the connection between two perceptrons as an inter-neuronal link. An illustration of such a graph is available on Figure 2.2.

Formally, an MLP is composed of a succession of at least three FC layers: the *input layer*, the *hidden layer(s)* and the *output layer*. Each of the inputs from the input vector is associated with one neuron of the input layer. The hidden layers contain as many nodes as wished and the output layer contains as many nodes as desired classification classes. Each neuron i from the output layer gives the output prediction $\hat{y}_i$ on the i-th associated class.

2.2.1 Forward propagation

The classification decision is made on the basis of the output provided by the last layer of the network. The information is propagated unidirectionally through the network starting from the input and ending at the output. This is called forward propagation. Let's see how this propagation works.

For each neuron, the activation (2.1) is first computed and passed through the activation function (2.2). The inputs of the first layer are obviously contained in the input vector. The inputs of the hidden and output layers are the outputs from the preceding FC layer. In order to generate the output, the information is therefore propagated feed forwardly through the network. The data flows iteratively from

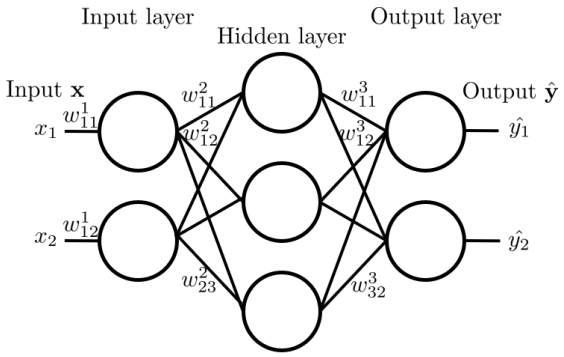


Figure 2.2: Graphical representation of a 3-layer MLP. The input layer is composed of 2 neurons, the hidden layer of 3 and the output layer of 2 neurons. Each output neurons provide a prediction on a class. For sake of simplicity, only a few weights are annotated but each edge between each node is weighted. The biases associated to each layer are not represented either.

layer to layer. It begins with the input layer and flows toward the output layer by passing through each hidden layer.

For clarity reasons, some notations are defined in order to distinguish the different parameters constituting the network. Suppose an MLP composed of m FC layers. The notation l_k is introduced to designate the k-th layer of the network (numbered from 1, the input layer, to m, the output layer).

The weight located on the inter-neuronal connection between the nodes i, located on the layer l_{k-1}, and the node j located on l_k will be noted w^k_{ij}. The same way, bias will be noted b^k_i which indicate the bias of the node i from the layer l_k. The neurons from the k-th layer are numbered from top to bottom as 1 to r_k (the number of neurons in the k-th layer).

The activation and output for the i-th node of the k-th layer are respectively given, provided an adaptation with the two previous equations, by the following relations:

$$\begin{aligned} a^k_i &= \sum_{j=1}^{r_{k-1}} w^k_{ji} o^{k-1}_j + b^k_i \\ &= \sum_{j=0}^{r_{k-1}} w^k_{ji} o^{k-1}_j \end{aligned} \tag{2.4}$$

$$o^k_i = \phi(a^k_i) \tag{2.5}$$

Note that the same trick as for the perceptron is used to compact the equation. Indeed, $b^k_i := w^k_{0i}$ and $o^{k-1}_0 := 1$.

An annotated neuron of an MLP is available on the next Figure:

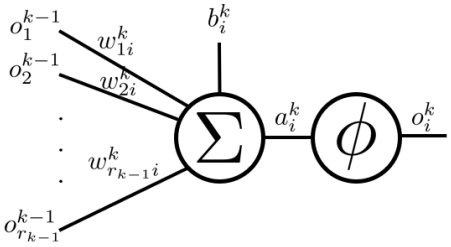


Figure 2.3: Graphical representation of the i-th neuron from the k-th layer in an MLP.

A. Activation function

For a reason that will be explained in the later subsection 2.4, devoted to the training of networks, it is necessary to have an activation function that is *differentiable and non-zero*.

For MLP networks, one usually finds two sigmoidal functions. Namely, the *hyperbolic tangent* and the *logistic function* which are respectively defined below and are illustrated on Figure 2.5.

$$\tanh(x) := \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.6}$$

$$\text{logistic}(x) := \frac{1}{1 + e^{-x}} \tag{2.7}$$

2.2.2 Limitations

Hornik, Stinchcombe, and White [18] introduced the *universal approximation theorem* that proves that any function can be approximated with a network containing at best, a single hidden layer. In other words, a chain formed by a succession of layers of perceptrons could approximate any function, whether it is linear or non-linear. MLP thus allows to overcome the limitations of the perceptron. This is why these networks are excellent candidates for classification and regression tasks.

Unfortunately, MLP is now considered insufficient for advanced computer vision tasks. It has two major drawbacks that make it noncompetitive today for classification. The first big disadvantage is that the total number of parameters evolves far too quickly with the number of layers ($\prod_{k=1}^{m} r_k + \sum_{k=1}^{m} 1$ for weights and biases). This introduces a redundancy in high dimensions which makes forward propagation inefficient. Another disadvantage comes from the fact that it takes flattened vectors as inputs disregarding the spatial information. The reader will easily understand that this is a quite a big concern for grid-like topology data such as images...

This lead us to introduce the networks of the next section. CNN addresses these two drawbacks by introducing a new layer that takes location information into account and drastically reduces the number of parameters by using a weight sharing architecture.

2.3 Convolutional neural networks

Convolutional Neural Network (CNN) is a class of networks belonging to deep learning that exploits the hierarchical structure of data by combining simpler and smaller patterns through filters.

These filters are called *convolution kernels* and have the particularity to have a shared-weight architecture. The kernels slide along the input data, the *input maps*, to provide an equivariant translation response, the *activation maps*. These filters allow the network to extract features from a data domain, the *receptive field*, by mapping those with a translation-independent function. This makes it possible to summarize a context while taking into account its spatial layout. These properties make these networks excellent candidates for processing grid-like topology data like images.

2.3.1 Forward propagation through different layers

CNN are made up of a succession of different distinct layers. An input map (like an image) is propagated feed forwardly as a feature map through the network before obtaining the prediction output on the last layer. Each layer maps an input volume to an output volume through its own differentiable function. In this subsection, classical CNN layers and the way they propagate the information are introduced.

A. Convolutional layer

The convolutional layer is the main layer of CNN. This layer consists of a set of K kernels composed of parameters, the *weights*. Each of these kernels is defined by a small width and height F which extends over the entire depth of the input volume D_1 i.e. the number of input maps. Therefore, a kernel is a 3D volume of dimensions $F \times F \times D_1$. Note that the input volume $I \in \mathbb{R}^{W_1 \times H_1 \times D_1}$ where W_1 and H_1 are respectively the width and height of the input maps.

During the forward phase, each of the K kernels convolves with the input volume to form an activation map. In other words, each filter slides over the width and height of the input volume and computes the dot products between the filter parameters and the input at any position in the volume. This produces a two-dimensional activation map that gives the responses of that filter at each

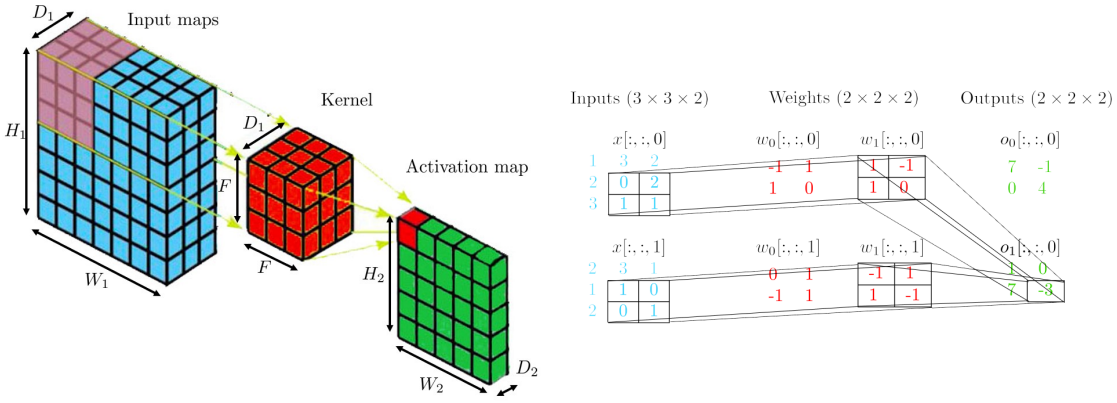
spatial position. Mathematically for the kernel W composed of weights $w_{m,n,c}$ and a bias b, mathematically:

$$(I * W)_{ij} := \sum_{m=1}^{F} \sum_{n=1}^{F} \sum_{c=1}^{D_1} w_{m,n,c} \cdot I_{i+m,j+n,c} + b \tag{2.8}$$

By combining the two-dimensional activation maps of the results of the convolutions of the K different kernels, an output volume of *depth* K is produced. The output is thus a volume $O \in \mathbb{R}^{W_2 \times H_2 \times D_2}$ where $D_2 = K$. A convolutional layer contains therefore $(F \cdot F \cdot D_1) \cdot K$ learnable weights as well as $b \cdot K$ learnable biases.

It should be noted that different hyper-parameters allow to control the size of the output maps as well as the size of the receptive field of the perceptrons by modifying the behavior of the convolutions. One can notably quote the *stride*, the *zero padding* and the *dilation rate* which will not be discussed here. Interested readers will find an excellent introduction to those in Dumoulin and Visin [11].

A visual summary of what has just been explained and an example is given on next Figure:



(a) 3D convolution using one kernel of size $F = 3$. The input volume contains $D_1 = 3$ input maps of height $H_1 = 7$ and width $W_1 = 7$ that produces an activation map of height $H_2 = 5$ and width $W_2 = 5$.

(b) Result of a two kernels convolutional layer convolving 2 feature maps (3×3) with two kernels of size 2 (2×2) resulting into two output maps of size 2 (2×2).

Figure 2.4: Convolutional layer

The *weight-sharing* architecture of the convolutional layer allows to drastically reduce the number of parameters compared to a FC layer. Finally, thanks to the translation invariant kernels, the network processes data by taking into account the spatial locality of the information.

B. Batch normalization layer

The batch normalization (BN) is a very common layer in CNN introduced in 2015 by Ioffe and Szegedy [21]. As its name suggests, it is used to normalize the inputs of a layer by performing a re-centering and rescaling operation on the latter.

This allows mainly two things. First and most obviously, to normalize the information that fluctuates in the network during inference. But it also and especially allows reducing[1] the *internal covariance shift* which complicates the training. More intuitively, the efficiency of a layer will depend on the distribution of inputs to which it is trained for. Also during training, several sources of randomness from this process causes a variation of the inputs distribution. This has the effect of slowing down the training which can be corrected via normalization.

Mathematically, this normalization of the inputs is performed by a scale and a shift operation respectively associated to the learnable parameters γ and β. Assume the batch D of size n and let the input samples to the layer x and the outputs y.

The transformation applied by the BN is given by:

$$\mathrm{BN}_{\gamma,\beta}(x) := y \leftarrow \gamma\hat{x} + \beta \tag{2.9}$$

Where $\hat{x}$ is a normalized version of the input x defined by:

$$\hat{x} := \frac{x - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}} \tag{2.10}$$

ϵ is an arbitrarily small constant added to the batch variance for numerical stability. μ_b and σ_b^2 are respectively the moving average on the batch D and the moving variance defined by:

$$\mu_b := \frac{1}{n}\sum_{x\in D} x \tag{2.11}$$

$$\sigma_b^2 := \frac{1}{n}\sum_{x\in D}(x - \mu_b)^2 \tag{2.12}$$

In summary, batch normalization is used to improve the generalization of the network. It is a form of model regularization that normalizes the input distributions by applying a re-shifting and rescaling.

C. Activation layer

An activation layer is simply a layer without any parameters that applies an activation function on the inputs.

[1]This is still a hotly debated topic. There does not yet seem to be a consensus at this level but here will be assumed that this is the case.

In the previous section on MLP, two different activation functions used in the neurons were introduced. For some reason occurring during the training of networks, Krizhevsky and Hinton [24] demonstrate that after a convolutional layer the *Rectified Linear Unit* (ReLU) activation is best. This function is defined as follows:

$$\text{ReLU}(x) := \max(0, x) \tag{2.13}$$

ReLU has no saturation for positive values and does not propagate negative information. This has the effect of introducing non-linearity in the decision function and in the network in general without affecting the receptive fields from the output of the convolutional layers. An illustration of the different activation already introduced is available on the next Figure.

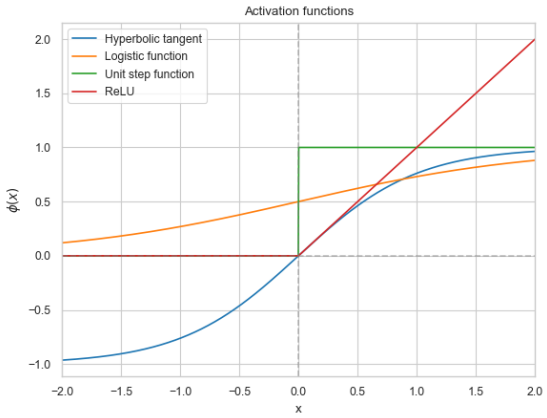


Figure 2.5: Illustration of different activation functions.

D. CBA block

The three previous described layers generally constitute the majority of a CNN model. Further, CNN architectures are often formed by blocks composed of a succession of these layers. A convolutional layer is usually followed by a batch normalization and an activation layer. Those blocks are referred to by their acronym as a *CBA* block. The CBA are then assembled to form the part of the network that will extract the features from the images.

E. Pooling layers

Pooling layers can be seen as a form of non-linear down-sampling layers which makes them very important for CNN. In fact, they allow two things:

1. Reduce the resolution (width and height) of the feature maps during forward propagation in order to reduce the computational complexity.

2. Extract and summarize the data from the receptive field of the layer.

In other words, it allows a synthesis of the features from the input map by performing a reduction in the amount of information propagated.

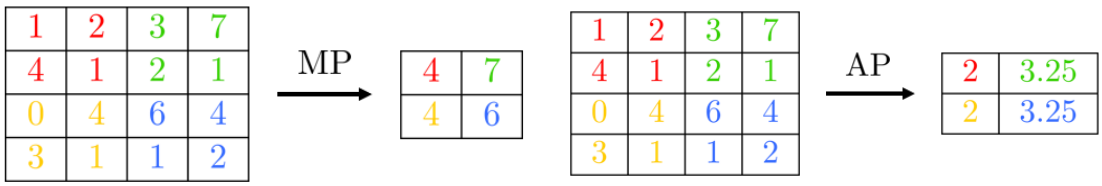
A pooling layer is determined by a set of hyper-parameters: the size of its receptive field, the size of the regions that constitutes the receptive field and the down-sampling action they apply. As with convolutional layers, it should be noted that there exist some hyper-parameters that allow the modification of the behavior of the receptive field that will not be discussed here.

Two distinct pooling down-sampling are usually distinguished:

1. *Max pooling* (MP): output the maximum values observed in each region.

2. *Average pooling* (AP): output the average value observed in each region.

A visual example of an MP and AP layer is available on Figure 2.6.



(a) Max pooling

(b) Average pooling

Figure 2.6: Illustration of pooling layers of kernel size 4×4. The receptive field is divided into 4 different regions of size 2×2 represented by a color. It is on each of these regions that the down-sampling action is applied.

F. Fully-connected layer

Those layers, described in section 2.2, are used at the end of the network to process the extracted features. The last FC layer usually provides the classification prediction. It is worth noting that the FC layers take as input a vector $\mathbf{x}$ while the layers presented above output a volume. A flattening of the data is generally performed in order to adapt the dimensions when passing from one convolutional to a FC layer.

G. Argmax, softmax and output of a CNN

In a multi-class image classification task with C different classes, it is common practice to use as the final layer for CNN a FC layer with a neuron associated for each of the classes. The output of these neurons are called the *logits*.

The logit associated to class $c \in \{0, \dots, C-1\}$ is denoted as $\hat{z}_c$. The vector of logits, composed by each of the logit associated with each of the classes, is noted $\hat{\mathbf{z}} \in \mathbb{R}^C$.

The choice of the class is then made by choosing the highest logit index via the argmax function. This is a form of winner-takes-all in which the class with the largest logit has output 1 while all other classes have as output 0.

However, in order to bring an additional nuance to the prediction, it is generally preferred to be able to associate a probability of correctness on the predicted class. This can be achieved using the *softmax* function $\sigma : \mathbb{R}^C \to [0, 1]$ which normalizes the inputs. The softmax function σ for the class c is defined as:

$$\sigma_c(\mathbf{z}) := \frac{e^{\mathbf{z}_c}}{\sum_{i=0}^{C-1} e^{\mathbf{z}_i}} \tag{2.14}$$

In the literature, this value is often considered as the probability of correctness associated to the class c. Strictly, $\hat{p}_c := \sigma_c(\hat{\mathbf{z}})$.

In fact, applying this function on the logit vector for each of the C classes allows ending up with a normalized vector $\hat{\mathbf{p}} \in [0, 1]^C$ composed by the probability of correctness associated to each of the classes. This vector thus gives a distribution on the output probability of correctness. This output is preferable to the previous binary one since it reveals uncertainties related to classification. In the following, the output of a CNN will be referred to as the latter.

Finally, the choice of the class is made on the basis of the highest observed probability: $\hat{y} = \text{argmax}_c\, \hat{\mathbf{p}}$. The output prediction pair is thus given by $(\hat{y}, \hat{p}_{\hat{y}})$ respectively as the predicted class and the predicted correctness probability associated to this class.

2.3.2 VGG

VGG is one of the most famous CNN architecture introduced in 2015 by Simonyan and Zisserman [35] at the 2014 ImageNet Challenge. The main innovation of the VGG compared to other architectures for this challenge is that it combines small receptive fields to form larger ones. This trick allows this architecture to reduce the number of total parameters while preserving its viewing aperture.

The idea is to associate small filters (3×3, the smallest possible kernel size to capture the notions of top/bottom, right/left and center) together. Indeed, a stack of two kernels has then an effective receptive field of 5×5 and a stack of three

kernels have then a receptive field of 7×7. One can easily understand that a stack of three layers of kernels has $3.(3.3) = 27$ parameters while a layer of 7×7 has 49. This reduction of the number of parameters facilitates the training of deeper networks.

A VGG architecture will therefore use a succession of convolution layer stacks followed by a ReLU activation. Each of these stacks will then be followed by a max pooling layer to reduce the resolution of the feature maps. The end of the network will be characterized by a few FC layers to process the features and provide the output prediction class.

A more recent improvement of this architecture consists in replacing the convolution/activation blocks by a CBA block, as described earlier.

Finally, an overview of the typical architecture of a VGG model not using the batch normalization layers is available below.

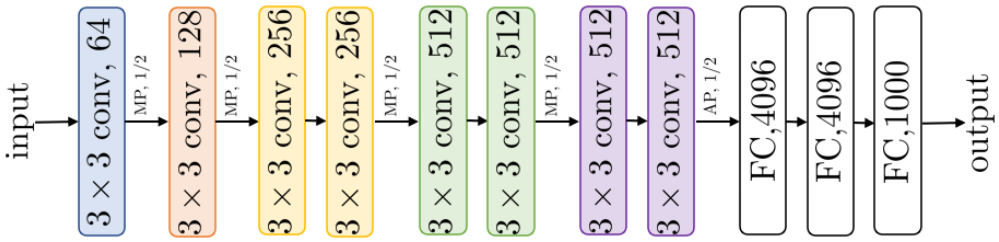


Figure 2.7: A typical VGG-11 model architecture. Each convolutional layer is denoted as $F \times F$ conv, K. Each pooling as either: AP or MP, $reductionratio$ and each FC layer as FC, r_k where r_k is the number of nodes.

2.4 Training of neural networks

So far, the parameters of the networks have been considered well suited for data classification. However, to achieve this result it is necessary to go through an optimization of the learnable parameters. This can be done through *supervised learning* during the *training* phase thanks to some labeled data. This section is dedicated to show the reader how this complex process of fine-tuning the parameters is done.

First, will be seen how the simplest possible neural network, the perceptron, is trained. This will allow us to move on to the training of more complex networks i.e. the training of MLP and CNN. Then, will be described the algorithm that allows training complete networks. This algorithm will be approached mathematically in order to provide the reader with a deep understanding of this subject as it will be necessary for the following. Finally, will be described some possible improvements that facilitate this heavy training task.

2.4.1 Train a single perceptron

In order for the perceptron, a binary classifier, to work correctly it is necessary to adapt its constituting parameters. Via a process of iterative updating of the latter, it is possible to make the perceptron converge thanks to the training set D, composed by n labeled samples of input-output pairs $(\mathbf{x}, y)$. From a notation's point of view, $\mathbf{x}$ is denoted as the input and y as the ground truth (either 0 or 1 here respectively for the negative or positive instance).

The training of the learnable parameters is carried out through the following steps:

1. Initialize all the weights and the bias to a small random value.

2. For each of the sample constituting the training set D, perform the following:

   (a) Compute the output prediction associated to the input sample $\mathbf{x}$ thanks to (2.3), the equation f of the perceptron:

   $$\hat{y} = f(\mathbf{x}; \mathbf{w}) \tag{2.15}$$

   (b) Update the parameters by applying a *delta rule* thank to the ground truth y of the sample:

   $$\mathbf{w} \leftarrow \mathbf{w} - \alpha(\hat{y} - y) \cdot \mathbf{x} \tag{2.16}$$

   Where $\alpha \in \mathbb{R}_0^+$ is the learning rate which determine the scale of the steps that the parameters take.

By performing iteratively the second step for the whole training set, the parameter values are optimized by shifting them towards other values that minimize the error between the perceptron prediction and the labeled ground truth. If the learning rate is well chosen and the dataset is linearly separable, this ultimately results into a perfect linear classifier on that training set.

2.4.2 Train a complete neural network

The training of more complete networks is not fundamentally different from the training of a simple perceptron. It also consists of a first initialization of the parameters followed by an optimization of these parameters via an update rule. The optimization, also iterative, will consist in minimizing an error computed on the basis of different predictions obtained during the inference of a labeled training set.

The difference with the simple perceptron comes from the inference which involves a succession of neurons with different parameters. In order to update these parameters appropriately, it is necessary to know the impact of each of these parameters on the error committed. It is precisely for this reason that updating this type of network is complex. It requires the knowledge of the *causality of each of the parameters* from the network in order to carry out an adapted update of the latter.

Intuitively, one can understand and summarize the training of a network with the following steps:

1. Initialize the parameters of the network.
2. Compute the predictions of the network and the error committed thanks to the labeled training set.
3. Identify and estimate the causality of the parameters in the error.
4. Apply a correction on the learnable parameters that is proportional to the error they have caused.

By repeating steps 2 to 4 several times, this protocol appropriately update the entire network by minimizing the error.

The estimation of what is called here the "causality" has been made possible by the *backpropagation algorithm*. Indeed, the latter allows computing the *gradient of the loss*, previously referred to as causality, function with respect to the weights of the network.

The objective of this subsection is to present these different steps that allow the training of neural networks. The initialization of the parameters will first be

presented before proceeding to the definition of the error function. Finally, the update rule of the parameters will be clearly defined.

Next subsection will focus on the way to compute the gradients, causalities, from the parameters thanks to the backpropagation algorithm.

A. Initialization

Glorot and Bengio [12] study the saturation of neurons across their activation and gradients through the different layers and iterations of training. In their study, they demonstrate the importance of having hidden layers during training that do not saturate in order to avoid the gradients from erasing and deteriorating the quality of the training.

They also observe that for a standard initialization the variance of the back-propagated gradients is decreasing within the different layers that constitute the network. They therefore propose an initialization of the weights called *normalized initialization* which allows the normalization of the gradients through the different layers. This has the effect of favoring a more uniform training across the layers and thus allow the network to achieve better performances.

This technique consists in initializing the weights according to a uniform distribution:

$$W \sim \mathcal{U}\left(\left[-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right]\right) \tag{2.17}$$

Where n_{in} and n_{out} are respectively the amount of input and output features associated to the layer.

B. Loss function

The *error function*, mainly called *loss function*, points out the error between the desired outputs $\mathbf{y}$, contained in the labeled training set, and the output predictions of the network $\hat{\mathbf{p}}$. The choice of a proper error function is very important. Indeed, it is this function that will intrinsically guide each update of the parameters during the entire training since the gradient are computed from this base. It is thus necessary that the value of the loss is high when the prediction of the network is poor; conversely low when the prediction is accurate.

There exist a multitude of different loss functions which differ according to the task to be carried out. During a multi-class classification task, the *cross entropy loss* is generally used.

This function, also called *negative log likelihood*, is defined by:

$$L(\mathbf{y}, \hat{\mathbf{p}}) := -\sum_{c=0}^{C-1} \mathbf{y}_c \log\left(\hat{\mathbf{p}}_c\right) \tag{2.18}$$

Where $\mathbf{y} \in \{0,1\}^C$ is the binary target vector, formed by 0 at each of the inputs and 1 at the corresponding index of the target. $\hat{\mathbf{p}}$ is the prediction vector at the output of the CNN, formed by the predictions on the classification of each class.

This loss presents two main advantages:

1. Its value increases when the predicted probability deviates from the true classification label.

2. Its value grows when the network outputs a vector that is hesitant about the chosen class. This has the effect of increasing the value of the gradients when the prediction is bad or undecided.

This accelerate the convergence of the network and makes the classification rule become sharper.

C. Update rule

The parameters of the network will be optimized thanks to a *gradient descent*. This gradient descent will be carried out in the hyperspace of those parameters in order to minimize the loss. To do this, each of the parameters will be updated independently and proportionally to its causality in the error. It is therefore necessary to compute the gradient associated with each of those.

For convenience, let us first define θ as a general collective notation to identify the parameters constituting the network that requires learning i.e. the weights and biases of each of the neurons. Let us then note D as the training set, composed by all the labeled samples.

The loss function can therefore be written as $L(D;\theta)$ and the error gradient associated to the parameter θ is written: $\partial L(D;\theta)/\partial\theta$.

The update rule of the parameters θ is given by:

$$\theta \leftarrow \theta - \alpha \frac{\partial L(D;\theta)}{\partial \theta} \tag{2.19}$$

Where $\alpha \in [0,1]$ is the learning rate.

This update rule is in fact a slightly modified delta rule that involves the causality aspect.

2.4.3 Backpropagation algorithm

The backpropagation is a general optimization method introduced in the 1970s that allows the automatic differentiation of complex nested functions by exploiting the properties of the *chain rule*. It is thanks to this algorithm that the gradients,

necessary for the update, are derived. It was first applied to neural networks in 1986 by Rumelhart, Hinton, and Williams [32].

This subsection aims to provide the reader with an intuitive and mathematical understanding of how the algorithm operates. The derivation of gradients for FC and convolutional layers will be presented here. The procedure being the same but equations different, one will start with the gradients of FC layers to facilitate the comprehension. The gradients of the convolutional layers will then be explained on the basis of the previous conclusions. It should be noted that will not be addressed here how backpropagation is achieved in other types of layers so as not to flood the reader with complex irrelevant equations.

A. Gradients of FC layers

As a reminder, a FC layer is composed of a set of perceptrons fully connected to the neurons of the previous and following layers.

The learnable parameters that need to be updated, whose gradient must be calculated to apply the update rule, are therefore the component weights of each of the neurons from this layer. In other words, the following term must be estimated:

$$\frac{\partial L(D;\theta)}{\partial w_{ij}^k}$$

In the following, the reader will be shown how to derive this partial derivatives. For sake of clarity, let us recall in the next Table the notations already introduced.

Notation	Description
l_k	k-th layer, numbered from input 1 to output m
r_k	number of nodes on the l_k layer
i	index of a node, numbered from 1 to r_k for the l_k layer
w_{ij}^k	weight (and bias) of node j on layer l_k starting from node i of layer l_{k-1}
a_i^k	activation of node i on the l_k layer
ϕ	activation function
o_i^k	output of node i for layer l_k

Table 2.1: Reminder of the introduced notations.

Our reasoning begins like this: it is known that the error is dependent on the output of the network but also that the output of the network is dependent of the activation associated to any node. In fact, the error is only a complex association of nested functions composed, among other things, of the activation of each of the nodes.

By applying the chain rule on the gradient error, one can split the loss into two different factors that will be easier for us to compute. Mathematically[2]:

$$\frac{\partial L}{\partial w_{ij}^k} = \frac{\partial L}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k} \tag{2.20}$$

Computation of $\partial a_j^k / \partial w_{ij}^k$: By recalling (2.4), the definition of the activation:

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k} \left(\sum_{l=0}^{r_{k-1}} w_{lj}^k o_l^{k-1} \right) \tag{2.21}$$

Which is simply equal to the following:

$$\boxed{\frac{\partial a_j^k}{\partial w_{ij}^k} = o_i^{k-1}} \tag{2.22}$$

The result is quite elegant. This factor is the output from the neurons of the previous layer associated to this weight. It is important to mention that this result is also an intermediate result that was computed during the forward propagation.

Computation of $\partial L / \partial a_j^k$ This part is slightly trickier. It is asked to the reader to hang on a little longer as half the work has already been done. In order to simplify the expression of future relations, a new term δ_j^k is introduced and defined by:

$$\delta_j^k := \frac{\partial L}{\partial a_j^k} \tag{2.23}$$

This *error* is by definition the gradient of the loss function according to the activation. In the following, it will be shown that the factor δ_j^k is dependent on the errors from the subsequent layer δ_i^{k+1}. Consequently, the computation of this factor must be carried out once the forward propagation has been completed and in the opposite direction, i.e. starting from the output layer and propagating towards the input layer. This is where the algorithm gets its name *back propagation* from, as opposed to forward propagation.

In order to demonstrate this, let us derive the error expression. Two cases must be distinguished:

1. *When the error is on the last layer i.e. when $l_k = m$*: the expression from the derivative of the loss function according to the activation of the output node j is:

$$\delta_j^m = \frac{\partial L(D;\theta)}{\partial a_j^m} \tag{2.24}$$

[2]Notice that the notations of the function inputs are dropped to avoid overloading the relations.

Yet, it is known that the prediction of the network associated to nodes j involves the softmax function which is itself dependent on the output of the last node. Mathematically:

$$\begin{aligned}\frac{\partial L(D;\theta)}{\partial a_j^m} &= \frac{\partial L(D;\sigma(o_j^m))}{\partial a_j^m} \\ &= \frac{\partial L(D;\sigma(\phi(a_j^m)))}{\partial a_j^m}\end{aligned} \tag{2.25}$$

By applying the chain rule, the following result is obtained:

$$\frac{\partial L(D;\sigma(\phi(a_j^m)))}{\partial a_j^m} = L'(D;\sigma(\phi(a_j^m))\sigma'(\phi(a_j^m))\phi'(a_j^m) \tag{2.26}$$

Where L', σ' and ϕ' are respectively the derivatives of the loss, softmax and activation function of the last layer according to a_j^m. To conclude with the next relation:

$$\boxed{\delta_j^m = L'(D;\sigma(\phi(a_j^m))\sigma'(\phi(a_j^m))\phi'(a_j^m)} \tag{2.27}$$

2. *Otherwise i.e. when* $k \in \{1,\dots,m-1\}$: by applying the same reasoning as above one can solve the error expression for the hidden layers and the input layer. By exploiting the chain rule again since the activation at layer l_{k+1} is a function of the activation at layer l_k:

$$\begin{aligned}\delta_j^k &= \frac{\partial L}{\partial a_j^k} \\ &= \sum_{l=1}^{r_{k+1}} \frac{\partial L}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k} \\ &= \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k}\end{aligned} \tag{2.28}$$

It is interesting to note that the obtained equation is independent of the biases, i.e., independent of $l = 0$, since the output o_0^k is constant.

Recalling (2.4), the definition of a_l^k:

$$\begin{aligned}\frac{\partial a_l^{k+1}}{\partial a_j^k} &= \frac{\partial \sum_{i=0}^{r_k} w_{ij}^{k+1}\phi(a_i^k)}{\partial a_j^k} \\ &= w_{jl}^{k+1}\phi'\left(a_j^k\right)\end{aligned} \tag{2.29}$$

Thus, by combining these last two equations, the *backpropagation formula* is derived:

$$\boxed{\delta_j^k = \phi'\left(a_j^k\right) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}} \tag{2.30}$$

To summarize what has just been derived, the gradient error with respect to the parameters w_{ij}^k is equal to the product between the output i of the layer l_{k-1} and the error of the node j:

$$\boxed{\frac{\partial L}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}} \tag{2.31}$$

The first factor depends directly on the loss function while the second factor depends only on the previous inputs. o_i^{k-1} can be computed during the inference i.e. during the forward propagation. Whereas δ_j^k involves the error function which can only be computed once L has been computed i.e. once the forward propagation is completed. It has been shown that δ^k is dependent on $delta^{k+1}$, so the computation of the error factor must be carried out starting from the last layer and working towards the input layer. This is known as *backward phase*. It is therefore obvious that the parameters can only be updated once all these factors have been computed i.e. when the backward and forward propagation are finished. It will also be necessary to store all the values of all these factors in memory.

To finish, the reader now understands from the above equations the remark made in the previous subsection about the derivability and non-zero of the activation function. Indeed, an undefined or null value of the activation actually prevents any potential update of the parameters. This is a key notion to understand for the rest of our work.

B. Gradients of convolutional layers

Now that the gradient equations for the FC layers are derived, it is time to move on to the convolutional layers. The principle is not fundamentally different from the one operated on the FC layers. The main difference comes from the particular arrangement of the weights. To recall what has been presented in subsection 2.3.1A., a convolutional layer takes advantage of a shared-weight architecture. It is composed of kernels which are themselves composed by the weights.

It is therefore intended to determine $\partial L/\partial w_{m,n}^k$, the error gradient associated with weight (m, n) from the kernel W of layer l_k[3] as well as $\partial E/\partial b^k$, the error

[3]The careful reader will note that the notations for the depth of volume inputs has been

gradient associated to the bias. By a similar reasoning to the one performed for the previous subsection, Jefkine [22] presents for the interested reader a nice and complete demonstration of the gradients derivation. Here, will only be presented a quick summary of what is being developed.

Let us start by defining the activation $a^k_{i,j}$ located at the coordinates (i, j) of the k-th activation map from the network. By definition (starting from the convolution (2.8) with $D_1 = 1$):

$$a^k_{i,j} = \sum_m \sum_n w^k_{m,n} o^{k-1}_{i+m,j+n} + b^k \tag{2.32}$$

The output vector $o^k_{i,j}$ of the associated activation is given by:

$$o^k_{i,j} = \phi(a^k_{i,j}) \tag{2.33}$$

Where ϕ is the ReLU activation function.

By applying the chain rule, deriving and replacing $\partial L/\partial a^k_{i,j}$ by his error $\delta^k_{i,j}$ via its definition (2.23):

$$\boxed{\frac{\partial L}{\partial w^k_{m,n}} = \sum_{i=0}^{H_2-F} \sum_{j=0}^{W_2-F} \delta^k_{i,j} o^{k-1}_{i+m,j+n}} \tag{2.34}$$

Simplifying the expressions via a little algebra and analysis, the following expression for the error is obtained:

$$\boxed{\delta^k_{i,j} = \phi'\left(a^k_{i,j}\right) \sum_{m=0}^{F-1} \sum_{n=0}^{F-1} \delta^{k+1}_{i-m,j-n} w^{k+1}_{m,n}} \tag{2.35}$$

It can be seen that the error gradient associated with the parameters of a convolutional layer is a combination of the errors from next layer, computed during backpropagation, with outputs from the previous layer, computed during forward propagation.

Intuitively, the expression of the gradients of a FC layer is found except that it is adapted to the convolution operation which involves shared weights and translation along the input volume.

C. The algorithm

In brief, the backpropagation algorithm works as follows[4]:

dropped in order to the simplify the notations. In fact, only the case $D_1 = 1$ is considered here in order to simplify the reader's life but the process remains generalizable for $D_1 \geq 1$.

[4]In practice, this algorithm does not operate precisely like this. It is subject to many optimizations which allow an acceleration of the computations.

1. **Forward phase**. For each of the labeled samples of the training set:
   (a) Compute the prediction results thanks to equations of the forward propagation (2.4) and (2.5).
   (b) Store in memory the predictions, activation and outputs from each node constituting each layer.
2. **Backward phase**. For each of the labeled samples of the training set:
   (a) Compute the loss thanks the ground truth and the predictions computed in the previous step.
   (b) Compute the gradients thanks to values previously saved in memory. Start on the last layer and back propagate iteratively layer by layer with the provided equations until the first layer.

D. Train an MLP

The training of a network of only FC connected layers i.e. an MLP can be realized in practice via the following steps:

1. **Initialization**. A random initialization of the parameters is performed. This process is described in subsection 2.4.2A..
2. **Computation of the gradients**. This step is performed using the back-propagation algorithm as described above. Once the training set is processed, compute the whole set of gradients.
3. **Update the parameters**. This step consists in updating the parameters via the update rule described in subsection 2.4.2C..

By iterating steps 2 and 3 several times, the parameters are updated with respect to their causality on the loss. This allows the model to be trained by minimizing the error and thus to converge to better performance.

A model is trained by performing several complete passes on the training set. An *epoch* consists in performing a complete run on all the samples of the training set.

E. Train a CNN

The training of a CNN can also be done with the same steps as described above.

The only difference is that the computation of the gradients is slightly modified. Indeed, a CNN is composed of different types of layers (convolutions, BN, pooling, ...).

These changes of layers within the network must be taken into account during the backpropagation. This results in a slightly more complex backward phase involving different gradient equations. The derivation of the gradients from all the different layers is not addressed here so as not to make this work unnecessarily complex.

In practice, the automatic differentiation of the different layers is automatically managed by the deep learning frameworks (TensorFlow uses autodiff [20]; Pytorch uses Autograd [1]). This saves the coder from having to reimplement the backpropagation each time.

2.4.4 Improve the learning

Network training is a long and difficult task that requires a lot of data and computing power. So far has been discussed how to train a network in a very concrete way.

This subsection addresses some essential points that, when put into practice, will optimize and facilitate the training.

A. Stochastic gradient descent

Equation (2.19) for updating the parameters of a network indicates that it is necessary to compute the gradients for the whole training set in order to update the weights. One speaks then about *batch gradient descent*. Although the gradient descent is guaranteed to converge to the global minimum for convex error functions and to a local minimum for non-convex, it is in practice not usable. Indeed, datasets do not usually fit in memory and the convergence of the model can be very slow since an update requires a lot of computing resources.

In contrast, a *Stochastic Gradient Descent* (SGD) involves updating of the parameters after each sample in the training set. This allows for faster convergence while making the computations feasible. Only due to the frequent updates of the parameters, SGD has a high variance on the minimization of the cost function.

A compromise between these two extremes is the *mini-batch gradient descent* which will be referred to as SGD by abuse of language. It consists in performing an update of the parameters after the process of a subset of the training set. It thus allows taking advantage of each of the previous techniques:

1. It reduces the variance on the update of the parameters which allows a more stable convergence.

2. It allows limiting the computational resources while offering the possibility of carrying out matrix optimizations which offer a faster learning.

Considering a mini-batch d as a subset of n samples from the training set D. The update rule of the parameters becomes:

$$\theta \leftarrow \theta - \alpha \frac{\partial L(d;\theta)}{\partial \theta} \tag{2.36}$$

For $n = N$ is found the gradient descent equation while for $n = 1$ is found the classical stochastic gradient descent equation.

B. Momentum

Sutskever et al. [36] explains why adding a momentum term when updating parameters in a learning algorithm is effective. The author shows that the momentum μ can increase the range of learning rate over which the system converges.

The addition of a momentum term consists of adding a portion from the previous update when updating the parameters. This has the effect of accelerating the gradient descent in the right direction by adding inertia to the movement. This inertia allows the damping of the oscillations caused by the various mini-batches.

By physical analogy, the *velocity* is defined by:

$$v^{t+1} \leftarrow \frac{\partial L(d;\theta)}{\partial \theta} + \mu v^t \tag{2.37}$$

An exponential notation is introduced here to explain the temporal dependencies of the data. The update rule of the parameters becomes:

$$\theta \leftarrow \theta - \alpha v^{t+1} \tag{2.38}$$

C. Weight decay

Krogh and Hertz [25] proved that a weight decay could improve generalization of neural networks by preventing overfitting.

The authors showed that adding a regularization term in the cost function during the learning process allows suppressing the irrelevant components of the weights by favoring the choice of simple weights. The added penalty is proportional to a parameter λ which will indicate its intensity.

By applying the weight decay and noting the general loss function L_0:

$$L(d;\theta) = L_0(d;\theta) + \frac{1}{2}\lambda \sum_i \theta_i^2 \tag{2.39}$$

Computing the gradient of the loss for the parameters, it is observed that the regularization term introduces a new term $\lambda\theta$ in the weight update equation which becomes:

$$\theta \leftarrow \theta - \alpha \left(\frac{\partial L(d;\theta)}{\partial \theta} + \lambda\theta \right) \tag{2.40}$$

D. Learning rate scheduler

In deep learning, it is common to refine the learning rate as the training process progresses in order to reduce the size of the update steps during the gradient descent. This allows converging more precisely towards the minimum of the loss function. There are many different learning rate schedulers, each with their own advantages/disadvantages.

Here will be used for this study the *multi-step learning rate scheduler* defined by a decay factor η which reduces the learning rate each time a *milestone* of epochs is reached. Mathematically:

$$\alpha_{\text{epoch}} = \begin{cases} \eta\alpha_{\text{epoch}-1}, & \text{if epoch in [milestones]} \\ \alpha_{\text{epoch}-1}, & \text{otherwise} \end{cases} \tag{2.41}$$

E. Data augmentation

Training neural networks requires a very large amount of data in order to avoid overfitting. This phenomenon is observed when a model learns a function with a large variance to fit perfectly with the training set. This has the effect of strongly deteriorating the performance of the network when it is tested on other data. *Data Augmentation* (DA) is a data-space solution to this problem. Thanks to a set of data transformation techniques that modify the size, quality or other features of the training set, it is possible to build models that have a better generalization.

Shorten and Khoshgoftaar [34] describe the different classical techniques used for learning with CNN. The focus is done here mainly on the following 3 transformations:

- *Normalization*: consists in normalizing the input channels of an image to ensure similar behavior of the network for all the inputs.
- *Flipping*: consists in flipping the image around its horizontal axis. Vertical flipping is also possible but less common as it completely transforms the data properties.
- *Cropping*: consists in modifying the dimensions of an input by cropping a central patch of each image. Random cropping can simulate an effect similar to translation.

Chapter 3

Background on quantization

The purpose of this Chapter is to provide the reader with a fundamental *understanding of quantization*. It will be divided into two distinct parts.

The first one will introduce the *mathematical formalism of quantization*. Different *systems using dither*, the method at the core of this study, will then be presented.

The second part will aim at introducing the *quantization of the activation within the neural networks*. The particular training that results from the quantization of activation will then be addressed.

3.1 Quantization

In signal processing, a *quantization consists in assigning a value from a predefined discrete set to each of the samples that constitute the continuous signal* in order to make it discrete.

In this Chapter, the mathematical background of the quantizers will first be introduced as well as the system that performs the quantization. *Dither*, a technique used in signal processing to *decorellate the quantization error*, introduced by the quantizer will then be presented. Dither is a key concept in this work that it is essential to understand for the following.

3.1.1 Scalar quantizer

Strictly, a *scalar quantizer* of size n is an application Q from $\mathbb{R}$ to a discrete set $Z = \{z_1, \ldots, z_n\}$. It is noted $Q : \mathbb{R} \to Z$ such that $Q(z) = z_k$ for $k \in \{1, n\}$.

A quantifier can also be seen as being a set of intervals belonging to the starting space, $[t_k, t_{k+1}]$, called *decision levels* to which corresponds a value of the arrival

space, z_k, called *reconstruction level*. The *step of quantization* Δ_k is then defined as being the difference between 2 decision levels: $\Delta_k = t_{k+1} - t_k$.

A. Uniform scalar quantizers

A *uniform scalar quantizers*, i.e. quantizers for which the quantization step is constant: $\Delta_k = \Delta \ \forall \ k \in \{1, n\}$. The graph associated with this kind of quantizer, looks like a staircase function with steps of uniform size. In the literature, the step size, Δ, is referred to as the *least significant bit* (LSB) since a change in the input signal level of a step width corresponds to a change of one LSB in the binary output.

In general, two different types of quantizers are found: the *mid-tread* and the *mid-riser*.

Mid-tread The quantization equation of a mid-tread quantizer of input z is defined as follows:

$$Q(z) = \Delta \cdot \left\lfloor \frac{z}{\Delta} + \frac{1}{2} \right\rfloor \tag{3.1}$$

Where $\lfloor \ \rfloor$ is the floor function.

Mid-riser The quantization equation of a mid-riser quantizer of input z is defined as follows:

$$Q(z) = \Delta \cdot \left\lfloor \frac{z}{\Delta} \right\rfloor + \frac{1}{2} \tag{3.2}$$

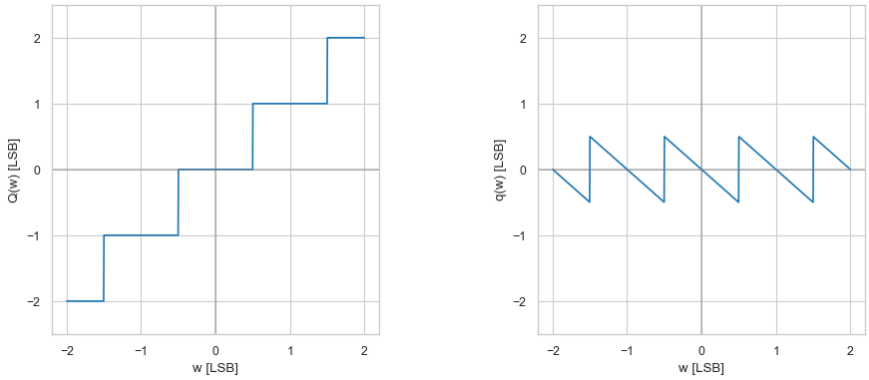
Unlike mid-tread, the mid-riser never outputs 0. This can be useful in some applications.

B. Quantization error

When quantifying a signal, a continuous space is projected onto a discrete space. This compression results in a loss of information. The distortion introduced is called the *quantization error*. It is noted q and defined by:

$$\boxed{q(z) := Q(z) - z} \tag{3.3}$$

The following image illustrates the transfer characteristics and the quantization error as a function of z for a uniform scalar quantizer. It can be seen that the error amplitude is always less than 0.5 LSB and is periodic of period 1 LSB over z.



(a) Transfer characteristics (b) Quantization error

Figure 3.1: Illustration of a mid-tread quantizer.

3.1.2 Multi-bit quantizing systems

Multi-bit quantizing systems are uniform scalar quantizers that encode their output over a predefined number of bits. A n-bit quantization thus has $|Z| = 2^n$ outputs associated to inputs distributed uniformly over the bounded input domain. In practice, bounded inputs are required in order to avoid saturation of the quantizer.

For the following, it is necessary to introduce some notations. The *input of the system* is noted x, the *output of the system* y and the *total error of the system*[1] ε such that:

$$\boxed{\varepsilon := y - x} \tag{3.4}$$

In the remainder, two different archetypes of quantization systems involving dither are presented. To be mentioned: the *non-subtractively dithered* (NSD) and the *subtractively dithered* (SD) systems.

But before presenting these systems, a presentation of a system without dithering will be made, the *undithered* system (UD).

A. Undithered systems

UD is the most classical system. The input of the system x is identical to the input of the quantizer z. Then, by definition the quantization error q is equal to the total error ε. The following relations are written:

$$\begin{aligned} y &= Q(x) \\ &= x + q(x) \\ &= x + \varepsilon_{UD} \end{aligned} \tag{3.5}$$

[1] In practice and by abuse of language, this term will be referred to as the quantization error.

An illustrative diagram of this system is available in Figure 3.2a.

B. Dithered systems

For the following two systems, a random process is introduced, ν, called *dither*, which is statistically independent of the input x and stationary. The dither is generally considered to be *independent and identically distributed*. This dither ν is added to the input of the system x in order to control its relation with the total error ε and the statistical properties of its output y. Note that the two following systems require the generation of randomness contrary to the preceding totally deterministic system.

The input of the quantizer, z, is defined by:

$$z = x + \nu \tag{3.6}$$

From the above equation, it can be observed that the result of the *quantization as well as the total error is stochastic*, unlike the previous system.

Non-subtractively dithered The idea of NSD systems is simply to apply quantization on this intermediate result z. The following relations are obtained:

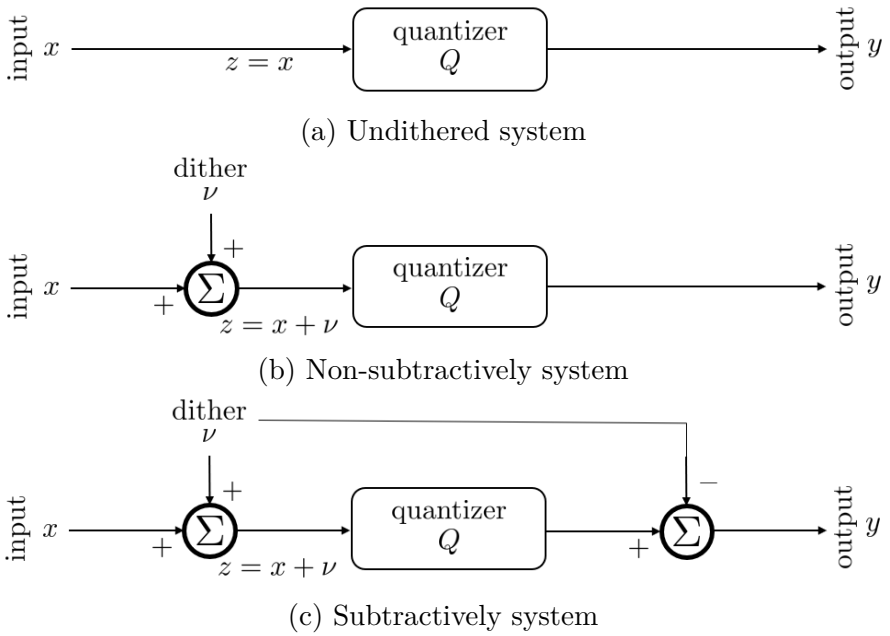
$$\begin{aligned} y &= Q(z) \\ &= x + \nu + q(x + \nu) \\ &= x + \varepsilon_{NSD} \end{aligned} \tag{3.7}$$

The total error is noted $\varepsilon_{NSD} = \nu + q(x + \nu)$. An illustrative diagram of this system is available in Figure 3.2b. The following section aims to show what this implies.

Subtractively dithered The idea of SD systems is simply to apply quantization on this intermediate result z and then subtract the output of the quantizer with this same dither ν. The following equations are derived:

$$\begin{aligned} y &= Q(z) - \nu \\ &= x + q(x + \nu) \\ &= x + \varepsilon_{SD} \end{aligned} \tag{3.8}$$

The total error is noted $\varepsilon_{SD} = q(x + \nu)$. The advantage of this archetype is that it makes the *dither term ν disappear in the output y* unlike the previous system with no subtraction. The main disadvantage of this approach is that the *outputs are non-discrete*. This also requires the ability to generate a random signal and store it for subtraction. An illustrative diagram of this system is available in Figure 3.2c.



(a) Undithered system

(b) Non-subtractively system

(c) Subtractively system

Figure 3.2: Summary diagram of the different multi-bit quantizer systems.

3.1.3 Output analysis

The inherent non-linearity introduced in the systems described above, whose description is misleadingly simple, makes its analytical analysis surprisingly difficult. This section provides an *understanding from of the benefits introduced by dither* by analyzing the errors of the systems as well as the mean output. First will be presented the UD systems in order to better highlight the beneficial effects of introducing dither in the other systems.

A. Undithered system

In 1948, Bennett [4] shows that the quantization error of a uniform quantizer ε_{UD} (which is equivalent to the total error for this archetype) can be approximated by a sequence of random variables uniformly distributed over $[-\Delta/2, \Delta/2]$. This sequence is decorrelated from the input signal x. In other words, the authors state that $\varepsilon_{UD} \overset{i.i.d.}{\sim} \mathcal{U}([-\Delta/2, \Delta/2])$ i.e. the *error can be approximated by a uniformly distributed additive white noise*. The authors further demonstrate that this approximation *is reasonable if and only if*:

1. The number of quantizer levels $|Z|$ or 2^n is sufficiently large.
2. The step size Δ is small enough.

3. The input probability density function p_x is smooth enough.

Those *conditions on the approximation turns out to be quite restrictive* for many applications and thus does not always allow a system to be described correctly. To convince ourselves of this, let us take a simple sinusoidal signal with an amplitude of less than 0.5 LSB as an input signal. The output of this signal through a mid-tread quantizer will be the null function and the quantization error will be the opposite of the input signal. The error is thus totally correlated to the input signal. In fact, the quantizer clearly does not respect the second condition.

More mathematically, the focus for this study will be on the output expectation. Starting from equation (3.9) of the undithered system, the following relations are derived:

$$\begin{aligned} \mathbb{E}[y] &= \mathbb{E}[x + \varepsilon_{UD}] \\ &= \mathbb{E}[x] + \mathbb{E}[q(x)] \end{aligned} \tag{3.9}$$

On average, it is observed that the output y will be equivalent to the input except for an error term. It can be noted that this error term will be a function of the input and will therefore be deterministic. The analytical characterization of this bias depends directly on the input, it cannot be derived in a general way. This is especially true for a signal that does not respect the 3 rules stated above.

B. Dithered system

In 1964, Schuchman [33] shows that *adding a particular dither to the system input before quantization allows the quantizer input signal to be forced to meet the conditions* of the approximation stated above by Bennett. To this end, Schuchman states in his paper two rather restrictive properties on the distribution that the dither must adopt. More precisely, the author shows that a dither ν distributed uniformly between $-\Delta/2$ and $\Delta/2$ allows the respect of these properties.

To put it another way, *the addition of a dither* $\nu \sim \mathcal{U}([-\Delta/2, \Delta/2])$ *to the input of a system* x *before a quantization* Q *allows turning the quantization error* q *independent of* x. This decorrelation also allows the error to be uniformly distributed on $[-\Delta/2, \Delta/2]$ and this no matter the input.

Non-subtractive dither Now, considering a dither as described above and taking equation 3.7 of the NSD, the following relations for the output expectation are obtained:

$$\begin{aligned} \mathbb{E}[y] &= \mathbb{E}[x + \varepsilon_{NSD}] \\ &= \mathbb{E}[x] + \mathbb{E}[\nu] + \mathbb{E}[q(x + \nu)] \\ &= \mathbb{E}[x] \end{aligned} \tag{3.10}$$

It is observed that *on average, the output is equivalent to the input.*

Subtractive dither By a similar reasoning, the *same conclusion* as for the subtractive dither is obtained via the following relations:

$$\begin{aligned} \mathbb{E}[y] &= \mathbb{E}[x + \varepsilon_{SD}] \\ &= \mathbb{E}[x] + \mathbb{E}[q(x + \nu)] \\ &= \mathbb{E}[x] \end{aligned} \tag{3.11}$$

C. Core idea of dither

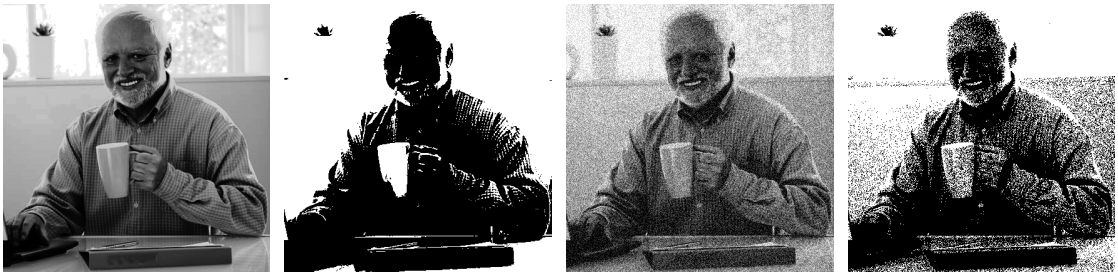
First, it has been shown for a non-dithered quantizer that, on average, the output is equal to the input with the exception of an error term introduced by the quantization. Further, it has also been shown that this bias, the quantization error, is independent of the input only if certain restrictive requirements are met. Otherwise, there is no mathematical guarantee at this level.

It has then been demonstrated that adding a dither ν to the input x of a quantization system makes the quantization error q independent of the input and uniformly distributed over a centered domain of width Δ for any input.

Thanks to these last properties, it has been demonstrated that on average the input x is equal to the output y. In other words, adding a dither to an input signal through a uniform quantization allows, thanks to an averaging process, to limit the impact of quantization.

This phenomenon can be observed more explicitly for the following Harold[2] [16] images. It is fairly straightforward to observe that the dithered (binary) version expresses more details than the simply quantized (also binary) version. Indeed, the *dither adds a touch of uncertainty to the quantizer which has the effect of smoothing the impact of the quantization*. The averaging process is automatically performed by the human ocular system.

[2]Besides looking great, Harold is an electrical engineer as I pretend to be.



(a) Original (b) UQ (c) Dithered (d) DQ

Figure 3.3: Image used as an example to illustrate the smoothing of dither on a binary quantized image. (a): Original image (grayscale). (b): Original binary quantized image (Undithered Quantified). (c): Dithered image (grayscale). (d): Dithered binary quantized image (Dithered Quantified).

3.2 Quantization applied to activation function

This section introduces the *activation quantizer and its effects on the network*. First, will be introduced mathematically this quantifier and then will be discussed the impact of the quantization of the activation on the training and how to adapt it.

3.2.1 ReLU quantizer

The study of this work focuses on the quantization of the ReLU activation, the main activation function of a convolutional network.

The first thing to observe to quantify this activation is that the standard ReLU function (*defined by* (2.13)) is unsaturated. Indeed, the image of this function is unbounded for the positive values. Since a n-bit quantizer only has 2^n outputs, it is necessary to clip the ReLU activation function. Choi et al. [7] investigate the impact of the clipping parameters by studying the performances of various saturation threshold. The latter reveals that there should be no performance issue using the saturation of values greater than 1[3].

The saturation threshold of the positive inputs will therefore be set by convention to 1. The *clipped ReLU* is thus defined by:

$$\text{ReLU1}(x) := \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } 0 < x \leq 1\ 3 \\ 1 & \text{elsewhere} \end{cases} \tag{3.12}$$

A uniform mid-tread scalar quantizer Q is then introduced in order to quantize on n-bit(s) the ReLU activation function. The step Δ of the quantizer will be defined by $1/2^{n-1} = 2^{1-n}$.

The quantifier Q on n-bit(s) applied to a clipped ReLU will have the following expression:

$$Q(\text{ReLU1}(x)) := \begin{cases} 0 & \text{if } x \leq 0 \\ 2^{1-n} \cdot \left\lfloor 2^{n-1}x + \frac{1}{2} \right\rfloor & \text{if } 0 < x \leq 1 \\ 1 & \text{elsewhere} \end{cases} \tag{3.13}$$

An illustration of ReLU clipped quantizers for 1 and 2 bits is available on Figure 3.4.

[3]This was checked empirically afterwards to make sure that there was effectively no loss of performance.

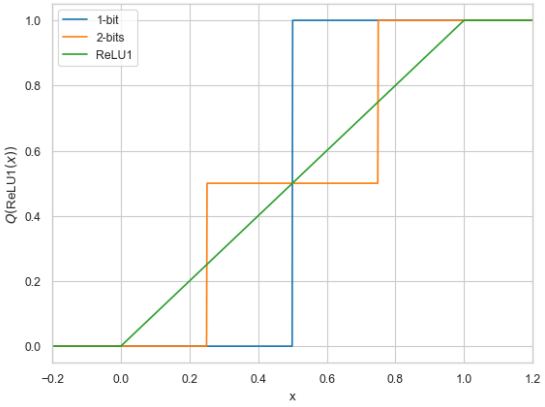


Figure 3.4: Mid-tread uniform scalar quantizer on a clipped ReLU activation for a quantization on 1 and 2 bits.

3.2.2 Gradient cancellation

The quantization of activation allows, on the one hand, to compress the activation, which is the goal of this compression technique, but it also implies another issue which concerns the training of networks.

The *problem lies more precisely in the estimation of the gradients during back-propagation*. Indeed, the computation from the gradient error associated with the different weights involves the derivative of the activation function (*cf.* (2.30) *and* (2.35) *for proof*).

Knowing the derivative of a staircase function is null on the whole of its domain, it is then obvious to understand that this *causes the cancellation of the gradients* whatever the activation is. This cancellation *thus prevents any possible update of the parameters* and consequently, the convergence during the SGD towards a minimum.

3.2.3 Straight-through estimator

To overcome this problem, there is a method called *straight-through estimator* (STE) introduced by Hinton [17] and studied by Bengio, Léonard, and Courville [3] which *solves this cancellation of the error gradients* during training.

This method consists in *replacing during the backpropagation the true derivative of the activation* (which would be null) *by an estimate* of what it should have been. The challenge is obviously to estimate this value as well as possible in order to adapt the SGD steps as well as possible. Indeed, the better the estimation, the more the steps will be adapted and the more the gradient descent converges towards a minimum of the loss function. It is assumed that a lower minimum offers better

performance to models.

The estimator of the activation function when training quantized activation networks will then be given by its continuous function, as if there were no quantization. Its associated derivative is therefore given by:

$$\mathrm{ReLU1}'(x) = \begin{cases} 1 & \text{if } 0 < x \leq 1 \\ 0 & \text{elsewhere} \end{cases} \tag{3.14}$$

It is worth noting that there exist several possible estimators. The one used here will simply be the most straight forward one i.e. the clipped ReLU.

Chapter 4

Motivations

This Chapter is intended to *motivate the use of dithered quantization on activation*. This choice will be motivated by first analyzing the *impact of quantization on training*. Then, the impact of *dither* on quantization will be shown and how it could potentially help by *refining the SGD steps during training*.

4.1 Impact of quantization on training

In order to *understand the real impact of quantization on the training of the network*, a simplistic example will be first introduced which will be later generalized to fit the situation. Suppose an MLP network consisting of several FC layers using the clipped ReLU as activation. The focus will be on the update of a parameter that constitutes one of these FC layers.

The background Chapter on neural networks explains how to update the parameters. Recalling the update rule introduced in subsection 2.4.4A. on stochastic gradient descent and the notations of a FC layer (*introduced in subsection 2.2*), the update rule is given by:

$$w_{ij}^k \leftarrow w_{ij}^k - \frac{\alpha}{N} \sum_{x \in d} \frac{\partial L_x}{\partial w_{ij}^k} \tag{4.1}$$

Where d is a mini-batch of size N and L_x is the loss computed using sample x.

From this upper equation, it can be seen that it is necessary to compute the value of the gradient loss with regard to this parameter for each of the samples constituting the mini-batch. Those values can be derived thanks to the backpropagation algorithm. Recalling definition (2.31) of the gradient of a FC layer and knowing that the activation function is quantized clipped ReLU, the following relation can be written:

$$\frac{\partial L_x}{\partial w_{ij}^k} = \delta_{x;j}^k Q(\text{ReLU1}(a_{x;ij}^{k-1})) \tag{4.2}$$

The above equation can be rewritten in the following form which makes the quantization error explicit as already done in equation (3.9) from the quantization background Chapter.

$$\frac{\partial L_x}{\partial w_{ij}^k} = \delta_{x;j}^k \left(\text{ReLU1}(a_{x;ij}^{k-1}) + \varepsilon_{x;ij}^{k-1} \right) \tag{4.3}$$

Where $\varepsilon_{x;ij}^{k-1}$ is the *quantization error on the clipped ReLU* of the associated sample x.

The update rule is therefore rewritten as follows:

$$\begin{aligned} w_{ij}^k &\leftarrow w_{ij}^k - \frac{\alpha}{N} \sum_{x \in d} \left(\delta_{x;j}^k \text{ReLU1}(a_{x;ij}^{k-1}) + \delta_{x;j}^k \varepsilon_{x;ij}^{k-1} \right) \\ &\leftarrow w_{ij}^k - \alpha \left(\frac{1}{N} \sum_{x \in d} \delta_{x;j}^k \text{ReLU1}(a_{x;ij}^{k-1}) + \frac{1}{N} \sum_{x \in d} \delta_{x;j}^k \varepsilon_{x;ij}^{k-1} \right) \end{aligned} \tag{4.4}$$

In order to slightly clarify the notations, the empirical average on the mini-batch d operator $\mathbb{E}_d$ is introduced.

The update of the parameters constituting the system can be thus noted in a very general way as:

$$\boxed{w_{ij}^k \leftarrow w_{ij}^k - \alpha \mathbb{E}_d \left[\delta_j^k \text{ReLU1}(a_{ij}^{k-1}) \right] - \underbrace{\alpha \mathbb{E}_d \left[\delta_j^k \varepsilon_{ij}^{k-1} \right]}_{\text{error term}}} \tag{4.5}$$

This last relationship teaches us one thing: the *update of the parameters constituting a system with quantized activation is equivalent to the update of a non-quantized system expect to an error term from the quantization.* Further, this *error term has absolutely no mathematical guarantee* on the value that it can take. It is quite possible that its value differs completely from one mini-batch to another as well as for each of the different iterations.

4.1.1 Bias in the gradient descent

To clarify the *impact of this error term in the gradient descent*, let us imagine the simplistic loss valley for a system composed of only 2 parameters to be optimized. This loss valley is illustrated in Figure 4.1.

Optimizing this system involves to iteratively update these parameters via a gradient descent in order to find a minimum. However, it just has been shown in the previous subsection that the quantization of the activation causes an error term into the gradients. This quantization error term has the *consequence of deviating*

the vector from its true trajectory, thus making the *minimization of the loss less efficient*.

Two different vectors are represented on the following imaginary loss valley image. The green one represents the *true computed gradient* step which does not involve the quantization error term. The red one represents the *gradient calculated during the backpropagation* when the quantization error is included. These vectors result from the combination of the gradients computed for each of the two parameters w_1, w_2. The fictitious quantization errors, being, of course, exaggerated to facilitate the understanding of the reader, causes a change in direction and intensity of the red vector with respect to the optimal green one. These changes in direction thus make the search for a minimum more complicated and consequently the SGD suboptimal.

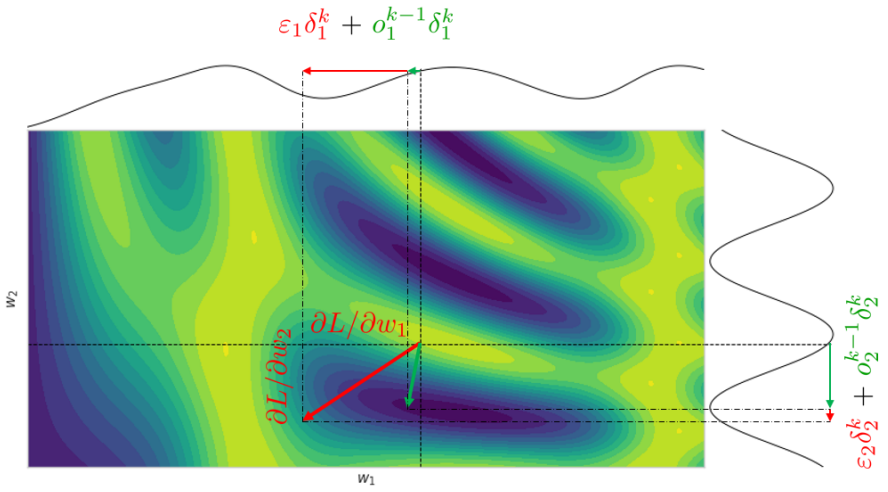


Figure 4.1: Valley of the loss function with regard to the two parameters of the model. Yellow represents the maximum values and blue the minimum values.

This simplistic scenario clearly exaggerates the quantization errors to make this explanation more explicit. The optimization of a real model is the subject of a SGD in a hyperspace of parameters with much higher dimensions. The quantization errors, however small, added together for such a high number of dimensions can still result in important changes of direction.

4.1.2 Gradient mismatch

It is found in the literature of compressed networks another way to explain the performance losses incurred following the quantization of the activation. This concern is known as the *gradient mismatch* and has namely been studied by [27], [23] and [7].

This phenomenon comes from the *difference observed between the activation during forward and the estimated activation during backward with the straight-through-estimator*. Indeed, this difference causes a mismatch which *makes it difficult to estimate the real steps to be taken*. This again prevents SGD from performing well since the minimization steps are impacted. The optimization of the parameters is then jeopardized, which in the end provides a model with poorer performance.

Many papers study different estimators more or less elaborated in order to reduce as much as possible this mismatch gradient and thus optimize the training of the networks ([10], [30], [6], [43]).

4.2 Erase quantization effects

This section introduces the *addition of dither in the ReLU quantization function* and demonstrates the properties on the output of this quantizer that follow from it.

4.2.1 Dither on quantized activation

Let us introduce in a very general way a dither ν which will be applied during the quantization of the activation.

The quantization system to be studied is the *non-subtractive dither* as described in subsection 3.1.2B.. Dither ν is uniformly distributed over a quantization step width (1 for 1-bit) centered over 0. The input of our quantization function defined previously by $\text{ReLU1}(x)$ in (3.13) therefore becomes $\text{ReLU1}(x) + \nu$ with $\nu \sim \mathcal{U}([-\Delta/2, \Delta/2])$.

The dithered clipped ReLU quantizer for 1-bit compression will therefore be defined as:

$$Q(\text{ReLU1}(x) + \nu) := \begin{cases} 0 & \text{if } \text{ReLU1}(x) + \nu \leq 0.5 \\ 1 & \text{elsewhere} \end{cases} \tag{4.6}$$

4.2.2 No quantization on average

Applying the same reasoning used to analyze the output of this quantifier as in subsection 3.1.3B., dedicated to this effect:

$$\begin{aligned} \mathbb{E}\left[Q(\text{ReLU1}(x) + \nu)\right] &= \mathbb{E}\left[\text{ReLU1}(x) + \varepsilon_{NSD}\right] \\ &= \mathbb{E}\left[\text{ReLU1}(x)\right] + \mathbb{E}\left[\varepsilon_{NSD}\right] \end{aligned} \tag{4.7}$$

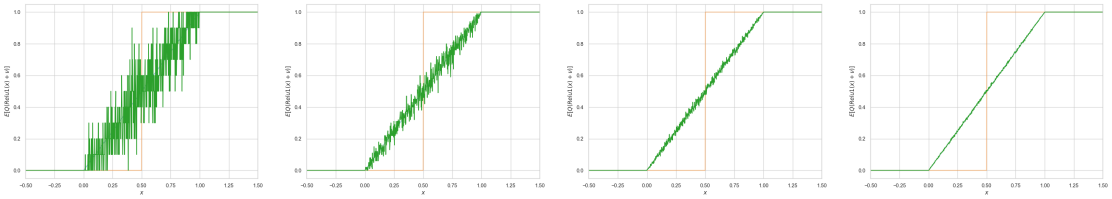
Since $\varepsilon_{NSD} \sim \mathcal{U}([-\Delta/2, \Delta/2])$ (property of the quantization error of an NSD), the following relationship is derived:

$$\boxed{\mathbb{E}\left[Q(\text{ReLU1}(x) + \nu)\right] = \mathbb{E}\left[\text{ReLU1}(x)\right]} \tag{4.8}$$

This demonstrates that dither allows to *erase on average the effect of quantization on the clipped ReLU function*.

In order to convince the reader of the beneficial effect of dither applied to the quantization of a clipped ReLU, a Monte Carlo simulation was performed to ensure that on average the effects of quantization were canceled. This empirical verification is done by averaging n different dither realizations associated with each of the inputs.

The following plots are obtained using a 1-bit quantization. It is clearly observed that with N growing, the initial curve of the clipped ReLU is found. Indeed, *the greater the number of realizations, the better the mean approximation will be and the more the effect of the quantization will disappear*. This is the core idea behind using dither into those quantizers.



(a) $n = 10$ (b) $n = 100$ (c) $n = 1000$ (d) $n = 10000$

Figure 4.2: Expected value of the quantized clipped ReLU using dither for various numbers of realizations.

4.3 Improve learning with dithered quantized activation

The first section of this Chapter informs us that the quantization of the activation introduces an error bias in the estimation of the gradients necessary for the update of the parameters. The bad estimation of this gradient implies a divergence of the steps realized during the minimization of the loss thus making the SGD suboptimal. Indeed, the error term $\alpha\mathbb{E}_d\left[\delta_j^k \varepsilon_{ij}^{k-1}\right]$ involved in the update rule has no mathematical guarantee of any kind.

However, it has just been shown in the previous section that the quantization error resulting from the quantization of the clipped ReLU can be controlled by

applying a dither. Indeed, adding the latter before quantization allows to *control the distribution of the error and therefore, on average, to erase the distortion introduced by this compression process.*

Let us see what impact this control of the error distribution has on the parameter update rule.

First, the notation $\varepsilon_{x;ij}^{NSD,k-1}$ is introduced to designate the quantization error committed on sample x using a dithered quantized activation. By the non-subtractive dither system assumption, it can be stated the following property:

$$\varepsilon_{x;ij}^{NSD,k-1} \sim \mathcal{U}\left(\left[-\frac{\Delta}{2}, \frac{\Delta}{2}\right]\right)$$

Next, knowing that the back propagated error $\delta_{x;j}^{k}$ is independent of the quantization error (*cf.* (2.30) *for proof*[1]), the subsequent statement can be made:

$$\delta_{x;j}^{k}\varepsilon_{x;ij}^{NSD,k-1} \sim \mathcal{U}\left([-\gamma, \gamma]\right)$$

With γ as a constant.

At this point, the discerning reader should quickly understand what is going on. Suppose now the training of the network is performed on a mini-batch d of infinite size i.e. $N \to \infty$. The error term appearing in the update rule can therefore be rewritten as:

$$\begin{aligned}\lim_{N\to\infty} \mathbb{E}_d\left[\delta_j^k \varepsilon_{ij}^{NSD,k-1}\right] &= \mathbb{E}\left[\delta_j^k \varepsilon_{ij}^{NSD,k-1}\right] \\ &= 0\end{aligned} \tag{4.9}$$

This important result indicates that the addition of a *dither allows erasing, via an averaging process, the problematic error term involved in the gradient update.* It therefore seems justified, assuming a sufficiently high but still finite batch size N, that addition of *dither would optimize the gradient descent* by erasing (at least partially) the negative impact of quantization.

Using dither when quantizing a clipped ReLU activation for a FC layer gives the following corrected update rule:

$$\boxed{\begin{aligned} w_{ij}^k &\leftarrow w_{ij}^k - \alpha\mathbb{E}_d\left[\delta_j^k \mathrm{ReLU1}(a_{ij}^{k-1})\right] - \alpha\mathbb{E}_d\left[\delta_j^k \varepsilon_{ij}^{NSD,k-1}\right] \\ &\approx w_{ij}^k - \alpha\mathbb{E}_d\left[\delta_j^k \mathrm{ReLU1}(a_{ij}^{k-1})\right] \end{aligned}} \tag{4.10}$$

In summary, the use of dither allows controlling the distribution of the quantization error. Via a mini-batch averaging process, the bias introduced by the

[1]Dither generations are independent from a layer to another layer, which guarantees the independence of this component.

compression process in the SGD steps is erased, which enables the convergence to be more efficient. This should theoretically lead to a model that performs better by refining the gradient steps i.e. by reducing the gradient mismatch. This should lead into an improvement of performance.

4.3.1 Generalization to convolutional layers

For sake of simplifications, the deletion from the negative impact of quantization during training has been demonstrated at this time only for FC layers. This subsection aims at *generalizing what has been previously shown to convolutional layers* since it is this type of layer that mostly constitutes CNN.

The procedure presented here is basically the same as the one derived for the FC layers. The update of the parameters constituting a convolutional layer requires the calculation of the gradient of the loss related to this parameter.

By starting from the loss gradient equation of a convolutional layer (2.34) and by applying the quantizer to the activation into this last relation, the loss gradient of sample x w.r.t a weight can be expressed as:

$$\begin{aligned}\frac{\partial L_x}{\partial w^k_{m,n}} &= \sum_{i=0}^{H_2-F}\sum_{j=0}^{W_2-F} \delta^k_{x;i,j} Q(\text{ReLU1}(a^{k-1}_{x;i+m,j+n})) \\ &= \sum_{i=0}^{H_2-F}\sum_{j=0}^{W_2-F} \delta^k_{x;i,j} \left(\text{ReLU1}(a^{k-1}_{x;i+m,j+n})) + \varepsilon^{k-1}_{x;i+m,j+n}\right)\end{aligned} \tag{4.11}$$

As the update rule for a convolutional layers parameter is the same as for the parameters of a FC layer, the rule can be written for a given mini-batch as:

$$\begin{aligned}w^k_{m,n} \leftarrow w^k_{m,n} &- \alpha \mathbb{E}_d \left[\sum_{i=0}^{H_2-F}\sum_{j=0}^{W_2-F} \delta^k_{i,j} \text{ReLU1}(a^{k-1}_{i+m,j+n}))\right] \\ &\underbrace{- \alpha \mathbb{E}_d \left[\sum_{i=0}^{H_2-F}\sum_{j=0}^{W_2-F} \delta^k_{i,j} \varepsilon^{k-1}_{i+m,j+n}\right]}_{\text{error term}}\end{aligned} \tag{4.12}$$

Where $\mathbb{E}_d$ is the empirical mean operator on the mini-batch d. Again, there is no guarantee of any kind on the value of the error term appearing in the above expression.

However, the distribution of the quantization error can be controlled by applying dither before quantization. Again, the quantization error of the NSD system is uniformly distributed on a quantization step width centered in 0. Since the back propagated error is independent of the quantization error (*cf.* (2.35) *for proof*), the following assumption can be made:

$$\begin{aligned}\lim_{N\to\infty} \mathbb{E}_d\left[\sum_{i=0}^{H_2-F}\sum_{j=0}^{W_2-F} \delta_{i,j}^k \varepsilon_{i+m,j+n}^{NSD,k-1}\right] &= \mathbb{E}\left[\sum_{i=0}^{H_2-F}\sum_{j=0}^{W_2-F} \delta_{i,j}^k \varepsilon_{i+m,j+n}^{NSD,k-1}\right] \\ &= \sum_{i=0}^{H_2-F}\sum_{j=0}^{W_2-F} \mathbb{E}\left[\delta_{i,j}^k \varepsilon_{i+m,j+n}^{NSD,k-1}\right] \\ &= 0\end{aligned} \tag{4.13}$$

It thus seems again reasonable to assume that the update rule of the parameters constituting a convolutional layer for a sufficiently high mini-batch size N is given by:

$$\boxed{w_{m,n}^k \overset{\simeq}{\leftarrow} w_{m,n}^k - \alpha \mathbb{E}_d\left[\sum_{i=0}^{H_2-F}\sum_{j=0}^{W_2-F} \delta_{i,j}^k \mathrm{ReLU1}(a_{i+m,j+n}^{k-1})\right]} \tag{4.14}$$

This result demonstrates that the *conclusions taken above regarding the optimization of the SGD steps during the gradient descent are generalizable to convolutional layers.*

4.4 Summary

The choice of dither is *motivated by two main arguments.*

The first one, which occurs during forward propagation, *cancels out the effect of quantization on average.* This could be useful for example when *performing several inferences* of the same sample with different dither realizations.

The second argument *concerns the training of networks* with quantized activation. The *quantization error causes the disruption of the step estimation during the SGD* making the optimization suboptimal. It was then hypothesized and demonstrated that dither allows, thanks to the averaging performed on a mini-batch, to *suppress these perturbations by reducing the gradient mismatch.* This should allow more efficient *optimization of the networks and could lead to better performing models.*

Chapter 5

Related works

Recent improvements in the computational capacities of computer systems and studies in the field of deep learning have made it possible to *train networks made up of more and more parameters*. The deeper and deeper networks are now *achieving increasingly remarkable performance*.

Only, this increase in the size of the networks, which is at the origin of the increase in performance, is accompanied by an increase in the computational costs but also in the storage costs of these networks. Although *training and inference are feasible at the server-GPU scale*, the *deployment of deep networks for embedded applications is still very challenging*. Indeed, the deployment of CNN on platforms with limited resources (energy consumption, computing capacity, storage capacity, etc.) requires network compression in order to reduce the cost of inference. It is today a hot topic in the literature which develops rapidly.

Nowadays, there are roughly three different techniques that allow model compression. These include *pruning* (Han, Mao, and Dally [15], Blalock et al. [5]), *efficient architecture design* (Wu et al. [40], Howard et al. [19], Tan and Le [37]) or *quantization* (Wu et al. [41], Yang et al. [42]). The focus for this work is on this last method.

5.1 Quantization of the network

Quantization consists in *compressing* via a quantizer the *parameters* constituting the model and/or its *activation* in order to *alleviate the application of its inference*, by example to use them on mobile devices. Mishra et al. [28] show that reducing the precision of weights and activation reduces the memory footprint, bandwidth and storage while also simplifying the requirements for hardware to efficiently support these operations. This makes it a very good candidate for applications with limited resources or dedicated low power hardware.

Courbariaux et al. [10] introduce a method to train *binarized neural networks* (BNN) using only binary weights and activation. This binary projection allows to drastically reduce memory size and accesses. In addition, most of the *arithmetic operations are replaced by bit-wise operations* which greatly *improves the power efficiency*. Although these hardware-friendly schemes allow boarding on mobile devices, they present one big disadvantage: the *accuracy loss* (Courbariaux, Bengio, and David [9]).

5.1.1 Quantization of the weights

Gong et al. [13] provide a study of different weight quantization models. They show that, thanks to multi-bit vector quantization using k-means, it is possible to reduce the size of the models while minimizing the loss of accuracy.

Later, Li, Zhang, and Liu [26] study a stronger quantization of the weights based on a ternary basis $\{-1, 0, 1\}$. Another approach studied by Courbariaux, Bengio, and David [8] consists in quantifying the weights in a binary way on ± 1. Both studies showed that *weight quantization coupled with an appropriate fine-tuning had no noticeable performance loss*.

A model quantized on the weights will be able to reach almost the *same performances as a full-precision model at the cost of a slower and more constrained training*. This has the advantage that it can be trained on a GPU server before being transferred to an embedded device.

5.1.2 Quantization of the activation

While quantization of weights reduces the size of the model, the *quantization of activation allows the replacement of the expensive inner products*. Indeed, those inner products, at the core of all network computations, can be replaced by logical and bit-counting operations.

In order to realize real-time processing applications on CPUs, Vanhoucke, Senior, and Mao [38] propose an 8-bit quantization of the activation. However, the quantization is applied after the training in order to avoid *problems related to the non-differentiability of the quantized activation*. This method implies a huge loss of performance when applied for stronger quantizations.

To overcome this problem, *methods using continuous approximation to the quantizer function in the backpropagation step* have been studied (BNN [10], XNOR-Net [30], DoReFa-Net [43]). Although these methods show good results for low-precision networks, there is *still a loss of performance* compared to a full-precision model.

Lin and Talathi [27] and Kim et al. [23] investigate the causes of this deterioration in accuracy. They showed that there exist a *gradient mismatch* between the

approximation of the *activation during backpropagation and the true value of the activation*. This has the effect of *impacting the stability of the SGD during training*. The SGD, being suboptimal, *impacts the loss minimization* and thus results in a *poorer model with reduced performance*. Choi et al. [7] investigate a more vectorial approach to estimate the gradient mismatch.

Ando et al. [2] study an elaborated dither algorithm of error diffusion within the activation maps in order to limit the total quantization error. This technique allows to slightly improve the performance of these models while requiring very few additional hardware resources.

Finally, our work proposes a *quantization error-oriented approach to explain the sub-optimality of the network training*. The provided method to solve this problem consists in *adding dither* during the quantization of the activation in order to *minimize the gradient mismatch* and thus to *optimize the estimation of the steps during the SGD*. This allows for a more efficiently trained model and requires thus no hardware modifications for inference while proposing a modification of the activation function during forward propagation without focusing on the estimators.

Chapter 6

Methodology

At this point, the reader should have a good overall idea of how CNN work as well how the training is operated using whether or not quantized activation. The quantization background Chapter further allowed to introduce different systems implementing dither as well as an analysis of the properties resulting from its application. The Chapter dedicated to the motivations then allowed the reader to understand the interest of adding dither during inference but also its impact during training. The Chapter on related works finally allows the latter to determine in the state-of-the-art the scope of our study and how it differs from what has already been done.

The objective of this Chapter is to *present the general methodology* applied to this study. In order to do that, the *research on the dither strategy* to be used will first be introduced. Then, the *experimental setup* used to perform the experiments will be presented. Finally, this Chapter will end with a section dedicated to the *description of the experiments performed and their purpose.*

6.1 Dither strategy

Gradient mismatch is one of the major challenges in training networks with quantized activation. *Many studies investigate different estimators* to best approximate the activation function during backpropagation via a continuous differentiable function. This is to *minimize the gradient mismatch that causes suboptimal training.* As a result of this, the compressed model fails to achieve the performance of a full-precision model.

Our study will work in the opposite direction. Instead of trying to adapt the activation function estimator during backward propagation, we try to *adapt the activation function during forward propagation to match* with the function usually used in non-quantized activation. We have indeed been able to verify the

beneficial theoretical impact of the dither application thanks to our analysis on the estimation of gradients which involves the quantization error. We have in fact demonstrated that the addition of dither could lead to a cancellation of this error which would allow correcting the SGD steps. Moreover, it has been shown that performing several generations of dither for a sample cancels on average the impact of quantization.

This section introduces the different opportunities related to the introduction of this dither as well as the strategy chosen to perform experiments.

6.1.1 Exploration of the dither hyper-parameters

The introduction of a dither in the ReLU activation quantizer raises numerous questions about the way to perform it:

- Where should it be applied? Only on some layers or everywhere?
- Should it be full random or should a common dither be applied for all activation maps?
- How wide should the dither be distributed ?
- Should it be shifted or should its distribution be centered in 0?
- Is there a better type of quantizer than another? Mid-riser or mid-tread?
- Should dither be applied to values above a certain threshold?
- Is it more interesting to use a non-subtractive or subtractive dither system ?

Due to the large number of possibilities to be tested on these issues, the studies were carried out by varying each of these hyper-parameters independently through a performance study on the different models. Those issues were therefore addressed in an *exhaustive trial-and-error research comparing classification performance from hundreds of networks* that have been trained using different strategies.

Those studies of the dither-related hyper-parameters are available in Appendix A for the interested reader but will not be discussed in this Chapter since they do not present very interesting conclusions. Only the chosen dither strategy which presents the best classification performances is presented here.

6.1.2 Retained dither strategy

The chosen strategy is to use *non-subtractive dither* with a *mid-tread quantizer*. The dither, *distributed over a quantization step width centered in 0*, is applied after the application of the ReLU function.

Mathematically, it is defined as such:

$$Q(\text{ReLU1}(x) + \nu) \text{ with } \nu \sim \mathcal{U}\left(\left[\frac{-\Delta}{2}, \frac{\Delta}{2}\right]\right)$$

Where the input of the layer is denoted by x.

The continuous estimator (STE) used for the training is the *clipped ReLU* defined by (3.12).

The dither is applied regardless of the value of x i.e. there is *no minimum or maximum threshold* before applying it. Further, the addition of dither is implemented on *all the layers with a new realization every single time*. In other words, a new dither is randomly generated each time an activation map is being processed within the network.

Finally, in order to observe more explicitly the impact of the addition of dither, our focus is restricted to *binary quantized activation*. Indeed, the accuracy gap between a 1-bit quantized activation and a full-precision model is much larger than the one observed using a 2-bit quantization or more. This will therefore allow to better observe the potential performance improvements. Since there is no evidence to the contrary, the conclusions obtained for 1-bit quantizations will be assumed to generalize to weaker quantizations.

6.2 Experimental setup

In the following experiments, the performance of CNN will be measured on an image classification task with the *aim of viewing the impact of dither*. The objective of this section is to describe the experimental setup for the experiments that will be performed.

6.2.1 Dataset

The dataset on which this work is based is the *CIFAR-10* which is a *labeled* dataset introduced by Krizhevsky and Hinton [24]. It consists of a set of 60000 images uniformly distributed in 10 different classes completely mutually exclusive. This dataset has become a *classic for performance benchmarking* on CNN.

The dataset is divided into three different subsets: the training set contains 45000 images, the validation set is composed of 5000 images and the testing set is composed of 10000 images. Note that the classes are uniformly distributed over these three different subsets. The samples constituting the dataset are 3 channels (RGB) images of 32×32 pixels. A preview of some of the images in the set is available in Figure 6.1.

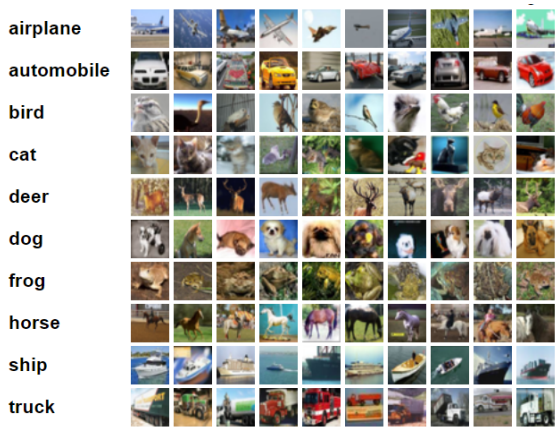


Figure 6.1: CIFAR-10 dataset. Ten random images of each of the 10 classes are shown here as an illustration.

This dataset was chosen for two main reasons. First, it is a *fairly simple set* on which simple networks can be trained fairly quickly while providing satisfying results. Second, this dataset, *widely used in the literature*, is very representative of the network performance. The conclusions obtained on this dataset are very often generalizable to other tasks.

6.2.2 Metrics

The study of the classification performances is mainly focused on one metric in particular: the *accuracy*. Nevertheless, the *calibration* of the networks has also been studied in this work but will only be presented in an appendix for the interested reader as it is less relevant.

A. Accuracy

The accuracy reports the classification performance of a network by indicating the *proportion of correct classifications*. This metric is defined for a dataset D as the ratio between the number of correct predictions and the total number classifications N. It is defined as:

$$\text{Accuracy}_D := \frac{1}{N} \sum_{x \in D} 1(\hat{y}_x = y_x) \tag{6.1}$$

Where $\hat{y}_x$ is the prediction class of the network computed on sample x and y_x is its ground truth belonging to the labeled dataset.

This metric can be calculated for the training, validation or testing set and is often expressed in percentages. The goal is obviously to obtain models with the

greatest possible accuracy.

Gap An important metric, derived from accuracy, that will be used extensively in this study is the *gap between the performances of a compressed and a non-compressed one*. It is defined by the difference in accuracy between a quantized model, what is actually obtained, and a full-precision model, upper limit reachable in performance. This difference is also expressed in percentages.

B. Calibration

A second study is carried out on the calibration performance of a model, i.e. *the capacity of a network to estimate its probability of correct prediction on a class.* This study is not presented directly in this work because it is believed to be not very relevant. Indeed, there exists calibration techniques of networks which can be carried out once the training is done. Guo et al. [14] study those different methods. The interested reader can nevertheless find our study of the calibration in Appendix B.

6.2.3 Architecture

The architecture used for this work is a version of *VGG-11_bn* adapted to the CIFAR-10 dataset. The description of this archetype has been done in the background Chapter on neural networks in subsection 2.3.2. Basically, the adaptation consists in reducing the number of FC layers and adjusting the number of kernels per convolution layer. A summary Table with the different layers is available below.

Beginning				Following			
Layers	**# kernels**	**Output shape (length × width × bands)**	**# parameters**	**Layers**	**# kernels**	**Output shape (length × width × bands)**	**# parameters**
Input		$32 \times 32 \times 3$		Convolution 5	512	$4 \times 4 \times 512$	1179648
Convolution 1	64	$32 \times 32 \times 64$	1728	Batch norm 5	512	$4 \times 4 \times 512$	1024
Batch norm 1	64	$32 \times 32 \times 64$	128	Q(ReLU1)		$4 \times 4 \times 512$	
Q(ReLU1)		$32 \times 32 \times 64$		Convolution 6	512	$4 \times 4 \times 512$	2359296
Max pool 1		$16 \times 16 \times 64$		Batch norm 6	512	$4 \times 4 \times 512$	1024
Convolution 2	128	$16 \times 16 \times 128$	73728	Q(ReLU1)		$4 \times 4 \times 512$	
Batch norm 2	128	$16 \times 16 \times 128$	256	Max pool 4		$2 \times 2 \times 512$	
Q(ReLU1)		$16 \times 16 \times 128$		Convolution 7	512	$2 \times 2 \times 512$	2359296
Max pool 2		$8 \times 8 \times 128$		Batch norm 7	512	$2 \times 2 \times 512$	1024
Convolution 3	256	$8 \times 8 \times 256$	294912	Q(ReLU1)		$2 \times 2 \times 512$	
Batch norm 3	256	$8 \times 8 \times 256$	512	Convolution 8	512	$2 \times 2 \times 512$	2359296
Q(ReLU1)		$8 \times 8 \times 256$		Batch norm 8	512	$2 \times 2 \times 512$	1024
Convolution 4	256	$8 \times 8 \times 256$	589824	Q(ReLU1)		$2 \times 2 \times 512$	
Batch norm 4	256	$8 \times 8 \times 256$	512	Max pool 5		$1 \times 1 \times 512$	
Q(ReLU1)		$8 \times 8 \times 256$		FC 1	10	10	5130
Max pool 3		$4 \times 4 \times 256$		Softmax		10	

Table 6.1: Detailed VGG-11_bn architecture used for this work.

This architecture was chosen for two main reasons. First, training this kind of model is simple, fast and does not require meticulous fine-tuning of the learning hyper-parameters. Second, it is consistently observed in the quantization literature that this architecture correctly generalizes the performance from other network architecture.

6.2.4 Training

The models are *initialized accordingly to the technique described in subsection 2.4.2A.* using a determined seed for the random generation. *Minimization of the loss function is achieved through an SGD* for which the values of the learning hyper-parameters are provided in the Table below.

Learning parameters		#epochs	200
		learning rate	$\alpha = \{10^{-1}, 10^{-2}\}$
Optimizer	SGD	mini-batch size	$N = 128$
		momentum	$\mu = 0.9$
		weight decay factor	$\lambda = \{10^{-3}, 10^{-4}, 10^{-5}\}$
Learning rate scheduler	multi-step	decay factor	$\eta = 0.2$
		milestones	$[100, 150, 175]$

Table 6.2: Learning hyper-parameters used for this work.

A *grid-search is performed on the learning rate and the weight decay* within the values enclosed into brackets to select only the best performing models. Finally, in order to avoid drawing hasty conclusions from the observation of outliers, *each*

experiment is conducted three times by using different random generation seeds. Indeed, this allows obtaining three different initializations, resulting into three different models.

6.3 Conducted experiments

The quantization of the activation leads to a quantization error that affects the step estimate during the SGD. It was then illustrated how this could have a negative impact on the training of a model and therefore make the training suboptimal. The addition of dither was then motivated by the fact that it allows a control on the quantization error distribution which on average help in canceling the error term from the SGD step estimation. It was subsequently hypothesized that it therefore enables a recalibration of the steps to be performed by erasing the noisy aspect from the gradients estimation.

This optimization of the SGD should allow to potentially reach a better minimum of the loss and thus converge towards a *more efficient model*. The objective of the experiments will be to *verify empirically if this increase is true*.

Further, it was also shown that the introduction of dither during quantization allowed, for several realizations, to erase the impact of quantization. A second set of experiments will be carried out to *study the performance obtained when several inferences* for the same sample with different generations of dither are performed.

6.3.1 Training using dither

The main objective of this first empirical study will be to verify if *dither increases the performance of the models*. This will be done on the basis of a comparison between a non-compressed model, i.e. the maximum performance achievable in practice, a classically quantized model, to have a starting point, and a model trained with dither, to observe the impact.

6.3.2 Multiple inferences

The second part of the experiments focuses on a set-based approach taking advantage of the stochastic inference offered by dither. Indeed, the classification is dependent on the generation of dither that has been performed. In other words, the decision of the chosen class varies from one realization to another. It has also been shown in the motivations Chapter that several realizations of dither for the same sample can cancel the quantization on average. This motivates the idea that *taking into account these different decisions could make classification more efficient.*

The study of the performance from this type of method will first be carried out by *varying the number of inferences*.

Then, a *comparison in performance and implementation with other set-based approaches* found in the literature will be made. One is *test-time augmentation*, which takes advantage of predictions made on augmented data during inference, and the other is the *ensemble method*, which uses a set of different networks for inference. These two methods are widely used in many applications and can even be combined together.

A. Test-time augmentation

Test-time augmentation (TTA) is a *data augmentation technique applied to the testing set* which consists in performing several inferences of the same transformed/augmented image and making the classification decision by comparing the different predictions obtained.

In practice, the augmentations performed must be chosen in order to maximize the chances of correctly classifying the samples. This strategic choice of augmentations is usually subject to a careful research. Indeed, the transformations to be carried out are found by fine-tuning the hyper-parameters in order to maximize the classification performances.

B. Ensemble method

This method is the most straight forward of the set-based methods. It consists in *using a multitude of different networks*, of the same architecture but from different initializations, to perform the inference.

C. Decision rules

A crucial element that has not yet been addressed is the *decision rule*, i.e. *how to make the classification choice based on the different results*.

Let n be the number of inferences realized for a single sample. Two decision strategies are available:

1. The choice of classification is made by *taking the class that is the most chosen within the different inferences*. This can be seen as a form of democratic voting where the most represented class is chosen in the end. The prediction confidence of the chosen class will then be defined by the number of times this class has been chosen divided by the number of inferences.

2. The second method consists in *averaging all the confidence output* by the network for each of the inferences. The class chosen will then be the one

with the highest average prediction probability and the associated prediction confidence will be this average value.

Many experiments have been carried out to identify which of these two methods is the most efficient. In practice, there is a similarity in classification performance for high n.

However, the first technique presented regularly suffers from indecision when the values of n are low. For example if two different classes are chosen when $n = 2$, it is impossible to know which one to take. For this reason, *the second method will be used.*

Chapter 7

Results and discussions

The objective of this Chapter is to *present and discuss the results of the experiments* carried out as described in previous Chapter at section 6.3.

The first part of the experiments will concern the improvement in performances through the training. The second part is dedicated to multiple inferences with dither.

7.1 Training using dither

The objective of this experiment is to *verify if the addition of dither during the quantization of the activation allows optimizing the training.* This will be checked by comparing the gap, i.e. the difference in accuracy between compressed and non-compressed networks, and this for 1-bit quantized using dither or not. The mentioned performances will be the one measured on the testing set.

In order to generalize the study more broadly, the networks will be trained on several portions of the training set. Indeed, the CIFAR-10 training set contains 45000 images but the number of experiments will be multiplied by working on the following portions: 0.1, 0.2, 0.4, 0.8 and 1.0 composed respectively by 4500, 9000, 18000, 36000 and 45000 samples.

The training of these networks has been carried out according to what has been described in the subsection 6.2.4 dedicated to this. For each model a grid search is performed to select only the best model. In order to keep the number of parameter updates constant for each experiment, an adaptation to the number of epochs has been performed according to the following formula:

$$\begin{aligned} \text{new \#epochs} &= \frac{\text{\#epochs}}{\text{training set portion}} \\ &= \frac{200}{\text{training set portion}} \end{aligned} \tag{7.1}$$

Finally, in order to make sure not to work with an outlier model, each experiment is performed with three different initializations to obtain three different models.

7.1.1 Full-precision

The training of full-precision networks, i.e. without any quantization, *reveals the maximum practical limit in classification performance*. Indeed, in theory, a perfectly compressed model should ideally reach the same performance as a non-compressed model.

The average accuracy obtained for each portion of the training set is available in Table 7.1. *The performances reached for the complete training set reach* 91.35% *on the testing set*. Which is a slightly lower result than what is observed in the state-of-the-art. This is probably due to a slightly suboptimal training. Indeed, we had to restrict the training to a limited number of epochs and parameters on the grid search so that it is feasible at our scale based on our computing capacity. However, this should not be a major concern since the training procedure is kept constant for all experiments. The conclusions obtained should therefore be generalizable.

7.1.2 Quantized activation

The training of the 1-bit quantized activation networks, reveals the *baseline in classification performance*. It is considered as the starting point.

Again, the obtained performances on the testing set are available in Table 7.1. On average, 86.81% *of the classifications are correct*, which is very slightly below the performances found in the literature. Probably for the same reasons as above...

7.1.3 Dithered quantized activation

The training of quantized dithered networks is slightly different from the previous networks and deserves some explanations. Indeed, the transition from the training phase, carried out with dither, to the inference phase, carried out without dither so that the model is deterministic, implies a change in the distribution of activation. This modification of the distribution from the feature maps at the output of the activation layer will result in a slight loss of performance due to the batch normalization layers.

As seen in subsection 2.3.1B. dedicated to BN layers, they are composed of two learnable parameters which are adapted according to the distributions of the features maps. A change from a dithered to a non-dithered model causes a slight modification of these distributions which disrupts the normalization operation. It is therefore necessary to perform a small fine-tuning of these parameters in order to adapt them again to the new distributions.

This is done during a second very short training session during which all learnable parameters are frozen except those from the BN layers. This process will be referred to as the fine-tuning of the BN layers.

The performances obtained on the testing set for the trained with dither networks followed by a fine-tuning of the BN, to adapt the inference to no dither, are available in Table 7.1. *The performances obtained indicate for training on the complete set an average accuracy of* 89.21%.

7.1.4 Comparing performances

The Table below shows the different results obtained during the experiments described above. The accuracy is obtained by averaging the three different models while the gap represents the loss of performance observed between the mentioned networks and the full precision (FP) networks. Baseline stands for 1-bit quantized networks while dithered stands for the quantized networks trained with dither and fine-tuned on the BN for an inference without dither. All the data below are expressed in percentages.

Metrics (%)	**Models**	**Portions of the training set**				
		0.1	**0.2**	**0.4**	**0.8**	**1.0**
Avg. accuracy	FP	76.82	83.09	87.32	90.01	91.35
	Baseline	72.59	78.70	83.13	86.27	86.81
	Dithered	**74.19**	**79.19**	**84.70**	**88.43**	**89.21**
Gap	Baseline	-4.23	-4.39	-4.19	-3.74	-4.54
	Dithered	**-2.63**	**-3.9**	**-2.62**	**-1.58**	**-2.14**

Table 7.1: Summary of the results obtained on the testing set by varying the portion of the training set.

The following Figure provides a more visual representation of these results.

The performance losses for a simply quantized model are on average equal to 4.22% for the different portions of the training set against 2.57% for a dithered training. *A global improvement of almost* 2% *is thus observed reaching* 2.4% *for the complete training set.*

The choice of a mini-batch of size 128 may seem weak to validate the hypothesis that an empirical mean can be replaced by an expectation[1]. However, it is necessary to mention that 352 mini-batches are performed per epoch, which represents 70400 averages performed on 128 images. Further, it is not on a single mini-batch for a

[1]As it is done in the demonstration of subsection 4.3

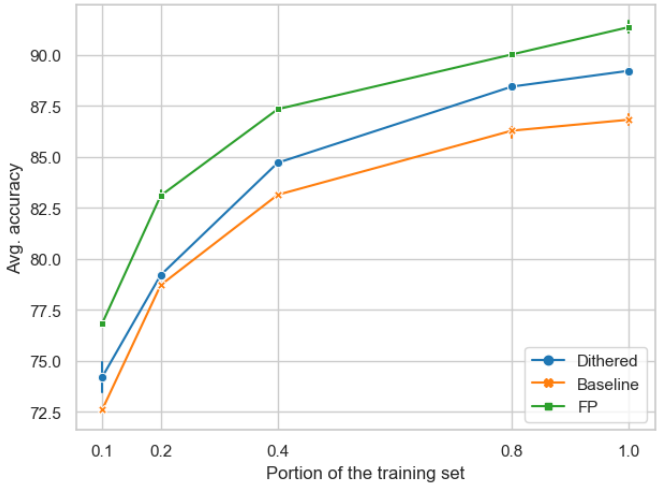


Figure 7.1: Plot from the evolution of the accuracy for various training set portions. Vertical bars show the 68% confidence interval.

single parameter that one must look at the averaging process but rather through the global averaging process. Indeed, the optimization of a network consists in an optimization of each of its parameters. The global averaging process can also be seen through those millions of different parameters.

Finally, this recurrent improvement in accuracy allows claiming that networks obtained by dithered training offer a better minimization of the loss than simply quantized training. The assumptions made in the motivations Chapter about optimizing the SGD by better estimate the step thus seems to be empirically confirmed. More precisely, the addition of dither during the training process allows to reduce the impact of the error term, which has for effect to improve the estimation of the steps performed to minimize the loss. In other words, *dither allows minimizing the gradient mismatch at the origin of a suboptimal SGD optimization.*

7.1.5 Other experiments

A range of other experiments were carried out during this research. The interested reader can find the following elements in some sections of the appendix.

- A study on the calibration of the networks completely similar to the experiments carried out can be found in Appendix B.
- All the experiments presented here were performed using mini-batches of size $N = 128$. A further study was carried out for different mini-batch sizes in order to check the impact on the average of the SGD steps. The results and discussions of this study can be found in Appendix C.

- Different model transfers have been performed in order to observe the impact of dither and fine-tuning of the BN layers. One example is the model transfer of a full precision to a quantized activation or a quantized dithered activation. The results and discussions of this study can be found in Appendix D.

7.2 Multiple inferences

The study carried out for this part of the experiments consists in *observing the performance obtained when several inferences* from different dither generations are made. As a reminder, this set-based approach is motivated by the fact that applying several dither realizations to the same sample allows on average to cancel the impact of quantization. Several inferences could thus allow the network to refine the classification more precisely.

First, a study of *accuracy as a function from the number of inferences* will be performed to observe the impact of dither during forward propagation. Then, in order to position this method among other set-based approaches, i.e. test-time augmentation and ensemble method, a *comparison in performance and implementation* will be done.

The following experiments are again performed on a 1-bit quantization of the activation. For sake of consistency, the networks used for the following experiments are the three models trained with dither from the previous section. Only, since the inference is performed with dither, the fine-tuning of the BN layers has not been performed. Finally, the decision rule used for the choice of the class is described in subsection 6.3.2C. dedicated to this purpose.

7.2.1 Influence from the number of realizations

In order to observe the impact from the number of realizations per sample, the performances are measured on the testing set by varying the amount of dither generations n to the following values: 1, 5, 10, 20, 50 and 100.

The average accuracy obtained from the three models for various numbers of realizations are available in the Table below.

Number of realizations	**1**	**5**	**10**	**20**	**50**	**100**
Avg. accuracy (%)	86.70	89.96	90.34	90.61	90.67	90.68

Table 7.2: Results obtained on the testing set by varying the number of realizations.

It is found that a single dither realization provides almost the same results as a simply quantized model. However, only *5 dither realizations per sample improve the performance by more than* 3%. A hundred dither realizations nearly reach the performance obtained by a non-compressed model.

In order to better illustrate the performance trend of this Table, the evolution of the accuracy has been plotted on the following Figure.

It is clear to see that *increasing the number of realizations improves the accuracy.* A simple visual check of this graph allows noticing the appearance of a horizontal

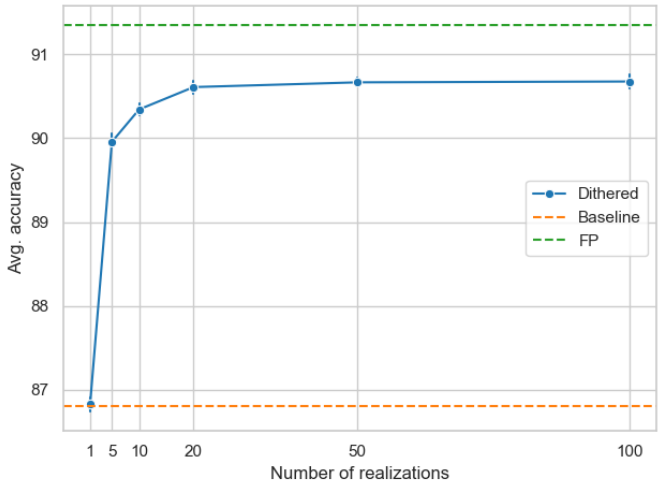


Figure 7.2: Evolution of the averaged accuracy obtained by varying the amount of dither realizations. The vertical bars indicate the confidence interval at 68%.

asymptote. This last one limits the accuracy of this dithered approach rather quickly (around 20 realizations).

On the graph are represented, the average accuracy observed for full precision (FP) networks and for simply quantized networks (baseline). Since those data are the result of a single inference, their comparison to those obtained from multiple inferences is not really relevant. Their only purpose is to serve as a benchmark to situate the reader.

According to [10], using 1-bit quantized networks (activations and weights) enables a drastic reduction in energy consumption (i.e., more than 32 times[2]) compared to full-precision networks. A study to find the pareto optimum by varying the number of realizations and the consumed energy per sample should permit to obtain in fine, for a certain number of realizations, an interesting compromise between accuracy and energy consumption.

7.2.2 Comparing set-based approaches

It has therefore been shown that the dithered approach involving several generations can improve classification performance. The following experiments will aim at comparing this dithered method with other approaches in order to position it with respect to what is done in the literature.

The study of these different approaches will be performed by *studying the*

[2]These data are extremely difficult to assess as they are extremely situation dependent.

accuracy[3] *on the testing set for a number of inferences n varying from 1 to 20.* The following sections aim to present very briefly how these measurements have been carried out.

A. Dithered approach

The procedure for this method is the same as the one described above.

B. Test-time augmentation

The choice of the transformations to be applied is a tedious optimization task in itself, especially as it depends on the number of realizations n.

In order not to complicate the task unnecessarily, the same random transformation as the one used during the training is chosen by taking care to reduce the size of the random cropping to 2 pixels maximum instead of 4. The performances were obtained by applying n random transformations for values ranging from 1 to 20 always ensuring that the first inference is the non-augmented image.

The models used to conduct this experiment are the three simply quantized networks used in the previous section under the name baseline.

C. Ensemble method

In order to compare the performance of this approach with the others, 20 different networks from different initialization were trained with 1-bit quantized activation using the same method as for baseline networks. Inferences were therefore made by varying the number of models from 1 to 20.

D. Comparison

The following graph presents the average accuracy observed for various n of the different set-based approaches.

This plot shows that from a performance point of view, the *dithered method* stands up very well against other approaches already studied in the literature. Its accuracy is particularly high for a low number of realizations, which can be a major advantage on an embedded device with limited resources. However, from an application point of view, this technique requires the ability of the device to *generate randomness*. Moreover, the inclusion of the dither before the quantization also implies *extra additions during the inference.*

TTA approach *does not perform very well from the accuracy point of view.* However, one should be careful to remain critical of these observations, remembering

[3] A similar study available in Appendix B has been performed on the calibration.

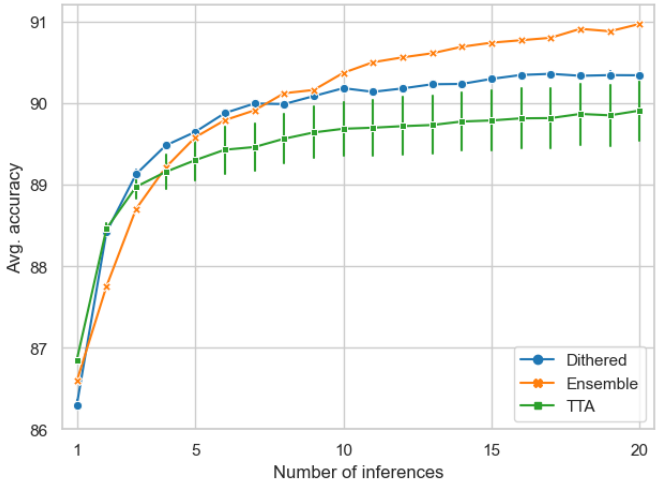


Figure 7.3: Evolution of the average accuracy for varying number of realizations using different set-based approaches. The vertical bars are the confidence interval at 68%.

that the transformations performed are randoms and not optimized. Indeed, the study of the augmentations to be applied has not been carried out to facilitate the task. From a concrete point of view on an embedded device, it is necessary to mention that *augmentation requires pre-processing of the images which implies additional operations*. The operating costs of these transformations obviously vary according to their complexity.

The *ensemble method* performs best when the number of networks n starts to be large enough but scores poorly before. It is also worth noting that this technique is by far the most restrictive for mobile applications. In addition to the fact that *training time increases linearly with the number of networks*, the *management of different models presents a whole bunch of problems*. These include *storage issue* of these networks and the *problems associated with the transition* between these different models.

Finally, the following Table provides the reader with an overview of the advantages/disadvantages of what has been just presented.

Approaches	Advantages	Disavantages
Dither	Good acc. when n is small Use of only one network No pre-processing of the images	Requires dither generation: - Need of random generation - Additional additions during inference
TTA	Good acc. when n is really small (Usually) No random generation Use of only one network	Weak acc. when n is large Requires pre-processing of the images: - Requires meticulous research of augmentations - Chosen transformations are highly situation dependent - Augmented copies requires processing
Ensemble	Good acc. when n is large No random generation No pre-processing of the inputs	Weak acc. when n is small Use of multiple networks: - Increase linearly the training time - Increase linearly the storage place on the device - Whole bunch of problems for the loading of the networks

Table 7.3: Summary of the different set-based approaches listing the advantages and disadvantages.

Chapter 8

Conclusion

Neural networks are getting more and more powerful at the cost of more and more learnable parameters. This makes inference complex for an application on embedded devices since their computational resources are limited. Therefore, there is a need to compress these networks in order to facilitate the inference process.

This can be achieved through quantization of activation. Only, this compression implies a loss in classification performance. The latter is due to a suboptimal training which can be explained by the gradient mismatch, i.e. the difference between the activation in the forward pass (quantized) and its estimator (non-quantized) in the backward pass. This causes a mismatch in the estimation of the steps to be taken in the SGD which prevents the network from converging to an optimized loss minimum.

A study of the impact of quantization during training allowed the analysis of this gradient mismatch. Indeed, it was shown via a quantization error-oriented approach that the distortion, introduced by quantization, affected the gradient descent by adding an undesired bias term to the performed steps. In order to solve this problem, it was demonstrated that addition of dither to the quantizer would help to optimize the gradient descent by minimizing this bias. This motivated the idea of training networks using dither.

It was also shown that several inferences using different generations of dither could, on average, negate the impact of quantization on activation. This motivates the set-based approach, which consists of making several inferences and performing the classification on the basis of the different obtained results. It might thus allow sharpening the classification rule and consequently increase the accuracy.

In order to verify these two assumptions, experiments were performed on CIFAR-10 using VGG-11 networks trained with the retained optimized dither strategy.

The experiments aiming at verifying the optimization of the training thanks to

dither were performed by comparing the accuracy of full-precision models, simply quantized models and dithered models. It has been demonstrated that an increase of more than 2% in accuracy (thus dividing the gap by half) is obtained thanks to the application of this particular training. This could confirm the assumptions about the optimization of the network generalization by reducing the gradient mismatch thanks to dither.

The experiments aiming at studying the performances of the set-based dithered approach has mainly shown two contributions.

First, the impact of the number of realizations, i.e. of multiple inferences, for a single sample could be determined. This allowed to show that a 3% improvement in performances was obtainable in only 5 realizations while the evolution of the performances reached saturation after a dozen realizations, which makes it possible to achieve almost the same performances as a full-precision model.

Second, the comparative approach has allowed us to qualitatively and quantitatively situate this dithered method among other set-based approaches. Indeed, this method is rather well defended from a classification performance perspective but also from an application point of view for an embedding on mobile devices.

Further works

For the moment, the conclusions have only been obtained for a single type of network applied to a single dataset. Admittedly, this architecture/dataset combination seems to generalize quite well in the literature. It is still interesting to check if other combinations also verify the same accuracy gain in order to generalize more confidently the obtained conclusions. It is also necessary to point out that no tests on dedicated hardware devices have been performed. This could have presented interesting results regarding the trade-off between energy consumption and accuracy.

Moreover, a comparison with the state-of-the-art from a performance point of view could not be made because the performances obtained for the full precision or simply quantized networks could never reach those observed in the literature. This is probably due to slightly incomplete training. Greater research capabilities could allow for more in-depth study with a more complete training. This would allow us to situate our contributions in relation to what has already been presented in the literature.

Finally, no empirical measurement of the quantization error within the network has been performed. This could have been interesting to identify more precisely the impact of dither. Further, Wannamaker [39] states in his paper that it is more

interesting in image processing to use a third order dither[1], i.e. a semicircular distribution, rather than a first order dither, i.e. the one studied here with a rectangular distribution. A complementary study using maybe second or third order dither could have provided some interesting results.

[1]It should be noted that this study was voluntarily discarded in order to limit the search space for hyper-parameters which was already quite large.

Bibliography

[1] *A Gentle Introduction to Torch.Autograd — PyTorch Tutorials 1.9.0+cu102 Documentation*. URL: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html.

[2] Kota Ando et al. "Dither NN: Hardware/Algorithm Co-Design for Accurate Quantized Neural Networks". In: *IEICE Trans. Inf. & Syst.* E102.D.12 (Dec. 1, 2019), pp. 2341–2353. ISSN: 0916-8532, 1745-1361. DOI: 10.1587/transinf.2019PAP0009.

[3] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. Aug. 15, 2013. arXiv: 1308.3432 [cs]. URL: http://arxiv.org/abs/1308.3432.

[4] W. R. Bennett. "Spectra of Quantized Signals". In: *The Bell System Technical Journal* 27.3 (July 1948), pp. 446–472. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb01340.x.

[5] Davis Blalock et al. *What Is the State of Neural Network Pruning?* Mar. 6, 2020. arXiv: 2003.03033 [cs, stat]. URL: http://arxiv.org/abs/2003.03033.

[6] Zhaowei Cai et al. *Deep Learning with Low Precision by Half-Wave Gaussian Quantization*. Feb. 3, 2017. arXiv: 1702.00953 [cs]. URL: http://arxiv.org/abs/1702.00953.

[7] Jungwook Choi et al. *PACT: Parameterized Clipping Activation for Quantized Neural Networks*. July 17, 2018. arXiv: 1805.06085 [cs]. URL: http://arxiv.org/abs/1805.06085.

[8] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. *BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations*. Apr. 18, 2016. arXiv: 1511.00363 [cs]. URL: http://arxiv.org/abs/1511.00363.

[9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. *Training Deep Neural Networks with Low Precision Multiplications*. Sept. 22, 2015. arXiv: 1412.7024 [cs]. URL: http://arxiv.org/abs/1412.7024.

[10] Matthieu Courbariaux et al. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. Mar. 17, 2016. arXiv: 1602.02830 [cs]. URL: http://arxiv.org/abs/1602.02830.

[11] Vincent Dumoulin and Francesco Visin. *A Guide to Convolution Arithmetic for Deep Learning*. Jan. 11, 2018. arXiv: 1603.07285 [cs, stat]. URL: http://arxiv.org/abs/1603.07285.

[12] 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*. Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. JMLR Workshop and Conference Proceedings, Mar. 31, 2010, pp. 249–256.

[13] Yunchao Gong et al. *Compressing Deep Convolutional Networks Using Vector Quantization*. Dec. 18, 2014. arXiv: 1412.6115 [cs]. URL: http://arxiv.org/abs/1412.6115.

[14] Chuan Guo et al. *On Calibration of Modern Neural Networks*. Aug. 3, 2017. arXiv: 1706.04599 [cs]. URL: http://arxiv.org/abs/1706.04599.

[15] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. Feb. 15, 2016. arXiv: 1510.00149 [cs]. URL: http://arxiv.org/abs/1510.00149.

[16] *Hide The Pain Harold*. Know Your Meme. URL: https://knowyourmeme.com/memes/hide-the-pain-harold.

[17] Geoffrey Hinton. *Neural Networks for Machine Learning, Coursera*. 2012.

[18] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer Feedforward Networks Are Universal Approximators". In: *Neural Networks* 2.5 (Jan. 1, 1989), pp. 359–366. ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8.

[19] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. Apr. 16, 2017. arXiv: 1704.04861 [cs]. URL: http://arxiv.org/abs/1704.04861.

[20] *Introduction to Gradients and Automatic Differentiation*. TensorFlow. URL: https://www.tensorflow.org/guide/autodiff.

[21] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. Mar. 2, 2015. arXiv: 1502.03167 [cs]. URL: http://arxiv.org/abs/1502.03167.

[22] Jefkine. *Backpropagation In Convolutional Neural Networks*. DeepGrid. URL: https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/.

[23] Hyungjun Kim et al. *BinaryDuo: Reducing Gradient Mismatch in Binary Activation Network by Coupling Binary Activations*. Feb. 16, 2020. arXiv: 2002.06517 [cs, stat]. URL: http://arxiv.org/abs/2002.06517.

[24] Alex Krizhevsky and Geoffrey Hinton. *Learning Multiple Layers of Features from Tiny Images*. University of Toronto, 2009, p. 60.

[25] Anders Krogh and John Hertz. "A Simple Weight Decay Can Improve Generalization". In: *Advances in Neural Information Processing Systems*. Vol. 4. Morgan-Kaufmann, 1992.

[26] Fengfu Li, Bo Zhang, and Bin Liu. *Ternary Weight Networks*. Nov. 18, 2016. arXiv: 1605.04711 [cs]. URL: http://arxiv.org/abs/1605.04711.

[27] Darryl D. Lin and Sachin S. Talathi. *Overcoming Challenges in Fixed Point Training of Deep Convolutional Networks*. July 8, 2016. arXiv: 1607.02241 [cs]. URL: http://arxiv.org/abs/1607.02241.

[28] Asit Mishra et al. *WRPN: Wide Reduced-Precision Networks*. Sept. 4, 2017. arXiv: 1709.01134 [cs]. URL: http://arxiv.org/abs/1709.01134.

[29] *Papers with Code - ImageNet Benchmark (Image Classification)*. URL: https://paperswithcode.com/sota/image-classification-on-imagenet.

[30] Mohammad Rastegari et al. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*. Aug. 2, 2016. arXiv: 1603.05279 [cs]. URL: http://arxiv.org/abs/1603.05279.

[31] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 1939-1471, 0033-295X. DOI: 10.1037/h0042519.

[32] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-Propagating Errors". In: *Nature* 323.6088 (Oct. 1, 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0.

[33] L. Schuchman. "Dither Signals and Their Effect on Quantization Noise". In: *IEEE Transactions on Communication Technology* 12.4 (Dec. 1964), pp. 162–165. ISSN: 2162-2175. DOI: 10.1109/TCOM.1964.1088973.

[34] Connor Shorten and Taghi M. Khoshgoftaar. "A Survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data* 6.1 (July 6, 2019), p. 60. ISSN: 2196-1115. DOI: 10.1186/s40537-019-0197-0.

[35] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. Version 6. Apr. 10, 2015. arXiv: 1409.1556 [cs]. URL: http://arxiv.org/abs/1409.1556.

[36] Ilya Sutskever et al. "On the Importance of Initialization and Momentum in Deep Learning". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 17–19, 2013, pp. 1139–1147.

[37] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. Sept. 11, 2020. arXiv: 1905.11946 [cs, stat]. URL: http://arxiv.org/abs/1905.11946.

[38] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. "Improving the Speed of Neural Networks on CPUs". In: *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*. 2011.

[39] Robert Alexander Wannamaker. "The Theory of Dithered Quantization : Ph.D. Thesis Presented to the University of Waterloo". In: (1997), p. 223.

[40] Bichen Wu et al. *Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions*. Dec. 3, 2017. arXiv: 1711.08141 [cs]. URL: http://arxiv.org/abs/1711.08141.

[41] Shuang Wu et al. *Training and Inference with Integers in Deep Neural Networks*. Feb. 13, 2018. arXiv: 1802.04680 [cs]. URL: http://arxiv.org/abs/1802.04680.

[42] Jiwei Yang et al. *Quantization Networks*. Nov. 27, 2019. arXiv: 1911.09464 [cs, stat]. URL: http://arxiv.org/abs/1911.09464.

[43] Shuchang Zhou et al. *DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients*. Feb. 1, 2018. arXiv: 1606.06160 [cs]. URL: http://arxiv.org/abs/1606.06160.

Appendix A

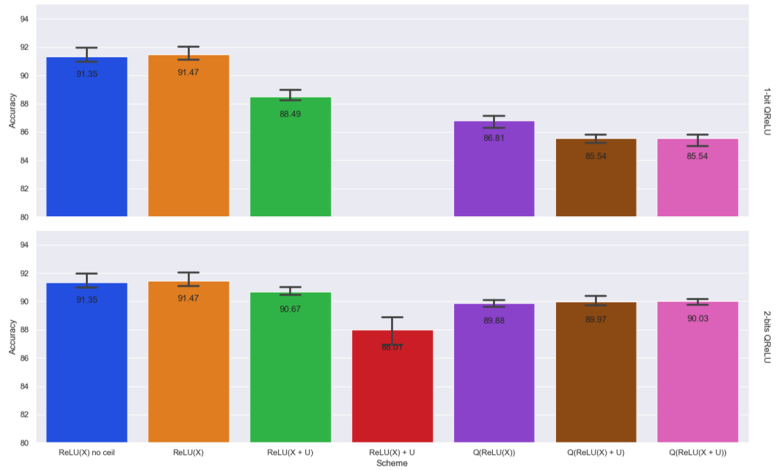
Hyper-parameters dither study

This Chapter aims at briefly presenting some results observed during the search for the best strategy to adopt regarding the addition of dither. There will be very few discussions of the results in this Chapter. Indeed, the objective is mainly to identify the best strategy from an accuracy point of view.

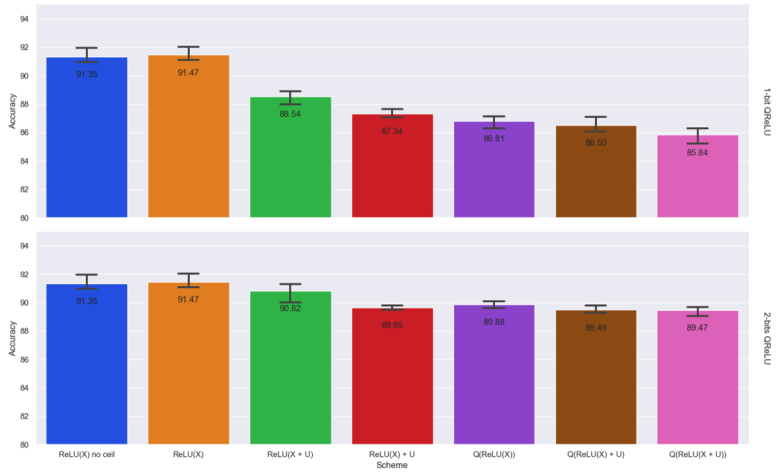
Moments and dither schemes

This section studies different dither schemes (basically, is it recommended to apply dither before or after the ReLU ?) in order to verify the impact of dither on training. The experiment consists in training networks with the activation function mentioned for quantized activation on 1 and 2 bits.

For networks involving dither (denoted as U instead of ν), two measurements are performed. Namely, inferences with and without dither. It is important to mention that the fine-tuning of BN layers has not been performed, for ease of use due to the number of tests to be performed, which explains the poor performance observed for these activation functions.



(a) No dither during inference.



(b) Dither during inference.

Figure A.1: Avg. accuracy on testing set for various activation functions. Each experiment was performed three times via three different initializations. The black vertical bars represent the 95 % ci.

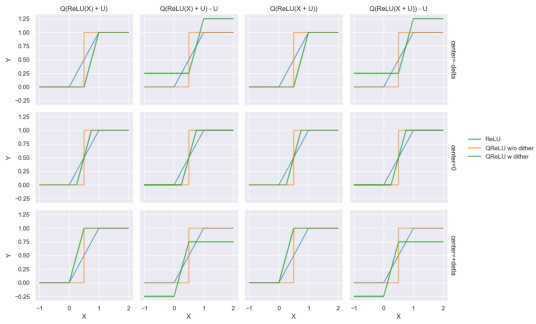
What is important to observe here is that there is a priori not really a better dither scheme to apply between Q(ReLU1(x) + ν) and Q(ReLU1(x + ν)). To be in line with the theory of dithered quantization, the scheme that applies dither after the activation function will be implemented.

These results also show the importance of fine-tuning BN layers. Indeed, performing only one inference with dither does not really make sense while suddenly stopping the use of dither during the inference leads to a drop in performance. Comparing the continuous scheme ReLU1(x) + ν using no dither during inference with ReLU1(x) + ν using dither during inference, allows evaluating the loss of performance caused by the non-adaptation of the BN layers. Hopefully, this drop can be completely recovered thanks to the fine-tuning of those (*cf. Appendix D*).

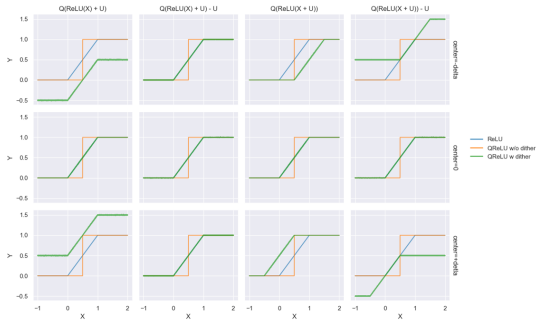
Widths and centers

This section aims to check what is the width, i.e. size of its distribution window, of dither to apply and how to center its distribution.

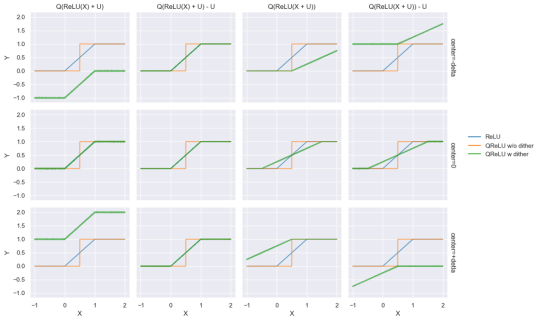
Here, three different widths are studied, namely $\Delta/2$, Δ and 2Δ where Δ is the quantization step. In addition to this, three different centering will also be tested, namely -Δ, 0 and Δ on different dither schemes in order to know which combination of centering/width to use for each of the following schemes: Q(ReLU1(x) + ν), Q(ReLU1(x + ν)), Q(ReLU1(x) + ν) - ν and Q(ReLU1(x + ν)) - ν.



(a) Width = 0.5



(b) Width = 1



(c) Width = 2

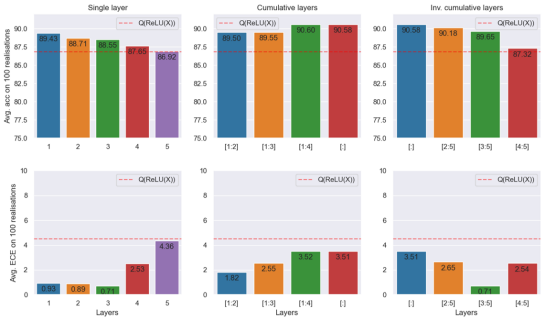
Figure A.2: Expected value of different dither strategies for varying centers and widths.

The approach adopted for the choice of the centering and the width is motivated by the fact that for several generations of dither, the quantized function ReLU must be equivalent to the ReLU function. The retained dither strategy therefore consists in using a centered in 0 dither whose width is equivalent to the quantization step.

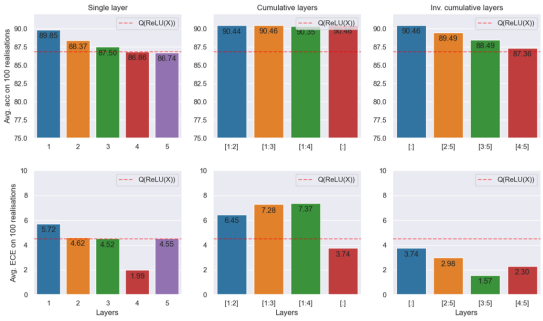
Layer-by-layer study

This section aims to study where it is most interesting to apply dither.

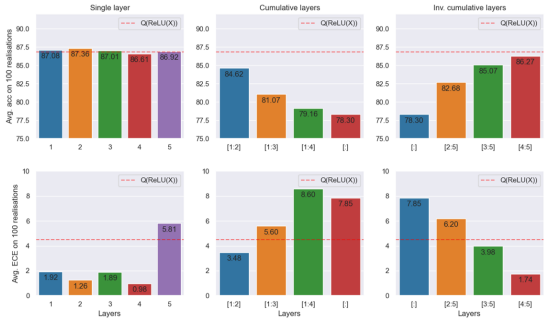
To do this, the architecture has been divided into 5 CBA blocks of different resolutions numbered from 1 to 5. Different tests were then performed in order to observe where to apply dither. The scheme studied here is the same as the one presented in this study, i.e. a centered dither of width equivalent to Δ applied to a 1-bit quantization function Q(ReLU1(x) + ν).



(a) Trained and inferred using dither on the mentioned layers.



(b) Trained using dither on all layers. Inferred using dither on the mentioned layers.



(c) Trained without using dither. Inferred using dither on the mentioned layers.

Figure A.3: Avg. accuracy and ECE (*cf. Appendix B for calibration*) using Q(ReLU1(X) + U) strategy with dither on various layers. No fine-tuning of the BN layers has been performed.

One can see that it is interesting to apply dither everywhere rather than on specific layers.

This concludes our search for hyper-parameters. Many other tests have been performed but a consequent pruning has been done to show only the essential.

Appendix B

Calibration study

The calibration of a model is defined as its ability to estimate the correctness of its classification. This Chapter aims at studying this metric through different experiments. First, the mathematical formalism will be defined and then the results obtained will be shown. Again, few discussions will be undertaken. The objective is really to present the results.

Metric

In the subsection 2.3.1G., was introduced the output process of the probability of correct classification, the *confidence*, via equation (2.14). This probability is expected to be as close as possible to the true probability of correct classification.

Formally, it is expected for a perfectly calibrated model that:

$$\mathbb{P}\left(\hat{y}_x = c \mid \hat{p}_{x;c} = p\right) = p, \ \forall\, p \in [0,1] \tag{B.1}$$

Where $\hat{y}_x$ denotes the output of the network computed for sample x while $\hat{p}_{x;c}$ denotes the confidence of the network on class c over sample x.

In 2017, Guo et al. [14] studies the calibration of deep learning models. To do this, they introduced the *Expected Calibration Error* (ECE) which will allow to empirically measure the calibration error for finite samples. The calculation of this error requires the estimation of the *expected accuracy* as well as the *average confidence* for each of the finite samples.

The author defines $M = 10$ finite samples by grouping the predictions associated with the testing set of a model into interval bins (each of size $1/M$). The set of indices B_m is composed by the samples corresponding to the predictions whose confidence falls in the interval $I_m =]\frac{m-1}{M}, \frac{m}{M}]$.

The expected accuracy associated with the finite sample B_m is given by:

$$\text{acc}(B_m) = \frac{1}{|B_m|} \sum_{x \in B_m} \mathbf{1}(\hat{y}_x = y_x) \tag{B.2}$$

The average confidence associated with the finite sample B_m is given by:

$$\text{conf}(B_m) = \frac{1}{|B_m|} \sum_{x \in B_m} \hat{p}_{\hat{y}_x} \tag{B.3}$$

A perfectly calibrated model will therefore have $\text{acc}(B_m) = \text{conf}(B_m)\ \forall\ m \in \{1, \ldots, M\}$.

The ECE consists of a weighted average of the different accuracy and confidences associated with each of the intervals. It is defined as:

$$\text{ECE} = \sum_{m=1}^{M} \frac{|B_m|}{x} \left|\text{acc}\left(B_m\right) - \text{conf}\left(B_m\right)\right| \tag{B.4}$$

Where n indicates the total number of samples.

A perfectly calibrated system will have an ECE that tends to 0 i.e. the confidence of the network perfectly indicates the probability of correctness of classification.

Measures

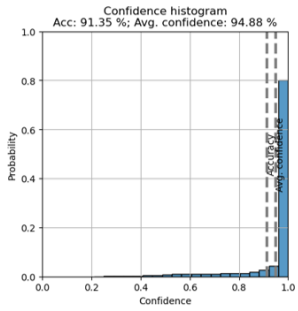
Experiments were performed on non-quantized, simply quantized and dithered quantized networks to observe the impact of the proposed training on the calibration.

Reliability diagrams and confidence histograms

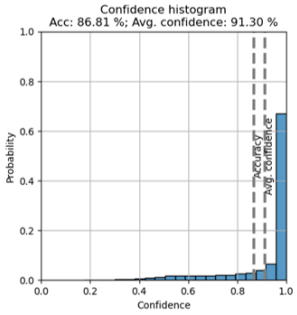
The plots in Figure B.1 represent the confidence histograms as well as the reliability diagrams as described in paper [14].

The confidence histogram shows the distribution of the confidences obtained on the testing set. The average accuracy and the average confidence are also shown.

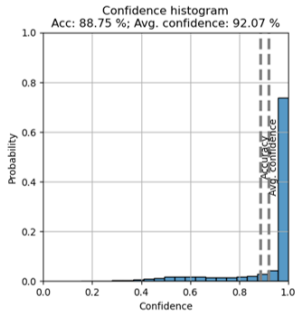
The reliability diagram shows in blue the real accuracy observed for the samples of the testing set whose confidence falls within the range of the different bins (calculated during the ECE) while the red indicates the error. Indeed, any deviation from the gray diagonal represents a bad calibration since it is expected that a calibrated network will obtain accuracy similar to its confidence.



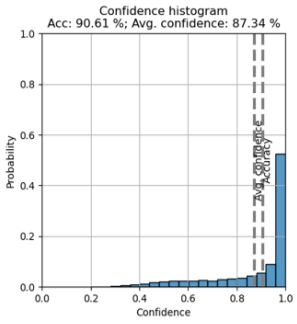
(a) ReLU1(x)



(b) Q(ReLU1(x))



(c) Q(ReLU1(x) + ν). No dither during inference.



(d) Q(ReLU1(x) + ν). Dither during inference. 20 realizations.

Figure B.1: Confidence histogram and reliability diagram observed on the test set for different networks using different activation functions.

The positive impact on the calibration between a dithered network and a simply quantized network is quite clearly observed. The calibration observed for a dither trained model, whose BN layers are fine-tuned, allows obtaining a natural calibration better than a non-quantized model. A dithered inference allows obtaining with 20 realizations an even smaller calibration error.

Training using dither and multiple inferences

A study similar to what was done in section 7.1.4 was performed on the ECE by varying the portion of the training set. The following results were obtained:

Metrics (%)	**Models**	**Portions of the training set**				
		0.1	**0.2**	**0.4**	**0.8**	**1.0**
ECE	FP	14.71	10.35	7.93	6.23	4.12
	Baseline	20.05	15.36	11.66	8.15	7.19
	Dithered	**11.23**	**9.54**	**6.45**	**5.10**	**3.38**

Table B.1: Summary of the results obtained on the testing set by varying the portion of the training set.

Again, from a general point of view, dithered training allows to obtain a more calibrated model than a simply quantized model but also than a full-precision model.

A study similar to what was done in section 7.1.4 was carried out on the ECE by performing several dithered inferences. The following results were obtained:

N	**1**	**5**	**10**	**20**	**50**	**100**
ECE (%)	4.52	2.24	2.88	3.22	3.46	3.54

Table B.2: Results obtained on the testing set by varying the number of realizations.

Making several inferences can therefore be interesting from a calibration point of view. However, the observed tendency indicates that there is an optimum of realizations to optimize the calibration error. This one is located at two inferences as shown in the following plot.

A comparison in calibration performance has been performed for different set-based approaches as in section 7.2.2D.. The results obtained are shown in Figure B.2.

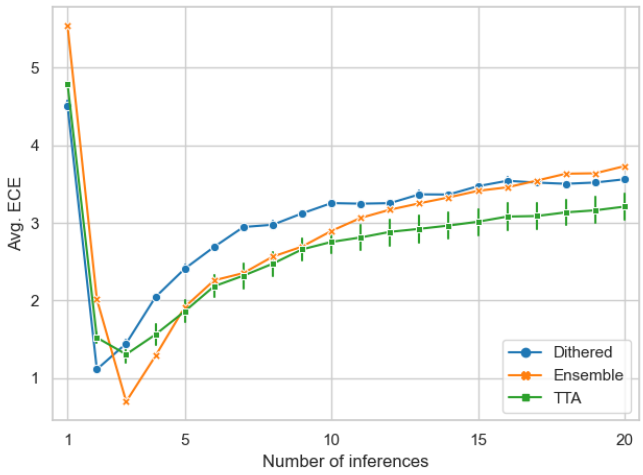


Figure B.2: Plot from the evolution of the ECE for various training set portions. Vertical bars show the 68% confidence interval.

k

Appendix C

Mini-batch size study

A study of the parameter N, i.e. the number of elements contained per mini-batch, was carried out in order to verify its impact on the training. Indeed, the optimization of the SGD thanks to dither was demonstrated by making the hypothesis that the parameter N was high enough for the empirical mean to be replaced by an expectation. This is obviously not the case in practice.

The objective of this experiment is to see if the latter has a visible impact on the training through the analysis of the accuracy on the testing set for networks trained using mini-batches of different sizes.

Each measurement was performed for three different models for the following N values: 32, 64, 128, 256 and 512. An adaptation of the learning rate has been performed so that the network is driven in a more or less similar way between different experiments.

The results obtained can be seen on the graph below for models using or not dither.

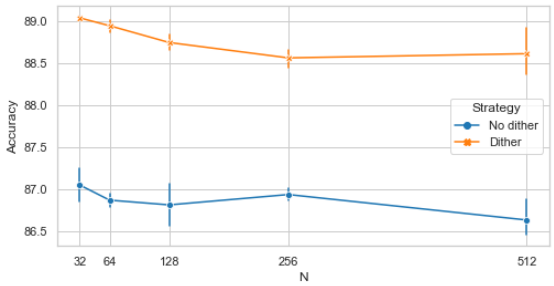


Figure C.1: Avg. accuracy obtained for varying size of mini-batches. The vertical bars present the 68 % ci.

Unfortunately, this study does not allow to observe any real trend of variation in performance in relation to the mini-batch size. We think that the slight variations

obtained are the consequences of a slightly different training due to this change in size. They do not necessarily reflect the consequences of the number of samples operating in the averaging process during the dithered training.

Appendix D

Models transfers study

Some experiments of transferring activation functions to another have been performed in order to observe the impact of the dither on the accuracy.

The networks are trained according to the activation given in the "Trained using" column while the inference is performed with the activation given in the "Inference using" column. The "FT" column indicates whether or not fine-tuning of the BN layers has been performed. Finally, the last column indicates the difference in accuracy observed on the testing set between the initial model and after the transfer. "A1" states for simply quantized, i.e. Q(ReLU1(x)) while FP states for full precision, i.e. no quantization using ReLU1(x).

Other transfer tests have been performed but are not shown here so as not to overload the Table. Again, the results obtained will not be discussed but are available here below.

Trained using	**Initial perf. (%)**	**Inference using**	**FT**	**End perf. (%)**	**Perf. diff. (%)**
FP	91.35	A1	Yes	69.98	-21.37
		A1 + dither	Yes	74.24	-17.11
FP + dither	87.34	FP	Yes	91.22	**+3.88**
A1	86.81	A1 + dither	No	12.34	-74.47
		A1 + dither	Yes	83.59	-3.22
A1 + dither	86.50	A1	No	85.54	-0.96
		A1	Yes	89.21	**+2.71**

Table D.1: Accuracy obtained for different model transfers trained with one activation function and inferred with another.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**