SAPIENZA
UNIVERSITÀ DI ROMA

Compression Techniques for Deep Quaternion Neural Networks

Faculty of Information Engineering, Informatics, and Statistics

Corso di Laurea Magistrale in Master in Engineering in Computer Science

Candidate

Riccardo Vecchi
ID number 1467420

Thesis Advisor

Aurelio Uncini

Co-Advisor

Simone Scardapane

Academic Year 2018/2019

Compression Techniques for Deep Quaternion Neural Networks
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: r.vecchi92@gmail.com

Contents

List of Figures

Chapter 1

Introduction

Recent progress in neural networks gave birth to the exploding field of Deep Learning, one of the most successful subset of Machine Learning. Since the beginning of this neural networks' renaissance, several architectures have been developed. One of these are neural networks based on the quaternion algebra. They are a promising architecture for several data and tasks, thanks to the hyper-complex algebra properties on which they are based.

The aim of this thesis is to extends and evaluate compression techniques used for "standard" neural networks in the quaternion domain. Compression has the potential to make a model highly efficient (both in terms of computing and storage), so that it can be deployed also in contexts previously impossible, as in mobile devices, or more simply, create more powerful models for the same computational budget.

1.1 Document Organization

In the first chapter we will present a general overview of the Machine Learning field. It is about its definition and how it is correlated to other disciplines.

In the second chapter we go deeper by introducing Artificial Neural Networks (ANNs), one of the many possible kind of models developed in Machine Learning. Then, the chapter concludes by describing the subject of the thesis, Quaternion Neural Networks (QNNs).

The third chapter establishes a mathematical and theoretical framework over the compression techniques that we want to develop and test on QNNs.

The fourth is about experimentation and discussions of the data collected from experiments. They range from testing some regularization functions introduced in this thesis to the implementation and tests complex pipeline that engage an ensemble of techniques (deep compression).

The last chapter (conclusion) sums up the results obtained in this work and try to highlight possible future directions of research.

Finally, an appendix for quaternion numbers is presented. It has no claim to be complete, but it is enough for understanding the thesis' work.

1.2 Machine Learning

Machine learning (ML) is the field of study that aims to programming computers to improve a performance criterion using example data or "past experience", hence without being explicitly programmed for doing so. It permits to extracts knowledge from data. Indeed, we can define learning as the capacity to improving with the experience at some task.

Actually, Machine Learning is a subset of the Artificial Intelligence (AI) field. The research goal of Artificial Intelligence is to create technology that allows computers to behave in an intelligent manner. Hence, most AI algorithms are developed to solve problems that animals and people solve continuously, like perception, knowledge representation, learning, reasoning, planning and so on.

In the ML context, a learner can be defined as an agent that, after having extracted experience from a finite set of examples, can generalize its knowledge to perform well also on new or unseen examples or tasks. The ability to generalize is central for a learner, indeed the variability of the inputs will be such that the training data can comprise only a tiny fraction of all possible samples.

Several models have been developed in the history of Machine Learning. Each one has its set of applications, problems and characteristics. The most important models are: Support Vector Machines (SVMs), Decision Trees, Genetic Algorithms (GAs), Artificial Neural Networks (ANNs) and so on.

1.2.1 Types of Learning Paradigms

There are three major types of learning paradigms in ML, given different tasks, approaches and the kind of data they use.

Supervised Learning

The first one is Supervised Learning. Stuart J. Russell and Peter Norvig describe Supervised Learning as the machine learning task of learning a function that maps an input to an output based on example input-output pairs[1]. In other words, Supervised Learning comprises algorithms used to learn a well-approximated mapping function $f(x)$ between input variables $x \in X$ and output variables $y \in Y$, so that output data can be derived for new input data provided.

All begins with a training dataset consisting of a set of training examples, pairs where an input object and its associated output value are provided to the machine. Once the function $f(x)$ is learned, the machine will make predictions of possible outcomes. Hence, the learning algorithm has to generalize from the training data to unseen situations in a reasonable and reliable way.

Two examples of Supervised Learning are Classification and Regression tasks, whose main difference is that in Classification problems the target value y takes discrete values (1 or 0 for binary problems, or from 0 to 9 like in the case of for digits classification); in a Regression problem instead, the target value is continuous (like the price of a house, which cannot be divided into a finite number of classes).

There are different learning algorithms that can be used, like Support Vector Machines, Naïve Bayes Neural Networks and Decision trees. Also, the applications for Supervised Learning are numerous and affect different fields, such as pattern recognition.

Unsupervised Learning

Unsupervised Learning[2] can be defined as a set of algorithms that can group unsorted and unlabeled data on the basis of similarities and differences. Hence, machines act on information received without prior training. Potentially, the self-organization of information from raw data typical of Unsupervised Learning algorithms makes them much more flexible than Supervised one. Indeed, even in the era of Big-Data, labeled datasets are not always available, and create one of them, is a costly and time-consuming operation. An hybrid between Supervised and Unsupervised Learning that tries to reduce the problem of scarcity of labelled data is called Semi-supervised learning.

Unsupervised Learning finds a wide range of applications and one of the most important of them is Cluster Analysis: here the uncategorized information are examined by the machine, which defined clusters grouping data on the basis of the presence or absence of commonalities previously identified. Another notably usage of Unsupervised algorithms is the density estimation in statistics. Also, it is applied in solving the famous "cocktail party problem", namely the task of separating sources from a mix of data.

In general, Unsupervised learning algorithms can perform more complex processing tasks than supervised learning ones, but unsupervised learning can be more unpredictable.

Reinforcement Learning

The third and last paradigm is called Reinforcement Learning (RL) and is an area of machine learning concerned with how some agents ought to take actions in an environment in order to maximize the idea of cumulative reward. Therefore, Reinforcement learning is all about making decisions sequentially: agents (algorithms) can take different actions interacting with the environment (which constitute a set of inputs) that is often mathematically described as a Markov decision process (MDP)[3].

The agent receives rewards by performing correctly and penalties for performing incorrectly, learning by maximizing its reward and minimizing its penalty (without intervention from a human).

The focus for an agent is finding a balance between exploration (of uncharted territory) and exploitation of current knowledge (Kaelbling et al., 2010)[4]. At start time the agent is "tabula rasa" namely it has no prior knowledge about the environment, hence in this phase is preponderant the exploration of the environment. As the agent continues to learn, its actions will be more and more based on the experience gathered, and not only on exploration of information, hence it will exploit its knowledge.

1.2.2 Deep Learning

ANNs have existed for quite a while (decades ago), but scientists struggled to get them to work. However, recent progress in the field gave birth to the successful field of Deep Learning.

Deep Learning, known also as Hierarchical Learning or Deep Structured Learning contains a large family of ML methods based principally on neural networks.

These architectures outperform other techniques in several tasks like computer vision (CV), natural language processing (NLP), speech recognition, drug discovery, medical image analysis, fraud detection, bioinformatics and so on. In some contexts they exceed also human performance[5, 6, 7].

The major factors who contributed to the rebirth of ANNs are:

- Progress in algorithms and computer science in general;
- Growing flood of on-line data (Big Data);
- Increasing computational power.

Their versatility is given also from the fact that the learning procedure can be supervised, unsupervised or semi-supervised.

In general, DL models are complex and difficult to optimize, but they offer several benefits.

For instance, one of the main advantage over other models in supervised learning tasks is that they can automatically do feature engineering, namely they can extract automatically the needed features and intermediate representations, lessening the use of harder computer vision algorithms or digital image processing techniques.

One of the main drawbacks (at least nowadays), is instead the DL model explainability and interpretability, indeed these models are often seen as black box systems. This fact could reduce their fields of application at only non critical sectors.

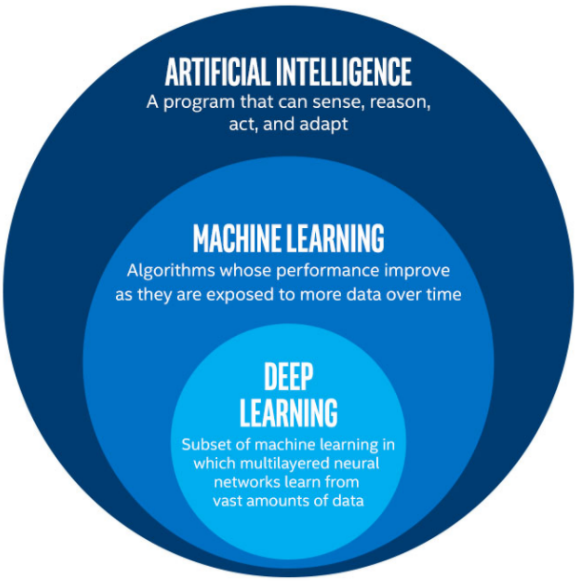

Figure 1.1. Relation among AI, ML and Deep Learning

Chapter 2

Artificial Neural Networks

2.1 Definition

Artificial neural networks (ANNs) are computing systems architectures (hardware or software) loosely inspired by the biological brain that can be thought as an huge neural network, or a collection of several ones. The simplest processing unit (PU) in a neural network is called the neuron.

Computers outperform brains in many tasks like numeric computation and symbols manipulations, but fail in other really simple for a biological brain, this fact is due to differences in the computational architecture[8, 9, 10]. Although single neurons are not very powerful and can perform only local computation, the human brain contains up to hundreds of billion of neurons highly interconnected (10^{14} to 10^{15} interconnections). Hence, the biological brain is a highly complex nonlinear and parallel computer that can perform motor control, perception, planning, pattern recognition and other tasks several times faster than the fastest digital computer, although its fundamental units are order of magnitude slower than the computers counterpart.

The computing architecture of the brain also gives to it many desiderable properties not present in von Neumann computers like low energy consumption, distributed representation and computation, generalization ability, fault tolerance (thanks to its distributed computation) and so on[11].

ANNs are the best fit that implements the main features of a biological brain. They can be implemented with electronic components or, more common today, simulated in software with computers.

An ANN is basically formed by several PUs linked together. Knowledge is acquired from outside stimuli with some learning mechanism. This knowledge is "stored" in the synaptic weights, hence learning means essentially modify these weights. In particular, these weights are modified such that the ANN is able to minimize a certain "cost function".

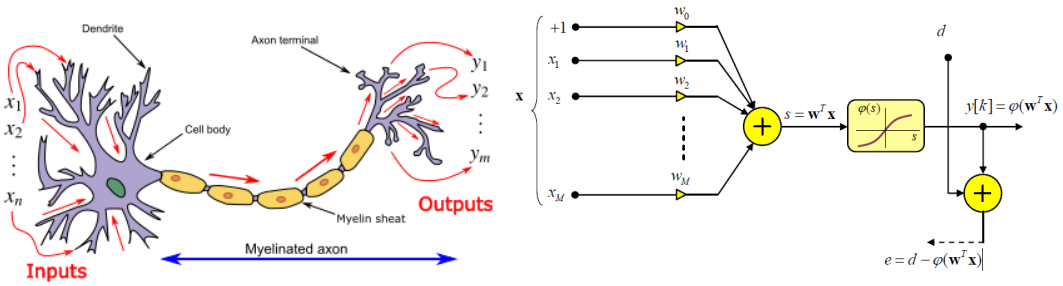

Figure 2.1. Comparison among a biological neuron model and a neuron based on the McCulloch & Pitts one[12]

A simple way to classify the several kind of ANNs is to identify them by:

- Network topology (or architecture);
- Neurons or processing units;
- The learning algorithms, namely the set of rules that, starting from an initial or random configuration of the network, specify the change to do to the network parameters to improve its performance on some task.

For which concern network architecture, there are two main classes of NNs: Feed Forward Networks and Recurrent Networks (RNN).

The difference between the two classes of networks is that the Feed Forward ones allow signals to travel only in one way (from input to output), while the latter type has feedback connections (mathematically they can be represented with a graph that contains loops).

RNNs exhibit a temporal dynamic behaviour[13] thanks to their structure, making them suitable for tasks containing undefined long sequences, as in the case of speech recognition[14].

Let define a basic mathematical framework about NNs by introducing the multilayer perceptron (MLP) without loss of generality for other kind of networks. MLPs are a class of feedforward networks organized in layers (at least three layers of nodes, that are: an input layer, one or more hidden layers and an output layer). This kind of network has a biological justification and a mathematical one, indeed, based on the famous Cybenko's theorem[15], MLPs are universal function approximators, namely they can approximate almost every continuous function given the right number of neurons.

Let $\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{w})$ be a MLP neural network where $\boldsymbol{x} \in \mathbb{R}^P, \boldsymbol{y} \in \mathbb{R}^Q$ and $\boldsymbol{w} \in \mathbb{R}^M$, representing a nonlinear transformation $f : \mathbb{R}^P \to \mathbb{R}^Q$ in which any continuous, multivariate function $\boldsymbol{d} = g(\boldsymbol{x})$ s.t. $g : \mathbb{R}^P \to \mathbb{R}^Q$ can be approximated as:

$$\| f(\boldsymbol{x}, \boldsymbol{w}^*) - g(\boldsymbol{x}) \|_\rho < \varepsilon \tag{2.1}$$

where $\boldsymbol{w}^*$ are the weights value of the MLP acquired with an "ideal" learning process, ρ is a given metric (e.g. Euclidean norm where $\rho = 2$) and ϵ an arbitrary small number.

Furthermore, we define:

- L: number of layers in the MLP;
- l: layer index;
- N_l: number of neurons in the l^{th} layer;
- $w_{ki}^{(l)}$: weight referring to the k^{th} neuron of the l^{th} layer which takes as input the output of the i^{th} of the $l-1^{th}$ layer;
- $w_{0k}^{(l)}$: bias relative to the k^{th} neuron of the l^{th} layer;
- $s_k^{(l)}$: linear combiner output for the k^{th} neuron of the l^{th} layer;
- $x_k^{(l)}$: output of the k^{th} neuron of the l^{th} layer;
- $\phi(\cdot)$: activation function s.t. $x_k^{(l)} = \phi\left(s_k^{(l)}\right)$.

The output y_k^l of a MLP network written in scalar form is defined as:

$$s_k^{(l)} = \begin{cases} s_k^{(l)} = \sum_{j=0}^{N_{l-1}} w_{kj}^{(l)} x_j^{(l-1)}, \quad x_0^{(l-1)} = 1 \\ x_k^{(l)} = \phi_k^{(l)}\left(s_k^{(l)}\right) \end{cases} for\, l = 1, 2, \ldots, L;\; k = 1, 2, \ldots, N_l; \tag{2.2}$$

Vector and matrix notation help to make notation more compact:

$$\begin{aligned} \boldsymbol{s}^{(l)} &= \left[\begin{array}{cccc} s_1^{(l)} & s_2^{(l)} & \ldots & s_{N_l}^{(l)} \end{array}\right]^T \\ \boldsymbol{w}_k^{(l)} &= \left[\begin{array}{cccc} w_{0k}^{(l)} & w_{1k}^{(l)} & \ldots & w_{N_l k}^{(l)} \end{array}\right]^T \\ \boldsymbol{\Phi}^{(l)} &= \left[\begin{array}{cccc} \phi_1^{(l)}(\cdot) & \phi_2^{(l)}(\cdot) & \ldots & \phi_{N_l}^{(l)}(\cdot) \end{array}\right]^T \\ \boldsymbol{x}^{l-1} &= \left[\begin{array}{cccc} 1 & x_1^{(l-1)} & \ldots & x_{N_{l-1}}^{(l-1)} \end{array}\right]^T \\ \boldsymbol{y}^{(l)} &= \left[\begin{array}{cccc} y_1^{(l)} & y_2^{(l)} & \ldots & y_{N_l}^{(l)} \end{array}\right]^T \end{aligned} \tag{2.3}$$

this way, the linear combiner result for the l^{th} layer is:

$$s^{(l)} = \left[\begin{array}{cccc} \boldsymbol{w}_1^{(l)T}\boldsymbol{x}^{(l-1)} & \ldots & \boldsymbol{w}_{N_l}^{(l)T}\boldsymbol{x}^{(l-1)} \end{array}\right]^T = \boldsymbol{W}^{(l)}\boldsymbol{x}^{(l-1)} \tag{2.4}$$

The weight matrix $\boldsymbol{W}$ is defined as:

$$\boldsymbol{W} \in \mathbb{R}^{(Nl \times N_{l-1})} = \left[\begin{array}{cccc} \boldsymbol{w}_1^{(l)t} & \ldots & \boldsymbol{w}_{N_l}^{(l)t} \end{array}\right]^T = \left[\begin{array}{cccc} w_{01}^{(l)} & w_{11}^{(l)} & \ldots & w_{01}^{(l)} \\ w_{02}^{(l)} & w_{12}^{(l)} & \ldots & w_{01}^{(l)} \\ & & \ddots & \\ \ldots & \ldots & & \ldots \\ w_{01}^{(l)} & w_{01}^{(l)} & \ldots & w_{01}^{(l)} \end{array}\right] \tag{2.5}$$

Hence, the output vector of the l^{th} layer can be written as:

$$\boldsymbol{x}^{(l)} = \left[\phi_1^{(l)}\left(\boldsymbol{w}_1^{(l)T}\boldsymbol{x}^{l-1}\right)\dots\phi_{N_l}^{(l)}\left(\boldsymbol{w}_{N_l}^{(l)T}\boldsymbol{x}^{(l-1)}\right)\right]^T = \boldsymbol{\Phi^{(l)}}\left(\boldsymbol{W}^{(l)}\boldsymbol{x}^{(l-1)}\right) \tag{2.6}$$

and the output of the MLP after L layers is:

$$\boldsymbol{y} = \boldsymbol{x}^{(L)} = \boldsymbol{\Phi}^{(L)}\left(\boldsymbol{W}^{(L)}\boldsymbol{\Phi}^{(l-1)}\left(\dots\boldsymbol{\Phi}^{(2)}\left(\boldsymbol{W}^{(2)}\boldsymbol{\Phi}^{(1)}\left(\boldsymbol{W}^{(1)}\boldsymbol{x}^{(0)}\right)\right)\right)\right) \tag{2.7}$$

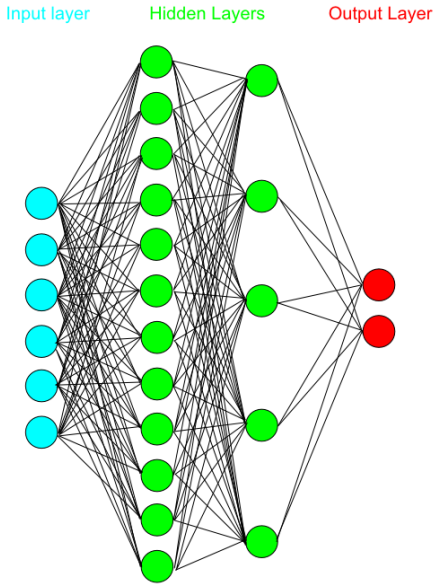

Figure 2.2. Instance of a Multilayer Perceptron

2.2 Activation Functions

An Activation Function (AF) combines in some manner the weights of the neuron and the corresponding set of inputs, hence defining the output of the unit.

The choice of a good activation function is essential in the build up of a good model. In particular, nonlinear AF allow small networks to solve nontrivial problems using small number of units.

AF must have some attractive features like being nonlinear, continuously differentiable, monotonic, approximate well identity function near the origin and so on.

Let define the Step Function, one of the first used AF for its simplicity:

$$\phi\left(x\right) = \begin{cases} 0 & x < 0 \\ 1 & x \geqslant 0 \end{cases} \tag{2.8}$$

A famous AF is the Logistic Function:

$$\phi(x) = \frac{R}{1 + e^{-x}} \tag{2.9}$$

where R is a parameter that control the curve's maximum value (the standard case is when $R = 1$).

Another function worthy of mention is the $\tanh(\cdot)$:

$$\phi(x) = \tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})} \tag{2.10}$$

Finally, one of the most AF currently used is the ReLU, defined as follows:

$$\phi(x) = \max(0, x) \tag{2.11}$$

A neuron that uses this AF is called ReLU (Rectified Linear Unit). ReLU is not differentiable in 0, but it carries several improvements[16] over others famous AFs. Moreover, ReLU is very efficient to compute.

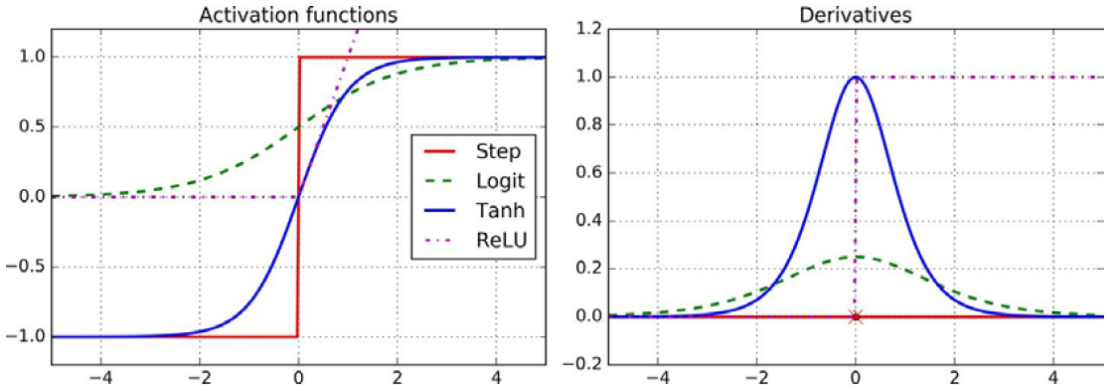

Figure 2.3. Summary of the principal Activation Functions and their derivatives[17]

2.3 Loss Functions

A loss functions (or cost function) is a computationally feasible function representing the price paid for inaccuracy of predictions of a given model, namely it is used to calculate how we are far from the desired behaviour of the system.

There are several type of loss functions, each with its mathematical properties and use cases.

For regression problems, one of the most used loss function is the Mean Square Error (MSE), called also quadratic loss:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2 \tag{2.12}$$

where Y_i and $\hat{Y}_i$ are our target variable and predicted value, respectively.

In classification problems instead, a well-suited loss function, especially in modern deep neural networks, is the cross entropy:

$$L(d, f(x)) = -\sum_{i=1}^{o} d_i \log(f_i(x)) \tag{2.13}$$

where d_i is the i^{th} desired output and $f_i(\cdot)$ is the i^{th} output of the model.

The cross entropy function is related to the concept of entropy introduced in information theory and to Kullback-Leibler divergence.

2.4 Gradient Descent and Backpropagation algorithms

As said before, there are several learning strategies for training neural networks. If we use a loss function, we can formulate the learning process as an optimization problem.

In that case, one of the most used algorithm is called Gradient Descent. It is a first-order iterative optimization algorithm used to find the values of parameters of a function f that minimize a loss function related to f. As the name of the algorithm says, it uses the concept of gradient of a function ∇f to optimize the objective function.

In the case of non convex functions, as in the case of loss function in neural networks, the algorithm does not guarantee to find the global minimum but only local minimum (in case of a convex function local minimum corresponds to global minimum). Despite this fact, Yann LeCun et al. proved that this is not a real obstacle in most cases[18].

In a Neural Network biases and weights of each artificial neuron are the parameters to optimize.

In general, for finding a local minimum iteratively, Gradient Descent updates each parameter w in the following way:

$$w_{n+1} = w_n - \gamma \nabla f(w_n) \tag{2.14}$$

Namely, we takes steps toward the opposite of the gradient. The γ parameter permits us to takes steps proportional to the steepest descent. Generally γ is far less than 1 and we call it learning rate. With an high learning rate γ we can move very fast toward a local minimum, but we risk overshooting the lowest point. Instead, with a very low γ, we can move in the direction of the negative gradient since we are recalculating it so frequently, hence a low learning rate is more precise, but the operation to reach the local minimum will last longer (calculating the gradient is time-consuming).

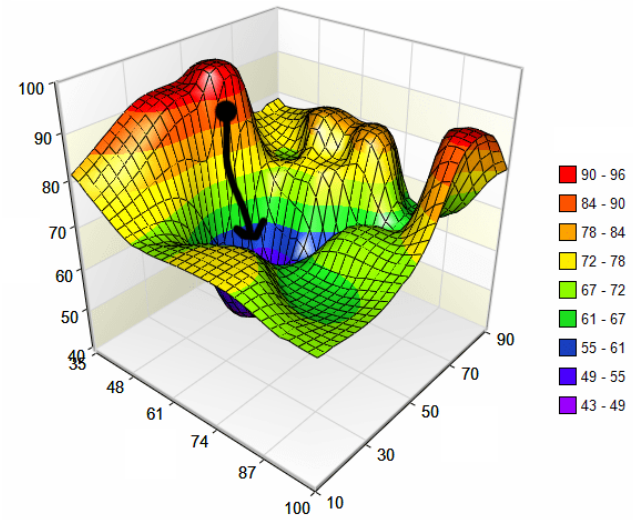

Figure 2.4. Loss function minimization in a 2D input space

It is interesting to note that gradient descent works in spaces of any number of dimensions, even in infinite-dimensional ones.

Obviously there are many versions of the Gradient Descent algorithm like Stochastic Gradient Descent (SGD), SGD with momentum and so on, but the previous concepts remain valid.

Stochastic Gradient Descent in particular, was introduced for addressing situations in which the training data size is huge. It is called stochastic because at each iteration, rather than computing the gradient ∇f, SGD randomly takes samples in subsets of the training data called mini-batches. Thus, there is no need to compute all the training data in batch. SGD is often more appealing than gradient descent because it offers a regularization effect and because there are cases in which with very few iterations the algorithm can find useful solution to the learning problem, without the need to visit all the training set.

It is worth also to mention a variant of SGD called Adam, that has been used in all the experiments performed for the thesis. The Adam optimization algorithm has recently seen broader adoption for deep learning applications. Its name derives from "Adaptive moment estimation" and, as the name suggests, it computes new learning rate for each parameter of the network. It is an algorithm that tries to take the best of 2 famous training algorithm, called AdaGrad and RMSProp.

For computing effectively the gradients, we use another algorithm called Back-propagation[19], that is easily the recursive application of the chain rule. The algorithm was introduced by several researchers in the 60's[20] and implemented on computers in 1970. It should be noted that back-propagation is not a training algorithm, but is used only to compute the gradients. Moreover, it is not specific to Deep Feedforward Networks like the CNN ones.

Back-propagation is a local process, namely each indivisible unit is independent in computing the gradient, it does not need to being aware of the full-state of the network.

Algorithm 1 Back-propagation algorithm for MLPs

Input **:**

$Network\ depth\ l$

$\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}\ weight\ matrices$

$\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}\ bias\ parameters$

$\boldsymbol{x}\ input\ value$

$\boldsymbol{t}\ target\ value$

Forward pass **:**

$h^{(0)} = \boldsymbol{x}$

for $k = 1, \ldots, l$ ***do***

$\boldsymbol{\alpha}^{(k)} = \boldsymbol{b}^{(k)} + W^{(k)}\boldsymbol{h}^{(k-1)}$

$\boldsymbol{h}^{(k)} = f\left(\boldsymbol{\alpha}^{(k)}\right)$

end for

$y = \boldsymbol{h}^{(l)}$

$J = L\left(\boldsymbol{t}, \boldsymbol{y}\right)$

Backward pass **:**

$\boldsymbol{g} \leftarrow \nabla_y J = \nabla_y L\left(\boldsymbol{t}, \boldsymbol{y}\right)$

for $k = l,\ l-1,\ 1$ ***do***

$Propagate\ gradients\ to\ the\ pre-nonlinearity\ activations:$

$\boldsymbol{g} \leftarrow \nabla_{\alpha(k)} J = g \odot f'\left(\boldsymbol{\alpha}^{(k)}\right)$

$\nabla_{W^{(k)}} J = \boldsymbol{g}\left(\boldsymbol{h}^{(k-1)}\right)^T$

$\nabla_{b^{(k)}} J = \boldsymbol{g}$

$Propagate\ gradients\ to\ the\ next\ lower-level\ hidden\ layer:$

$\boldsymbol{g} \leftarrow \nabla_{h^{(k-1)}} J = \left(W^{(k)}\right)^T \boldsymbol{g}$

end for

The symbol $\odot$ in the pseudo-code of the algorithm denotes the element-wise product, also known as the Hadamard product.

The version of Back-propagation proposed here works for MLPs that have been extensively presented before, but there are more general versions that support networks of any structure.

Over time various improvements have been made to the BP algorithm, boosting its performance through techniques like dynamic programming (for avoid doing the same computations multiple times) and several other ones, for instance [21].

2.5 Regularization and overfitting

A problem that can occur when training a model is that of overtraining or, as called in statistic, overfitting.

While undertraining, or underfitting, happens when a statistical model cannot adequately capture the underlying data structure (resulting in parameters missed in that model and so poor predictive performance), overfitting can be considered as the opposite situation.

Indeed, in statistics overfitting is "the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably"[22]. So an overfitted model is a model containing more parameters than can be justified by the data. The essence of overfitting is to have unknowingly extracted some of the residual "variation" as if it represented underlying model structure. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. But these concepts do not apply to new data provided and negatively impact the models ability to generalize.

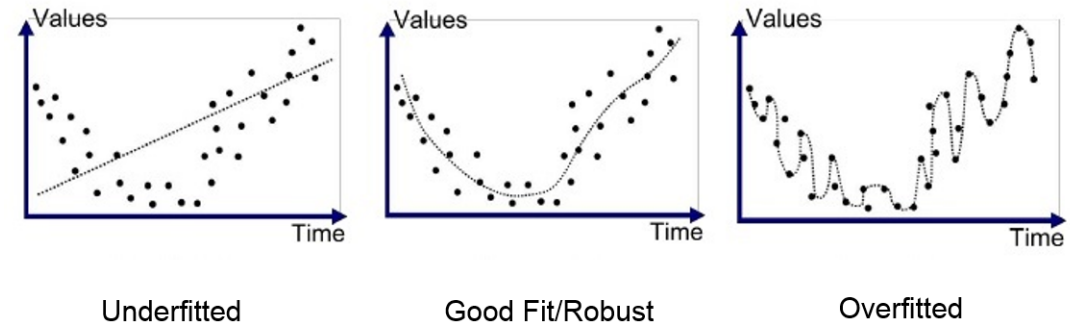

Figure 2.5. Underfitted, good fit and overfitted models

In order to prevent overfitting (but also ill-posed problems), regularization is usually applied. It is a technique that constrains, regularizes or shrinks the coefficient estimates towards zero. In other words, this technique discourages learning a more complex or flexible model, in order to avoid the risk of overfitting.

Therefore regularization reduces the variance of the model without substantial increase in its bias.

Regularization terms are generally added to the loss functions used. If $\mathcal{L}(\theta)$ is the loss function to minimize, a regularized version is:

$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \mathcal{G}(\theta) \tag{2.15}$$

where $\mathcal{G}$ is the regularization function (e.g. Quaternion sparse regularization) and λ is a so-called regularization factor, useful for balancing the actual loss and the need to regularize the network.

The tuning parameter λ controls the impact on bias and variance. As the value of λ rises, it reduces the value of coefficients, reducing the variance as well. This increase in λ is beneficial until a certain limit as it is only reducing the variance without losing any important properties in the data. But after that limit, with further increase in λ the model starts losing important properties, giving rise to bias in the model and thus underfitting. Therefore, the value of λ should be carefully selected.

There exists several types of regularizations that deal with different aspects and goals like Dropout[23], an inexpensive but powerful method in which the network randomly (and temporarily) drops some units at each training iteration for preventing overfitting and other unwanted results.

Without the pretension of presenting all forms of regularization, we will mathematically introduce two type that are extensively used in the experiments of the thesis and are also most used one in the field.

The first regularization function introduced is the L_1:

$$L_1(\boldsymbol{w}) \triangleq |\boldsymbol{w}| \tag{2.16}$$

where $\boldsymbol{w}$ is any weight or bias of the network.

Another regularization worth to mention is the L_2 that can be defined as:

$$L_2(\boldsymbol{w}) \triangleq ||\boldsymbol{w}||_2^2 \tag{2.17}$$

L_2 is also known as "weight decay" for its property of giving penalty proportional to the magnitude of the weights.

2.6 Convolutional Neural Networks

A convolution neural network, often shorten with ConvNet or CNN, is a particular kind of artificial neural network inspired directly from the visual cortex of animals[24, 25]. Hence, CNNs are principally developed for computer vision tasks, although their use is not limited to only this case. They can be viewed as a regularized version of multilayer perceptrons. The last one indeed, is more prone to overfitting given its full-connected architecture. CNNs, instead take advantages of hierarchical pattern in data.

A CNN can be comprised of several type of layers, of which the main ones are convolutional layers, pooling layers and fully connected ones. In the following a brief presentation of these kinds of layers:

- Convolutional layers - are the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input.

- Pooling layers - A pooling layer is also referred to as a sub-sampling layer. In this category there are also several layer options, with max pooling being the most popular. This basically takes a filter (normally of size 2x2) and a stride, then applies it to the input volume and outputs the maximum number in every subregion. Another instance of pooling functions is the average pooling that return the mean across the subregion. The intuitive reasoning behind the pooling layer is that once we know that a specific feature is in the original input volume, its exact location is not as important as its relative location to the other features. An advantage is that this layer drastically reduces the spatial dimension (length and width change but not the depth) of the input volume (hence less parameters/weights) and moreover it can control the overfitting.

- Fully-connected layers - After several convolutional and pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have connections to all activations in the previous layer, as seen in regular neural networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

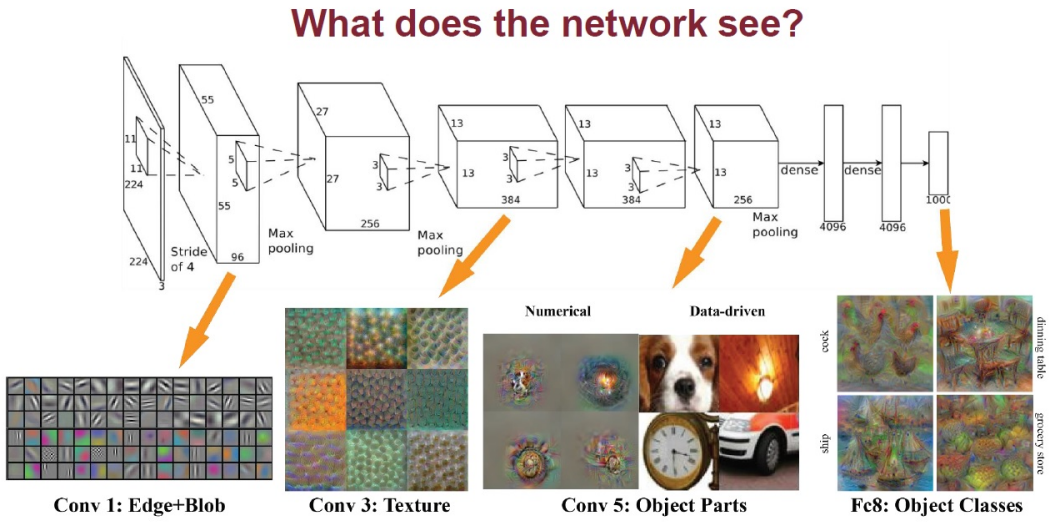

Figure 2.6. Hierarchical complexity of patterns in a CNN

Another important aspect of CNNs are that they are translation invariant, namely the results are not affected by the 2D position of objects/structures within the image.

This fact is due to the parameter sharing typical of this kind of network.

On the contrary, other transformations of the patterns in an image can affect final results as in the case of scaling and rotation.

This weak of CNNs is often solved with the use of data augmentation that consists in augmenting the training set by adding also pre-processed versions (rotated, scaled, mirrored) of the training set images.

2.7 Quaternion Neural Networks

Several works have noted benefits in generalizing weights of neural networks from the set of real numbers $\mathbb{R}$ to the complex field $\mathbb{C}$ as [26, 27].

Quaternion Neural networks (QNNs) takes one step further and generalize complex neural networks as well.

For simplicity, we will refer to the model of the Multi-Layer Perceptron defined before and we will extends this model to the hyper-complex algebra $\mathbb{H}$.

One of the few differences between the two models is on the type of weights used.

The weight matrix $\boldsymbol{W}$of a quaternion MLP can be defined as:

$$\boldsymbol{W} \in \mathbb{H}^{(Nl \times N_{l-1})} = \begin{bmatrix} \boldsymbol{w}_1^{(l)t} & \dots & \boldsymbol{w}_{N_l}^{(l)t} \end{bmatrix}^T = \begin{bmatrix} w_{01}^l & w_{11}^l & \dots & w_{01}^l \\ w_{02}^l & w_{12}^l & \dots & w_{01}^l \\ \dots & \dots & \dots & \dots \\ w_{01}^l & w_{01}^l & \dots & w_{01}^l \end{bmatrix} \tag{2.18}$$

where $\mathbb{H}$ is the corresponding quaternion algebra.

In order to perform backpropagation on this kind of networks activation function and cost functions have to be differentiable with respect all components (r, i, j, k) for each quaternion parameter. Fortunately, this is easily obtained by considering the quaternion chain rule, as shown by Chase Gaudet and Anthony Maida[28].

Obviously, QNNs can also work on real data. In the case the input is not quaternion-valued, it should be expanded appropriately. For instance, in a classification task with gray-scale image a common choice is to put the gray-scale image in the r channel and initialize to 0 the vector part, while in a RGB image the vector part (i, j, k) is filled with RGB channels respectively and the real part is zero-filled.

The articulated structure of a quaternion and its relative algebra in respect to a real numbers can benefit effectiveness of neural networks in several ways.

2.7.1 Quaternion Convolution

Let extend the standard convolution operation in the quaternion domain as done in [28].

Given a quaternion matrix $\boldsymbol{W} = \boldsymbol{A} + i\boldsymbol{B} + j\boldsymbol{C} + k\boldsymbol{D}$ where $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$ are real matrices, and a quaternion vector $\boldsymbol{h} = \boldsymbol{w} + i\boldsymbol{x} + j\boldsymbol{y} + k\boldsymbol{z}$ where $\boldsymbol{w}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ are real vectors, we define the quaternion convolution between $\boldsymbol{W}$and $\boldsymbol{h}$ in the following matrix representation:

$$\begin{bmatrix} \boldsymbol{R}(\boldsymbol{W} * \boldsymbol{h}) \\ \boldsymbol{I}(\boldsymbol{W} * \boldsymbol{h}) \\ \boldsymbol{J}(\boldsymbol{W} * \boldsymbol{h}) \\ \boldsymbol{\mathcal{K}}(\boldsymbol{W} * \boldsymbol{h}) \end{bmatrix} = \begin{bmatrix} \boldsymbol{A} & -\boldsymbol{B} & -\boldsymbol{C} & -\boldsymbol{D} \\ \boldsymbol{B} & \boldsymbol{A} & -\boldsymbol{D} & \boldsymbol{C} \\ \boldsymbol{C} & \boldsymbol{D} & \boldsymbol{A} & -\boldsymbol{B} \\ \boldsymbol{D} & -\boldsymbol{C} & \boldsymbol{B} & \boldsymbol{A} \end{bmatrix} * \begin{bmatrix} \boldsymbol{w} \\ \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{z} \end{bmatrix} \tag{2.19}$$

Hence, the definitions given above are enough for easily inferencing several kinds of quaternion networks like Quaternion Convolutional Neural Networks (QCNNs), Quaternion Recurrent Neural Networks (QRNNs).

2.7.2 QNN performance

As pointed out by [28, 29] Quaternion neural networks can outperform classical neural networks on several tasks like classification, segmentation, denoising and so on, given the same budget of parameters. This fact is due to the quaternion algebra and its operations: they manage to capture internal latent dependencies.

Moreover, some data are naturally well-represented by quaternion matrices: as an example, in colored images each RGB pixel can be represented with a quaternion number as explained by [29].

Real-valued neural networks instead, when processing colored images (and in general multi-channels inputs), will merge these channels by summing up the convolution results and will give in output one single channel per kernel.

This type of processing works well in many situations, but it has some evident drawbacks[30], because the network will not learn the articulated interrelationship between different channels, as in the basic case of three RGB channels losing valuable information. Moreover, in real-valued networks there is an higher risk of overfitting because by simply summing up the results of the convolution operations, the networks has many degrees of freedom for learning kernels.

Furthermore, the quaternion ability to represents spatial rotations may make quaternion networks more robust also when there is rotational variance. The ability to represents orientations and rotations is mathematically defined and explained in the Appendix A.

In the fourth chapter, before presenting experiments of the novel techniques developed for the thesis, there will a performance comparison between quaternion-valued neural networks and classical ones.

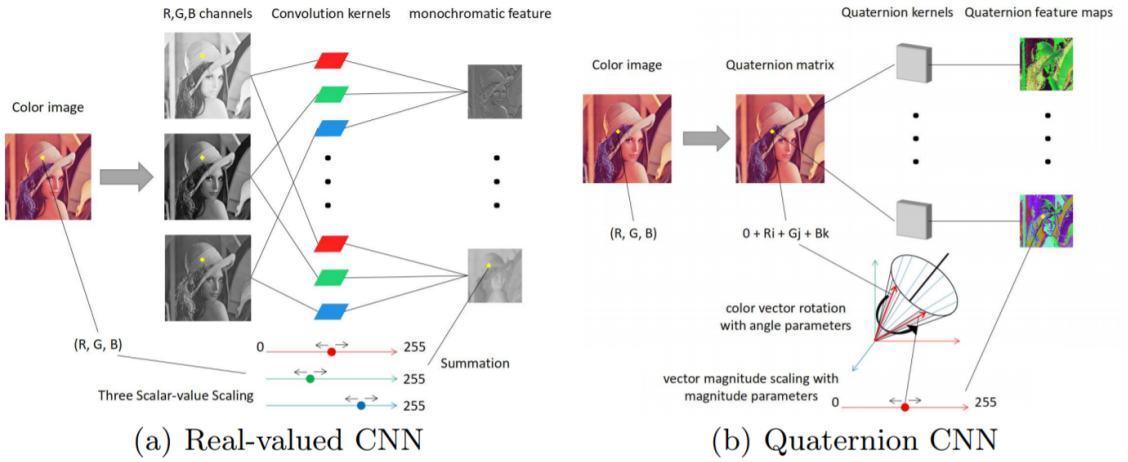

Figure 2.7. Illustration on the difference in the processing of RGB channels between R-CNNs and Q-CNNs on convolution layers from [29].

Chapter 3

Experimental Contribution

3.1 Group-level regularization

The aim of this section is to describe a regularization function thought to induce sparsity not only on single weights of the networks, but also to force an entire set of weights to 0 if their contribution to the output of the network is almost insignificant. A more structured sparsity can be effective to take advantage of hardware-specific acceleration and for obtaining extremely compact networks as proved by previous works like [31, 32].

For quaternion neural networks, it is interesting to define a group as a quaternion weight (four real numbers) and see if introducing two new regularization functions based on this idea it is possible to cause a quaternion-level sparsity.

The first regularization proposed is the *Quaternion sparse regularization*, defined as follows:

$$R_Q(\boldsymbol{W}) \triangleq \sum_{\boldsymbol{w} \in \boldsymbol{W}} \|\boldsymbol{w}\| = \sum_{\boldsymbol{w} \in \boldsymbol{W}} \sqrt{a^2 + b^2 + c^2 + d^2} \tag{3.1}$$

where $\boldsymbol{W}$ is the set of all weights of the network and a, b, c, d are the coefficient of a quaternion weight $\boldsymbol{w}$. The regularization as defined makes sense because it corresponds to use the quaternion norm as metric. For further information about the quaternion algebra, see Appendix A.

The Quaternion sparse regularization could still be sub-optimal, since single real weights are not enforced to 0. Then, an extension to the previous regularization is the *Sparse Quaternion Lasso regularization*:

$$R_{QL}(\boldsymbol{W}) \triangleq R_Q(\boldsymbol{W}) + R_{L_1}(\boldsymbol{W}) \tag{3.2}$$

where R_{L_1} uses the standard L_1 norm.

In general, R_{QL} could have two different regularization factor, one for L_1 part and another for the R_Q part, much like an elastic net regularization. However, for simplicity I choose to maintain a single regularization factor λ for the R_{QL} tuning.

In the chapter 3 are presented experimental data on the effectiveness of the two proposed regularization function under different datasets and problems.

3.2 Quaternion Batch Normalization

Batch normalization is a technique introduced in 2015[33] that enhances the training speed, performance and stability of artificial neural networks. It works by adjusting and scaling the activations, normalizing hence the input layer. Although the effect of batch normalization is proved, the reasons behind its effectiveness are under discussion. The authors attributes its effectiveness to the reduction of internal covariate shift.

Conceptually, the normalization step should be conducted over the entire training set to estimate mean and variance more precisely. But generally, neural networks are trained iteratively over subsets of the training set called mini-batch.

Hence, in its original formulation, the algorithm is:

Algorithm 2 Batch Normalizing Transform, applied to activation x over a mini-batch.

Input : $Values\ of\ x\ over\ a\ mini-batch:\ \mathcal{B} = \{x_{1\dots m}\}$;
$Parameters\ to\ be\ learned\ \gamma, \beta$

Output : $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad mini-batch\ mean$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad mini-batch\ variance$$

$$x_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad normalization$$

$$y_i \leftarrow \gamma x_i + \beta \equiv BN_{\gamma,\beta}(x_i) \qquad scale\ and\ shift$$

Batch-size is a key hyper-parameter to choose for the training of a neural network, indeed a wrong one can hinder or slow-down the training. If the neural networks uses some Batch Normalization layers, the tuning of this hyper-parameter is even more important, indeed this technique works well only when statistics of the mini-batch approximate well those of the entire population.

It is interesting to test the effectiveness of the standard Batch Normalization in regards to Quaternion Neural Networks and also compare it with a natural extension to the hyper-complex algebra, hence introducing the Quaternion Batch Normalization, hereinafter referred as QBN.

In order to define the QBN, firstly we need to extend statistical features (mean and variance) used in Batch Normalization to the quaternion domain.

In [34] they have been defined as:

$$\begin{aligned} QE(x) &= \frac{1}{T}\sum_{i=1}^{T} q_0 + q_1\boldsymbol{i} + q_2\boldsymbol{j} + q_3\boldsymbol{k} \\ &= \bar{q}_0 + \bar{q}_1\boldsymbol{i} + \bar{q}_2\boldsymbol{j} + \bar{q}_3\boldsymbol{k} \end{aligned} \tag{3.3}$$

$$\begin{aligned} QV(x) &= \frac{1}{T}\sum_{i=1}^{T} (x - QE(x))(x - QE(x))^* \\ &= \frac{1}{T}\sum_{i=1}^{T} (\Delta q_0^2 + \Delta q_1^2 + \Delta q_2^2 + \Delta q_3^2) \end{aligned} \tag{3.4}$$

The quaternion mean $QE(x)$ is still a quaternion, while the variance $QV(x)$ is a real number similar to the real case.

It is interesting to note that each element of $QE(x)$ works as a mean for its respective component, while $QV(x)$ is shared across all the four components.

Then the definition of QBN for an input feature x_i can be found from [35] and is:

$$\begin{aligned} QBN(x_i) &= \gamma\left(\frac{x_i - QE(x)}{\sqrt{QV(x) + \varepsilon}}\right) + \beta \\ &= \gamma\left(\frac{q_0^i - \bar{q}_0}{\sqrt{QV(x) + \varepsilon}}\right) + \beta_0 \\ &+ \left(\gamma\left(\frac{q_1^i - \bar{q}_1}{\sqrt{QV(x) + \varepsilon}}\right) + \beta_1\right)\boldsymbol{i} \\ &+ \left(\gamma\left(\frac{q_2^i - \bar{q}_2}{\sqrt{QV(x) + \varepsilon}}\right) + \beta_2\right)\boldsymbol{j} \\ &+ \left(\gamma\left(\frac{q_3^i - \bar{q}_3}{\sqrt{QV(x) + \varepsilon}}\right) + \beta_3\right)\boldsymbol{k} \end{aligned} \tag{3.5}$$

As in the standard case, also the QBN introduces 2 parameters, namely for each output, there is real number γ that initializes to 1 and represents stretch scale, and a quaternion parameter β that initializes to 0 and represents shift scale. The constant ε is infinitesimal and is introduced for numerical stability.

Moreover, given that γ is the factor that scales the input for each output neuron (or channels for CNN), it is interesting to test also an L_1 regularization over the γ parameter of the QBN. Indeed, imposing this regularizer to the parameter γ should be another way that leads to group-level (quaternion) or, more broadly, neuron sparsity. Hence, it is worthwhile to compare the performance of this solution and the previously presented *Quaternion sparse regularization* and *Sparse Quaternion Lasso regularization.*

This intuition comes from Gordon et al.[36] and [37].

3.3 Deep compression for Quaternion Networks

Deep compression[38] is a three stage pipeline composed by pruning, trained quantization and Huffman coding. This technique is capable to reduce the storage requirement of neural networks up to 49 times the original size without loss of accuracy. Other advantages are that these compressed networks are often faster (although it depends on the hardware used) and have better energy efficiency. In a mobile-centric world where compute and storage resources are limited and in which there are privacy concerns, these gains can do the difference between run NNs workloads in cloud or in local devices. An instance of the successful application of the technique of Deep compression can be found also in the paper of Iandola et al.[39], in which the authors develop a minimal neural network called "SqueezeNet" that has the same accuracy of AlexNet over the ImageNet dataset, but with far less parameters and space required.

In this section is presented a consistent, but slightly modified version of the pipeline introduced in the original paper.

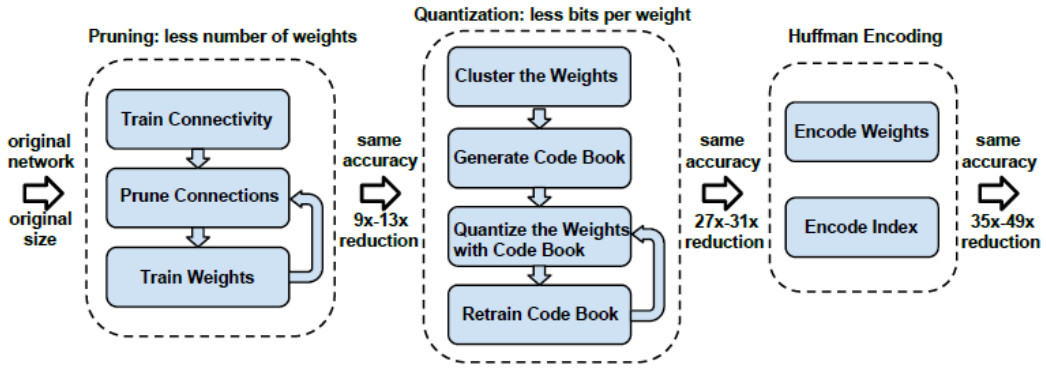

Figure 3.1. Deep compression pipeline from [38]

Pruning aims to delete small-weight connections that contribute nearly nothing to the output of the network. The *Sparse Quaternion Lasso regularization* presented before fits very well this job, given also the results (see next chapter) in respect of the other regularizations techniques. Hence, as pruning technique will be used the R_{QL} regularization, forcing every weight lower than a given threshold (10^{-3} in our case) to zero. The first difference with respect to the original pipeline is that the model will not be re-trained after pruning.

The second stage concerns weight sharing, namely given a budget of N weights allowed per layer, we will found n centroids with the K-means clustering algorithm, that is one of the most simple unsupervised algorithm. The initialization of the centroids can be of three different types, that are linear, density and random initialization, although the paper proves that the most effective is the linear one. Hence, we will use only the linear initialization in the experiments.

All weights of a layer of the neural network will be updated to the value of the nearest centroid. Hence, k original weights $W = \{w_1, w_2, \ldots, w_k\}$ will be partitioned into

n buckets $C = \{c_1, c_2, \ldots, c_n\}$, where we should have $k \gg n$ for minimizing the so-called within-cluster sum of squares ($WCSS$):

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{w \in c_i} \mid w - c_i \mid^2 \tag{3.6}$$

This suggest also to use an optimized structure for storing the weights, that will be called "codebook" in order to maximize the entropy of the information.

Weight sharing will be tested in two versions: the first will apply weight sharing for each quaternion layer (similar to the paper's implementation, except for the quaternion-nature of the weights) and the other will handle each quaternion component (r, i, j, k) as a separate layer. The idea is that see the quaternion layer as an indivisible unit, it could be more resistant to quantization in respect of the original proposal with the same number of bit.

Obviously, after the weight sharing step the accuracy of the NNs could be severely degraded, depending on the number of different weights allowed. Hence, after having determined the centroids with the K-means clustering algorithm and applied weight sharing, there is another step of training to fine-tune the weights. If the quantization is not too aggressive in the number of bits (and thus the number of centroids), fine tuning could restore the accuracy lost in the weight's quantization giving though a network that is squeezable of at least one order of magnitude.

In particular, for retraining the model after having stored its shared weights in a codebook structure, we computes the gradients through a back-propagation phase, as in the standard case. Then we algebraically sums all the gradients of weight that belong to the same cluster, hence using the indicator function $\mathbb{1}(\cdot)$, the centroid index of a weight w by I_w and the loss $\mathcal{L}$, the gradient for a centroid C_n will be:

$$\frac{\partial \mathcal{L}}{\partial C_n} = \sum_{w \in W} \frac{\partial \mathcal{L}}{\partial w} \mathbb{1}(I_w = n) \tag{3.7}$$

The last step compress the codebook structure with the famous Huffman encoding that aims to maximize the entropy of the information. Given the previous steps, there will be a lot of redundant information. In this work, Gzip will be used as a proxy for the Huffman encoding, although it is more complex than the latter.

The most valuable things discovered by the authors of the paper is that these techniques (pruning, weight sharing, encoding) work better together, because one step helps the next one. Indeed, if we prune as much as possible the network, the quantization part in the weight sharing will have less loss of precision keeping the number of centroids fixed.

Chapter 4

Implementation and Results

4.1 Tools & libraries

For the technical implementation of the thesis and the carry out of experiments many tools have been used. Here a list of the most useful ones.

Python Programming Language

All neural network models and techniques have been developed with the Python programming language in its latest version 3.7.3. Python is an high-level, interpreted and general purpose programming language. It uses garbage collection and is dynamically typed.

Python is a very versatile language, indeed it supports several programming paradigms as the object-oriented, procedural and functional one.

Moreover, the design philosophy of the language is to emphasize the code readability by enforcing some syntactic rules and by having introduced code style guidelines since its creation.

It is actually the de-facto standard for application in data science, machine learning, scientific applications and other related disciplines. This fact is due to its ever expanding collection of libraries for these kind of disciplines. The results is that is one of the most demanded and fast-growing languages in the world.

Figure 4.1. Python programming language logo

PyTorch - Machine Learning framework

One of these is PyTorch: it is an open-source ML framework for Python developed by Facebook. The library strengths are support to tensor computation (with syntax similar to NumPy), GPU acceleration support, built-in deep learning support and autograd system for automatic differentiation.

Figure 4.2. Logo of the Machine Learning framework PyTorch

Google Colaboratory

Google Colaboratory is another tool that has been used to accelerate development and testing.

It is a cloud-based platform in which there is a free Jupyter notebook environment and let access to powerful computing backend with support for hardware acceleration (GPU, TPU). Neural Networks for their own nature are computational demanding and are embarrassingly parallel, hence they are a good fit for parallel resources like GPU that, compared to CPU, can speed-up computation up to hundreds of times[40, 41, 42, 43]. In particular Google Colaboratory provides a CUDA environment with the professional Nvidia Tesla K80 GPU (12 GB of GDDR5) with very few usage restrictions. Moreover, the environment has already the most used libraries, e.g. libraries for numerical computation like NumPy, for data visualization (Matplotlib, Pandas and so on) and obviously the most used ML framework TensorFlow and PyTorch.

Figure 4.3. Google Colaboratory's logo

Dataset

Dataset are a collection of data. In Machine Learning, where algorithms try to learn from data, they are a focal point of the work. Indeed a too small dataset or a badly balanced one can be an obstacle in the reaching of the goal (small and badly balanced are relative to the task to learn and the targeted performance to achieve).

One of the most simple dataset available in the Machine Learning research is MNIST (Modified National Institute of Standards and Technology database). The dataset

is composed by 70,000 gray-scale images of handwritten digits (from 0 to 9), of which 60,000 for the training phase and the last 10,000 for testing purpose.

A more complex dataset is CIFAR-10 (Canadian Institute For Advanced Research), which has a collection of 60,000 32x32 RGB images that belongs to 10 different classes (6,000 images per class).

The choice of these two datasets for this work is due both to their relative simplicity and for the fact that in neural network literature they are studied and used in massive way, make them suitable to test and benchmark new techniques.

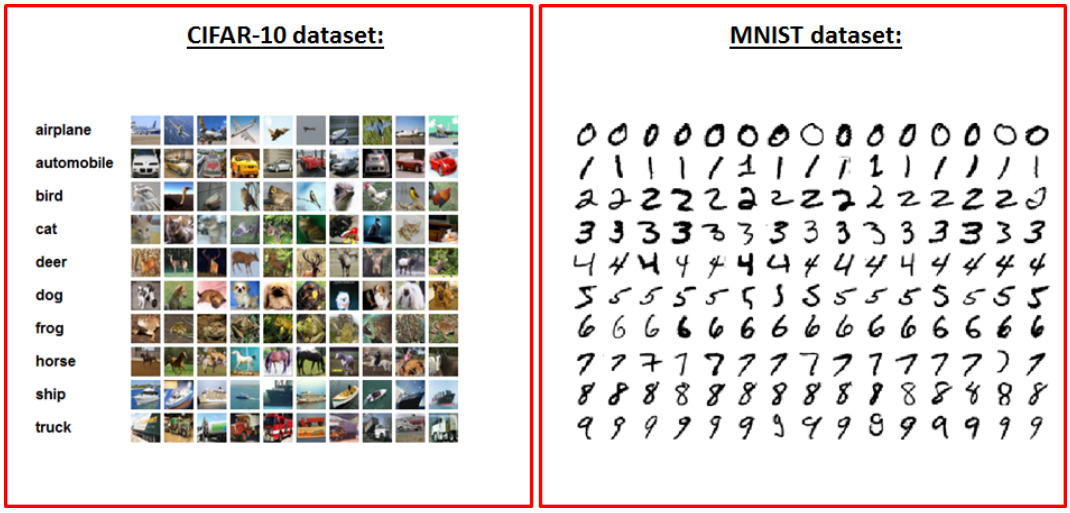

Figure 4.4. Samples from CIFAR-10 and MNIST datasets

Pytorch Quaternion Neural Networks Library

Implementation-wise, a starting point for this work was found at [44], developed by the ML researcher Titouan Parcollet. The library contains quaternion primitives (operations), quaternion layers (convolution, linear and so on) and networks samples.

After some fixes and extensions, the library becomes a good basis for developing and testing compression techniques for quaternion NNs.

In particular, the first step was to do a major refactoring of the library, in which the code has been made device-agnostic (It can run on CPU, GPU or whatever accelerator will be supported in the future by PyTorch) and deprecated methods has been replaced by new ones (PyTorch is constantly updated to support new features, enhance performance, simplify deployment of models, or simply fix bugs).

After this initial work, new regularizations, the Deep Compression pipeline and the Quaternion Batch Normalization have been developed and integrated in the library together with several clients to test them.

The updated library is then a fork of the original one and can be found at

https://github.com/Riccardo-Vecchi/Pytorch-Quaternion-Neural-Networks.

The GitHub repository contains also a Google Colab Notebook that is a turnkey environment for testing the techniques developed for the project without download

the code. Comfortable forms let the user testing several combination of parameters and hyper-parameters without the need to modify in first person the code.

Below there is the tree structure of the library:

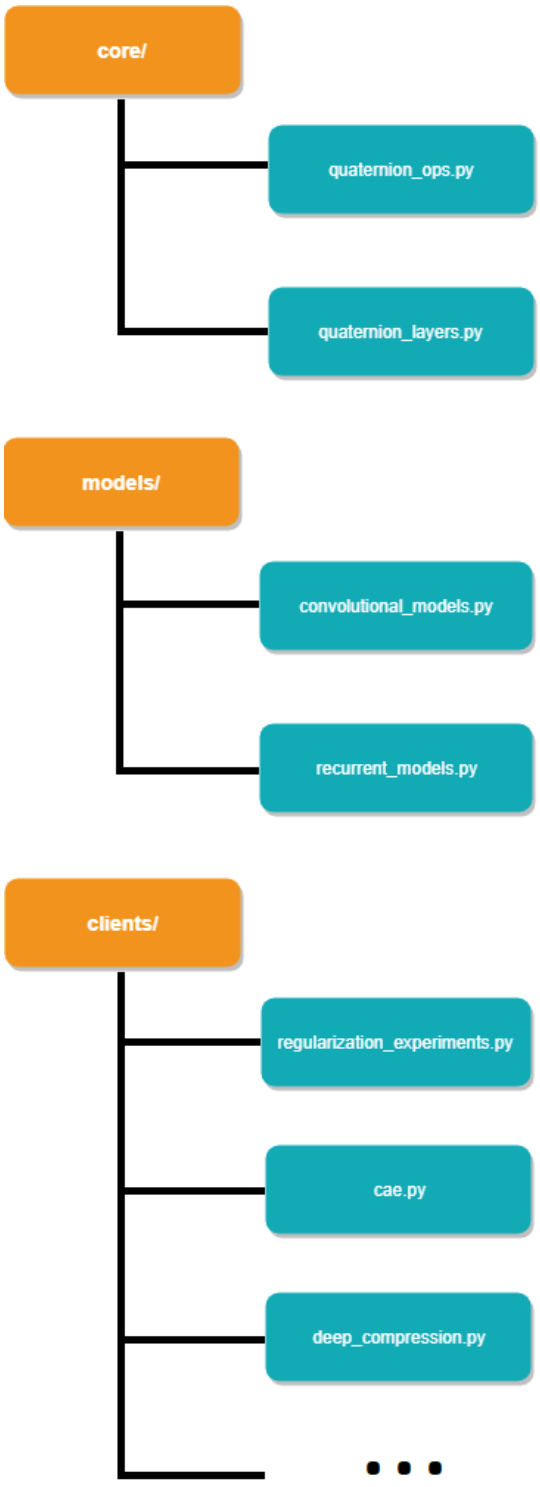

Figure 4.5. Tree structure of the extended library

4.2 Quaternionic vs Real-valued Neural Networks

Before the testing of the novel techniques introduced in the chapter 3, we want to test the effectiveness of QNNs in comparison to the standard ones.

Two experiments will be presented in this section on convolutional models, since this thesis is focused on them for the sake of brevity, although also recurrent ones have proved their effectiveness in several tasks.

CIFAR-10 Classification

In this experiment, we will test real-valued CNNs and QCNNs on a classification task over the CIFAR-10 dataset and the performance metric will be the maximum accuracy reached.

The experiment will be performed three times and results will be averaged to have more confidence in the data and avoiding outlier outcomes.

No form of regularization will be used to obtain a comparison less noisy as possible.

We avoid to test also with MNIST dataset because it is a too simple dataset, indeed the problem of classify samples from this dataset can be considered already solved effectively with standard CNNs.

In general, the number of parameters and the depth are directly proportional to the expressive power of a network. Since in literature we found that quaternion-valued networks can outperforms standard ones thanks to their inherent capacity to learn internal relationship among data, we will make them smaller in comparison to the standard one and still try to obtain an higher test accuracy.

Dataset	Type	Depth	# Parameters	Training epochs	η	Batch size
CIFAR-10	R-CNN	7	191,414	50	0.002	400
	Q-CNN	6	68,232			

Table 4.1. Parameters and hyper-parameters of the networks.

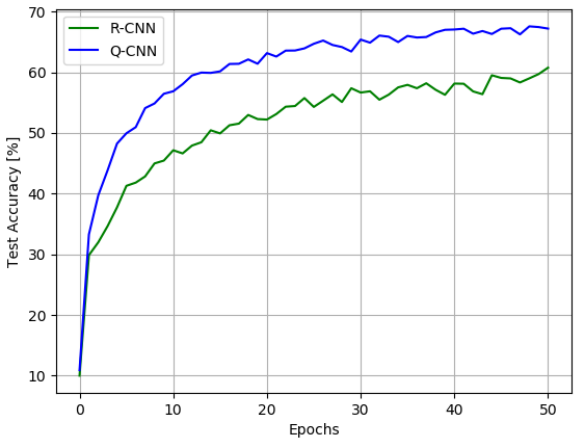

Figure 4.6. Comparison between R-CNN and Q-CNN over the CIFAR-10 dataset

Dataset	Measure	R-CNN	Q-CNN
CIFAR-10	Test accuracy [%]	60.75	67.19
	Training time [secs]	415.93	438.24

Table 4.2. Summary of the average results from the classification experiment above

The Quaternion variant outperforms the standard CNN by reaching an average accuracy of 6.44% higher, despite having almost a third of parameters and being shallower.

As said in the section 2.7.2, this fact is explainable by remembering that quaternion convolutional layers do not lose structural information about the RGB channels. Indeed, QCNNs process colored pixels as single multi-dimensional entities, while standard CNNs consider pixels as three different and separated values.

In addition to not losing color information, QCNNs impose an implicit regularizer on the architecture of network as proved by [29].

More in general, when there are multidimensional input features, QCNN can learn both internal and global relations among data.

Color reconstruction

In the second experiment instead, we will reproduce the paper [45], in which the authors proved that a Quaternion Convolutional Auto-Encoder (QCAE) manages to reconstruct colours from gray-scale images well, while the real-valued Convolution Auto-Encoder (CAE) fails in this simple task.

An autoencoder is a neural network that tries to reconstruct its input, hence it is used in unsupervised problems. It is composed by an encoding part and a decoding part. In general the hidden layers have dimension smaller than the input, hence the network need to learn efficient encoding of data.

The client used for the test was already included in the public library provided by [44]. The training is done on a single training image from the KODAK dataset, while the testing phase is done on an another image from the same dataset.

Type	Depth	# Parameters	Training epochs	η
QCAE	4	6,444	500	0.002
CAE	4	24,785	500	0.002

Table 4.3. Parameters and hyper-parameters of the networks

Although the quaternion auto-encoder has almost four times less parameter than the classical counterpart, it manages to outperform the latter already from the first 100 iterations.

A graphical proof of this result is given in the next page.

Figure 4.7. Color reconstruction comparison between Real-valued network (left) and Quaternion-valued network (right) after 0, 100, 200, 300, 400, 500 iterations respectively.

4.3 Comparison among regularization functions

In this paragraph will be presented an experimental comparison among the standard regularization functions and the new ones specifically designed for quaternion networks. The functions are tested against multiple datasets and each experiment have been performed 3 times to have more confidence on the results obtained.

Algebraically speaking, in the case of real-valued neural networks, the *Quaternion sparse regularization* becomes a simple L_1, while the *Sparse Quaternion Lasso regularization* becomes a double L_1.

Implementation-wise instead, (as it is implemented in the project library) *Quaternion sparse regularization* does nothing for real-valued Network, while the *Sparse Quaternion Lasso regularization* becomes a simple L_1.

Hence, in the following tests only quaternion neural networks will be used.

Each regularization function has been tuned with the maximum regularization factor λ possible and maintaining a comparable accuracy level among them (for the fairness of the experiment).

As this paragraph concerns more the sparsity capability of regularization functions, the accuracy plots will not be showed (they are very little informative given that we will try to maintain them very similar among experiments), instead, the mean accuracy will be reported in the summary table, below the plots.

For calculating the sparsity, every weight $\boldsymbol{w}$ such that $|\boldsymbol{w}| < 10^{-3}$ is computed as 0.

Degenerate cases in which the network lost 100% of its weights are not computed in the mean results, but the experiment were repeated.

In the following table there are the parameters used in the experiments:

Dataset	# Parameters	Training epochs	η	Mini-batch size
MNIST	11,688	20	0.002	400
CIFAR-10	474,920	50	0.002	400

Table 4.4. Parameters and hyper-parameters of the networks

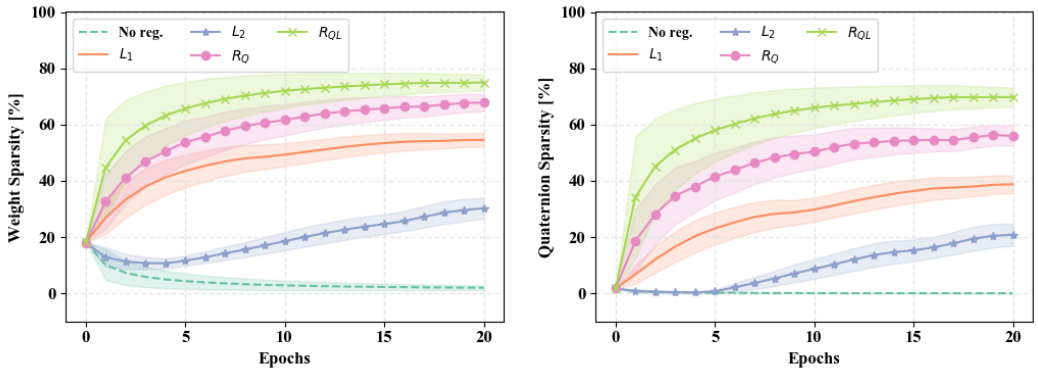

Figure 4.8. Comparison of regularization functions L_1, L_2, R_Q, R_{QL} and base case (no regularization) for MNIST. Regularization factors: $\lambda_{L_1} = 3 \cdot 10^{-3}, \lambda_{L_2} = 0.2, \lambda_{R_Q} = 7.5 \cdot 10^{-3}, \lambda_{R_{QL}} = 2.7 \cdot 10^{-3}$.

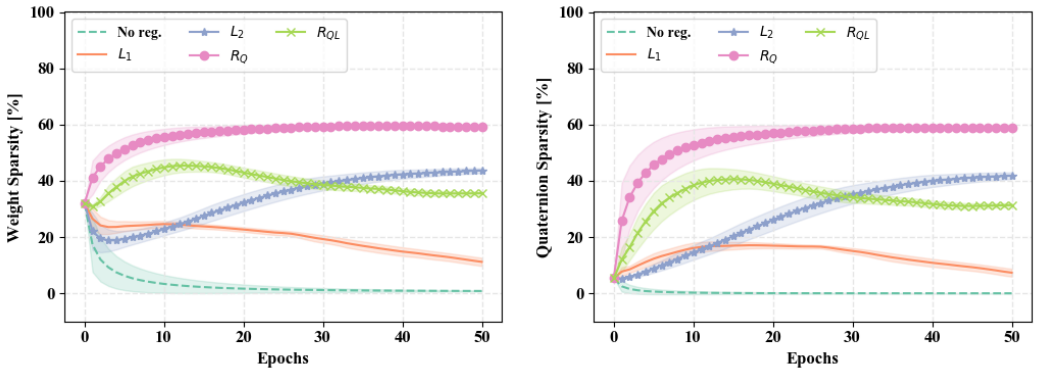

Figure 4.9. Comparison of regularization functions L_1, L_2, R_Q, R_{QL} and base case (no regularization) for CIFAR-10. Regularization factors: $\lambda_{L_1} = 2 \cdot 10^{-6}, \lambda_{L_2} = 5 \cdot 10^{-5}, \lambda_{R_Q} = 5.3 \cdot 10^{-6}, \lambda_{R_{QL}} = 3.4 \cdot 10^{-6}$.

Some observation can be drawn from the graphs above.

Firstly, it can be seen that L_2 regularization has different performance between the two datasets. In the case of MNIST it does not enforce sparsity very well, both for weight and quaternion sparsity, while in CIFAR-10 tend to be worse only than R_Q. In the case of real-valued networks, L_2 is not famous to be good at sparsifying. On the contrary, L_1 is considered good in literature, although in the case of CIFAR-10 it loses performance over time for our network. The L_1 term that is in R_{QL} probably is the reason why also R_{QL} has a descend arc for which concern sparsity in this dataset. As said before, we will avoid for simplicity to use two different regularization terms λ for the two terms of R_{QL}, but probably with a good tuning it could outperforms all the other techniques, including R_Q. R_{QL} is the best regularization tested in the case of MNIST, while R_Q has more consistent performance over the two datasets. For classification problems is clear that both R_Q and R_{QL} have desirable properties, indeed they enforce weight and quaternion sparsity well. Graphically, this can be viewed as the height difference between the two lines (weight and quaternion) is smaller than in the case of using L_1 or L_2.

The average results are summed up by the following table:

Dataset	Measure	No reg.	L_1	L_2	R_Q	R_{QL}
MNIST	Test accuracy [%]	98.95	96.69	93.46	96.81	96.29
	Training time [secs]	179.35	183.17	185.43	184.22	186.22
	Weight Sparsity [%]	1.71	54.68	34.08	68.40	75.08
	Quaternion Sparsity [%]	0.00	40.01	23.45	53.82	69.42
CIFAR-10	Test accuracy [%]	71.30	72.58	73.43	71.03	73.20
	Training time [secs]	924.77	1146.89	1160.48	1151.50	1168.37
	Weight Sparsity [%]	0.77	8.58	44.24	59.29	35.91
	Quaternion Sparsity [%]	0.00	4.77	42.61	58.73	31.89

Table 4.5. Summary of the average results from classification experiments

It is interesting also to view the weight's norm in a layer at the change of the regularization used with some histograms generated with TensorBoard:

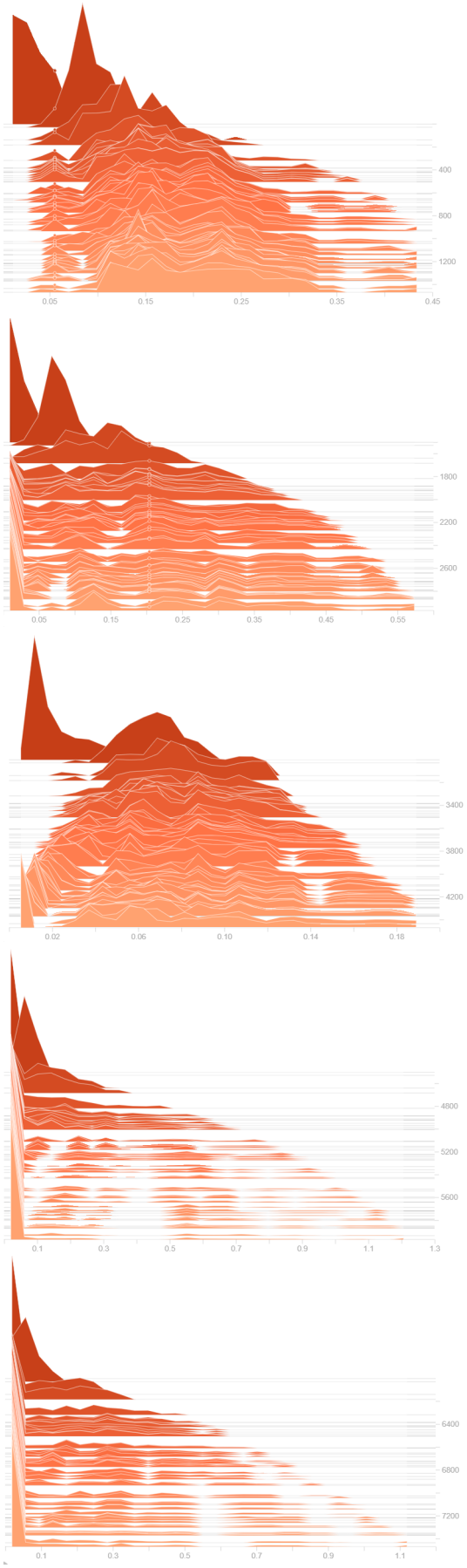

Figure 4.10. Histograms of weights' norm over time for the first quaternion convolutional layer for the neural network used for MNIST. From top to bottom: no regularization, L_1, L_2, R_Q, R_{QL}.

One could wonder if the same good results can be reached using the new regularizations for other tasks and architectures. Indeed we are interested in techniques as general as possible.

For instance we can test the effectiveness of regularization functions with the previously presented problem of the reconstruction of colours[45]. The problem is quite simple from a complexity point of view, hence we expect a greater level of sparsity in respect to the classification problems previously tested.

Moreover, the unsupervised problem uses a Quaternion Convolutional Auto-Encoder (QCAE), hence we can test our regularizations over another architecture.

To measure the quality of a compressed image in respect to the original one, there is the need to introduce some measurement suitable for the purpose.

Let introduce two standard measures: $PSNR$ (Peak signal-to-noise ratio) and $SSIM$ (structural similarity).

The first is a measure of the ratio between the maximum possible power of a signal and the power of the noise that reduce the quality of its representation[46]. Hence the $PSNR$ (in dB) is defined as:

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{\sqrt{MSE}}\right) = 20 \cdot \log_{10}(MAX_I) - 10 \cdot log_{10}(MSE) \tag{4.1}$$

where MSE is the mean squared error and MAX_I is the maximum possible pixel value of the image.

On the other hand, the latter measure $SSIM$ is more specifically designed to predict the perceived quality of digital contents and thus, is an improvement on more traditional methods like the $PSNR$. Its formula[47] is the following:

$$SSIM\,(x, y) = \frac{(2\mu_x\mu_y + c_1)\,(2\sigma_{xy} + c_2)}{\left(\mu_x^2\mu_y^2 + c_1\right)\left(\sigma_x^2 + \sigma_y^2 + c_2\right)} \tag{4.2}$$

where x and y are two windows of an image of equal size NxN, μ_x, μ_y are the average of x and y respectively, σ_x^2, σ_y^2 their variance and c_1, c_2 two variables to stabilize the division with weak denominator.

For completeness, in the experiments both measures will be presented.

The iterations have been doubled to 1000 in respect to the previously presented color problem, while the network used is exactly the same. Each regularization function has been tuned as much as possible.

Regularization	$\boldsymbol{\lambda}$
No reg.	-
L_1	$1 \cdot 10^{-4}$
L_2	$5 \cdot 10^{-3}$
R_Q	$3 \cdot 10^{-4}$
R_{QL}	$1.5 \cdot 10^{-4}$

Table 4.6. Regularization factors used in the compression experiments on the QCAE architecture

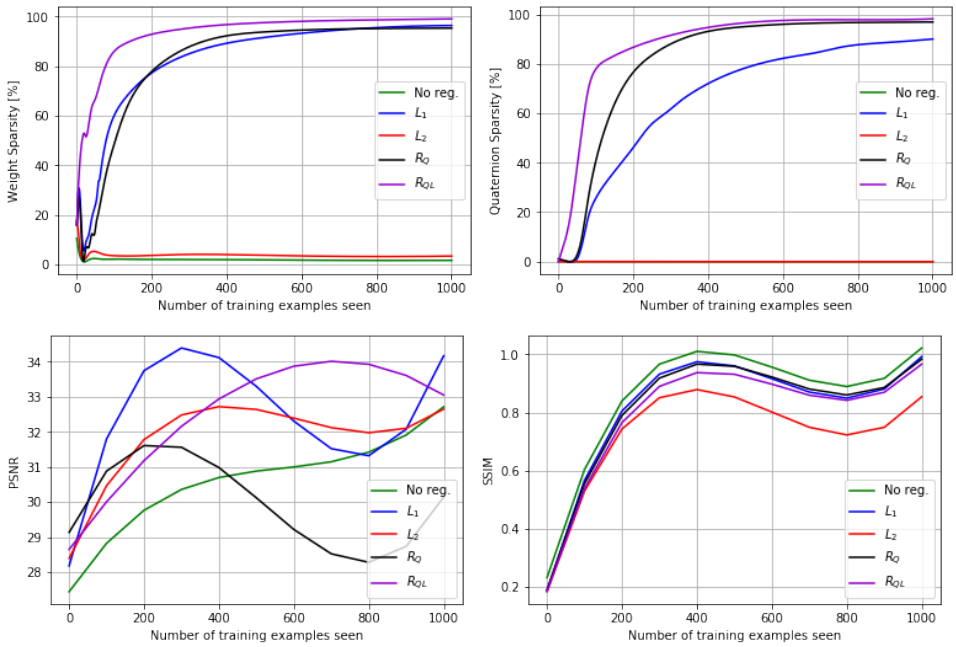

Figure 4.11. Comparison of regularization functions L_1, L_2, R_Q, R_{QL} and base case (no regularization) for the coloring task with a Quaternion Convolutional Auto-Encoder

In our tests, the $PSNR$ chart is very noisy and in general has a limited scope of validity as pointed out by previous works[48, 49]. Hence, since $SSIM$ is a more reliable measure for which concern the human perception, we will refer to it for our comparison.

Measure	**No reg.**	$\mathbf{L_1}$	$\mathbf{L_2}$	$\mathbf{R_Q}$	$\mathbf{R_{QL}}$
PSNR	33.06	34.52	32.34	29.12	32.53
SSIM	0.96	0.93	0.79	0.92	0.91
Weight Sparsity [%]	1.75	96.59	3.35	96.72	99.15
Quaternion Sparsity [%]	0.00	89.96	0.00	95.25	97.98
Total time	23.24	24.17	23.77	23.62	24.91

Table 4.7. Summary of the results for compression experiment after 1000 iterations

We can note that also in this task, both R_Q and R_{QL} outperform all the other techniques, with a moderately better result for the latter. The L_2 regularization is slightly better than not using regularization at all, while L_1 is behind R_Q for which concern quaternion sparsity.

4.4 Quaternion Batch Normalization Experiments

4.4.1 Quaternion Batch Normalization Vs Batch Normalization

Initially, it is interesting to see if the generalized Batch Normalization can bring some performance increases in regards to the classical one, that is the BN out-of-the-box implementation of PyTorch.

Keeping fixed hyper-parameters and avoiding any form of regularization that could result in noisy results, we have the following results:

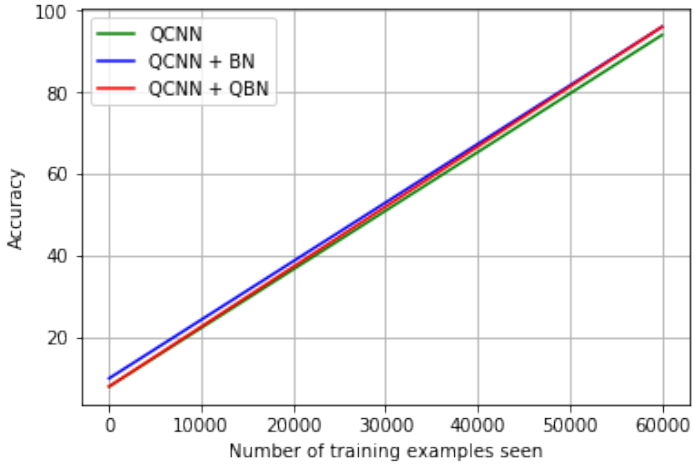

Figure 4.12. Comparison among QBN (red), BN (blue) and base case (green) for the MNIST dataset. Hyper-parameters: $\eta = 0.001$, mini-batch size = 400, 1 epoch.

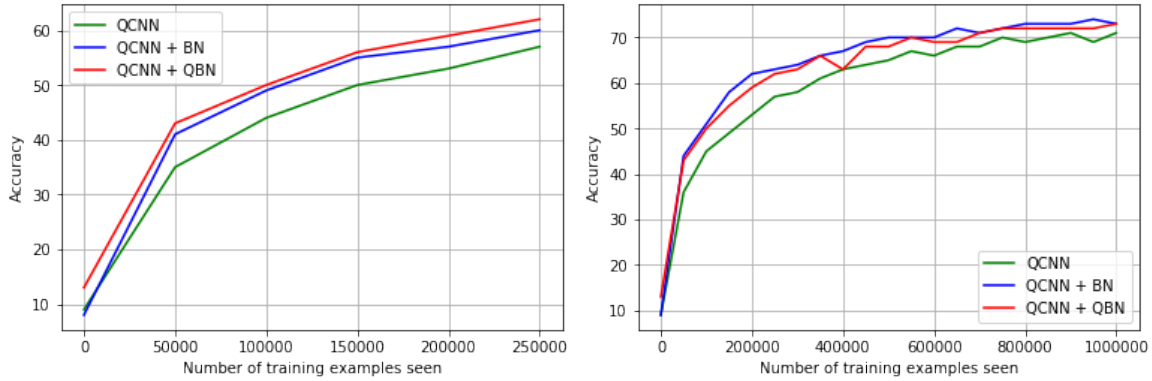

Figure 4.13. Comparison among QBN (red), BN (blue) and base case (green) for the CIFAR-10 dataset. Hyper-parameters: $\eta = 0.001$, mini-batch size = 400 and 5 and 20 epochs respectively.

We can note immediately that, although we saw a performance increase by using QBN in respect to BN in the case of MNIST, the dataset is too simple for doing a deeper analysis. Indeed after only an epoch it reaches almost a 99% accuracy when using one of the two batch normalization techniques. This can move the accuracy "bottleneck" on the architecture of the network itself.

Hence, we will concentrate only on testing the techniques on the CIFAR-10 dataset. The results obtained are summed up by the following table:

Experiment	# Parameters	Test Accuracy [%]	Training time [secs]
QCNN	474,920	71.23	219.26
QCNN + BN	475,888	73.07	219.59
QCNN + QBN	475,525	74.13	233.65

Table 4.8. Test effectiveness of QBN against BN without any regularization on CIFAR-10. Hyper-parameters: $\eta = 0.001$, mini-batch size = 400, 20 epochs.

In terms of absolute accuracy, QBN seems to be slight better than BN, but we should make more testing to be sure of this advantage. Because of this thesis is focused on compression techniques, we will focus on using QBN as a regularizer in the next section.

Maybe, the main pros of QBN is in its robustness against the shrinking of the batch size in comparison to its real counterpart. Indeed, given the same hyper-parameters and shrinking the batch-size of CIFAR-10 (2X) for testing purpose, we have:

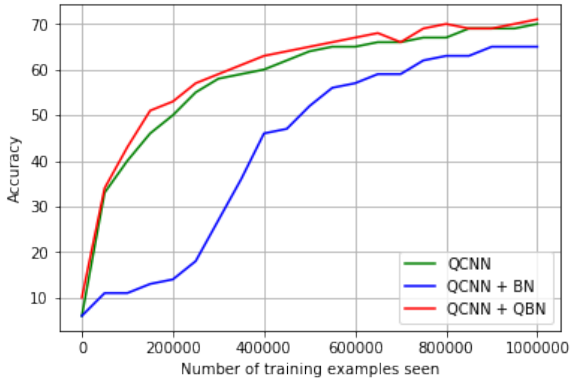

Figure 4.14. Comparison among QBN, BN and base case for the CIFAR-10 dataset with a restricted batch size (200), $\eta = 0.001$, 20 epochs.

Experiment	Test Accuracy [%]	Training time [secs]
QCNN	70.37	230.20
QCNN + BN	65.95	235.88
QCNN + QBN	71.01	272.54

Table 4.9. Summary of the results for QBN and BN when batch size is halved for CIFAR-10 dataset. Hyper-parameters: $\eta = 0.001$, mini-batch size = 200, 20 epochs.

This fact can still be explained with the quaternion's algebra properties.

Indeed, the traditional scalar mean of the classical Batch Normalization is substituted by the vectorial mean in the QBN. The variance is still a real number, but its calculus benefits from the more robust mean. The result is that the QBN layer, can estimates sufficiently well mean and variance of the dataset population also when the batch-size is reduced.

Hence, also in this case the quaternion algebra helps the network to retain useful information. A similar case can be found in the field of Compresses Sensing in which the "block sparsity" of quaternion sparse signals helps in increasing the recovery rate in respect to real case, as illustrated by [50, 51].

4.4.2 Quaternion Batch Normalization as a regularizer

As pointed out in the theoretical part, applying an L_1 regularization to the γ parameters of the QBN layers could lead to a sparse network, in the sense of group/structured sparsity. It could be even more effective of R_Q and R_{QL}, indeed regularizing γ means regularizing entire channels/neurons and this could lead to delete entire parts of the networks in few iterations. Hence, we will measure also neuron sparsity in this section.

Given this introduction, we can set two main experiments:

1. Comparison between *Quaternion sparse regularization (R_Q)* and L_1 regularization of the γ parameters of QBN. The latter will be denoted simply by $L_1(\gamma)$;
2. Comparison between *Sparse Quaternion Lasso regularization (R_{QL})* and L_1 regularization applied both to γ parameters of QBN and to the weights of the network. The new and last form of regularization will be denoted as $L_1(w+\gamma)$.

In the testing of R_{QL} and R_Q one wonders if it is fair to test them with or without Quaternion Batch Norm. For completeness, both these variants will be presented.

When R_{QL} (that contains also the L_1 regularization term) is used in a network with QBN, the γ parameter is not regularized, for the same reasons of experimental fairness. Each experiment has been performed three times, hence plots and tables of results presented are averaged.

From experiments, it emerged that a good initialization of γ is essential to the effectiveness of regularization (and accuracy). Without loss of generality, we found that a good compromise for accuracy and sparsity was initialize $\gamma = 0.1$ both for $L_1(\gamma)$ and $L_1(w+\gamma)$ and for $QBN + R_{QL}$ and $QBN + R_Q$.

Other parameters and hyper-parameters used are summed up by the following table:

Dataset	**# Parameters**	**Training epochs**	$\boldsymbol{\eta}$	**Mini-batch size**
MNIST	~12K	20	0.001	400
CIFAR-10	~475K	50	0.001	400

Table 4.10. Parameters and Hyper-parameters used for experiments in this section

Finally, the regularization factors that were found effective for each regularization function and dataset tested:

Dataset	**Regularization**	$\boldsymbol{\lambda}$
MNIST	R_Q	$7.5 \cdot 10^{-3}$
	R_{QL}	$2.7 \cdot 10^{-3}$
	$QBN + R_Q$	$8 \cdot 10^{-3}$
	$QBN + R_{QL}$	$6 \cdot 10^{-3}$
	$L_1(\gamma)$	2.5
	$L_1(w+\gamma)$	$1.5 \cdot 10^{-2}$

Dataset	**Regularization**	$\boldsymbol{\lambda}$
CIFAR-10	R_Q	$5.3 \cdot 10^{-6}$
	R_{QL}	$3.4 \cdot 10^{-6}$
	$QBN + R_Q$	$4 \cdot 10^{-4}$
	$QBN + R_{QL}$	$2.5 \cdot 10^{-4}$
	$L_1(\gamma)$	$9 \cdot 10^{-2}$
	$L_1(w+\gamma)$	$4.5 \cdot 10^{-4}$

Table 4.11. Regularization factor for each regularization function and dataset

1° experiment

In the following the R_Q regularization will be tested in the variant with QBN.

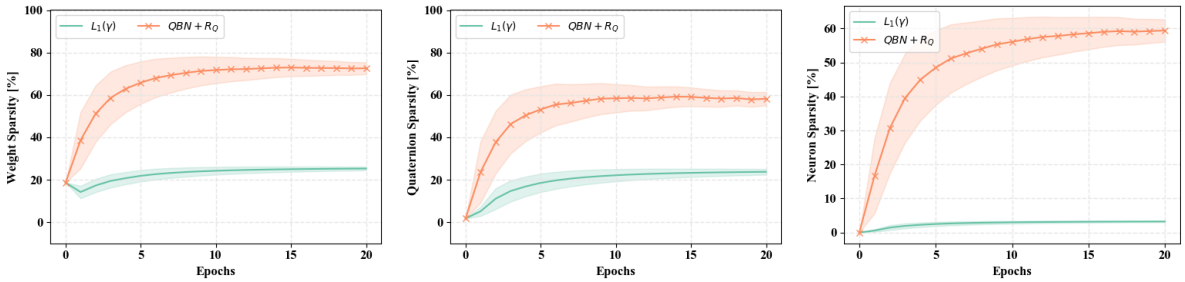

Figure 4.15. Comparison between QBN+R_Q and QBN+$L_1(\gamma)$ for MNIST

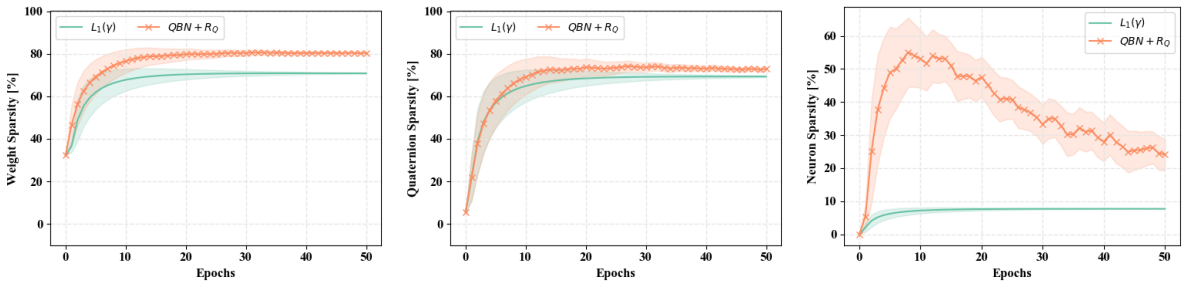

Figure 4.16. Comparison between QBN+R_Q and QBN+$L_1(\gamma)$ for CIFAR-10

Dataset	Measure	QBN + R_Q	QBN + $L_1(\gamma)$
MNIST	Test accuracy [%]	97.29	96.69
	Training time [secs]	192.51	233.89
	Weight Sparsity [%]	72.91	25.53
	Quaternion Sparsity [%]	60.08	24.14
	Neuron Sparsity [%]	60.38	3.30
CIFAR-10	Test accuracy [%]	73.97	67.66
	Training time [secs]	1344.49	1343.00
	Weight Sparsity [%]	81.60	70.72
	Quaternion Sparsity [%]	75.89	69.25
	Neuron Sparsity [%]	21.27	7.71

Table 4.12. Average results obtained when comparing $QBN + R_Q$ and QBN+$L_1(\gamma)$

In the following the R_Q regularization will be tested without QBN.

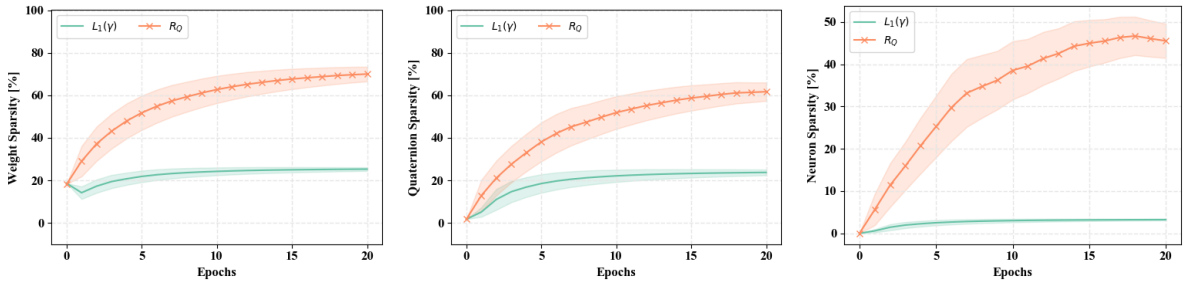

Figure 4.17. Comparison between R_Q and QBN+$L_1(\gamma)$ for MNIST

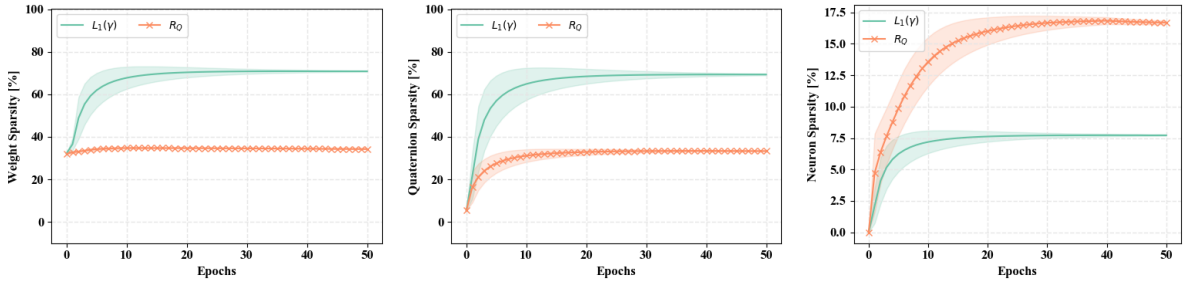

Figure 4.18. Comparison between R_Q and QBN+$L_1(\gamma)$ for CIFAR-10

Dataset	Measure	$\mathbf{R_Q}$	QBN + $\mathbf{L_1(\gamma)}$
MNIST	Test accuracy [%]	96.67	96.69
	Training time [secs]	180.13	233.89
	Weight Sparsity [%]	71.97	25.53
	Quaternion Sparsity [%]	63.60	24.14
	Neuron Sparsity [%]	42.14	3.30
CIFAR-10	Test accuracy [%]	76.36	67.66
	Training time [secs]	1156.23	1343.00
	Weight Sparsity [%]	34.18	70.72
	Quaternion Sparsity [%]	33.26	69.25
	Neuron Sparsity [%]	16.65	7.71

Table 4.13. Average results obtained when comparing R_Q and QBN+$L_1(\gamma)$

2° experiment

In the following the R_{QL} regularization will be tested in the variant with QBN.

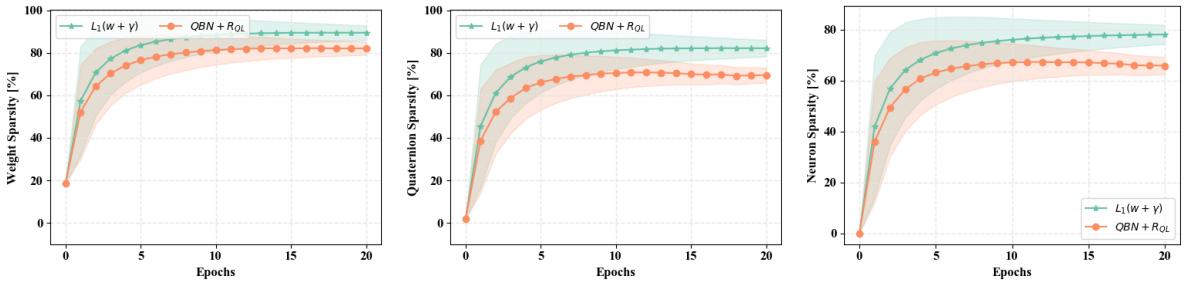

Figure 4.19. Comparison between QBN+R_{QL} and QBN+$L_1(w+\gamma)$ for MNIST

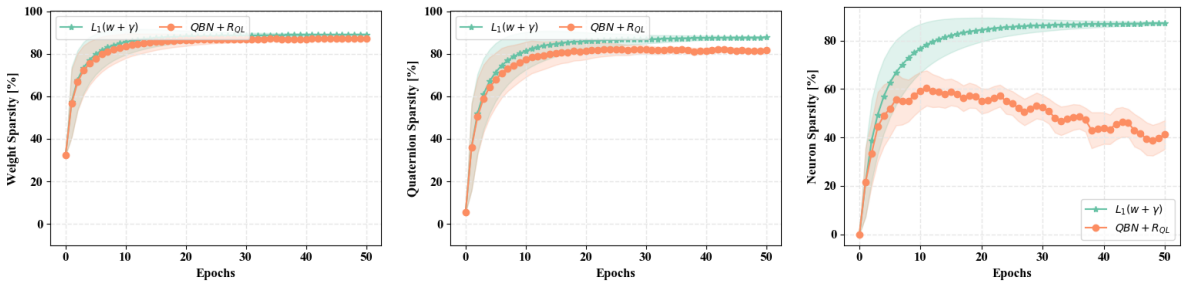

Figure 4.20. Comparison between QBN+R_{QL} and QBN+$L_1(w+\gamma)$ for CIFAR-10

Dataset	Measure	QBN + $\mathbf{R_{QL}}$	QBN + $\mathbf{L_1}(w+\gamma)$
MNIST	Test accuracy [%]	95.98	95.23
	Training time [secs]	302.78	299.19
	Weight Sparsity [%]	81.90	89.13
	Quaternion Sparsity [%]	70.17	81.86
	Neuron Sparsity [%]	65.25	78.62
CIFAR-10	Test accuracy [%]	71.61	71.38
	Training time [secs]	1372.04	1344.00
	Weight Sparsity [%]	87.31	89.06
	Quaternion Sparsity [%]	82.42	87.72
	Neuron Sparsity [%]	50.06	87.22

Table 4.14. Average results obtained when comparing $QBN + R_{QL}$ and QBN+$L_1(w+\gamma)$

In the following the R_{QL} regularization will be tested without QBN.

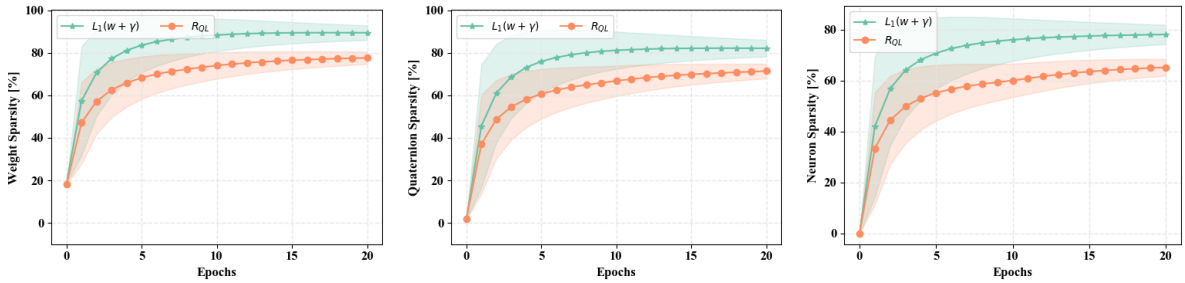

Figure 4.21. Comparison between R_{QL} and QBN+$L_1(w+\gamma)$ for MNIST

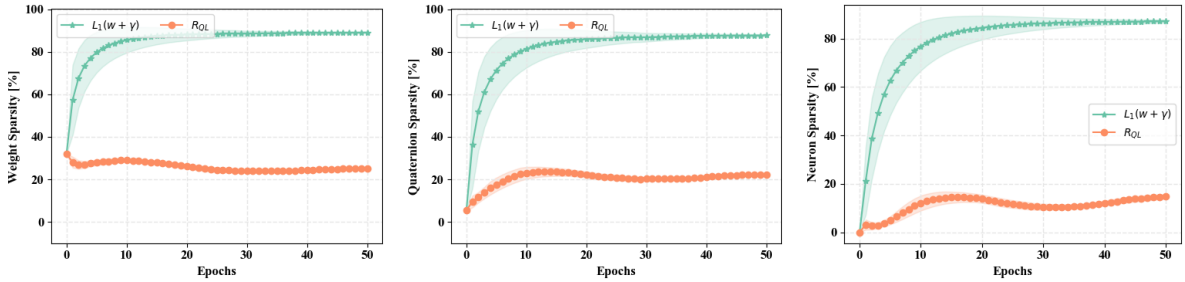

Figure 4.22. Comparison between R_{QL} and QBN+$L_1(w+\gamma)$ for CIFAR-10

Dataset	Measure	$\mathbf{R_{QL}}$	QBN + $\mathbf{L_1}(w+\gamma)$
MNIST	Test accuracy [%]	96.27	95.23
	Training time [secs]	278.38	299.19
	Weight Sparsity [%]	78.62	89.13
	Quaternion Sparsity [%]	73.05	81.86
	Neuron Sparsity [%]	66.67	78.62
CIFAR-10	Test accuracy [%]	76.54	71.38
	Training time [secs]	1164.71	1344.00
	Weight Sparsity [%]	24.52	89.06
	Quaternion Sparsity [%]	22.00	87.72
	Neuron Sparsity [%]	15.52	87.22

Table 4.15. Average results obtained when comparing R_{QL} and QBN+$L_1(w+\gamma)$

The first thing to note is that QBN improve also effectiveness of R_Q and R_{QL} in the case of CIFAR-10.

Indeed, sparsity performance, both for weight sparsity and for the stricter quaternion one, are greater or at least equal to the case in which we avoid the use of the batch normalization in every case. For MNIST this effect is less evident, and we have cases in which QBN improves effectiveness of R_Q and R_{QL} in sparsifying the weights of the network and others in which pure R_Q and R_{QL} have slightly greater performance quaternion and neuron sparsity metrics.

We note also that $L_1(\gamma)$, that regularizes only the γ parameters is not very effective in sparsify NNs.

On the other hand, it is interesting that even confronting the best-case scenario for R_Q and R_{QL} with $L_1(w+\gamma)$ (that regularizes the γ parameters and the weights of the networks), the latter outperforms all the previous regularization techniques by a great margin and is also more elegant from a mathematical point of view.

Notably, $L_1(w+\gamma)$ exceeds the other techniques in all metrics (weight, quaternion, neuron) in both the datasets tested. In the case of MNIST, $L_1(w+\gamma)$ surpasses the second-best regularization $QBN + R_{QL}$ of 7.23%, 11.69% and 13.37% for weight, quaternion and neuron sparsity, respectively, while maintaining a comparable accuracy. Also with the CIFAR-10 dataset, it manages to reach an average of 87.22% neuron sparsity losing only 0.23% of accuracy in respect of $QBN + R_{QL}$ that reach however only 50.06% neuron sparsity.

In conclusion, we found that QBN can effectively be used as a regularizer as long as a good initialization of the hyper-parameter γ is performed while also putting a loss on the weights of the network.

Regularizing only γ, as in the case of $L_1(\gamma)$ seems to be not very effective to sparsify the network, rather it hinder the learning process by flatten the input layer to 0 (we remember that γ is the stretch scale that will be multiplied to the input).

If we consider also the other advantages that QBN allows in terms of speeding-up learning and performances, we could think of using a "Swiss-army-knife" technique that also regularizes and shrinks the network.

Probably the most outstanding result is in the plot of neuron sparsity in the case of the CIFAR-10 dataset: only $L_1(w+\gamma)$ manages to sparsify neurons, $QBN+R_Q$ and $QBN + R_{QL}$ are good to sparsify weights/quaternions but the number of neurons that are contributing little continues to oscillates very sharply. This phenomenon can be better seen in a plot without smoothing. Hence, for completeness, in the next page there is a summing up with weight sparsity, quaternion sparsity, neuron sparsity and accuracy for all the regularizations proposed. The following plots are showed as raw as possible, without smoothing, to enhance their real behaviour and patterns.

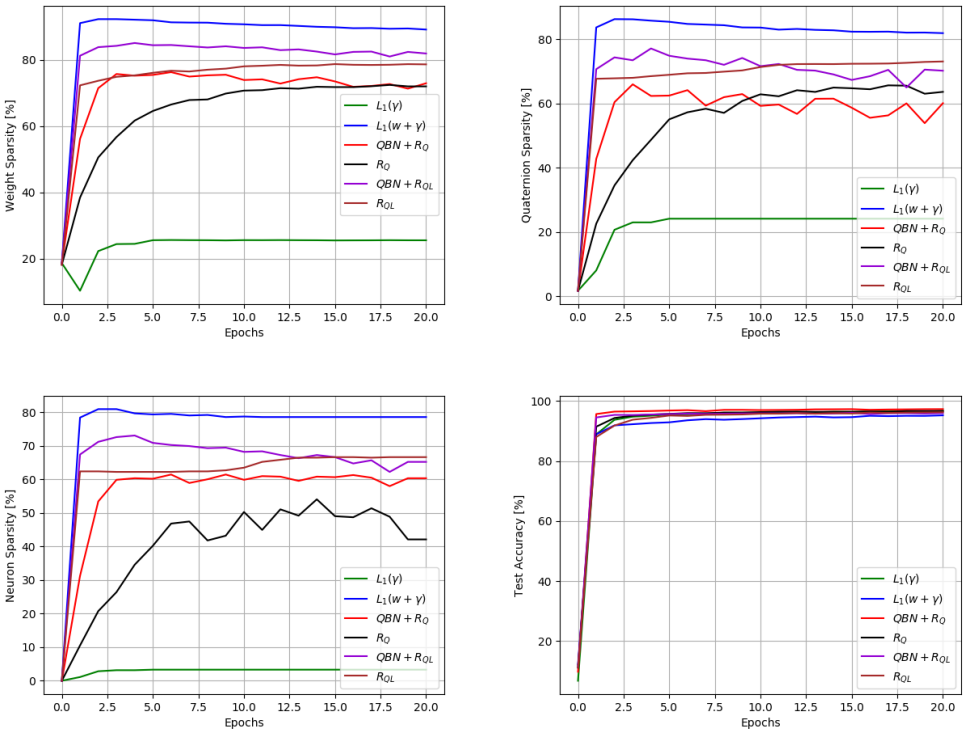

Figure 4.23. Average results for all regularizations proposed on the MNIST dataset

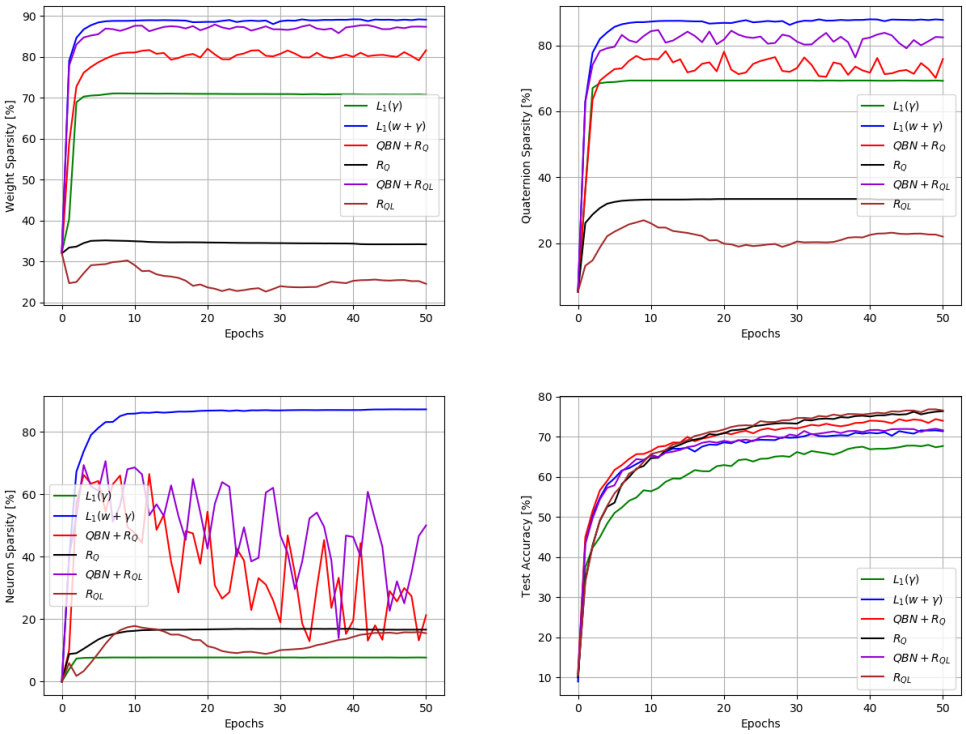

Figure 4.24. Average results for all regularizations proposed on the CIFAR-10 dataset

4.5 Deep compression pipeline evaluation

Deep compression is a very powerful technique, but requires a lot of tuning, given the many hyper-parameters it has. Indeed, in addition to the standard learning rate η used in training, there is an extra learning rate η_C for the centroid's fine tuning, the number of iterations for this last step, the number of bits desired for the weight sharing, and obviously the regularization function and relative multiplication factor that are essential for the training and pruning steps.

Technical details and implementation can be found in the online library presented in the section 4.1. As in the case of the Deep Compression paper, also the library will populate and manipulates a (simplified) codebook structure. It uses a format of file that tries to have the highest entropy possible.

The experiments have been conducted on the networks for MNIST and CIFAR-10 previously presented.

In the table below there are also the other hyper-parameters needed to specify exactly the Deep Compression pipeline. Only the number of bits used in the weight sharing will be omitted because will be varied in the experiments. Given the many combinations possible, other hyper-parameters are presented as-is because they were found to work the best at testing time.

Given the consistent performance across all datasets studied, the regularizer chosen for pruning the network of the inexpensive connection is the *Quaternion sparse regularization* R_Q. Regularizations that arise from the exploits of parameters of the QBN are not considered for the ambiguity on how to treat the QBN layer in the codebook in the case of quaternion weight sharing and for avoiding to introduce the Batch Normalization that could disrupt too match the original pipeline.

Dataset	# Parameters	Training epochs	η	η_C	λ_{RQ}	Batch size
MNIST	11,688	10	0.002	0.0001	$7.5 \cdot 10^{-3}$	400
CIFAR-10	474,920	30	0.002	0.0001	$5.3 \cdot 10^{-6}$	400

Table 4.16. Parameters and Hyper-parameters for the Deep Compression Experiments

Naïve weight sharing - 3 bits

Dataset	After...	Test Accuracy [%]	Parameters	Compression Rate
MNIST	Training	96.47	45.1KB	-
	Pruning	96.46	18.4KB	2.45X
	Weight Sharing	95.16	5.72KB	7.88X
	Fine Tuning	95.41	5.72KB	7.88X
CIFAR-10	Training	70.74	1.72MB	-
	Pruning	70.74	1.24 MB	1.38X
	Weight Sharing	65.09	212 KB	8.07X
	Fine Tuning	68.81	212 KB	8.07X

Table 4.17. Summary of the results for the deep compression pipeline. Every weight $\boldsymbol{w}$ such that $|\boldsymbol{w}| < 10^{-3}$ is pruned. Naïve weight sharing with each component of quaternion seen as a separate layer.

Naïve weight sharing - 2 bits

Dataset	After...	Test Accuracy [%]	Parameters	Compression Rate
MNIST	Training	96.89	44.4KB	-
	Pruning	96.90	19.8KB	2.24X
	Weight Sharing	91.78	4.68KB	9.49X
	Fine Tuning	93.94	4.68KB	9.49X
CIFAR-10	Training	71.14	1.68MB	-
	Pruning	71.16	977KB	1.72X
	Weight Sharing	40.57	167KB	10.06X
	Fine Tuning	64.32	167KB	10.06X

Table 4.18. Summary of the results for the deep compression pipeline. Every weight $\boldsymbol{w}$ such that $|\boldsymbol{w}| < 10^{-3}$ is pruned. Naïve weight sharing with each component of quaternion seen as a separate layer.

From the summary tables above, we can see that the Deep compression pipeline is very effective in compressing the quaternion networks. In our experiments we reached up to 10X compression from the original trained network. This is beneficial not only for storage, but it can also be exploited by specialized hardware to have both a consistent speed-up and a greater energy efficiency.

In the case of MNIST is simpler to lower the bits used in the quantization, thanks to the more effective regularization, which in turn is due to the simplicity of the problem.

Quaternion weight sharing - 2 bits

As second experiment, it is interesting to evaluate what happen if we use the "quaternion weight sharing". In this case we will compare only the accuracy of the naïve weight sharing and the quaternion one for seeing if we can retain useful information when using the latter. This means that fixing the number of bits, we will have effectively 4X less weights available in respect to the classical weight sharing with the same level of quantization. We have the following results:

Dataset	After...	Test Accuracy [%]	Parameters	Compression Rate
MNIST	Training	96.89	44.4KB	-
	Naïve Weight Sharing	91.78	4.68KB	9.49X
	Quaternion Weight Sharing	89.11	4.65KB	9.50X
CIFAR-10	Training	70.18	1.69MB	-
	Naïve Weight Sharing	41.56	167KB	10.06X
	Quaternion Weight Sharing	39.50	170KB	10.05X

Table 4.19. Summary of the results for the deep compression pipeline. Every weight $\boldsymbol{w}$ such that $|\boldsymbol{w}| < 10^{-3}$ is pruned. Quaternion weight sharing

Although the quaternion weight sharing loses a bit of accuracy in respect to the naïve version, we have to remember that we used the same number of bits, although the quaternion one have to fit 4X more parameter in a single centroid. Hence it is a more appropriate quantization method for quaternion networks. Probably a good compromise between accuracy and compression is allow a slightly greater number of bits for quantization in the case of quaternion weight sharing in respect to the standard one. In that case, the quaternion version could outperform the standard one both in accuracy and compression figures.

Chapter 5

Conclusions

This thesis introduces several techniques to extends the benefits of quaternion-valued Neural Networks to as many applications as possible, following the work done with complex neural networks and real ones.

Two new forms of regularization have been presented (R_Q and R_{QL}), specifically thought for QNN, together with the implementation and evaluation of both Quaternion Batch Normalization and Deep Compression pipeline, that have been studied also in non-conventional manners.

In particular, Quaternion Batch Normalization has been used also as a regularization layer, by adding to the standard loss function a regularization term for the γ parameter (stretch scale) of the QBN layer. This corresponds to try to remove entire channels/neurons that contributes almost nothing to the overall performance. These additional regularizations have been called $L_1(\gamma)$ and $L_1(w + \gamma)$.

Furthermore, after having tested successfully the Deep Compression pipeline for quaternion networks, a new form of weight sharing has been tested for them, namely we tried to adapt weight sharing considering the quaternion nature of the model to see if this can somewhat benefits the compression task.

The findings are that both the regularization functions R_Q and R_{QL} outperforms the classical L_1 and L_2 by a large margin. Indeed, also if we consider only the weight sparsity metrics (not quaternions), these two regularization functions still perform better than the others. In particular, R_{QL} seems to be better than R_Q in all situation, except in the case of the CIFAR-10 dataset.

For which concern the quaternion-based weight sharing, we found that it retains more information than the standard one, although further testing should be done.

Last but not least, we found that $L_1(\gamma)$ is not very effective, while $L_1(w + \gamma)$ set the state-of-the-art regarding QNN sparsification, dwarfing the results obtained with R_Q and R_{QL}, probably thanks to its "neuron-level effect" coupled with an L_1 regularization over weights.

A lot of work still needs to be done to optimize and assess quaternion nets capabilities. Moreover, as a future research path, it would be interesting to continue the trend to extends neural networks weights and operations to more complex structures (e.g. octonions[52] and sedenions) and see if there can be further benefits to extends them to more sophisticated algebras.

Appendix A

Quaternion Algebra

Quaternions are elements of the hyper-complex space $\mathbb{H}$, a direct extension of the complex numbers. They were introduced by William Rowan Hamilton in 1843[53].

They are useful in several applications like pure and applied mathematics, engineering, computer graphics, physics, bioinformatics, quantum mechanics and so on[54, 55, 56, 57].

One reason of their versatility is due to quaternion's capacity to representing 3D space rotation. In computer graphics, for instance, quaternions are inherently superior in representing 3D rotations/orientation, indeed they are more compact (less storage required) and need fewer calculations in respect to rotation matrices. Also, a rotation represented by quaternions is immune from the gimbal lock problem (the loss of one degree of freedom in a three-dimensional) that affects other form of representations, like Euler angles.

A quaternion $q \in \mathbb{H}$ can be represented in the following form:

$$q = a + b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k} \tag{A.1}$$

in which $a, b, c, d \in \mathbb{R}$ and i, j, k are three imaginary units compared to the one of the complex numbers. The symbols i, j, k can be interpreted as unit-vectors pointing along the three spatial axes.

The cost of generalizing the complex field to the hyper-complex one leads to the losing of the commutative property for multiplication, namely $q_1 q_2 \neq q_2 q_1$ in general, while associative and distributive laws continue to apply.

Non commutativity is summed up by the following multiplication table:

$\boldsymbol{x}$	$\mathbf{1}$	$\boldsymbol{i}$	$\boldsymbol{j}$	$\boldsymbol{k}$
$\mathbf{1}$	1	i	j	k
$\boldsymbol{i}$	i	-1	k	$-j$
$\boldsymbol{j}$	j	$-k$	-1	i
$\boldsymbol{k}$	k	j	$-i$	-1

Table A.1. Multiplication table

Famous is the formula carved by Hamilton into the stone of Brougham Bridge:

$$\boldsymbol{i^2} = \boldsymbol{j^2} = \boldsymbol{k^2} = \boldsymbol{ijk} = -1 \tag{A.2}$$

A quaternion can be divided into scalar part $\mathcal{S}(q) = a$ and vectorial part $\mathcal{V}(q) = b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k}$.

The conjugate of a quaternion q is defined as:

$$q^* = \mathcal{S}(q) - \mathcal{V}(q) = a - b\boldsymbol{i} - c\boldsymbol{j} - d\boldsymbol{k} \tag{A.3}$$

We can express the norm of a quaternion with:

$$\|q\| = \sqrt{qq^*} = \sqrt{q^*q} = \sqrt{a^2 + b^2 + c^2 + d^2} \tag{A.4}$$

Having defined the norm of a quaternion, we can define its inverse as:

$$q^{-1} = \frac{q^*}{||q||^2} \tag{A.5}$$

Moreover, it is important to define the Hamilton product for two quaternions q_1, q_2. It can be defined as:

$$\begin{aligned} q_1 \bigotimes q_2 = {} & a_1a_2 + a_1b_2i + a_1c_2j + a_1d_2k \\ & + b_1a_2i + b_1b_2i^2 + b_1c_2ij + b_1d_2ik \\ & + c_1a_2j + c_1b_2ji + c_1c_2j^2 + c_1d_2jk \\ & + d_1a_2k + d_1b_2ki + d_1c_2kj + d_1d_2k^2 \end{aligned} \tag{A.6}$$

It is also interesting to note that quaternion numbers can be represented in at least two ways: with $2x2$ complex matrices or with $4x4$ real matrices.

Since it is common to perform quaternion arithmetic with real numbers, it is useful to embed $\mathbb{H}$ into a real-valued representation: an injective homomorphism from $\mathbb{H}$ to the matrix ring $M(4, \mathbb{R})$ where $M(4, \mathbb{R})$ is a $4x4$ matrix $\in \mathbb{R}$.

Hence, a possible (not unique) representation using a $4x4$ real matrix for the quaternion number $q = a + b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k}$ is the following:

$$\begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix} =$$
$$a \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} + b \begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix} + c \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} + d \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{A.7}$$

A unit quaternion (also called versor) is a quaternion of norm 1 that provides a good proxy for representing rotations and orientation in three dimensional spaces. A classic vector $v \in \mathbb{R}^3$ can be seen as a pure quaternion (the real part is zero). Hence a convenient mapping from vectors and quaternions is the following:

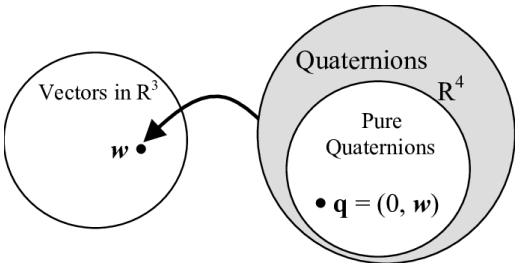

Figure A.1. One-to-one correspondence between pure quaternions and vectors in $\mathbb{R}^3$

Let consider a unit quaternion as $q = q_0 + \boldsymbol{q}$. This implies that $q_0^2 + ||\boldsymbol{q}||^2 = 1$. Hence, there exists some angle θ such that:

$$1. \cos^2\theta = q_0^2,$$
$$2. \sin^2\theta = ||\boldsymbol{q}||^2$$

This $\theta \in [0, \pi]$ is unique and we have that $\cos\theta = q_0$ and $\sin\theta = ||\boldsymbol{q}||$. Hence q can be expressed in terms of the unit vector $\boldsymbol{u} = \frac{\boldsymbol{q}}{||\boldsymbol{q}||}$:

$$q = \cos\theta + \boldsymbol{u}\sin\theta \tag{A.8}$$

Let define an operator on vectors $v \in \mathbb{R}^3$ by using the unit quaternion q:

$$L_q(\boldsymbol{v}) = q\boldsymbol{v}q^* = \left(q_0^2 - ||\boldsymbol{q}||^2\right)\boldsymbol{v} + 2\left(\boldsymbol{q}\cdot\boldsymbol{v}\right)\boldsymbol{q} + 2q_0\left(\boldsymbol{q}\times\boldsymbol{v}\right) \tag{A.9}$$

Hence for describing pure rotations with quaternions we can use the theorem[58]:

Theorem

For any unit quaternion

$$q = q_0 + \boldsymbol{q} = \cos\frac{\theta}{2} + \boldsymbol{u}sin\frac{\theta}{2} \tag{A.10}$$

and for any vector $v \in \mathbb{R}^3$ the action of the operator

$$L_q(\boldsymbol{v}) = q\boldsymbol{v}q^* \tag{A.11}$$

on $\mathbf{v}$ is equivalent to a rotation of the vector through an angle θ about $\boldsymbol{u}$ as the axis of rotation.

Bibliography

[1] Stuart J. Russell, Peter Norvig, "Artificial Intelligence: A Modern Approach, Third Edition*", Prentice Hall, 2010*

[2] P. Dayan., "Unsupervised learning", *The MIT Encyclopedia of the Cognitive Science, 1999.*

[3] van Otterlo, M.; Wiering, M. "Reinforcement learning and markov decision processes. Reinforcement Learning. Adaptation, Learning, and Optimization", *ISBN 978-3-642-27644-6, (2012)*

[4] Kaelbling, Leslie P.; Littman, Michael L.; Moore, Andrew W, "Reinforcement Learning: A Survey", *Journal of Artificial Intelligence Research. 4: 237–285, 1996*

[5] Krizhevsky, Alex; Sutskever, Ilya; Hinton, Geoffry, "ImageNet Classification with Deep Convolutional Neural Networks", *NIPS Neural Information Processing Systems, Lake Tahoe, Nevada, 2012*

[6] Ciresan, Dan; Meier, U.; Schmidhuber, J., "Multi-column deep neural networks for image classification", *IEEE Conference on Computer Vision and Pattern Recognition: 3642–3649. arXiv:1202.2745 (2012)*

[7] Silver, David; Huang, Aja; Maddison, Chris J.; Guez, Arthur; Sifre, Laurent; Driessche, George van den; Schrittwieser, Julian; Antonoglou, Ioannis; Panneershelvam, Veda; Lanctot, Marc; Dieleman, Sander; Grewe, Dominik; Nham, John; Kalchbrenner, Nal; Sutskever, Ilya; Lillicrap, Timothy; Leach, Madeleine; Kavukcuoglu, Koray; Graepel, Thore; Hassabis, Demis "Mastering the game of Go with deep neural networks and tree search", *Nature. 529 (7587): 484–489, 2016*

[8] B. Widrow, M. A. Lehr, "30 years of adaptive neural networks: Perceptrons madaline and backpropagation", *Proc. IEEE, vol. 78, no. 9, pp. 1415-1442, (1990)*

[9] A. K. Jain, J. Mao and K. M. Mohiuddin, "Artificial neural networks: A tutorial", *Computer, vol. 29, no. 3, pp. 31-44, (1996)*

[10] D.E. Rumelhart & E, J.L. Mcclelland, and the PDP Research Group, "Parallel Distributed Processing (PDP): Explorations in the Microstructure of Cognition", *Volume 1, MIT Press, Cambridge, Massachusetts (1986)*

[11] John von Neumann, "The Computer and The Brain", *New Haven and London Yale University press, ISBN-13: 978-0300181111, 1958*

[12] W.S. McCulloch and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity", *Bull. Mathematical Biophysics, Vol, 5, pp. 115-133, 1943*

[13] Miljanovic and Milos, "Comparative analysis of Recurrent and Finite Impulse Response Neural Networks in Time Series Prediction", *Indian Journal of Computer and Engineering. 3 (1), 2012*

[14] Li, Xiangang; Wu, Xihong, "Constructing Long Short-Term Memory based Deep Recurrent Neural Networks for Large Vocabulary Speech Recognition", *arXiv:1410.4281, 2014*

[15] G. Cybenko, "Approximation by Superposition of a Sigmoidal Function", *Math. Control Signals Systems, No.2, Springer-Verlag, New York (1989)*

[16] Vinod Nair and Geoffrey E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines" *ICML, 2010*

[17] Geron A. "Hands-On Machine Learning with Scikit-Learn and TensorFlow", *O'Reilly, 2017*

[18] LeCun, Yann; Bengio, Yoshua; Hinton, Geoffrey "Deep learning", *Nature. 521 (7553): 436–444, 2015*

[19] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning representations by back-propagating errors", *Nature, 323, 533-536, (1986)*

[20] Schmidhuber, Jürgen "Deep learning in neural networks: An overview", *Neural Networks. 61: 85–117. arXiv:1404.7828 (2015)*

[21] Yann LeCun, Leon Bottou, Genevieve B. Orr and Klaus-Robert Müller, "Efficient BackProp", *Neural Networks: Tricks of the Trade, pages 9–50. Springer, 1998*

[22] Definition of "overfitting" in statistics, *Oxford Dictionary*

[23] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever and Ruslan R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors", *arXiv:1207.0580v1, (2012)*

[24] Hubel, D. H.; Wiesel, T. N. "Receptive fields and functional architecture of monkey striate cortex", *The Journal of Physiology. 195 (1): 215–243. doi:10.1113/jphysiol.1968.sp008455* (1968)

[25] Fukushima, Kunihiko "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position", *Biological Cybernetics. 36 (4): 193–202 (1980)*

[26] Chiheb Trabelsi, Olexa Bilaniuk, Ying Zhang, Dmitriy Serdyuk, Sandeep Subramanian, João Felipe Santos, Soroush Mehri, Negar Rostamzadeh, Yoshua Bengio and Christopher Joseph Pal, "Deep Complex Networks", *ICLR 2018*

[27] Moritz Wolter and Angela Yao, "Complex gated recurrent neural networks", *Proceeding NIPS'18 Proceedings of the 32nd International Conference on Neural Information Processing Systems Pages 10557-10567, 2018*

[28] Chase Gaudet and Anthony Maida, "Deep Quaternion Networks" *arXiv:1712.04604v3, 2018*

[29] Xuanyu Zhu, Yi Xu, Hongteng Xu and Changjian Chen, "Quaternion Convolutional Neural Networks" a*rXiv:1903.00658v1, 2019*

[30] Xu, Y., Yu, L., Xu, H., Zhang, H., Nguyen, T., "Vector sparse representation of color image using quaternion matrix analysis", *IEEE Transactions on Image Processing 24(4), 1315–1329 (2015)*

[31] Simone Scardapane, Danilo Comminiello, Amir Hussain and Aurelio Uncini, "Group Sparse Regularization for Deep Neural Networks", *arXiv:1607.00485v1, 2016*

[32] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, Hai Li, "Learning Structured Sparsity in Deep Neural Networks", *arXiv:1608.03665v4, 2016*

[33] Ioffe, Sergey, Szegedy and Christian, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" *arXiv:1502.03167, 2015*

[34] J. Wang, T. Li, X. Luo, Y.-Q. Shi, R. Liu, and S. K. Jha, "Identifying computer generated images based on quaternion central moments in color quaternion wavelet domain" *IEEE Trans. Circuits Syst. Video Technol., to be published. doi: 10.1109/TCSVT.2018.2867786 (2018)*

[35] Yin, Qilin & Wang, Jinwei & Luo, Xiangyang & Zhai, Jiangtao & Jha, Sunil & Shi, Y.Q.. "Quaternion Convolutional Neural Network for Color Image Classification and Forensics". *IEEE Access. PP. 1-1. 10.1109/ACCESS.2019.2897000. (2019)*

[36] Gordon, Ariel D, Elad Eban, Ofir Nachum, Bo Chen, Tien-Ju Yang and Edward Choi, "MorphNet: Fast & Simple Resource-Constrained Structure Learning of Deep Networks." *IEEE/CVF Conference on Computer Vision and Pattern Recognition (2018): 1586-1595. (2018)*

[37] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan and Changshui Zhang, "Learning Efficient Convolutional Networks through Network Slimming" *IEEE International Conference on Computer Vision (ICCV) (2017)*

[38] Song Han, Huizi Mao and William J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding" *arXiv:1510.00149v5, 2016*

[39] Iandola, Forrest N; Han, Song; Moskewicz, Matthew W; Ashraf, Khalid; Dally, William J; Keutzer, Kurt "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size" *arXiv:1602.07360 (2016)*

[40] Oh, K.-S.; Jung, K. "GPU implementation of neural networks", *Pattern Recognition 37 (6): 1311–1314 (2004)*

[41] Chellapilla, K., Puri, S., and Simard, P.,"High performance convolutional neural networks for document processing", *International Workshop on Frontiers in Handwriting Recognition, 2009*

[42] Raina, Rajat; Madhavan, Anand; Ng, Andrew Y., "Large-scale Deep Unsupervised Learning Using Graphics Processors", *Proceedings of the 26th Annual International Conference on Machine Learning. ICML 2009*

[43] Sze, Vivienne; Chen, Yu-Hsin; Yang, Tien-Ju; Emer, Joel, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey" *arXiv:1703.09039, 2017*

[44] Titouan Parcollet, *https://github.com/Orkis-Research/Pytorch-Quaternion-Neural-Networks, 2018*

[45] T. Parcollet, M. Morchid and G. Linarès, "Quaternion Convolutional Neural Networks for Heterogeneous Image Processing", *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 8514-8518, Brighton, United Kingdom, 2019*

[46] Zhou Wang; Alan C. Bovik, "Modern Image Quality Assessment", *Morgan & Claypool Publishers ISBN-13: 978-1598290226, 2006*

[47] Wang, Zhou; Bovik, A.C.; Sheikh, H.R.; Simoncelli, E.P., "Image quality assessment: from error visibility to structural similarity", *IEEE Transactions on Image Processing 13 (4): 600–612 (2014)*

[48] Huynh-Thu, Q.; Ghanbari, M., "Scope of validity of PSNR in image/video quality assessment", *Electronics Letters. 44 (13): 800 (2008)*

[49] Huynh-Thu, Quan; Ghanbari, Mohammed "The accuracy of PSNR in predicting video quality for different video scenes and frame rates", *Telecommunication Systems. 49 (1): 35–48 (2012)*

[50] Jiasong Wu, Xu Zhang, Xiaoqing Wang, Lotfi Senhadji and Huazhong Shu, "L1-norm minimization for quaternion signals", *arXiv:1202.5471v1 (2012)*

[51] Agnieszka Badeńska, Łukasz Błaszczyk, "Compressed sensing in the quaternion algebra", *arXiv:1704.08202v2, (2017)*

[52] Jiasong Wu, Ling Xu, Youyong Kong, Lotfi Senhadji and Huazhong Shu, "Deep Octonion Networks", *arXiv:1903.08478, 2019*

[53] W. R. Hamilton, "On quaternions; or on a new system of imaginaries in algebra", *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, vol. 25, no. 163, (1844)*

[54] Ken Shoemake, "Animating Rotation with Quaternion Curves", *Computer Graphics, Presented at SIGGRAPH '85, 1985*

[55] Girard, P. R. "The quaternion group and modern physics", *European Journal of Physics. 5: 25–32, (1984)*

[56] J. M. McCarthy, "Introduction to Theoretical Kinematics", *MIT Press, 1990*

[57] Shu, Jian-Jun; Ouw, L.S., "Pairwise alignment of the DNA sequence using hypercomplex number representation", *Bulletin of Mathematical Biology. 66, arXiv:1403.2658, (2004)*

[58] J. B. Kuipers, "Quaternions and Rotation Sequences", *Princeton University Press, 1999*