Desenvolvimento e Avaliação de uma Biblioteca de Redes Neuronais de Valores Complexos.

PEDRO MANUEL FERREIRA ALVES
julho de 2024

Development and Evaluation of a Complex-Valued Neural Network Library.

The `Renplex` open-source Project

Pedro Manuel Ferreira Alves
Student No.: 1220523

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Artificial Intelligence Engineering.

Supervisor: Doutor Luiz Felipe Rocha de Faria, Professor Coordenador do Instituto Superior de Engenharia do Instituto Politécnico do Porto

Porto, July 1, 2024

Abstract

Complex-Valued Neural Networks (CVNN) have shown to be a promising type of Artificial Neural Networks (ANN) when compared to its real-valued counter-parts. However, it has been a research field where authors autonomously developed and tested CVNN with no common tools or library to module them.

This Master Thesis presents a library called Renplex capable of modulating CVNN as an open-source project for research and even for small scale applications. Although not suitable for beginners in the field of ANN or programming, the library provides a low-level interactive with Machine Learning (ML) pipeline, in order to accurately control CVNN evaluation.

To test the library's core functionalities, architectures such as Complex-Valued Multi-Layer Perceptron, Auto-encoder and Convolutional Neural Network were trained. These achieved test results that outperformed their real-valued counterparts for the MNIST dataset and a synthetically generated dataset for signal reconstruction. Such improvement on performance, has been previously stated throughout literature. It consisted in greater test accuracy (or lower loss values), more stability in training, faster convergence in terms of epochs needed, greater capability of generalization, and subsequently less prone to over-fitting.

This work will introduce a new tool for exploring CVNN, capable of scaling and potentially uncovering many of their hidden potentials for ML-related tasks.

Keywords: Neural Networks, Complex-Valued Neural Networks, Complex Back-propagation, Complex Activation Functions.

Resumo

Redes Neuronais de Valores Complexos (CVNN), têm revelado ser um tipo de Rede Neuronais Artificiais (ANN) promissoras quando comparadas com Redes Neuronais de Valores Reais (RVNN). No entanto, tem sido uma àrea de estudo em que autores desenvolvem e testam CVNN sem o uso de uma ferramenta ou biblioteca em comum para as modelar.

Nesta Tese de Mestrado é apresentada um biblioteca chamada Renplex, capaz de modelar CVNN, sendo este um projeto para auxiliar em estudos de investigação e desenvolvimento bem como para aplicações simples. Apesar de não ser apropriada para utilizadores inexperientes nas áreas de ANN e programação, esta biblioteca providencia uma interação de baixo-nível com o processo de Aprendizagem Automática (ML), para que CVNN sejam avaliadas com rígor.

Para testar as functionalidades essenciais da biblioteca, arquiteturas como Perceptron de Multi-Camadas, Auto-Codificador e Rede Neuronal Convolucional, foram treinadas. CVNN permitiu obter melhores resultados que as RVNN para o dataset de MNIST e para um dataset gerado sintéticamente para reconstrução de sinal. Esta melhoria de resultados de teste está assente na literatura. Consistem em melhor acurácia e/ou função de perda, mais estabilidade de treino, convergência rápida (com menos épocas), melhor capacidade de generalização, e consequentemente, menos propício a um super-ajuste.

Este trabalho introduz uma nova ferramenta para explorar CVNN, capaz de escalar e potencialmente desvendar uma diversidade de potencialidades relacionadas com tarefas de ML.

Palavras-chave: Neural Networks, Complex-Valued Neural Networks, Complex Back-propagation, Complex Activation Functions.

Acknowledgement

I would like to express my heartfelt appreciation to all the people, both in my academic and personal circles, who have guided and supported me on this journey.

First and foremost, I am grateful to my supervisor for accepting my proposed topic of study, for overseeing my work, and for making our interactions fruitful.

I extend my sincere thanks to my family for their unwavering support and constant presence.

A special thank you goes to my girlfriend and her family. Your continuous support, and encouragement, has given me the motivation to strive for excellence and become a better person and researcher.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

AF	Activation Function.
ANN	Artificial Neural Network.
CAF	Complex Activation Function.
CBP	Complex Back-Propagation.
CNN	Convolutional Neural Network.
CS	Computer Science.
CV	Computer Vision.
CV-CNN	Complex-Valued Convolutional Neural Network.
CVNN	Complex Valued Neural Networks.
DL	Deep Learning.
KAF	Kernel Activation Function.
RVNN	Real Valued Neural Network.
SotA	State-of-the-Art.

Chapter 1

Introduction

This chapter will give the reader some context on what will be the main subject of this Master Thesis. The present work will be an attempt to tackle a problem presented by the literature: The scarcity of computational libraries that modulate Complex Valued Neural Networks (CVNN) (Bassey, Qian, and X. Li 2021), which will be diving deeply alongside with the main objectives on the up-coming sections.

1.1 Contextualization

This Master Thesis explores a type of Neural Networks referred to as CVNN. These are a computing systems, based on the way a biological brain operates. Despite being driven by data, the difference between CVNN and, for the sake of this work, Real Valued Neural Network (RVNN), is the fact that they incorporate complex numbers[1] as their trainable or non-trainable parameters (Clarke 1990). This can range from weights, biases or activation functions, but also the dynamics and operations involved inside the layers and even the training process itself, changes substantially.

This poses the question of why is it relevant to study these types of Networks?

One of the first reasons, lies in the fact that, some data extracted by sensors, is inherently, complex-valued. Applications related to radar images of earth's surface (Sunaga, Natsuaki, and Hirose 2019), electromagnetic waves (Hirose and Yoshida 2012a; Mandic and Goh 2009), speech localization (Tsuzuki et al. 2013) and MRI signals (Virtue, Yu, and Lustig 2017), data which is inherently represented in the complex domain, have already emerged with generally far greater results when compared to RVNN.

The second reason is because they offer more "expressiveness" (Bassey, Qian, and X. Li 2021; Lee, Hasegawa, and Gao 2022), in other words, richer data representation that might encode more information related to a certain task. Matter of fact, CVNN have also comparable or better performance when compared to RVNN, in tasks where the input data is in the real domain, for instance in image classification (Nafisah, Rachmadi, and Imah 2018), segmentation (Ceylan and Yaçar 2013; Saraswathi and Srinivasan 2014) and wind prediction (Çevik, Acar, and Çunkaş 2018).

Finally, the third reason is a greater capacity for generalization (Lee, Hasegawa, and Gao 2022). This fact becomes apparent in the results obtained by (Hui Zhang et al. 2021) where an extract can be visualized in Figure 1.1.

[1]Any number that belongs to the mathematical domain $\mathbb{C}$.

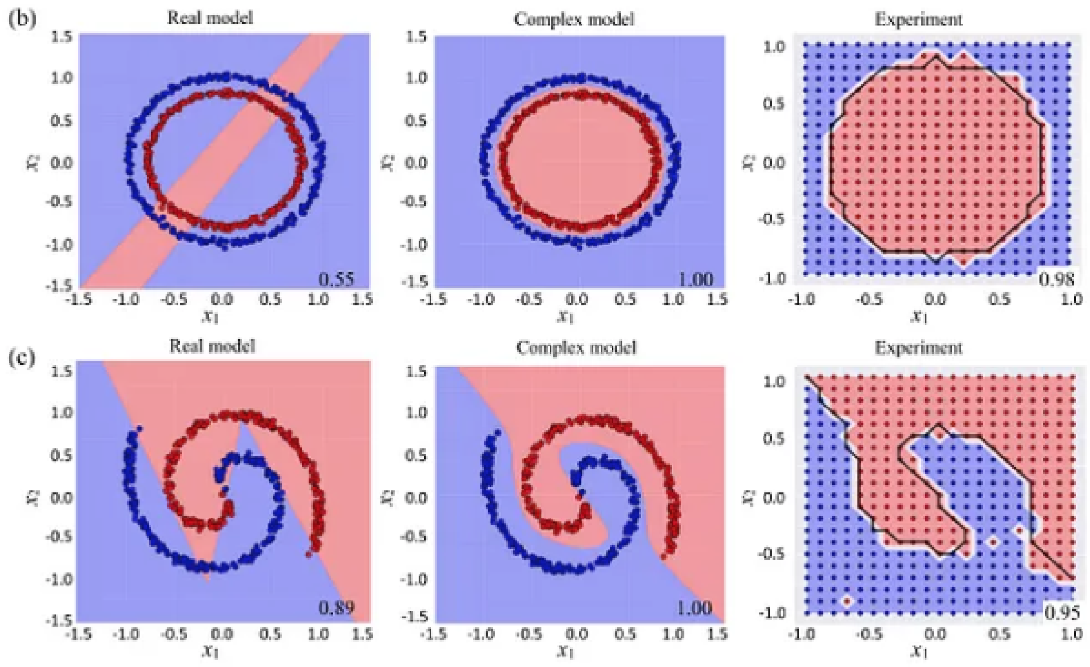


Figure 1.1: Portion of the results from the study (Hui Zhang et al. 2021), that shows the generalization capabilities of a CVNN compared to a RVNN.

RVNN are more widely utilized since their implementation is considerably easier, there is a variety of tools to modulate them with little background knowledge, and less computationally expensive. Nevertheless, there is an untapped potential related to these CVNN as it was alluded. (Lee, Hasegawa, and Gao 2022).

Furthermore, to give some fundamental context, complex numbers can be represented in the Euclidean form,

$$z = x + iy, \tag{1.1}$$

where $\mathcal{R}\{z\} = x$, $\mathcal{I}\{z\} = y$ are respectively the real and imaginary component of z, and i is the imaginary unit. Moreover, they can be represented in polar form,

$$z = \rho e^{i\phi}, \tag{1.2}$$

where e is the Euler's constant, $\rho = \sqrt{x^2 + y^2}$ is the absolute value and $\phi = \arctan\left(\frac{y}{x}\right)$ is the phase of z. In this polar representation, a z number can represent an electromagnetic signal[2] with an amplitude of ρ, and a phase $\phi = \phi(t) = \omega t + \phi_0$, being ϕ_0 the initial phase. By having the time samples of a signal, one can perform tasks with a CVNN with this signals as inputs or targets for instance.

This improvement on performance, if taken out of context, might lead to a miss-conception that CVNN are just RVNN in two dimensions. The root of this misunderstanding stems from a viewing a complex number as two real numbers, which in fact, it is not. In (Hirose and Yoshida 2012a), it was proven otherwise that the multiplication of complex numbers, actually limits the degrees of freedom of the network, thus being something entirely different.

[2]This can be relevant for instance in fiber-optic or wireless communications.

All these extra nuances may be able to represent, as it was stated by Hirose (2012), a "Super-Brain by Enrichment of the Information Representation". As it will become apparent, the engineering of complex activation functions and the various learning methods available for a CVNN are among the unique options not present in classical RVNNs. These innovations have the potential to significantly enhance performance in solving more challenging tasks.

1.2 Problem Definition

These networks were explored in a more theoretical level around 2012, and recently (2018 on-wards) some successful applications have been emerging. Nevertheless, this topic is underexplored, particularly in these CVNNs regarding the development of tools and libraries that allow one to explore such models (Bassey, Qian, and X. Li 2021).

In that sense, there is a small number of tools and the already existent ones do not provide a solid foundation to model CVNNs with close contact to its pipeline. Some were built on top of existing libraries meant for RVNN modeling, while others discontinued with no further updates. Additionally, on this small list, there are publicly unavailable tools, which would be a drawback to the Computer Science (CS) community, given the already existence of extremely popular and reliable RVNN open-source tools. Such tools will be reviewed, with more detail, in Chapter 2.

1.2.1 Objectives

The objective of this Master Thesis is twofold:

- Firstly, the main objective, is to build a library, which will be named Renplex, using the Rust programming language (Klabnik and Nichols 2018), which is capable of modeling these CVNN with possibility to have as much control as possible over the ML pipeline. Such library will be public and released as an open-source project to tackle the problem described. Repository of this library can be found here https://github.com/Pxdr0-A/renplex.git;
- Secondly, is to test some of the library's functionalities specifically in the comparison of performance between CVNN and another popular library that models RVNN TensorFlow. This is to ensure that it is working as intended and ready to be used for research and applications. (Abadi et al. 2023).

The reason for choosing Rust as the programming language for this library, because it is a systems programming language that offers low-level memory management if needed with great performance and an emerging popularity (Stack Overflow 2023). This makes it possible to produce a library capable of achieving some much needed runtime efficiency for network training, but specially, with notable scaling capabilities and security (Klabnik and Nichols 2018).

Additionally, Tensorflow is going to be the RVNN modeling library for comparison due to being a very popular machine learning framework, but also a framework. On top of that, the author of this reasearch has more experience with it, thus minimizing execution errors.

To meet these two objectives, there will be a set of tasks involved. On one hand, to allow for this customization, the library should ensure the ability to specify the precision of the calculations (32-bit or 64-bit float), provide a set of activation functions and layers to scaffold a personalized network. Ensuring these requirements, will allow to tackle the problem of the restrict CVNN modeling.

On the other hand, to provide a concise comparison between CVNN and RVNN, the proposed pipeline will go as follows:

- Only one optimization method will be explored for the CVNN. This method the most analogous to the conventional back-propagation algorithm (Rumelhart, Hinton, and Williams 1986): the fully back-propagation algorithm (Jose Agustin Barrachina et al. 2023). For each dataset addressed in this Master Thesis, equivalent architectures between RVNN and CVNN will be trained and compared with each other's test performances;
- A special task regarding signal processing, where CVNN typically out-perform will be considered to demonstrate that the developed models in the library are working as intended and in agreement with literature results.

This pipeline will ensure a fair comparison and demonstrate the usability of these CVNNs, as a consequence, hopefully tackling the main problem of scarcity in viable CVNN modeling tools in the open-source community for research purposes and real-world applications.

Chapter 2

State-of-the-Art

For this State-of-the-Art (SotA), only an overview of the literature is given as the primary objective is to build a library that modulates CVNN. Inasmuch, the first step is to study relevant applications for CVNN, which will give some insights on the data that our library should be able to handle. Additionally, it exists in the literature a vast number of approaches to develop a CVNN, for such, the mathematical theory surrounding this topic will be addressed. Lastly, the already existent libraries are described to help define what the mentioned library adds to the body of knowledge to the research community.

The search was performed by extracting the most relevant studies for the development of the library using the string "complex-valued neural network" as the main keywords. It consisted in finding papers with popular applications that have the potential of being applied in the library, as well as, theory that can be used for defining the architecture of a CVNN. Some extremely advanced or SotA procedures for modeling CVNN were kept out of this work, since the intention is just to design a simple library that can scale for common CVNN applications. Databases included in the search are the following: Google Scholar IEE Xplore, arxiv.org, Springer Link and Science Direct.

2.1 Applications

The majority of CVNN applications come from the fact that the training data is written in the complex domain. Meanwhile, the rest relies on strategies to cast the data from the real domain into the complex domain. This section is subdivided into the various areas of application.

2.1.1 Signal Processing

Signal processing was the first application found for the topic at hand (Hirose 2012) and it is a vast field. For simplicity, this subsection is subdivide in applications related to Wireless Communications and Audio. Although, the nature of the data used is similar between some fields, there are some nuances to it.

Wireless Communications

Electromagnetic waves that constitute the signals present in wireless communications, are more mathematically accurate when represented by complex numbers. If a certain problem arises that can be solved by training a neural-based model, CVNN can be considered.

One of the focus on this field is using CVNN for signal coherence (Hirose and Yoshida 2012b; R. Wu, H. Huang, and T. Huang 2017). It consists in providing the time signature, for instance, an electromagnetic signal to the inputs of the CVNN, with the objective the reconstruct the coherent source signal, with as little Signal-to-Noise-Ratio as possible. Hirose and Yoshida (2012b) describe a generalization for this problem and demonstrates that CVNN can achieve better performance than its real counter-part, whereas R. Wu, H. Huang, and T. Huang (2017) goes deeply in the specific learning method to be applied for these cases. Current work will not contain the latter learning strategy, still it will address a similar task as in Hirose and Yoshida (2012b) at Chapter 4, as it is one of the most fundamental tasks of CVNN.

Also, the field for developing signal equalizers is where CVNN provides satisfactory solutions (X. Hong et al. 2014; S. Liu et al. 2017; Uncini et al. 1999; You and D. Hong 1998). Such procedure, aims to mitigate the cross-modulation effects between the in-phase and quadrature-phase of the traveling signal, and it has been addressed since the late 90's (Uncini et al. 1999; You and D. Hong 1998). This application goes in a similar fashion to the one described in the previous paragraph by S. Liu et al. (2017).

Channel estimation is also an crucial aspect of wireless communications (Murata, T. Ding, and Hirose 2015; Yuan et al. 2019). In this case, studies aim to classify a certain channel's characteristics. Herein, CVNN receives the exchange signal in the complex domain. In the case of (Yuan et al. 2019), the authors implemented an auto-encoder architecture that a CVNN should be capable of supporting. Alongside channel estimation, CVNN managed to surpass Real Deep Learning (DL) models on the task of specific emitter identification (Y. Wang et al. 2021).

With the introduction of the 5G mobile network, a recent study shows an application CVNN in the massive multiple-input multiple-output (MIMO) (Tiba and youhong 2023) . The motivation lies purely on the fact that current MIMO detection is done by RVNN, which does an additional step of converting the complex data, that the signal naturally possesses, into real data. Not only there is loss of information, but also, unnecessary computational demand. Study shows that, a CVNN can provide better performance in when compared to current detectors and reduce the computational cost (Hirose and Yoshida 2012b; Tiba and youhong 2023). Some equally relevant studies precede this recent one, such as, (Marseet and Sahin 2017) where authors also use the architecture that was later used replicated by (Yuan et al. 2019) in the channel estimation problem.

Generic studies on signal processing, have also been conducted over 20 years ago (Kim and T. Adali 2000; Kim and Tülay Adali 2002). The latter authors experimented with specific activation functions that improve the performance of CVNN signal processing capabilities. Still, some more recent studies pick up from this point for further improvements and extensions (Scardapane et al. 2018).

Audio

Regarding audio analysis with CVNN, some applications emerge in the field of speech recognition (Hayakawa, Masuko, and Fujimura 2018). This application comes from the direct translation of incoming sound/wave signals from speech, which are already, by default, encoded in complex numbers with an amplitude and a phase. The speech signal is not analyzed in a recursive way, but a batch of the signal is analyzed to decode possible existent speech within it. A CVNN does outperform a RVNN in the task (Hayakawa, Masuko, and Fujimura

2018). Hu et al. (2020), also approaches the topic of analyzing speech. Nonetheless, in this specific case, the task is not to recognize but to enhance the signal. Similar to the channel estimation or specific emitter identification (Y. Wang et al. 2021), herein, a speech signal is introduced with noise and a CVNN is tasked to enhance the quality of the signal, which is in fact able to achieve satisfactory results.

The above problem of enhancing an audio signal was also applied for the mp3 format in (Al-Nuaimi, Faijul Amin, and Murase 2012). The objective was to recover an encoded signal, as close as possible to the unmodified, with the ideal architecture, one can get more suitable results when compared to an equivalent RVNN (Al-Nuaimi, Faijul Amin, and Murase 2012).

Strikingly, CVNN also found an application in music by means of a classification task of retrieving meta-information about a song (Kataoka, Kinouchi, and Hagiwara 1998), or memorizing a sequence of notes of a melody (Kinouchi and Hagiwara 1996). In spite of (Kataoka, Kinouchi, and Hagiwara 1998) using Recurrent Neural Networks, the procedures is similar when compared to real numbers.

2.1.2 Image Processing & Computer Vision

When it comes to Image Processing and Computer Vision (CV) models, CVNN also exhibit promising results, however, the way data is handle can be different from the signal processing procedure.

In satellite imagery as in the example of f TerraSAR-X datasets, satellites can provide aside from a normal image, information about the polarization of the light received (Gleich and Sipos 2018). This in itself, can be represented in the complex domain together the classical image and feed onto a Complex-Valued Convolutional Neural Network (CV-CNN). Nevertheless in (Gleich and Sipos 2018) the authors implement a CV-CNN with substantial results given the nature of the data.

Whereas, during data pre-processing stage for CVNN, not all data has an explicit complex notation associated. Y. Liu, H. Huang, and T. Huang (2014) created a model for hand gesture recognition, where the data is initially represented in pixels. The images are pre-processed with a CV tool to get the main features out of the image, such as angles between fingers, length, etc. These coordinates are then written in the complex plane where the images can finally be processed by a CVNN with a performance that matches current applications (Y. Liu, H. Huang, and T. Huang 2014). Although not with the same detail, another work pre-dated this exact issue (Hafiz, Amin, and Murase 2011).

Other studies, circle around facial recognition on distinguishing between the two genders. Still features are extracted from an image with CV tools. Despite that, Amilia, Sulistiyo, and Dayawati (2015) performed a mapping that defines $1 + 0.5\imath$ as male and $0 + 0.5\imath$ as female instead of the real typical values of 0 and 1.

In DL for Image Recognition, the typical CVNN pipeline can be either converting an image's pixels to complex numbers with no imaginary component or to extract the features of the image and find some mapping to the complex plane, with these features (Chiheb, Bilaniuk, Serdyuk, et al. 2017; S. Gu and L. Ding 2018). One simple example can be for instance, getting the intensity of the transitions and respective angles with a Sobel Operator (Sobel 2014), and the intensities and angles can be mapped to the absolute values and phases of the complex numbers respectively. Another option is to compress the features since a complex number can encode two numbers, the number of inputs can be resized to $N/2$ where N is the number of real inputs (S. Gu and L. Ding 2018). CVNN have also been employed in similar problems involving crowd counting (Matlacz and Sarwas 2018), where one can divide the complex components in features as stated previously or just give a default value to the imaginary part (Chiheb, Bilaniuk, Serdyuk, et al. 2017; Matlacz and Sarwas 2018).

Furthermore, in the Image Processing realm, CVNN can be used to reconstruct images that are blurred (I. Aizenberg, Alexander, and Jackson 2011; I. Aizenberg, Paliy, et al. 2008) similarly to signal reconstruction, but with mapping to real numbers and without requiring DL models.

To wrap up this selection of applications, it is important to acknowledge that CVNN are equally suitable outside classification or segmentation tasks, such as the ones reviewed up until now. A very recent study by Luo et al. (2024) provides a solution for compressing images based on CVNN with greater robustness against adversarial attacks when compared to RVNN.

2.1.3 Health

CVNN make an appearance in the health sector. Applications such as Electroencephalography (EEG) and Medical Resonance Imaging (MRI) where both signal can be divided in a real and an imaginary component. In the studies (Du, Riddell, and X. Wang 2023; Peker, Sen, and Delen 2016; J. Zhang and Y. Wu 2017) the authors explore possible usages in EEG-related diagnosis. While (Du, Riddell, and X. Wang 2023) dive in a more generic study on trying to address if CVNN are viable for EEG applications, (Peker, Sen, and Delen 2016; J. Zhang and Y. Wu 2017) take a more pragmatic approach in trying to apply it to classifying sleep stages, and epilepsy diagnosis, respectively. In the former, there is the use of the complex convolution (will be later addressed) operation given the initial complex signal, where authors were able to match human experts performance (J. Zhang and Y. Wu 2017). In the latter, authors consider the relevance of the EEG as the gold standard for epilepsy diagnosis (Pillai and Sperling 2006) to create a model that performs such evaluation, however, as opposed to the previous procedure, they do not rely on convolutional layers. Instead, the authors implement a Multi-Layered Perceptron (Rumelhart, Hinton, and Williams 1986) CVNN with k-folds cross-validation to accurately be used for epilepsy diagnosis (Peker, Sen, and Delen 2016).

In the MRI scenario, (E. Cole et al. 2021; E. K. Cole et al. 2020), a cross-section of an image is defined in the complex plane with polar coordinates. The reconstruction process significantly reduces the amount of time patients need to remain still, so the authors explore Complex Convolutional Layers that perform this task. Being the input a matrix of complex values, CVNNs were able to achieve higher quality of image reconstruction (E. Cole et al. 2021; E. K. Cole et al. 2020). Still in the topic of MRI, the task of identifying tissue parameters, based on cross-sections, was also tackled with CVNN in (Virtue, Yu, and Lustig 2017) and outperformed RVNN purposely designed for the task.

It is of high importance to note that operations such as the Fast Fourier Transform, can be implemented in CVNN, which subsequently analyzes, for example, medical images regarding mammography for digital watermarking Olanrewaju et al. (2011).

2.1.4 Other Applications

Although multiple cases indicate that CVNN typically surpasses RVNN in terms of performance metrics, CVNN is generally slower to train. As previously mentioned, the study by Hui Zhang et al. (2021) develops for the first a computing ship specialized for complex computation involved in a complex-valued multi-layer perceptron. With this computing chip, the authors analyzed fundamental logic gates that adapted better to non-linearities when compared to RVNN. The IRIS (Fisher 1936) and MNIST (LeCun, Cortes, and Burges 1998) dataset were also studied with slightly different strategies both achieving greater results than RVNN.

Interestingly enough, CVNN found its way onto stock prediction (Jia, B. Yang, and W. Zhang 2018; H. Wang, B. Yang, and Lv 2017). Stocks data is not written in the complex domain, therefore, authors described a method casting those values in a unitary complex value. This was by defining a phase based on the real data point, maximum value of the set, and minimum value of the set (Y. Wang et al. 2021). Both procedures also used different optimization algorithms for updating the weights, respectively, Particle Swarm Optimization (Eberhart and Kennedy 1995) and Cuckoo search (X.-S. Yang and Deb 2014).

There is also a Thesis that is worth mentioning on using and exploring Deep Complex-Valued Recurrent Neural Networks (Mönning 2019).

2.2 Theory Behind CVNN

Multiple approaches have been taken into considerations when it comes to designing a CVNN and many references have already dived deep into how should one structure a CVNN depending on the task at hand. This section will be divided into two sub-sections. First about how the literature has approached the problem of defining a Complex Activation Function (CAF) and how can the back-propagation be implemented in these networks. Some notes about the Complex Back-Propagation (CBP) algorithm will be included based on the literature.

Related to a CVNN library (Jose Agustin Barrachina et al. 2023), provides also a detailed explanation on how to implement such networks code-wise, as well as, (Abdalla 2023) also provides a great and up-to-date summary on the many possibilities to design a CVNN. For development purposes, former studies will be important for the library implementation.

2.2.1 Complex Activation Function

The first steps onto the development of a CVNN, targeted CAF (Clarke 1990; Georgiou and Koutsougeras 1992). There is an inherent problem of this neural networks that upon providing a complex argument to an activation function typically used in RVNN[1], one would observe that its derivatives are not contained/limited in the complex domain. This condition is important for the stability of the gradients. Next on this section, it will be addressed some CAF studied in the literature that were implemented in the library.

The most standard example is the identity activation function or simply no activation first introduced by Widrow, McCool, and Ball (1975). It is a function that is useful for drawing signals in the output layer for instance but it is prone to exploding/vanishing gradient (Hirose 2012). The problem with the exploding/vanishing gradient can be tackled by normalizing z

[1]A standard example can be the Sigmoid function for instance

with the modulo function (Amari 1995; Hirose 2012), nevertheless it limits z to a circumference of radius 1 which is specially useful for the non-gradient based approach for learning (Bassey, Qian, and X. Li 2021).

Although slightly unstable, the hyperbolic tangent function can also be used, being one of the first activation functions to be experimented with (Kim and Tulay Adali 2000).

To tackle the problem of limited derivative, Benvenuto and Piazza (1992) suggested the split-type activation functions. This one consists in applying a well-known activation function use in RVNN like sigmoid (Cox 1958) to both the real and imaginary part separately (split-type A) or the amplitude and phase (split-type B) (Abdalla 2023), regardless, such function does provide the much needed quick differentiation but it does not represent a fully CAF with the possibility to incorporate correlations between real and imaginary component just like previous ones. By expanding this reasoning, one can define a set o split-functions based on RVNN activation functions making a very direct analogy between networks.

$$f_s(z) = f(\mathcal{R}\{z\}) + if(\mathcal{I}\{z\}), \tag{2.1}$$

where $f(x)$ is a function with limited derivative in the real domain like the sigmoid, or some non-linear function like ReLU (Glorot, Bordes, and Bengio 2011). Within the same context, one can have complex non-linear/parametric functions in the complex domain taking in consideration the phase, which is that case for the zReLU function (Guberman 2016).

$$f(z) = \begin{cases} z \text{ if } \arg(z) \in \left[0, \frac{\pi}{2}\right] \\ \\ 0 \text{ otherwise} \end{cases} \tag{2.2}$$

All these functions were later referred to as non-analytical (Scardapane et al. 2020), and those that involved the necessity to compute absolute values, also fall in the same category.

Another non-analytical CAF worth noting is the Cardioid Function (Virtue, Yu, and Lustig 2017). It is easily differentiable to help in the CBP and carries a simple expression of basic computation, given by,

$$f(z) = \frac{1}{2}\left(1 + \cos(\arg(z))\right) z. \tag{2.3}$$

The above mentioned CAF, describes the main core types of activation in the complex domain that the library will implement. Nonetheless, in the next paragraph will be mentioned some CAF that have proved to provide comparable or better performance but were not yet implemented in the library.

Although used in some specific contexts, the modReLU function is a variant of the ReLU function in the complex domain first introduced by Arjovsky, Shah, and Bengio (2016). This function can be prone to training since it possesses a threshold that needs to be defined unlike the split-ReLU function for instance.

$$f(z) = \mathrm{ReLU}(|z| + b)\frac{z}{|z|}, \tag{2.4}$$

where b is the threshold.

Authors in (Scardapane et al. 2020) describe the usage of these so called Kernel Activation Function (KAF) that when incorporated in a CVNN, show better results when compared to RVNN for the standard dataset MINIST (LeCun, Cortes, and Burges 1998; Scardapane et al. 2020). Although they are not used in this library, it is important to mention their existence that they provide good performance on CVNN models and could be implemented in the library. To avoid getting too technical in this SotA, broad terms, KAF are constructed with a kernel that is easily differentiable and limited in a weighted sum along a grid. Typical KAF would read as,

$$g(z) = \sum_{n=1}^{D} \sum_{m=1}^{D} \alpha_{n,m} \kappa_{\mathcal{C}} \left(z, d_n + \imath d_m \right), \tag{2.5}$$

with D being the dimension of the grid, $\kappa_{\mathcal{C}}$ the kernel (can be for instance a Gaussian function), d_n, d_m are parameters of the grid and $\alpha_{n,m}$ the mixing parameters (Hofmann, Schölkopf, and Smola 2008; W. Liu, Principe, and Haykin 2011).

2.2.2 Complex Learning Procedure

In later sections, a more detailed walk-through on the CBP algorithm, to be used in this Master Thesis, will be provided as well as other possible learning alternatives. This section, briefly discusses some of the already available options in the literature.

Core Procedures

One can divide into two main topics, gradient-based and a non-gradient based learning (Abdalla 2023; Bassey, Qian, and X. Li 2021).

With the gradient approach, the objective is, as known from the classical neural networks, to compute the gradient of the cost/loss function. The loss can be calculated in the complex domain for a single training sample like so,

$$\mathcal{L} = \sum_{n} |a_n - t_n|^2 , \tag{2.6}$$

where $a_n \in \mathbb{C}$ represents the activation on the last layer per unit n and $t_n \in \mathbb{C}$ the desired output from the training sample.

The back-propagation can be performed with almost the same procedure as in a RVNN, however, one can either analyze the adjustment to the weights at the real and imaginary level individually (split CBP) (Benvenuto and Piazza 1992) or at the weight as an entire complex number (full CBP) since they obey the same differential properties (Abdalla 2023; Bassey, Qian, and X. Li 2021; Hirose 2012; Kim and Tulay Adali 2000; S. Li et al. 2006). Moreover, one must recognize the foundation required to reach the complex gradient of the loss within the Wirtinger Calculus (Wirtinger 1927). From surveys (Bassey, Qian, and X. Li 2021; Lee, Hasegawa, and Gao 2022), one can see that the full CBP is more commonly used and the split CBP is gradually becoming obsolete due to not considering the correlations between real and imaginary parts individually.

One thing that CVNN bring of new when compared to RVNN is that one can apply a non-gradient based approach for training the network. The error correction occurs at the phase level being the reason why the modulo activation function is so important. The correction is defined and discussed in (N. Aizenberg et al. 1973) and as stated, does not involve computing any derivatives. For a simple feed-forward CVNN, the expression for the error correction can be found in (Abdalla 2023) and its interpretation is that the error is corrected at the neuron level (individually). As the error begins at the output layer, in this approach, it still needs to be propagated backwards. The implementation of this algorithm in the library is incomplete.

Further Considerations

The mentioned core procedures already underwent some improvements, or simply new additional methods have been adopted. Among them, Q. Liu et al. (2017) proposed a more efficient algorithm for updating the weights of a CVNN. It was achieved by separating the training method of the input layer, from the output layer. In the latter, the update to the output neuron's weights is first calculated with the least squares method. Getting this result, error is propagated with CBP using Gradient Decent to the input's weights. Authors reported a better convergence and generalization capabilities (Q. Liu et al. 2017).

Modifications to the learning algorithm with momentum optimization or even the Adam optimizer, also provide same benefits as in the RVNN (Kingma and Ba 2014; Kotsovsky, Batyuk, and Yurchenko 2020). The same applies for adaptive complex-valued step sizes in gradient descent, however, this procedure can only be applied to fully CVNN (Zhao and H. Huang 2023, 2024).

Convolution operation is well defined in the complex plain and, just like in the RVNN, calculations can be parallelized. In spite of the slight increase in complexity, CV-CNN outperform a Convolutional Neural Network (CNN) (Chatterjee et al. 2022; Guberman 2016).

Last but not least, if designed properly, CVNN have been shown to be more robust than a its real counter-parts, specially in signal processing-related applications (Neacșu et al. 2022).

2.3 Exploration of Existent Libraries

As stated by (Bassey, Qian, and X. Li 2021), there is the need for libraries targeted for complex-valued computations such as in CVNN, hence the objective of this Master Thesis. Very few full-fledged libraries or toolboxes can be found in the literature that modulate CVNN. To the best of the authors knowledge, only the following references were found: (J Agustin Barrachina 2022; Jose Agustin Barrachina et al. 2023; Chiheb Trabelsi 2017; Cruz, Mayer, and Arantes n.d.; Dramsch and Contributors 2019; Gürüler and Peker 2015).

Within this group one finds, in its majority, libraries that are built on top of an already existent machine learning framework meant for RVNN modeling like Keras. (J Agustin Barrachina 2022; Jose Agustin Barrachina et al. 2023; Chiheb Trabelsi 2017; Cruz, Mayer, and Arantes n.d.; Dramsch and Contributors 2019) all use Keras / TensorFlow as a back-end which may limit the amount of available operations and architectures, further performance optimizations and efficient memory management, given the unique approaches, as observed from the current SotA, that CVNN may require. Additionally, all of these tools have not been updated since 2 years and were developed typically for a one-time-usage.

Gürüler and Peker (2015) has also developed a tool for signal process, nonetheless, from (Peker, Sen, and Delen 2016) it suggests that was built for this specific usage.

2.4 Data Protection & Ethical Aspects

Given the nature of the objective of this Thesis, it is not going to involve managing data of confidential or bio-metric related that might invoke any privacy policy. A synthetic signal reconstruction dataset will be produced and used for testing CVNN as well as and the MNIST dataset (LeCun, Cortes, and Burges 1998) in two applications.

Current state of the library is not yet capable of training large scale models, or even generative models in the dangerous category of the European Commission (2021).

Chapter 3

Library Implementation

This chapter will be dedicated to describing in detail how the library was developed. The theory and calculus involved in structuring a CVNN will be described as some, fundamental notions like forwarding a signal and training a CVNN. Afterwards, it will be briefly addressed how the library is structured followed by the main algorithms implemented to perform the necessary calculus related to CVNN tasks. This chapter will be wrapped up with some guidelines on how to operate with the library.

3.1 Theory & Calculus

The development of this library required some fundamental notions of complex analysis, therefore while going through the theory behind the dynamics of a CVNN in this library, essential Complex Analysis concepts will also be presented. First, establishing some fundamental concepts regarding Artificial Neural Network (ANN), second, how a CVNN forwards a signal, including a small discussion around complex activation functions, and third, how a CVNN learns with the fully complex back-propagation algorithm.

A majority of the fundamental notions involved in this chapter come from Wintinger Calculus (Wirtinger 1927).

3.1.1 Fundamentals

A ANN can have a high-level representation as a multivariate function whose parameters are its weights and biases.

$$\mathbf{y} = f_{w_1,w_2,\ldots,b_1,b_2,\ldots}(\mathbf{x}), \tag{3.1}$$

where f is the function that represents the network, w_i, b_i its weights and biases respectively, $\mathbf{x}$ the input features of a certain data point to be forwarded through the network, yielding the output features $\mathbf{y}$.

ANN are composed of layers which take a set of n_i input features, perform some operation on these features and output a set of n_o output features. Such operations, in the real domain, consist of e.g. computing a weighted sum or convolution. One can also visualize a layer as a function,

$$\mathbf{y}^{(l)} = g_{w_1,w_2,\ldots,b_1,b_2,\ldots}(\mathbf{x}^{(l)}), \tag{3.2}$$

being g the function that represents the layer, $\mathbf{y}^{(l)}$ the input and $\mathbf{x}^{(l)}$ the output features of a layer l, and w_i, b_i the weights and biases, respectively, of the layer (not necessarily the same set as in equation (3.1)).

Now, functions f and g are the same only if the ANN is composed of one layer. ANN typically contain multiple layers where mathematically speaking,

$$\mathbf{y} = g^{(L)} \circ \ldots \circ g^{(2)} \circ g^{(1)}(\mathbf{x}), \tag{3.3}$$

with $\circ$ denoting the chain composition operator, L being the total number of layers in the ANN, and $f(\mathbf{x}) = g^{(L)} \circ \ldots \circ g^{(2)} \circ g^{(1)}(\mathbf{x})$. Figure 3.1 depicts a high-level representation of an ANN according to the formalism used.

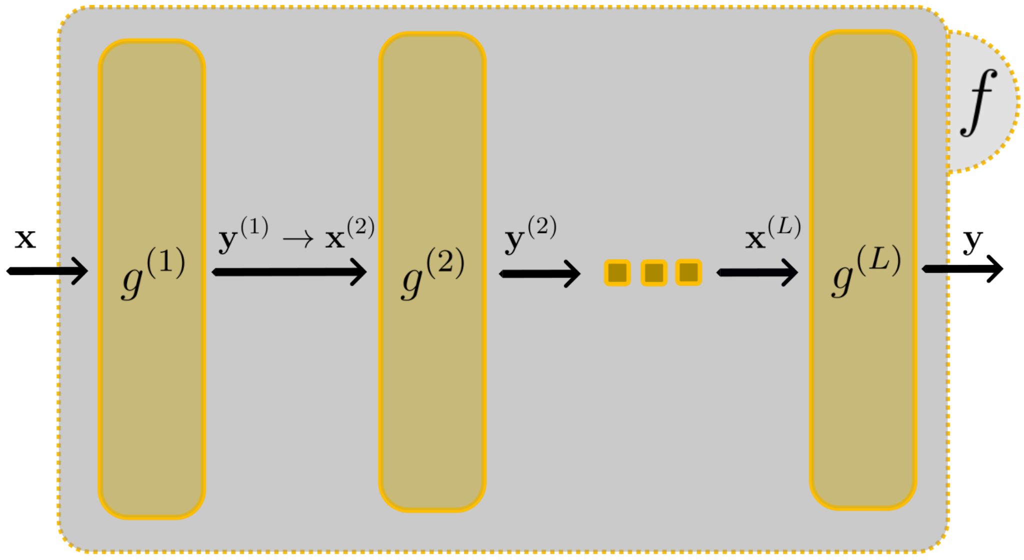


Figure 3.1: High-level representation of an ANN through function composition showing the dynamics of input $\mathbf{x}$ all the way to the output y through L layers.

Processing Unit

Layers contain an array of neurons or processing units (for short, unit). Each unit is capable of accessing the entire length of input features to that layer and it is the element that possesses the parameters of the network (weights and biases). The layer's number of output features is mediated by the number of units present in that layer.

Since each layer has its own sets of units, to keep track of all parameters, we will establish the following definitions:

- $\mathbf{w}_i^{(l)}$ - weights of neuron i of layer l. Weights can be a scalar, vector, matrix or generally speaking, a tensor;
- $b_i^{(l)}$ - scalar bias value of neuron i of layer l;
- $n_I^{(l)}$ - number of input features of layer l;
- $n_O^{(l)}$ - number of units or number of output features of layer l;
- $I^{(l)}$ - input feature shape of layer l;

- $O^{(l)}$ - output feature shape of layer l.

Upon receiving the input features, unit i in layer l will perform some computation expressed by a function $\widetilde{h}^{(l)}$, resulting in one output feature per unit. The quintessential example of such computation is the weighted sum, where if a neuron receives a vector of layer input features $\mathbf{x}'$,

$$\mathbf{q}_i^{(l)} = \widetilde{h}^{(l)}(\mathbf{w}_i^{(l)}, b_i^{(l)}, \mathbf{x}^{(l)}) = \mathbf{w}_i^{(l)} \cdot \mathbf{x}^{(l)} + b_i^{(l)}, \tag{3.4}$$

being $\widetilde{h}^{(l)} : \mathbb{R}^{n_I^{(l)}} \to \mathbb{R}$, $\mathbf{q}_i^{(l)}$ the pre-activation value of neuron i of layer l and "$\cdot$" denotes the dot product. In this example, $\mathbf{q}_i^{(l)}$ happens to be a scalar[1] and it was decided to name it pre-activation since the last step of the processing unit computation, is the application of an activation function.

The derivative of the activation function $h^{(l)}$ must yield a limited function. This function will receive the pre-activation value from a unit i of layer l to filter them to an appropriate range for the network, returning the final output feature or the activation value $\mathbf{a}_i^{(l)}$ of the unit,

$$\mathbf{a}_i^{(l)} = h^{(l)}(\mathbf{q}_i^{(l)}), \tag{3.5}$$

where $\mathbf{a}_i^{(l)}$ has the same dimensions as $\mathbf{q}_i^{(l)}$. Two standard examples of $h^{(l)}$ where $h^{(l)} : \mathbb{R} \to \mathbb{R}$ can be, for instance, the sigmoid function σ or hyperbolic tangent tanh operating. The layer l's output will be a vector of length $n_O^{(l)}$ containing all $\mathbf{a}_i^{(l)}$ for $i = 1, 2, ..., n_O^{(l)}$. On a side note, one can see that this process inside a unit is also a composition of functions,

$$\mathbf{a}_i^{(l)} = h^{(l)} \circ \widetilde{h}^{(l)}(\mathbf{w}_i^{(l)}, b_i^{(l)}, \mathbf{x}^{(l)}) \tag{3.6}$$

The main gist of a RVNN when compared to a CVNN is that $\mathbf{w}_i^{(l)} \in \mathbb{C}^{I^{(l)}}$ and $b_i^{(l)} \in \mathbb{C}$ for $i = 1, 2, ..., n_O^{(l)}$. This small modification on the network's parameters carries out major consequences in the traditional dynamics of RVNN.

3.1.2 Signal Forwarding

To forward a signal through an ANN, one must define the layers of the network, i.e. defining the functions g^l for $l = 1, 2, ..., L$ where L is the number of layers in the network, and calculate the chain composition of functions given input $\mathbf{x}$ as expressed in equation (3.3). To define the function $g^{(l)}$ where $g^{(l)} : \mathbb{R}^{I^{(l)}} \to \mathbb{R}^{O^{(l)}}$, one must first understand its action. Since a layer l will return $n_O^{(l)}$ output features then,

$$\begin{aligned}\mathbf{y}^{(l)} = g^{(l)}(\mathbf{x}^{(l)}) = \\ [h^{(l)} \circ \widetilde{h}^{(l)}(\mathbf{w}_1^{(l)}, b_1^{(l)}, \mathbf{x}^{(l)}), h^{(l)} \circ \widetilde{h}^{(l)}(\mathbf{w}_2^{(l)}, b_2^{(l)}, \mathbf{x}^{(l)}), ..., h^{(l)} \circ \widetilde{h}^{(l)}(\mathbf{w}_N^{(l)}, b_N^{(l)}, \mathbf{x}^{(l)})]\end{aligned} \tag{3.7}$$

with $N = n_O^{(l)}$. Indexing the output features yields,

[1] In general terms, these quantity can also be a tensor depending on the pre-activation operation.

$$\mathbf{y}^{(l)} = \mathbf{a}_i^{(l)} = h^{(l)} \circ \widetilde{h}^{(l)}(\mathbf{w}_i^{(l)}, b_i^{(l)}, \mathbf{x}^{(l)}). \tag{3.8}$$

With this in mind, one no longer needs the abstract function $g^{(l)}$. The layer l can be defined by the pre-activation function $\widetilde{h}^{(l)}$ and the activation function $h^{(l)}$. It is important to emphasize that, due to abuse of notation, it is not clear that $\widetilde{h}^{(l)}$ and $h^{(l)}$ can be different functions from $\widetilde{h}^{(l+1)}$ and $h^{(l+1)}$, which in fact can in cases where the network has multiple layers. If $L = 1$ forwarding a signal through the network can be trivial like so,

$$\mathbf{y}_i = \mathbf{a}_i^{(1)} = h^{(1)} \circ \widetilde{h}^{(1)}(\mathbf{w}_i^{(1)}, b_i^{(1)}, \mathbf{x}). \tag{3.9}$$

For a generic L, the entire forward dynamic is summarized as,

$$\mathbf{a}_i^{(l)} = h^{(l)} \circ \widetilde{h}^{(l)}(\mathbf{w}_i^{(l)}, b_i^{(l)}, \mathbf{a}_j^{(l-1)}) \tag{3.10}$$

where $\mathbf{a}_i^{(l=0)} = \mathbf{x}_i$ being the input to the ANN ($l = 0$ is a virtual layer) and $\mathbf{a}_i^{(l=L)} = \mathbf{y}_i$ the prediction of the network. The generic indices i, j, are used to represent that different layers can have different lengths of input and output features, i.e., typically $n_I^{(l)} \neq n_I^{(l-1)}$. Additionally, the number of input features of each layer must be equal to the number of output features of the previous layer, i.e. $n_I^{(l)} = n_O^{(l-1)}$. Figure 3.2, presents an illustrative summary of the notation and dynamics addressed up until this point.

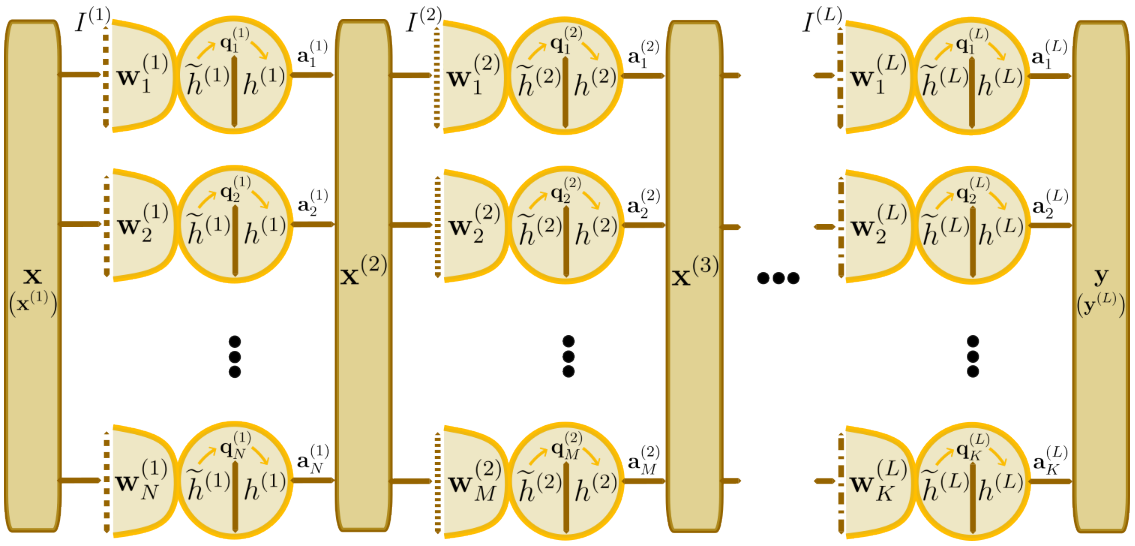

$I^{(1)}$
$I^{(2)}$
$I^{(L)}$
$\mathbf{x}$ $(\mathbf{x}^{(1)})$
$\mathbf{x}^{(2)}$
$\mathbf{x}^{(3)}$
$\mathbf{y}$ $(\mathbf{y}^{(L)})$
$\mathbf{w}_1^{(1)}$ $\mathbf{q}_1^{(1)}$ $\widetilde{h}^{(1)}$ $h^{(1)}$ $\mathbf{a}_1^{(1)}$
$\mathbf{w}_2^{(1)}$ $\mathbf{q}_2^{(1)}$ $\widetilde{h}^{(1)}$ $h^{(1)}$ $\mathbf{a}_2^{(1)}$
$\mathbf{w}_N^{(1)}$ $\mathbf{q}_N^{(1)}$ $\widetilde{h}^{(1)}$ $h^{(1)}$ $\mathbf{a}_N^{(1)}$
$\mathbf{w}_1^{(2)}$ $\mathbf{q}_1^{(2)}$ $\widetilde{h}^{(2)}$ $h^{(2)}$ $\mathbf{a}_1^{(2)}$
$\mathbf{w}_2^{(2)}$ $\mathbf{q}_2^{(2)}$ $\widetilde{h}^{(2)}$ $h^{(2)}$ $\mathbf{a}_2^{(2)}$
$\mathbf{w}_M^{(2)}$ $\mathbf{q}_M^{(2)}$ $\widetilde{h}^{(2)}$ $h^{(2)}$ $\mathbf{a}_M^{(2)}$
$\mathbf{w}_1^{(L)}$ $\mathbf{q}_1^{(L)}$ $\widetilde{h}^{(L)}$ $h^{(L)}$ $\mathbf{a}_1^{(L)}$
$\mathbf{w}_2^{(L)}$ $\mathbf{q}_2^{(L)}$ $\widetilde{h}^{(L)}$ $h^{(L)}$ $\mathbf{a}_2^{(L)}$
$\mathbf{w}_K^{(L)}$ $\mathbf{q}_K^{(L)}$ $\widetilde{h}^{(L)}$ $h^{(L)}$ $\mathbf{a}_K^{(L)}$

Figure 3.2: Base architecture of a neural and framing with the formalism introduced in this chapter. In this illustration, $N = n_O^{(1)}$, $M = n_O^{(2)}$ and $K = n_O^{(L)}$. To not make the illustration to dense , the bias parameter was omitted, nevertheless, its framing is analogous to the weights.

3.1.3 Complex Activation Functions

At a high-level, this procedure applies for both RVNN and CVNN. The major difference happens when one looks at the function $\widetilde{h}^{(l)}$ and $h^{(l)}$ since now they map values in the complex domain. Proceeding to examining the example of the pre-activation function of the weighted sum in equation (3.4) in the complex domain $\mathbb{C}$. The first instance where this domain offers some non-linearity in the calculations, is in the product between two complex numbers inside the scalar product between $\mathbf{w}_i^{(l)}$ and $\mathbf{x}' = \mathbf{a}_j^{(l-1)}$. Considering two complex numbers $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$, their multiplication can be expressed in function of each one's real and imaginary components,

$$z = z_1 z_2 = (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + y_1 x_2), \tag{3.11}$$

where $\mathcal{I}\{z_1\}$ and $\mathcal{I}\{z_2\}$ have an important role in modulating $\mathcal{R}\{z\}$. The second instance is in choosing an activation function for the network.

From this point onward, it is important to start considering the properties of complex numbers and its consequences on a CVNN. Since the weights and biases are now complex, even if the values of the input features carry no imaginary component, the forwarded signal will eventually pick up an imaginary component.

For a function to be viable candidate as an CAF $h^{(l)} : \mathbb{C}^{O^{(l)}} \to \mathbb{C}^{O^{(l)}}$ for a given layer l, its derivative needs to have a limited codomain. In the real domain, the previously given examples of σ and tanh are actually both limited in its original form and its derivative. However, if we take for instance function σ and provide it a complex argument, it will result in a function that does not have a limited codomain, nor a limited derivative, due to the exponentiation of complex numbers. A typical work around is to define a Real-Imaginary-Type function or Split-Function, as shown in equation (3.12).

$$h^{(l)}(z) = h_r^{(l)}(\mathcal{R}\{z\}) + i h_r^{(l)}(\mathcal{I}\{z\}), \tag{3.12}$$

with $h_r^{(l)} : \mathbb{R}^{O^{(l)}} \to \mathbb{R}^{O^{(l)}}$ a function with limited derivative. Given that $h_r^{(l)}$ is limited, ensures that $h^{(l)}$ and its derivative is also limited, therefore, a viable candidate as a CAF.

A group of split functions can be created based on real-valued Activation Function (AF) which are present in the library but also two more viable CAF. The library also implements the zReLU function

$$h^{(l)}(z) = \begin{cases} z \text{ if } 0 \leqslant \arg(z) \leqslant \frac{\pi}{2} \\ 0 \text{ otherwise} \end{cases}, \tag{3.13}$$

Additionally, it also implements an adaptation of the cardioid curve, as a function,

$$h^{(l)}(z) = \frac{1}{2}(1 + \cos(\arg(z)))z \tag{3.14}$$

Both functions being explicitly dependent on the phase of z and with only its derivative as a limited function.

3.1.4 Complex Back-Propagation

In section 3.1.1 it was defined a set of functions and composition of functions that, at the time, seemed to be just some abstraction. However, these functions will be important for implementing an algorithm based on the gradient descent optimization method (A.-L. Cauchy 1847), that will make it possible for the CVNN to learn through data. This requires knowledge in Complex Analysis, specifically complex function derivatives, and using the complex chain rule in the composition of functions to implement Complex Back-Propagation (Rumelhart, Hinton, and Williams 1986).

Before going through complex differentiation, let us quickly define a complex loss function. Although a simple concept, it is crucial for implementing complex gradient descent (or optimization algorithms in general).

Typically, in the real domain, one can define the RVNN loss function as,

$$\mathcal{L}_n = \left(\mathbf{a}_n^{(L)} - r_n\right)^2, \tag{3.15}$$

where $\mathcal{L}_n$ represents the error of neuron n given last layer's activation $\mathbf{a}_n^{(L)}$ and r_n the target output feature result (from the dataset). A mean or a sum can be performed along the flatten values of $\mathcal{L}_n$ for instance to a single value of the loss function expressed in equation (3.16)

$$\mathcal{L} = \sum_{n}^{n_O^{(L)}} \mathcal{L}_n, \tag{3.16}$$

In the complex domain, not a lot changes in the error expression $\mathcal{L}_n : \mathbb{C}^{O^{(L)}} \to \mathbb{R}^{O^{(L)}}$, since a complex codomain is not going to be considered. The complex error function can be,

$$\mathcal{L}_n = \left|\mathbf{a}_n^{(L)} - r_n\right|^2, \tag{3.17}$$

with $\left|\mathbf{a}_n^{(L)} - r_n\right|$ being the absolute value of the complex number resultant from $\mathbf{a}_n^{(L)} - r_n$.

Complex differentiation

Differentiating a complex function is going to be the most important task to perform in the complex gradient descent (as a consequence, complex back-propagation), and it is important to recognize certain types of complex functions.

In complex analysis, an Holomorphic function is a function that obeys the Cauchy-Riemann equations (A. L. Cauchy 1814). Given a function $f(z) = u(x,y) + iv(x,y)$ where $z = x + iy$, the Cauchy-Riemann equations read,

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y}, \; \frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}. \tag{3.18}$$

If such a function f obeys these equations, then its derivative with respect to z can be easily taken according to,

$$\frac{\partial f}{\partial z} = \frac{\partial u}{\partial x} + i\frac{\partial v}{\partial x}. \tag{3.19}$$

However, in this scenario of CVNN, functions typically do not obey equations in (3.18). These functions are called non-holomorphic, and equation in (3.19) no longer holds true. To compute the derivative of a non-holomorphic function, one must first layout the chain rule between a complex function and its argument's real and imaginary part like so,

$$\begin{cases} \dfrac{\partial f}{\partial z} = \dfrac{\partial f}{\partial x}\dfrac{\partial x}{\partial z} + \dfrac{\partial f}{\partial y}\dfrac{\partial y}{\partial z} \\ \\ \dfrac{\partial f}{\partial \bar{z}} = \dfrac{\partial f}{\partial x}\dfrac{\partial x}{\partial \bar{z}} + \dfrac{\partial f}{\partial y}\dfrac{\partial y}{\partial \bar{z}} \end{cases}, \tag{3.20}$$

being $\bar{z}$ the complex conjugate of z. Here, we are considering also the derivative with respect to $\bar{z}$ because both x and y, have an implicit dependence in z and $\bar{z}$ as show in equation (3.21),

$$\begin{cases} x = \dfrac{z + \bar{z}}{2} \\ \\ y = \dfrac{z - \bar{z}}{2i} \end{cases}. \tag{3.21}$$

We can use this dependency to simplify equation in (3.20) yielding,

$$\begin{cases} \dfrac{\partial f}{\partial z} = \dfrac{1}{2}\left(\dfrac{\partial f}{\partial x} - i\dfrac{\partial f}{\partial y}\right) \\ \\ \dfrac{\partial f}{\partial \bar{z}} = \dfrac{1}{2}\left(\dfrac{\partial f}{\partial x} + i\dfrac{\partial f}{\partial y}\right) \end{cases}. \tag{3.22}$$

These derivative expressions are valid for both holomorphic and non-holomorphic functions, however, one would see that the second is null for the holomorphic case. This is an important fact since, computationally-wise, if one knows from the start that a function is holomorphic: calculating its conjugate derivative is not necessary; a more straight-forward expression can be taken; last but not least, no need for storing unnecessary null data regarding the conjugate derivative and perform further calculations with just zeros.

The function definitions established in Section 3.1.1, will help categorize each one to simplify the back-propagation algorithm as much as possible.

Holomorphic & Non-Holomorphic functions in a CVNN

Pre-Activation Function $\widetilde{h}^{(l)}$ The most common example of a $\widetilde{h}^{(l)}$, is the (complex) weighted sum showed in equation (3.4), which for the sake of this library, only layer logics based on the weighted sum will be used (common weighted sum and convolution). First point is to know if such function is holomorphic, therefore, one needs to apply the Cauchy Riemann equations in 3.18. If it proves that the complex summation function $f_1(z) = z + z_0$ and the complex multiplication function $f_2(z) = zz_0$ are holomorphic, due to the properties of holomorphic functions (Wirtinger 1927), we can prove that the sum of complex products, is holomorphic (which is what a weighted sum is). For f_1 there is,

$$\begin{cases} u(x, y) = x + x_0 \Rightarrow \dfrac{\partial u}{\partial x} = 1,\ \dfrac{\partial u}{\partial y} = 0 \\ \\ v(x, y) = y + y_0 \Rightarrow \dfrac{\partial v}{\partial x} = 0,\ \dfrac{\partial v}{\partial y} = 1 \end{cases}, \tag{3.23}$$

with $z_0 = x_0 + iy_0$. Equation (3.23) shows that the summation function is holomorphic. For f_2,

$$\begin{cases} u(x,y) = xx_0 - yy_0 \Rightarrow \dfrac{\partial u}{\partial x} = x_0,\ \dfrac{\partial u}{\partial y} = -y_0 \\ \\ v(x,y) = xy_0 + yx_0 \Rightarrow \dfrac{\partial v}{\partial x} = y_0,\ \dfrac{\partial v}{\partial y} = x_0 \end{cases}, \tag{3.24}$$

where Cauchy-Riemann Equations also hold. Therefore yields,

$$\widetilde{h}^{(l)}(\mathbf{w}_i^{(l)}, b_i^{(l)}, \mathbf{a}_j^{(l-1)}) = \mathbf{w}_i^{(l)} \cdot \mathbf{a}_j^{(l-1)} + b_i^{(l)},\ \forall \mathbf{w}_i^{(l)}, \mathbf{a}_j^{(l-1)}, b_i^{(l)} \in \mathbb{C}, \tag{3.25}$$

being an holomorphic function. According to equation (3.19), one can determine what are going to be the relevant partial derivatives for the back-propagation algorithm in equation (3.26),

$$\begin{cases} \dfrac{\partial \widetilde{h}^{(l)}}{\partial \mathbf{a}_j^{(l-1)}} = \mathbf{w}_i^{(l)} \\ \\ \dfrac{\partial \widetilde{h}^{(l)}}{\partial \mathbf{w}_i^{(l)}} = \mathbf{a}_j^{(l-1)} \\ \\ \dfrac{\partial \widetilde{h}^{(l)}}{\partial b_i^{(l)}} = 1 \\ \\ \left(\dfrac{\partial \widetilde{h}^{(l)}}{\partial \bar{\mathbf{a}}_j^{(l-1)}} = \dfrac{\partial \widetilde{h}^{(l)}}{\partial \bar{\mathbf{w}}_i^{(l)}} = \dfrac{\partial \widetilde{h}^{(l)}}{\partial \bar{b}_i^{(l)}} = 0 \right) \end{cases}. \tag{3.26}$$

Complex Activation Functions $h^{(l)}$ Considering the set of activation functions present in this library, all happen to be non-holomorphic with the exeption of zReLU. By reviewing the split-function in equation (3.12), one proves that all split-functions are non-holomorphic, by applying the Cauchy-Riemann Equation in equation (3.27),

$$\begin{cases} u(x,y) = u(x) = h_r^{(l)}(\mathcal{R}\{z\}) \Rightarrow \dfrac{\partial u}{\partial x} = \dfrac{\partial h_r^{(l)}}{\partial x},\ \dfrac{\partial u}{\partial y} = 0 \\ \\ v(x,y) = v(y) = h_r^{(l)}(\mathcal{I}\{z\}) \Rightarrow \dfrac{\partial v}{\partial x} = 0,\ \dfrac{\partial v}{\partial y} = \dfrac{\partial h_r^{(l)}}{\partial y} \end{cases}. \tag{3.27}$$

Given the explicit dependency of $u = u(x)$ and $v = v(y)$, its derivatives with respect to x and y respectively, are always going to be different aside from when $x = y$. Nevertheless, since the entire domain of the split-functions is used, differentiation is performed with equation (3.22) resulting in,

$$\begin{cases} \dfrac{\partial h^{(l)}}{\partial z} = \dfrac{1}{2}\left(\dfrac{\partial h_r^{(l)}}{\partial x} + \dfrac{\partial h_r^{(l)}}{\partial y}\right) \\ \\ \dfrac{\partial h^{(l)}}{\partial \bar{z}} = \dfrac{1}{2}\left(\dfrac{\partial h_r^{(l)}}{\partial x} - \dfrac{\partial h_r^{(l)}}{\partial y}\right) \end{cases}. \tag{3.28}$$

Complex Loss Function $h^{(l)}$ Differentiating the complex error function in equation (3.17) is the starting point to determine the entire gradients of the network, since it is going to dictate what is the downward direction to optimize the error for each neuron, and as a consequence, downward direction in the entire loss function surface.

First checking if the error function in equation (3.17), is holomorphic, and for simplicity, considering $\left|\mathbf{a}_n^{(L)} - r_n\right|^2 = |z|^2 = x^2 + y^2$,

$$\begin{cases} u(x,y) == x^2 + y^2 \Rightarrow \dfrac{\partial u}{\partial x} = 2x,\ \dfrac{\partial u}{\partial y} = 2y \\ \\ v(x,y) = 0 \Rightarrow \dfrac{\partial v}{\partial x} = 0,\ \dfrac{\partial v}{\partial y} = 0 \end{cases}. \tag{3.29}$$

So this tells that the complex absolute value function is a non-holomorphic function, therefore, the loss function is only holomorphic for error values of zero (which is going to be never a reachable value). One needs to consider the non-holomorphic derivatives, which after some basic algebra yields,

$$\begin{cases} \dfrac{\partial \mathcal{L}_n}{\partial \mathbf{a}_n^{(L)}} = \bar{\mathbf{a}}_n^{(L)} - \bar{r}_n \\ \\ \dfrac{\partial \mathcal{L}_n}{\partial \bar{\mathbf{a}}_n^{(L)}} = \mathbf{a}_n^{(L)} - r_n \end{cases}. \tag{3.30}$$

Complex Chain Rule

To implement an optimization method based on the complex gradient descent, only rests to define a chain rule for propagating derivatives.

Supposing two generic complex functions A and B have a dependency in z and in $\bar{z}$ in case A and B are non.holomorphic. If there is a complex function $C = B \circ A$, then to compute the chain rule, one must consider z and $\bar{z}$. So, the chain rule for $B \circ A$ where both functions are non-holomorphic is,

$$\begin{cases} \dfrac{\partial (B \circ A)}{\partial z} = \dfrac{\partial B}{\partial A}\dfrac{\partial A}{\partial z} + \dfrac{\partial B}{\partial \bar{A}}\dfrac{\partial \bar{A}}{\partial z} \\ \\ \dfrac{\partial (B \circ A)}{\partial \bar{z}} = \dfrac{\partial B}{\partial A}\dfrac{\partial A}{\partial \bar{z}} + \dfrac{\partial B}{\partial \bar{A}}\dfrac{\partial \bar{A}}{\partial \bar{z}} \end{cases} \tag{3.31}$$

where properties such as,

$$\begin{cases} \dfrac{\partial \bar{A}}{\partial z} = \mathrm{conj}\left(\dfrac{\partial A}{\partial \bar{z}}\right) \\ \\ \dfrac{\partial \bar{A}}{\partial \bar{z}} = \mathrm{conj}\left(\dfrac{\partial A}{\partial z}\right) \end{cases}, \tag{3.32}$$

hold for these derivatives where, for readability, $\mathrm{conj}(z) = \bar{z}$. However, if A is holomorphic,

$$\begin{cases} \dfrac{\partial (B \circ A)}{\partial z} = \dfrac{\partial B}{\partial A}\dfrac{\partial A}{\partial z} \\ \\ \dfrac{\partial (B \circ A)}{\partial \bar{z}} = \dfrac{\partial B}{\partial \bar{A}}\dfrac{\partial \bar{A}}{\partial \bar{z}} \end{cases}, \tag{3.33}$$

else if B is holomorphic,

$$\begin{cases} \dfrac{\partial (B \circ A)}{\partial z} = \dfrac{\partial B}{\partial A}\dfrac{\partial A}{\partial z} \\ \\ \dfrac{\partial (B \circ A)}{\partial \bar{z}} = \dfrac{\partial B}{\partial A}\dfrac{\partial A}{\partial \bar{z}} \end{cases}, \tag{3.34}$$

else if A and B are both holomorphic,

$$\begin{cases} \dfrac{\partial (B \circ A)}{\partial z} = \dfrac{\partial B}{\partial A}\dfrac{\partial A}{\partial z} \\ \\ \dfrac{\partial (B \circ A)}{\partial \bar{z}} = 0 \end{cases}. \tag{3.35}$$

These expressions are very much important in the Complex Back-Propagation algorithm as it will become apparent.

Complex Gradient Descent & Back-Propagation

In the task of optimizing a neural network based on the gradient descent optimization method, the objective is to determine the gradient of the loss function like so,

$$\begin{cases} \dfrac{\partial \mathcal{L}}{\partial \mathbf{w}_i^{(l)}} = \dfrac{\partial \left(\mathcal{L} \circ f\right)}{\partial \mathbf{w}_i^{(l)}} = \dfrac{\partial \left(\mathcal{L} \circ \left(g^{(L)} \circ \ldots \circ g^{(2)} \circ g^{(1)}\right)\right)}{\partial \mathbf{w}_i^{(l)}} \\ \\ \dfrac{\partial \mathcal{L}}{\partial b_i^{(l)}} = \dfrac{\partial \left(\mathcal{L} \circ f\right)}{\partial b_i^{(l)}} = \dfrac{\partial \left(\mathcal{L} \circ \left(g^{(L)} \circ \ldots \circ g^{(2)} \circ g^{(1)}\right)\right)}{\partial b_i^{(l)}} \end{cases}, \tag{3.36}$$

since $g^l = h^{(l)} \circ \widetilde{h}^{(l)}$ and the only adjustable parameters of the network are $\mathbf{w}_i^{(l)}$ and $b_i^{(l)}$ for every $l = 1, 2, ..., L$ and $i = 1, 2, ..., n_l^{(l)}$[2].

This derivative is not straightforward to determine, because the only starting point is from the loss function it to optimize it with an explicit dependence on only last layer's activation $\mathbf{a}_n^{(L)}$. The way one determines the full derivative is by applying the back-propagation algorithm.

[2]The index i is implicitly dependent on l

First, the loss function's dependency with respect to last layer's weights and biases, $\mathbf{w}_n^{(L)}$ and $b_n^{(L)}$ respectively is,

$$\begin{cases} \dfrac{\partial \mathcal{L}}{\partial \mathbf{w}_n^{(L)}} = \dfrac{\partial \left(\mathcal{L} \circ \mathbf{a}_n^{(L)}\right)}{\partial \mathbf{w}_n^{(L)}} \\ \\ \dfrac{\partial \mathcal{L}}{\partial b_n^{(L)}} = \dfrac{\partial \left(\mathcal{L} \circ \mathbf{a}_n^{(L)}\right)}{\partial b_n^{(L)}} \end{cases} . \tag{3.37}$$

Because of the properties mentioned in equation (3.32) and the fact that $\mathcal{L} : \mathbb{C}^{O^{(L)} \otimes n_O^{(L)}} \to \mathbb{R}$ results in,

$$\begin{cases} \dfrac{\partial \mathcal{L}}{\partial \bar{\mathbf{w}}_i^{(l)}} = \text{conj}\left(\dfrac{\partial \mathcal{L}}{\partial \mathbf{w}_i^{(l)}}\right) \\ \\ \dfrac{\partial \mathcal{L}}{\partial \bar{b}_i^{(l)}} = \text{conj}\left(\dfrac{\partial \mathcal{L}}{\partial b_i^{(l)}}\right) \end{cases} . \tag{3.38}$$

Going back to section 3.1.4, $\mathbf{a}_n^{(L)}$ as function of $\mathbf{q}_n^{(L)}$, is a non-holomorphic function just like $\mathcal{L}(\mathbf{a}_n^{(L)})$. Therefore, one can use equation in (3.31) to determine the derivative compositions,

$$\begin{cases} \dfrac{\partial \mathcal{L}}{\partial \mathbf{w}_n^{(L)}} = \dfrac{\partial \mathcal{L}}{\partial \mathbf{a}_n^{(L)}} \dfrac{\partial \left(\mathbf{a}_n^{(L)} \circ \mathbf{q}_n^{(L)}\right)}{\partial \mathbf{w}_n^{(L)}} + \dfrac{\partial \mathcal{L}}{\partial \bar{\mathbf{a}}_n^{(L)}} \dfrac{\partial \left(\bar{\mathbf{a}}_n^{(L)} \circ \mathbf{q}_n^{(L)}\right)}{\partial \mathbf{w}_n^{(L)}} \\ \\ \dfrac{\partial \mathcal{L}}{\partial b_n^{(L)}} = \dfrac{\partial \mathcal{L}}{\partial \mathbf{a}_n^{(L)}} \dfrac{\partial \left(\mathbf{a}_n^{(L)} \circ \mathbf{q}_n^{(L)}\right)}{\partial b_n^{(L)}} + \dfrac{\partial \mathcal{L}}{\partial \bar{\mathbf{a}}_n^{(L)}} \dfrac{\partial \left(\bar{\mathbf{a}}_n^{(L)} \circ \mathbf{q}_n^{(L)}\right)}{\partial b_n^{(L)}} \end{cases} . \tag{3.39}$$

Nevertheless, $\mathbf{a}_n^{(L)}(\mathbf{q}_n^{(L)})$ (result of function $\widetilde{h}^{(L)}$) does not have an explicit dependency in $\mathbf{w}_n^{(L)}$ but $\mathbf{q}_n^{(L)}(\mathbf{w}_n^{(L)}, b_n^{(L)}, \mathbf{a}_m^{(L-1)})$ has (result of function $h^{(L)}$). As shown in section 3.1.4 that $\mathbf{q}_n^{(L)}$ is holomorphic so it is possible to apply equation in (3.34). Establishing the following definitions in equation (3.40) beforehand to make the equations more readable,

$$\begin{cases} \partial \mathcal{L}_n^{(L)} \equiv \dfrac{\partial \mathcal{L}}{\partial \mathbf{a}_n^{(L)}} \dfrac{\partial \mathbf{a}_n^{(L)}}{\partial \mathbf{q}_n^{(L)}} \\ \\ \partial \bar{\mathcal{L}}_n^{(L)} \equiv \dfrac{\partial \mathcal{L}}{\partial \bar{\mathbf{a}}_n^{(L)}} \dfrac{\partial \bar{\mathbf{a}}_n^{(L)}}{\partial \mathbf{q}_n^{(L)}} \end{cases} . \tag{3.40}$$

The derivative of the loss function with respect to the weights and biases of the last layer is,

$$\begin{cases} \dfrac{\partial \mathcal{L}}{\partial \mathbf{w}_n^{(L)}} = \left(\partial \mathcal{L}_n^{(L)} + \partial \bar{\mathcal{L}}_n^{(L)}\right) \dfrac{\partial \mathbf{q}_n^{(L)}}{\partial \mathbf{w}_n^{(L)}} \\ \\ \dfrac{\partial \mathcal{L}}{\partial b_n^{(L)}} = \left(\partial \mathcal{L}_n^{(L)} + \partial \bar{\mathcal{L}}_n^{(L)}\right) \dfrac{\partial \mathbf{q}_n^{(L)}}{\partial b_n^{(L)}} \end{cases} . \tag{3.41}$$

Now the next step is to determine the derivative of the loss function with respect to the weights and biases of the previous to last layer represented in equation (3.42),

$$\begin{cases} \dfrac{\partial \mathcal{L}}{\partial \mathbf{w}_m^{(L-1)}} = \left(\partial \mathcal{L}_m^{(L-1)} + \partial \bar{\mathcal{L}}_m^{(L-1)}\right) \dfrac{\partial \mathbf{q}_m^{(L-1)}}{\partial \mathbf{w}_m^{(L-1)}} \\ \\ \dfrac{\partial \mathcal{L}}{\partial b_m^{(L-1)}} = \left(\partial \mathcal{L}_m^{(L-1)} + \partial \bar{\mathcal{L}}_m^{(L-1)}\right) \dfrac{\partial \mathbf{q}_m^{(L-1)}}{\partial b_m^{(L-1)}} \end{cases} , \tag{3.42}$$

where in this expression, all values are known aside from $\partial \mathcal{L}_m^{(L-1)}$ and $\partial \bar{\mathcal{L}}_m^{(L-1)}$.

The key to the back-propagation algorithm, is determining the derivative of the loss function with respect to the previous to last layer's activation such that one can keep iterating backwards until the input layer ($l = 1$) is reached. Therefore yields,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_m^{(L-1)}} = \sum_n \left[\left(\partial \mathcal{L}_n^{(L)} + \partial \bar{\mathcal{L}}_n^{(L)}\right) \frac{\partial \mathbf{q}_n^{(L)}}{\partial \mathbf{a}_m^{(L-1)}} \right], \quad \frac{\partial \mathcal{L}}{\partial \bar{\mathbf{a}}_m^{(L-1)}} = \operatorname{conj}\left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}_m^{(L-1)}}\right) \tag{3.43}$$

where the sum over the units n, means that each neuron m of layer $L-1$ contributes equally to every neuron's activation of layer L.

Now one can use equation in (3.43) to determine the values of $\partial \mathcal{L}_m^{(L-1)}$ and $\partial \bar{\mathcal{L}}_m^{(L-1)}$ and insert them in equation (3.42) to propagate the derivatives backwards.

This Complex Back-Propagation is implemented by the proposed library and a summary can be visualized in Figure 3.3. In upcoming sections, some small technical details regarding the implementation of this algorithm will be discussed and some other operations relevant to CVNN modeling.

3.2 Structure of the Library

The library is structured in 8 modules:

- `act` : Activation function definitions and interface for computing activation values;
- `cvnn`: Core module that contains structures and tools for modulating CVNN architectures.
- `dataset`: Interface for processing datasets automatically;
- `err`: Minimal error handling enumerations;
- `init`: Enumerations for computing layer initialization techniques;
- `input`: Enumerations for handling static input and output features shapes for layers;

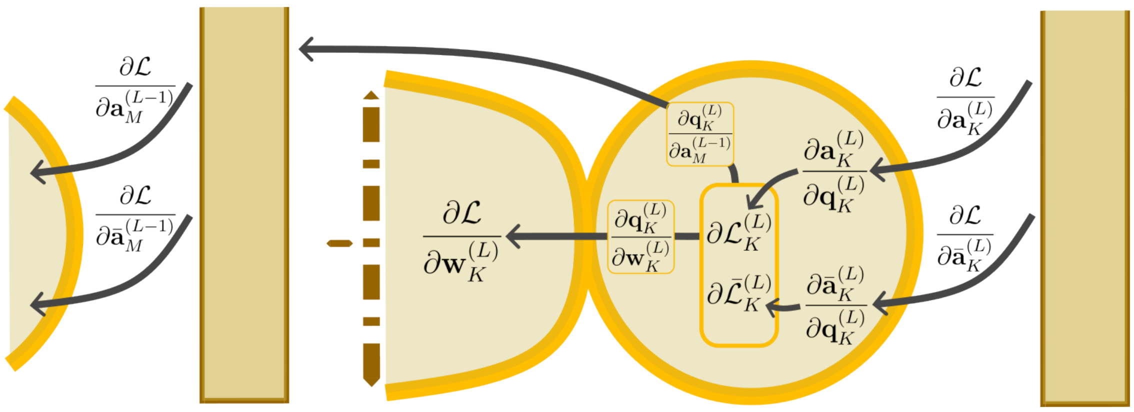

$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_M^{(L-1)}}$
$\frac{\partial \mathcal{L}}{\partial \bar{\mathbf{a}}_M^{(L-1)}}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_K^{(L)}}$
$\frac{\partial \mathbf{q}_K^{(L)}}{\partial \mathbf{a}_M^{(L-1)}}$
$\frac{\partial \mathbf{q}_K^{(L)}}{\partial \mathbf{w}_K^{(L)}}$
$\partial \mathcal{L}_K^{(L)}$
$\partial \bar{\mathcal{L}}_K^{(L)}$
$\frac{\partial \mathbf{a}_K^{(L)}}{\partial \mathbf{q}_K^{(L)}}$
$\frac{\partial \bar{\mathbf{a}}_K^{(L)}}{\partial \mathbf{q}_K^{(L)}}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_K^{(L)}}$
$\frac{\partial \mathcal{L}}{\partial \bar{\mathbf{a}}_K^{(L)}}$

Figure 3.3: Depiction summarizing the complex back-propagation algorithm, at the neuron level, used in Renplex. Arrows indicate calculations and one can see that, computational-wise, the majority of the derivatives can be calculated in parallel. Derivatives related to the bias were omitted for a more concise illustration and here $M = n_O^{(L-1)}$ and $K = n_O^{(L)}$
.

- `math`: Various mathematical utilities concerning, for instance, complex float, matrix definition and operations and random number generation;
- `opt`: Definition and computing loss function and its derivatives.

A brief overview on the core module `cvnn`.

`cvnn` Module The core module `cvnn`, only possesses two additional modules: `layer` and `network`. `layer` module is arguably the most important as it contains all the layers implemented in the library each one with its own logic, being:

- `DenseCLayer`: Contains a matrix of weights, vector of biases (as many as there is units), and an activation function. It represents a fully connected layer that performs the common weighted sum in equation (3.4) given a vector of scalar input features;
- `Flatten`: Flattens out a vector of matrix input features into a vector of scalar input features;
- `ConvCLayer`: Contains a matrix of weights (filters/kernels), a vector of biases and an activation function. This layer computes the convolution between every matrix input feature against every filter. The number of filters will dictate the number of output features (matrices);
- `Reduce`: A generalized pooling operation that asks for the user the operation to be applied on each block of each input feature matrix. Each matrix block of every input feature will be reduced into a scalar for down-sampling data.

The `network` module only defines the `struct` of a network, being a vector of layers, and underlying operations such as running the complex gradient descent optimization for a batch of data, adjusting the weights of the layers accordingly, calculating loss and accuracy, and forwarding and intercepting signals.

Having this quick layout of the library, the main implemented features, operations and critical algorithms will be addressed.

More details on the structure of the library can be viewed in the Github repository Pxdr0-A/renplex.git, it is also published in the crates.io, the official package manager for Rust in https://crates.io/crates/renplex. Documentation for this library can also be accessed in https://docs.rs/renplex/0.1.1/renplex/.

3.3 Main Implementations

To give some insights on the operation and performance of the library, in this section, there will be an overview on the most critical tasks executed in the library.

3.3.1 Matrix-Vector Multiplication

The Rust implementation for this operations goes as follows:

```rust
pub fn mul_slice(
  &self,
  rhs: &[T]
) -> Result<Vec<T>, OperationError> {

  // error handling
  ...

  // map rows of the matrix
  let res = self.rows_as_iter().map(|elm| {
    // create a row and vec iterator
    elm.into_iter()
    .zip(rhs.iter())
    // perform scalar product
    .fold(T::default(), |acc, elm| {
      acc + (*elm.0 * *elm.1)
    })
  }).collect::<Vec<_>>();

  Ok(res)
}
```

Listing 3.1: Overview on the Matrix-Vector multiplication implemented in the Renplex Library using Rust.

Here, the slice `rhs:&[T]` is representing a vector and `&self` is a reference in memory to a matrix. Afterwards, rows of the matrix are taken and mapped to a folded value that represents the (complex) scalar product between matrix row and vector. The collection of these folded values forms the result vector.

The majority of these algorithms were built with some flexibility in case there is need for parallelization or eventual GPU utilization. This algorithm is easy to add concurrency in the aspect that each folded value resultant from scalar matrix row and vector, can be computed independently.

3.3.2 2D Convolution

As for the 2D convolution, the high-level layout of the algorithm used can be viewed in Listing 3.2.

```rust
pub fn convolution(
  &self,
  kernel: &Self
) -> Result<Self, OperationError> {

  // extract kernel and matrix shape
  let k_shape = kernel.get_shape();
  let initial_shape = self.get_shape();

  // error handling
  ...

  // determine the final shape
  let final_shape = [
    initial_shape[0] - (k_shape[0]-1),
    initial_shape[1] - (k_shape[1]-1)
  ];

  // go through the number of final rows
  let convolved_body = (0..final_shape[0])
    .into_iter()
    .flat_map(|i| {
      // get a retangular matrix to slide the kernel
      let slider = self.get_slider(i, k_shape[0]);

      // slide the kernel
      let conv_row = slider.slide(&kernel)

      // returns a convolved row
      conv_row
  }).collect::<Vec<_>>();

  // convert a flatten matrix to the final shape
  Ok(convolved_body.to_matrix(final_shape).unwrap())
}
```

Listing 3.2: Overview on the Convolutional Product implemented in the Renplex library using Rust.

The way this convolution is going to be performed is by calculating the convolution per rows of the final shape. This is, it is going to get a "slider" from the matrix which is requesting as many rows as there are rows in the kernel[3] and make the kernel slide through it to compute the scalar product per block. This yields a convolved row that will collected into a vector later converted to a matrix as the result.

[3]For example, if the kernel shape is 3×4 it will request 3 rows.

Such implementation guarantees also an easy parallelization if one wishes to compute convolved rows concurrently by getting a reference to a slider of the matrix in whatever thread needed.

Some examples of convolutions performed by this algorithm can be found in Figure 3.4, with an example of a handwritten 4 in the MNIST dataset (LeCun, Cortes, and Burges 1998).

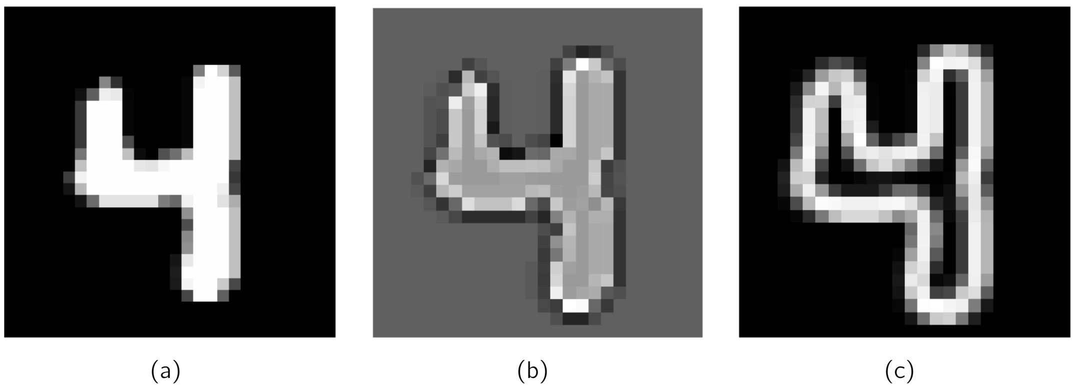


Figure 3.4: A handwritten "4" of the MNIST dataset in (a), and the results for the applications of two different convolutional filters using the Renplex library. In (b), the application of a 3×3 shapen filter and in (c) the application of the sobel operator (Sobel 2014).

In this three Figures we have in 3.4(a) the original image, in 3.4(b) the application of the sharpen filter,

$$\begin{pmatrix} 0.0 & -1.0 & 0.0 \\ -1.0 & 5.0 & -1.0 \\ 0.0 & -1.0 & 0.0 \end{pmatrix},$$

and in 3.4(c) the application of the Sobel Operator (Sobel and Feldman 1968), both filters with a size of 3×3.

In both of these procedure there is the downsize of the image because the convolution is reducing a 3×3 block to 1 value, thus effectively reducing dimension by 2 rows and 2 columns, in this case.

3.3.3 2D Down-Sampling & Up-Sampling

A crucial set of operations involved in convolutional neural networks is the and down-sampling up-sampling.

Down-Sampling For the down-sampling it was used a general pooling operation similar to the `skimage.measure.block_reduce` function present in the scikit-image (Van der Walt et al. 2014) python library, it divides the matrix in blocks and reduces them to single values based on a certain function. This can be viewed on Listing 3.3.

```
pub fn block_reduce(
  &self,
  block_size: &[usize],
    block_func: impl Fn(&[T]) -> T
) -> Result<Self, OperationError> {

    let matrix_shape = self.get_shape();

    // error handling
    ...

    // determine the final shape
    let final_shape = [
      matrix_shape[0] / block_size[0];
      matrix_shape[1] / block_size[1];
    ];

    // go through the number of final rows
    let reduced_body = (0..final_shape[0])
      .into_iter()
      .flat_map(|i| {
        // get a retangular matrix to slide the kernel
        let slider = self.get_slider(i, block_size[0]);

        // reduce the slider per block
        // (stride of block_size[1])
        let reduced_row = slider
          .reduce(block_size, block_func)

        // returns the reduced slider
        reduced_row
    }).collect::<Vec<_>>();

    // convert a flatten matrix to the final shape
    Ok(reduced_body.to_matrix(final_shape).unwrap())
}
```

Listing 3.3: Overview on the Reduce operation on images implemented in the Renplex library using Rust.

It is very similar to the 2D convolution, however it adds a stride of `block_size[1]` and a custom operation on the block aside from weighted sum (scalar product). Concurrency can be used in the same way as the convolution.

Up-Sampling This library only implements on up-sampling technique which is fractional up-sampling and it is used here to propagate the derivatives through a reduce layer. Such technique, receives a block shape saying how much one wants to increase the size of the image by padding in between pixel values (let us call it inner padding), and a kernel that performs a padded convolution after this inner padding is performed. This implementation is slightly bigger than the above ones mentioned so we are just going to show a high-level implementation on Listing 3.4.

```
pub fn fractional_upsampling(
    &self,
    block_size: &[usize],
    kernel: &Self
  ) -> Result<Matrix<T>, OperationError> {

    let matrix_shape = self.get_shape();
    let mut matrix_rows = self.rows_as_iter();

    // error handling
    ...

    // appropriate padding size wrapping pixles
    let inner_pad = calc_inner_pad(block_size);

    let final_shape = [
      matrix_shape[0] * block_size[0],
      matrix_shape[1] * block_size[1]
    ];

    let mut res = Vec::new();
    let n_rows = matrix_shape[0];
    // add inner paddings to the entire image
    // go through all pixels in the image
    for _ in 0..n_rows {
      // add upper padding
      for _ in 0..inner_pad[0] {
        // add as many rows as upper paddings
        res.extend(vec![T::default(); final_shape[1]]);
      }
      // add row with padding in between
      for row_elm in matrix_rows.next().unwrap() {
        // left inner pad
        res.extend(vec![T::default(); inner_pad[1]]);
        res.push(*row_elm);
        // right inner pad
        res.extend(vec![T::default(); inner_pad[2]]);
      }
      // add lower padding
      for _ in 0..inner_pad[3] {
        // add as many rows as lower paddings
        res.extend(vec![T::default(); final_shape[1]]);
      }
    }

    // create a result matrix
    let matrix_res = Matrix::from_body(
      res,
      [final_shape[0], final_shape[1]]
    );
    let kernel_shape  = kernel.get_shape();

    // perform padded convolution so that
    // dimensionality is not lost
    // and values are added in between
    let out = matrix_res
      // by default the filter should be [3, 3]
      .pad((kernel_shape[0]-2, kernel_shape[0]-2))
      .convolution(kernel)
      .unwrap();

    Ok(out)
}
```

Listing 3.4: Overview on the Fractional Up-sampling implementation in the Renplex library using Rust.

Typically the kernel used in fractional up-sampling is,

$$\begin{pmatrix} 0.25 & 0.50 & 0.25 \\ 0.50 & 1.00 & 0.50 \\ 0.25 & 0.50 & 0.25 \end{pmatrix},$$

where, to up-sample both real and imaginary parts equally in a complex image, this kernel has to be composed of real values.

On Figure 3.5 we give an example of an average pooling being performed, and reverting back to the original dimensions with fractional up-sampling of an image with real pixels. Process reads from 3.5(a) to 3.5(c).

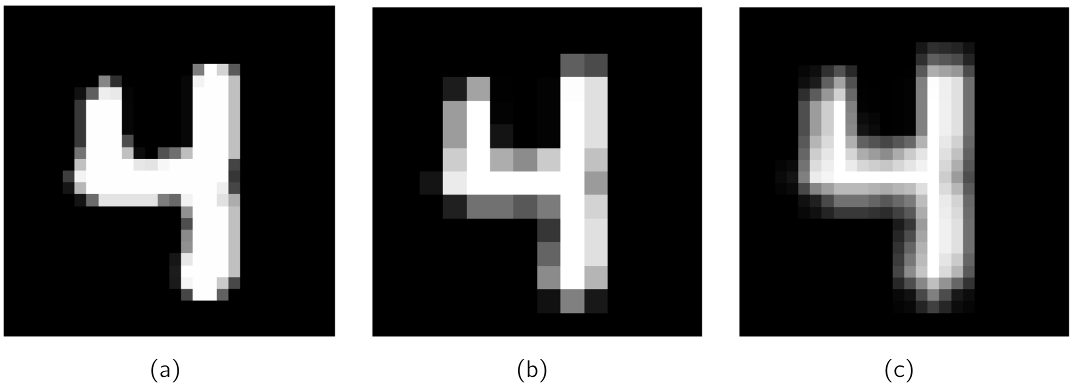


Figure 3.5: A handwritten "4" of the MNIST dataset in (a), and the results for the applications of average pooling (b) and fractional up-sampling (c) in succession using the Renplex library

3.3.4 Layer Initialization

Various methods for layer initialization were implemented in this library but they circle around a core method of generating complex numbers. When generating random complex numbers, this library generates a random number between 0 and a certain value defined by the user.

This value is going to be the absolute value of the complex number, then it generates a random phase between $(0, 2\pi)$. The complex number is then converted to real and imaginary part.

In spite of initialization methods like He Initialization (He et al. 2015), Xavier, Xavier Glorot (uniform and normal distribution) (Glorot and Bengio 2010) being used in RVNN, an adaptation of this methods were implemented. Each initialization method's distribution targets the absolute value of the complex number and the phase is again generated according to a uniform distribution.

Listing 3.5 shows how to generate two random complex numbers with this adapted versions Xavier Glorot (Uniform) and He initialization.

```rust
use renplex::init::InitMethod;

let ref mut seed: &mut u128 = 63478262957;

// number of input features of a certain layer
let ni: usize = 64;
// number of output features of a certain layer
let no: usize = 16;

// value generated according to Xavier Glorot Uniform
let xav_glo_num = InitMethod::XavierGlorotU(ni + no)
  .gen(seed);

// value generated according to He Initialization
let he_init_num = InitMethod::HeInit(ni)
  .gen(seed);
```

Listing 3.5: Demonstration of how to generate complex random numbers with Xavier Glorot Uniform and He initialization respectively.

since each one of these methods his based of on either $n_I^{(l)}$, $n_O^{(l)}$ or both.

3.3.5 Complex Back-Propagation

Initially, two pipelines for the complex back-propagation were tested. The current implemented pipeline can be viewed in Listing **??**.

```rust
pub fn gradient_opt(
  &mut self,
  data: Dataset<T, T>,
  loss_func: &ComplexLossFunc,
  lr: T
) -> Result<(), ForwardError> {

  let n_layers = self.layers.len();

  // error handling
  ...

```

```
    // weights and biases bgradients to accumulate
    let mut dldw_per_layer = Vec::new();
    let mut dldb_per_layer = Vec::new();
    let mut _total_params: usize = 0;

    // allocation of memory for the gradients
    // ...

    // make an iterator of the data points
    let (inputs, targets) = data.points_into_iter();

    let batch_size = inputs.len();
    for (input, target) in inputs.zip(targets) {
      // collect all activations of the network
      let mut activations = self
        .collect_acts(input)
        .unwrap()
        .into_iter()
        .rev();

      // initial prediction
      let initial_pred = activations.next().unwrap();
      // initial value of loss derivatives
      let mut dlda = T::d_loss(
        initial_pred,
        target,
        &loss_func
      ).unwrap().to_vec();
      // initial conjugate derivative of loss
      let mut dlda_conj: Vec<T> = dlda
        .iter()
        .map(|elm| { elm.conj() })
        .collect();

      // reversed layers iterator
      let rev_l = self.layers.iter().rev();
      // layers and activations reversed iterator
      let act_l_rev = activations.zip(rev_l).enumerate();
      // activations are consumed here recursively
      for (l, (prev_act, layer)) in act_l_rev {
        // check if layer propagates derivatives
        if layer.propagates() {
          let dldw; let dldb;
          // compute derivatives of the layer l
          // loss and conj loss derivative are being updated
          (dldw, dldb, dlda, dlda_conj) = layer
            .comp_grad(&prev_act, dlda, dlda_conj)
            .unwrap();

          // update gradients of the batch
          dldw_per_layer[n_layers-1-1].add_slice_mut(&dldw)
            .unwrap();
          dldb_per_layer[n_layers-1-1].add_slice_mut(&dldb)
            .unwrap();
        }
      }
      // drop unecessary memory
      drop(dlda); drop(dlda_conj);
    }
    // divide the gradient by the count of data samples
    let scale_param = lr / T::usize_to_complex(batch_size);

    // iterator containing the gradients per layer
    let update_iter = dldw_per_layer
      .into_iter()
      .zip(dldb_per_layer.into_iter())
      .zip(self.layers.iter_mut());

    update_iter.for_each(|((mut dldw, mut dldb), layer)| {
      if layer.propagates() {
        // scale the gradients
        dldw.mul_mut_scalar(scale_param).unwrap();
        dldb.mul_mut_scalar(scale_param).unwrap();
        // adjust the weights and biases of that layer
        layer.neg_conj_adjustment(dldw, dldb).unwrap();
      }
    });

    Ok(())
  }
```

Listing 3.6: Overview of the gradient optimization algorithm (using back-propagation) implementation in the Renplex library using Rust.

Where in Listing **??**, `self` now is a network `struct`/instance. This gradient optimization, for each data point in the batch, collects all activations of the network, reverses them, and starts clearing this features as they are needed for the derivative computation of each layer. After accumulating all of the calculated derivatives, it averages them out and multiplies by the learning rate and uses the result to adjust the weights and biases of each respective layer.

The alternative approach (not shown here as a listing), consumes much less memory, because it fetches the activation needed for the current derivative computation. However, the downside is that it has to forward the signal every time it needs to fetch and activation, yielding a considerable performance overhead. For the purpose of this library, and analyzed datasets, there were no problems concerning memory, nevertheless, it is a good future alternative to consider for more memory consuming networks.

3.4 Library Usage

To create a CVNN in the library, one must first define the layers to be added by initializing them as preferred. Let us see how to instantiate every single layer that the library implements so far.

3.4.1 Layer Definition

`DenseCLayer` One initializes a complex fully connected layer by indicating the input shape that it will receive, number of units, a CAF, a weight initialization method and a seed for random number generation. With the number of input features (scalars) and the number of units, the initialization function will generate a matrix of weights using the initialization method provided. An example of initialization can be visualized in Listing 3.7.

```rust
use renplex::math::cfloat::Cf32;
use renplex::input::IOShape;
use renplex::act::ComplexActFunc;
use renplex::cvnn::layer::dense::DenseCLayer;

let ref mut seed: &mut u128 = 63478262957;

// define complex number with 64bits
// 32 bits for each real and imaginary part
type Precision = Cf32;

// number of scalar input features
let ni = 64;
// number of scalar output features (units)
let no = 16;

// input features are scalars (vetor of values)
let input_shape = IOShape::Scalar(ni);

let dense: DenseCLayer<Precision> = DenseCLayer::init(
  input_shape,
  no,
  ComplexActFunc::RITSigmoid,
  InitMethod::XavierGlorotU(ni + no),
  seed
).unwrap();
```

Listing 3.7: Initialization of a complex dense layer.

`ConvCLayer` To initialize a convolutional layer one must give the number of input features and filters (which will be the number of output features), the sizes of the filters, activation function, kernel initialization method and a seed. The depth of the filters is the number of input features n_i and is automatically assigned in the initialization. This becomes apparent on Listing 3.8.

```rust
use renplex::math::Complex;
use renplex::math::cfloat::Cf32;
use renplex::input::IOShape;
use renplex::act::ComplexActFunc;
use renplex::cvnn::layer::conv::ConvCLayer;

let ref mut seed: &mut u128 = 63478262957;

// complex number with 64bits
// 32 bits for each real and imaginary part
type Prec = Cf32;

// number of input features
// i.e. number of channels of the images
let ni = 1;
// number of filters
let filters = 8;
// shape of the filters
// (without depth)
let f_shape = [3, 3];

let conv_layer: ConvCLayer<Prec> = ConvCLayer::init(
  IOShape::Matrix(1),
  filters,
  k_size,
  ComplexActFunc::RITReLU,
  InitMethod::HeInit(filters * f_shape[0] * f_shape[1]),
  seed
).unwrap();
```

Listing 3.8: Initialization of a complex convolutional layer.

The way this layer propagates derivatives fundamentally lies in equation (3.43), however, with method of calculation that is noteworthy.

$\partial\mathcal{L}_L^{(n)}$ and $\partial\bar{\mathcal{L}}_L^{(n)}$ are easy to compute since they are based on straightforward element-wise operations, i.e. one can just flatten out any matrix/tensor to perform the operations. The nuance, lies in the derivative of the pre-activation $\mathbf{q}_L^{(n)}$ with respect to the weights $\mathbf{w}_L^{(n)}$ and the activation of the previous layer $\mathbf{a}_{L-1}^{(m)}$. Turns out that, if one simplifies all weighted sum expressions underlying the convolution operation, the entire derivative can boil down also convolution operations. With some basic algebra, from these derivatives yield,

$$\begin{cases} \dfrac{\partial \mathbf{q}_n^{(L)}}{\partial \mathbf{w}_n^{(L)}} = \mathbf{a}_m^{(L-1)} * \mathbf{q}_n^{(L)} \\[2ex] \dfrac{\partial \mathbf{q}_n^{(L)}}{\partial \mathbf{a}_m^{(L-1)}} = \mathbf{q}_n^{(L)} *_p \operatorname{flip}\left(\mathbf{w}_n^{(L)}\right) \end{cases} \tag{3.44}$$

where $*$, $*_p$ represents the convolution and padded convolution product respectively and flip(.) indicates the operation of flipping a matrix. This is the way Renplex is calculating the convolutional derivatives.

`Reduce` Due to the amount of operations involved, although it does not carry any parameters of the network, the `Reduce` layer has a rather extensive initialization process. One needs to define, how many matrix input features will receive, the block shape of the operation, a block function that reduces a matrix block to a scalar and finally an interpolation kernel that will be used in the back-propagation algorithm to up-sample the derivatives to previous layers. In Listing 3.9, an example of an average pooling layer using the `Reduce` layer abstraction.

```
use renplex::math::Complex;
use renplex::math::cfloat::Cf32;
use renplex::cvnn::layer::reduce::Reduce;

// define complex number with 64bits
// 32 bits for each real and imaginary part
type Prec = Cf32;

// number of input features
let ni = 8;
// block shape
let b_shape = [2, 2];
let block_func = |block: &[Prec]| {
  // length of the flatten block
  let b_len = block.len();
  // sum of the elements
  let sum = block.into_iter().reduce(|acc, elm| {
    acc + *elm
  });

  // calculate the mean and return it
  sum / Prec::complex_to_usize(b_len)
};
// utility for quick access to the sharpen filter
let interp_k = Matrix::get_sharp_kernel();

let avg_pooling: Reduce<Prec> = Reduce::init(
  ni,
  b_shape,
  // store block_func in the heap
  Box::new(block_func),
  interp_k
).unwrap();
```

Listing 3.9: Initialization of a Mean Pooling layer using the Reduce layer abstraction.

In training, `Reduce` will fractional up-sample $\frac{\partial \mathcal{L}_n}{\partial \mathbf{a}_n^{(l)}}$ to $\frac{\partial \mathcal{L}_n}{\partial \mathbf{a}_m^{(l-1)}}$, and respective conjugate.

`Flatten` The flatten layer is a straightforward layer to define. One simply needs to define the shape of the input features and how many there are as shown in Figure 3.10.

```rust
use renplex::math::Complex;
use renplex::math::cfloat::Cf32;
use renplex::cvnn::layer::flatten::Flatten;

// input matrix shapes
let input_shape = [28, 28];
// number of input features
let ni = 8;

let flatten_layer: Flatten = Flatten::init(
  input_shape,
  ni
);
```

Listing 3.10: Initialization of a flatten layer.

3.4.2 Network Construction

After initializing a layer, that layer can be added in a `CNetwork` instance. However, this `stuct` only receives one type, the reason being is because, if it was a set of types belonging to a certain class, it would needed to rely on heap memory usage[4]. To tackle this problem, the library offers a static interface (enumeration) (Listing 3.11) to map a layer through the `wrap()` method, to a common type `CLayer` where it contains all the possibilities of layers developed inside the library.

```rust
pub enum CLayer<T> {
  Dense(DenseCLayer<T>),
  Convolutional(ConvCLayer<T>),
  Reduce(Reduce<T>),
  Flatten(Flatten)
}
```

Listing 3.11: Enumeration of the general layer interface.

To add the first layers one must use the `add_input()` method. From that point forward, the `add()` method will keep adding layers to the network where the last added layer is assumed to be the output layer. After that, the `gradient_opt()` method in Listing 3.6 can calculate the gradients for a certain batch of data to update the network with.

```rust
use renplex::math::Complex;
use renplex::math::cfloat::Cf32;
use renplex::dataset::Dataset;
use renplex::opt::ComplexLossFunc;
```

[4]Accessing memory from the heap, typically introduces some performance overhead when compared to accessing in the stack (Stroustrup 2013)

```rust
use renplex::cvnn::layer::CLayer;
use renplex::cvnn::network::CNetwork;

let mut network: CNetwork<Cf32> = CNetwork::new();

network.add_input(conv_layer.wrap()).unwrap();
network.add(avg_pooling.wrap()).unwrap();
network.add(flatten.wrap()).unwrap();
network.add(dense.wrap()).unwrap();

// define loss function
let loss_func = ComplexLossFuntion::Conventional;
// history of loss function values
let loss_vals = Vec::new();
// define a real learning rate
let learning_rate = Cf32::new(1.0, 0.0);

// initialize a batch of data
let mut data_batch: Dataset<Cf32, Cf32> = Dataset::new();
// extract a unique batch of data points
// can be done in any logic (default order, randomized, ...
 )
for _ in 0..batch_size {
  // collect data points from a file
  let data_point = ...;
  // add point to the dataset
  data_batch.add_point(data_point);
}

// calculate the initial loss for the batch of data
let loss = network::loss(
  data_batch,
  &loss_func
).unwrap()
// add loss value to history
// (for optimization algorithms for instance)
loss_vals.push(loss);

// train 1 batch of data
network.gradient_opt(
  data_batch,
  &loss_func,
  learning_rate
).unwrap();

// this pipeline can be repeated to perform an epoch
// and repeated again for as  many epochs choosen
```

Listing 3.12: A short example on how to build a batch of data and train it in a previously constructed network.

With this lower level approach, one can implement any optimization algorithm with the learning rate and change it accordingly to loss functions values or performance metric. The pipeline can also be repeated as many epochs it is needed to reach the desired results.

Chapter 4

Library Evaluation

In this Chapter, the library will be evaluated against two simple datasets: MNIST (LeCun, Cortes, and Burges 1998) and a synthetically generated signal reconstruction dataset. A comparison will be made with equivalent RVNN architectures using Tensorflow for the mentioned datasets. First it will given some brief insights on the performance of this library given its architecture, then some studies regarding adequate learning rates for CVNN modulated by the library while being compared with learning rates on RVNN. Then training some light ANN architectures against these datasets, namely, Multi-Layer Perceptron and a small Convolutional Neural Network. Meanwhile results will be presented alongside a small discussion on each network's performance.

4.1 Performance Remarks

With the development of this library, the focus was not on achieving comparable levels of performance as with RVNN frameworks such as Tensorflow. The library was developed without concurrency features, however, it was built with that in mind for adding concurrency if needed. Minimal performance and memory usage optimizations circled around, not cloning data structures and recycling memory as much as possible via references, condensing derivative computations in common loops, and compiling the code with optimization flags.

Having this in consideration for the following results, here are the resultant average runtime of each dataset and architecture per batch:

	FC (22886)	AE (87760)	CNN (32410)
SR	-	531ms	-
MNIST	23ms	-	936ms

Table 4.1: Batch run-times in mili-seconds, for the two datasets used (SR stands for the synthetic signal reconstruction dataset) and different architectures with respective number of parameters in parenthesis. FC stands for Fully connected (Multi-Layer Perceptron), AE is Auto-Encoder, CNN is convolutional neural network.

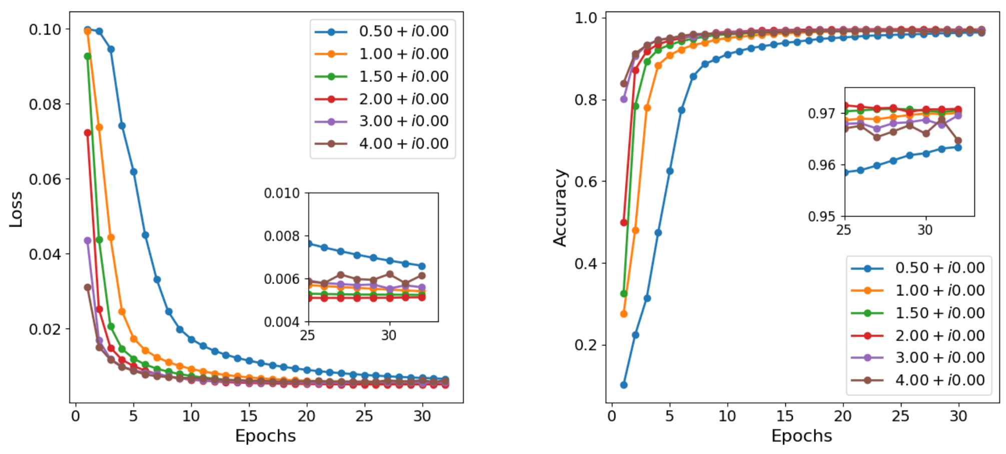
4.2 `Renplex`'s Learning Rates

To perform the upcoming tests, for each dataset it was performed a series of test with regards to the learning rate. In here, it will be shown for the example of the MNIST dataset using a fully connected architecture showed in Section 4.3.1, how the learning rate was selected. It is noteworthy that the MNIST dataset is not a dataset with complex values, therefore, to feed it to a CVNN, one simply defines each pixel as a "complex value" where the imaginary component is null.

To keep the tests short so that the computations would not become to much time expensive, we decided to limit the number of epochs and fit the learning rate so that the model learns in the interval. Fully connected networks learn slower when compared to convolutional neural networks so it was used 32 and 16 epochs respectively. The criteria used for selecting a learning rate for the referred interval were simple:

- Able to stabilize the model for around 5 epochs in the local minimum;
- Exhibit little to no oscillations at the local minimum.

Figure 4.1 shows various learning curves with test results (for different learning rates) where values around 1.5 and 2.0 appear to be good choices.



(a) (b)

Figure 4.1: Test loss and accuracy curves for the MNIST dataset concerning various real learning rates on the complex-valued multi-layer perceptron for 32 epochs. As the learning rate increases, convergence time improves but also stability decreases.

In the study (Huisheng Zhang and Mandic 2015), shows that it might be beneficial to use an imaginary learning rate to speed up training and achieve slightly results. One of the options is to have a specific adaptive learning rate that can be implemented with the library, however due to its complexity, its implementation was skiped. The other option is to have a small imaginary component, that it is not as beneficial as the adaptive approach but also showed some improvements. The latter was tested and Figure 4.2 shows the results where one can see that there is this slight benefit in convergence time but, for this case, the local minimum value is around the same for each curve.

The process discussed just now, was also applied for every Tensorflow architecture developed. In the same context, Figure 4.3 shows learning curves obtained with equivalent architecture with Tensorflow. One can notice that, such learning rates are slightly higher and learning

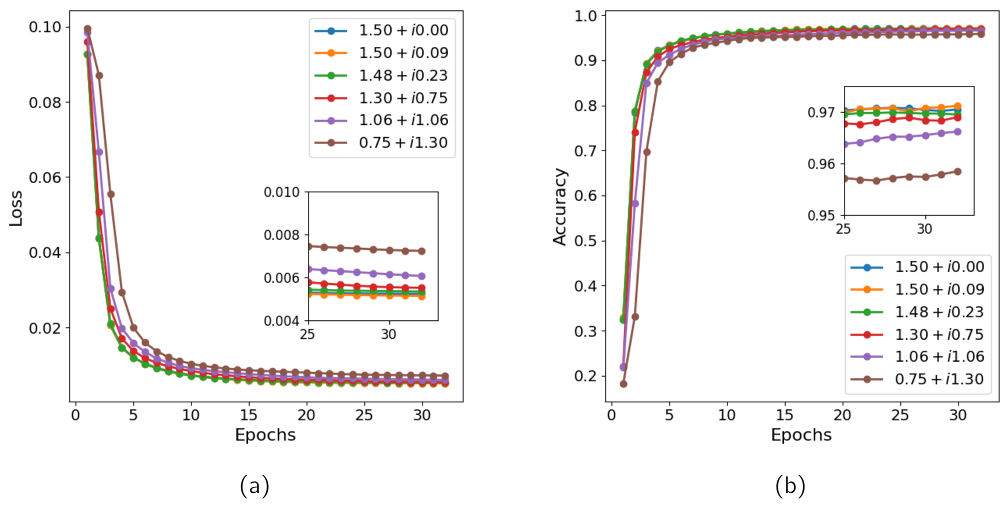


Figure 4.2: Test loss and accuracy curves for the MNIST dataset concerning various imaginary learning rates on the complex-valued multi-layer perceptron for 32 epochs. The studied learning rates follow an increase in phase with values 0, $\frac{\pi}{20}$, $\frac{\pi}{10}$, $\frac{\pi}{6}$, $\frac{\pi}{4}$ and $\frac{\pi}{3}$ with absolute value of 1.50.

curves more unstable when compared with the CVNN architecture, in other words, Tensorflow, has more difficulty training the model in the same amount of epochs that the CVNN does with stability (and higher accuracy).

4.3 Fully Connected Neural Network Applications

The current section will show two distinct applications of fully connected CVNN. First, a layout for classification using the MNIST dataset, then an auto-encoder type of architecture using dense layers for signal reconstruction. For the purpose of proving that the library works as intended, while minimizing runtime, the focus will be on very simple architectures constructed for the purpose of this Master Thesis only.

On a side note, it is important to note that, since the implementation of cross entropy Loss function is not well defined or explored in the literature, we used the mean squared error loss function for both Renplex and Tensorflow. This is without loss of generality since the derivative of the cross entropy (in composition with the softmax activation that is necessary), is the same as the derivative of the mean squared error, so the derivatives are propagated equally in both scenarios. This only plays a roll in optimization algorithms that pick up the loss function values for determining next iterations of the learning rate, which is also something that we will not be addressing in this simple case of studies.

4.3.1 MNIST Dataset

The task at hand with this dataset is: given a single channel input image of 28 $\times$ 28 pixels containing a handwritten digit, train the network to identify which digit is. For such, that output layer will need have 10 units where the neuron with greatest activation classifies the digit it represents. Based on the Multi-Layer Perceptron architecture (Rumelhart, Hinton,

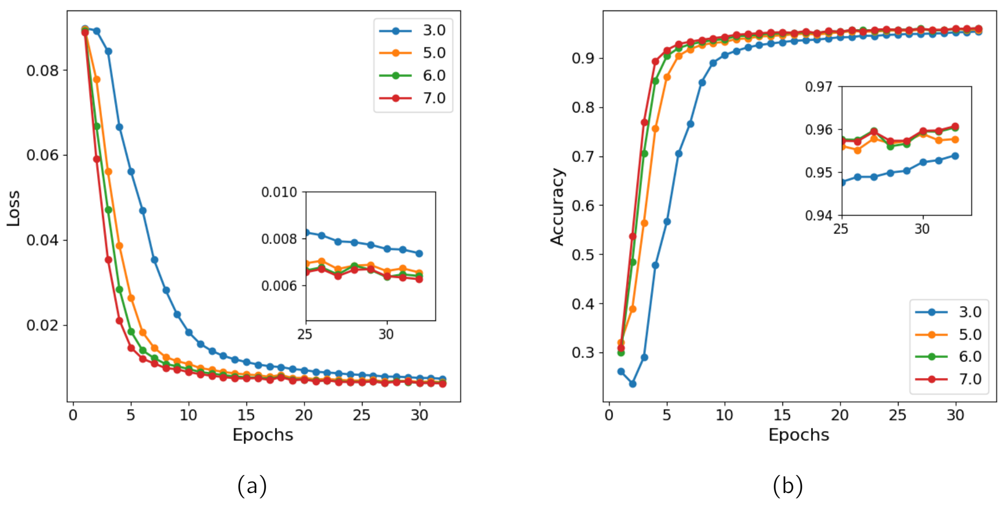


Figure 4.3: Test loss and accuracy curves using Tensorflow for the MNIST dataset concerning various learning rates on the multi-layer perceptron for 32 epochs.

and Williams 1986), we will suggest for this study the following architecture consisting of 5 layers in total:

- 1 - Flatten Layer to map the 28×28 matrix to a vector of dimension 784;
- 2 - Dense Layer, 28 units, (split) sigmoid activation;
- 3 - Dense Layer, 16 units, (split) sigmoid activation;
- 4 - Dense Layer, 16 units, (split) sigmoid activation;
- 5 - Dense Layer, 10 units, (split) sigmoid activation.

The differences between RVNN and CVNN architectures are marked with the parenthesis. Instead of having the sigmoid activation function, one has an equivalent one which we choose to be and Real-Imaginary-Type or split sigmoid function,

$$\sigma_S(z) = \sigma\left(\mathcal{R}\{z\}\right) + i\sigma\left(\mathcal{I}\{z\}\right). \tag{4.1}$$

With this setup the following results in Figure 4.4 were obtained. These results, as the followings, were obtained by running 6 different random seeds through 32 epochs and averaging out all points of all learning curves obtained, then drawing the mean learning curve inside a region with the size of plus/minus 2 times the standard deviation of each epoch value. For both networks, it was used batches of 100 images and the default data split with the MNIST dataset (the same for all instances where we study MNIST in this master thesis).

The results show that the CVNN over-performed the RVNN and even offered greater stability when reaching the local minimum. Additionally, CVNN seemed to have converged in less epochs compared to the RVNN. We would also like to note that the selected learning rates were in accordance with the pipeline in Section 4.2, as will all the results in this chapter.

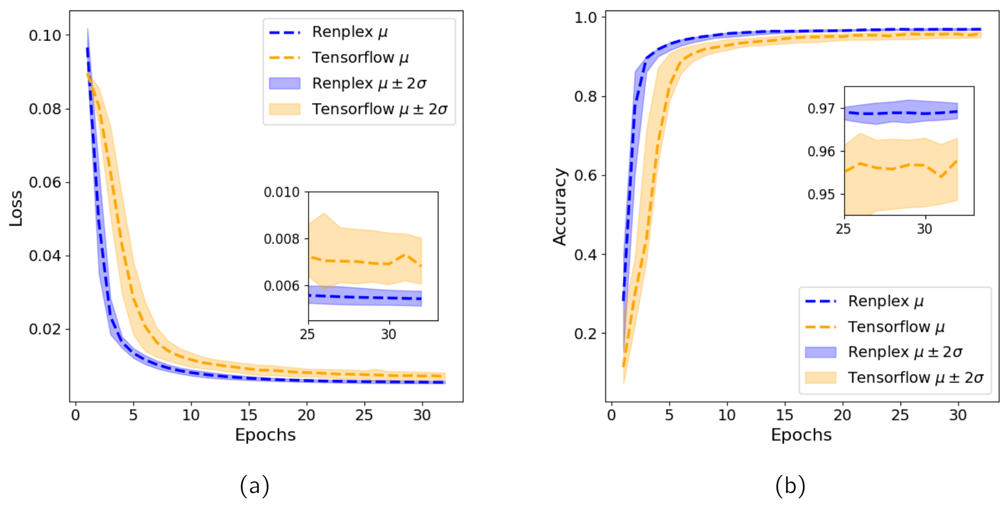


Figure 4.4: Comparison of mean test loss and accuracy curves between Renplex and Tensorflow for the MNIST dataset with equivalent multi-layered perceptron-based neural network models. The μ represents the average curve of all learning curves concerning 6 seeds, and 2σ two times the standard deviation related to every loss and accuracy value for each of the 32 epochs.

4.3.2 Synthetic Signal Reconstruction Dataset

In this next test of the library,the task that was performed was discussed in the state-of-the-art as very common to be executed by a CVNN. The network will be provided a noisy (complex) signal and its objective is to guess what the clean signal $s(t)$ is behind the noise. For this task we generated a Synthetic Dataset.

Method for Generating the Dataset

To generate this dataset we first considered a plain wave signal,

$$y(t) = \rho e^{i(\omega t+\phi)} \tag{4.2}$$

where ρ is the amplitude, ω frequency and ϕ the phase of the signal. In a more practical sense, t is the sampling vector or simply the time variable.

One way one could generate the dataset is by generating a various random combinations of ρ, ω and ϕ, produce a signal $y_i(t)$, which would be the dependent variable, and then add Gaussian noise to it to get $x_i(t)$ being each respective independent variable. Nevertheless, for this dataset we decided to form more complex (normalized) signals composed of multiple plain waves like so,

$$y(t) = C\sum_{n}^{N} \rho_n e^{i(\omega_n t+\phi_n)}, \tag{4.3}$$

where C is some normalization constant, the amplitudes and frequencies ρ_n and ω_n respectively, were both randomly uniformly distributed between 0.1 and 1 and all ϕ_n were uniformly distributed between 0 and 2π. The number of waves N, for each signal $y(t)$ of the dataset, were integers randomly uniformly distributed between 1 and a given limit number of waves to add which we considered 8 for upcoming tests. To have on average at least one complete cycle of the signal, the vector t goes from 0 to 4π with 512 samples.

To generate the dependent variables, i.e. noisy input signal to the network $x(t)$, one would just add Gaussian noise $g_\sigma(t)$ with a mean of 0 and a defined standard deviation governed by a threshold σ_0 which in the following cases was 0.025. To have a proportional noise to intensity, the former was used and a threshold for noise and increased with the sample's intensity by the following expression,

$$\sigma = \sigma_0 \left(1 + \frac{|y(t)|}{20}\right), \tag{4.4}$$

for every given instant t such that $x(t) = y(t) + g_\sigma(t)$. On Figure 4.5 we show four examples of pairs $(x(t), y(t))$ used for the dataset were one can see that multiple amplitudes and frequencies might be contained on the signal and that the noise remains proportional throughout different absolute values of the samples.

Regarding training and testing data, 20000 pairs of noisy and clean signal respectively were generated for both training and testing pipeline. All formed with random number of waves each with its own unique and random amplitudes, frequencies and phases.

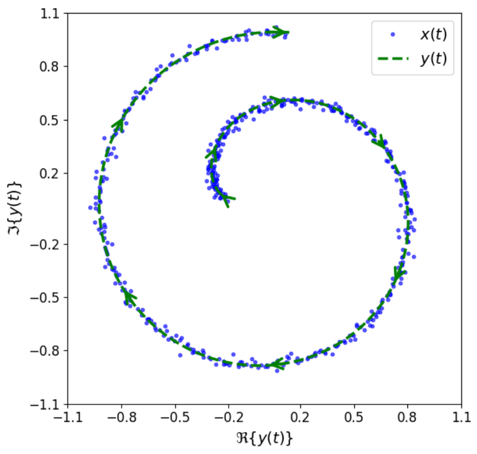
Results

To extract the results in Figure 4.6, it was used the CVNN architecture equivalent to the Auto-Encoder one. It consisted of:

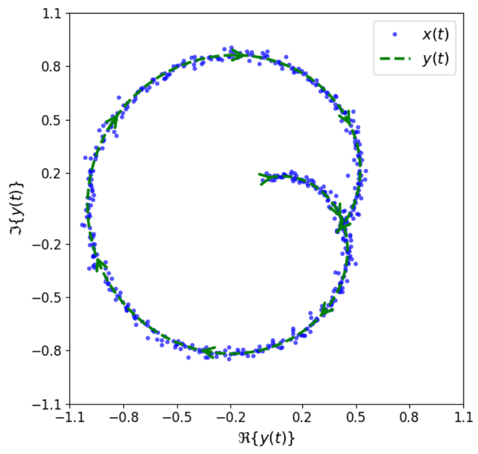
- 1 - Dense Layer, 128 units, (split) hyperbolic tangent activation. (This layer receives the signal encoded in a vector with 512 elements);
- 2 - Dense Layer, 32 units, (split) hyperbolic tangent activation;
- 3 - Dense Layer, 16 units, (split) hyperbolic tangent activation;
- 4 - Dense Layer, 32 units, (split) hyperbolic tangent activation;
- 5 - Dense Layer, 512 units, no activation.

However, since the signal is complex in its nature, one needs to find a way to feed it to a RVNN. In Tensorflow, it was created two equal networks where one trains on the real part of the signal and the other on the imaginary part of the signal. The predicted outputs of each model are combined in a single output and loss is then calculated according to the error in equation (3.15). This studies in Figure 4.6 were performed for 32 epochs and to select the learning rate an additional note was taken into account: output signals must have little to no signal-to-noise-ratio. If a small enough learning rate is given in such a task, the network simply tries to minimize the mean squared loss with no consideration whatsoever to the noise of the prediction, yielding at the end a very good result in terms of loss, but something that does not resembles a clear signal.

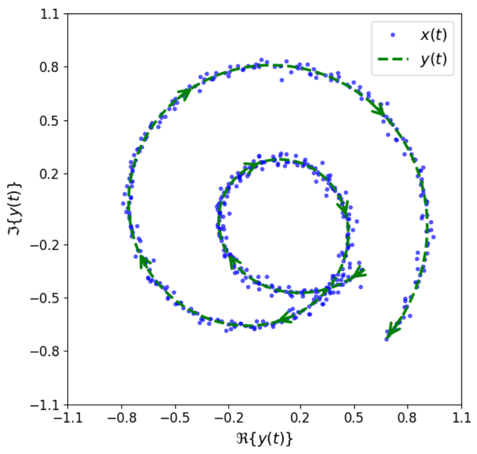
These results show again that a CVNN model outperforms RVNN model and also offer more stability while converging to a local minimum, in spite of initiating training at slightly worst loss values. Additionally it also trains quicker epoch-wise since it soon over-took the RVNN more or less at the 4 epoch mark. The capability of generalization becomes apparent if we visualize some predictions made from both CVNN and RVNN in Figure 4.7. This figure



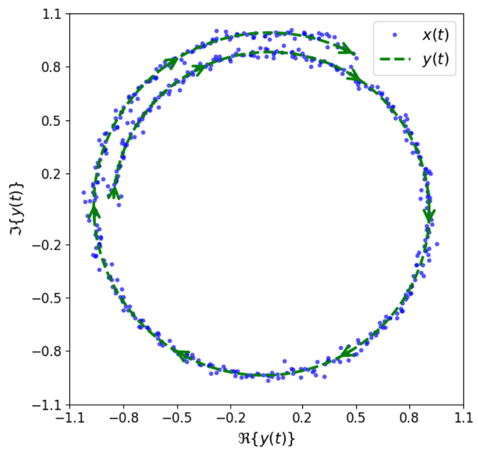
(a)



(b)



(c)



(d)

Figure 4.5: Four pairs of independent $x(t)$ and dependent variables $y(t)$ from the signal reconstruction dataset. Samples are represented in the complex plane, in green the supposed smooth signal and in blue dots the signal with Gaussian Blur. Arrows indicate the direction of propagation of the signal.

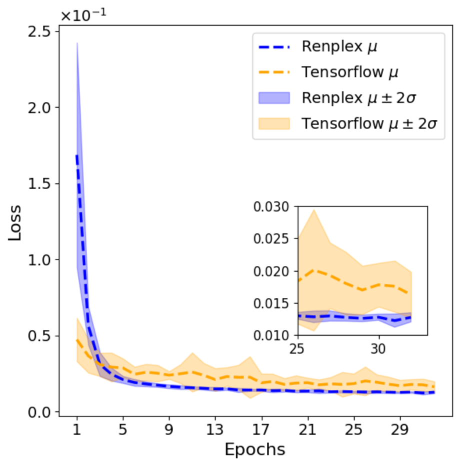


Figure 4.6: Comparison of mean test loss and accuracy curves between Renplex and Tensorflow for the Synthetic Signal Reconstruction dataset with equivalent auto-encoder-based neural network models. The μ represents the average curve of all learning curves concerning 6 seeds, and 2σ two times the standard deviation related to every loss and accuracy value for each of the 32 epochs.

suggests that, a RVNN has difficulties in continuously curving the signal in the complex plane, executing less smooth curves when compared to a CVNN prediction. Another disadvantage of the RVNN model is that it was used a "split-model" where one creates two independent models, one to deal with the real part and another to deal with the imaginary part of the signal, which is double the memory. The other option for the RVNN would be to have an input of 1024 (far all real and imaginary values separately) instead of 512 but that would be changing the architecture to much. Additionally, it actually produced less accurate results when compared to the split model.

4.4 Convolutional Neural Network Applications

Since the library is also capable of performing 2D convolutions, this section shows a simple application of a CVNN in comparison with a RVNN, in the same classification task with the MNIST dataset. This example will consist in a light Convolutional Neural Network architecture where the convolutional operation inside the complex convolution layer is performed according to what was described in Section 3.3.2.

4.4.1 MNIST Dataset

It was decided to use the MNIST dataset and an architecture that did not involved a considerable amount of parameters. It consisted in:

- 1 - Convolutional Layer, 8 filters, 3×3 kernel size, (split) ReLU activation;
- 2 - Average Pooling Layer (showed in Section 3.9), 2×2 block;
- 3 - Convolutional Layer, 16 filters, 3×3 kernel size, (split) ReLU activation;
- 4 - Dense Layer, 16 units, (split) sigmoid activation;

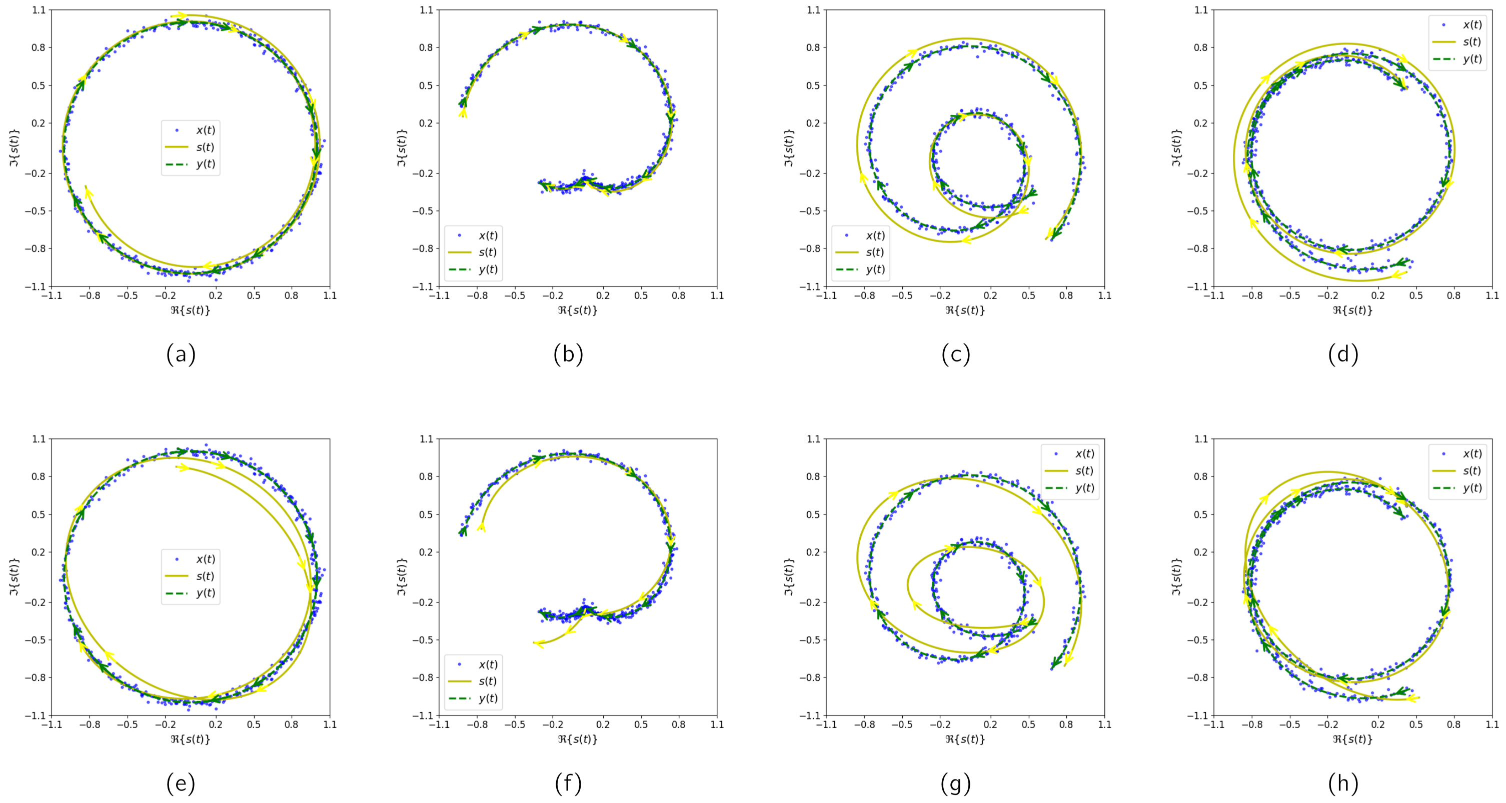
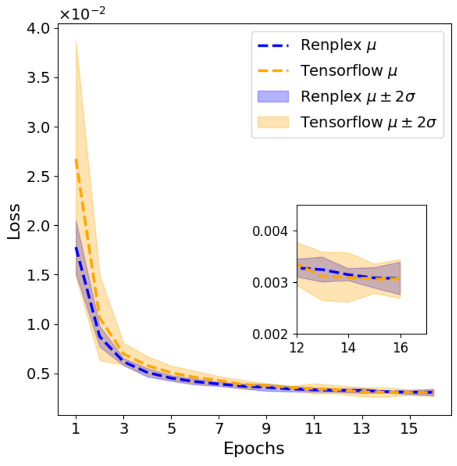


Figure 4.7: A set of 4 predictions in yellow ($s(t)$) executed from by the Renplex model from (a) to (d) and Tensorflow model(s) from (e) to (h). Tensorflow here is using two models in parallel, one for dealing with the real and the other for the imaginary part. $x(t)$ is the Gaussion noise version of $y(t)$ and $s(t)$ the attempt for reconstruction of $y(t)$. Signals in (a) and (e) are pure plane waves and the rest contain multiple frequencies (up to 8).

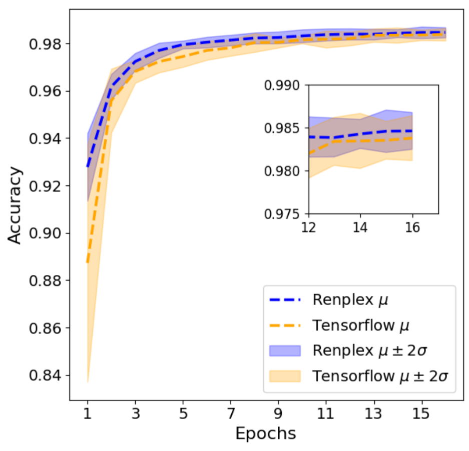
- 5 - Dense Layer, 10 units, (split) sigmoid activation.

The 8 input filters, make sure that first layer's activation does not store a lot of features, substantially reducing memory usage, and to make sure that the convolution layer performs the major role on the task, a reduced number of Dense units was added. Tests ran for 16 epochs due to these types of networks training faster but also more time expensive. Result on Figure 4.8, show a comparable performance between the RVNN and CVNN both in accuracy and solution stability.

In spite of the network modeling being almost analogous between RVNN and CVNN, it is important to notice that Tensorflow does not incorporate fractional up-sampling as a method for propagating derivatives backwards through the average polling layer. In fact, it has a method much more reliable almost mimicking the inverse process of the average pooling (Abadi et al. 2023). In Renplex's case, for easiness in implementation and quick release of the library, we decided to have a common operation for propagating derivatives through the Reduce Layer which is not ideal and may hinder performance.



(a)



(b)

Figure 4.8: Comparison of mean test loss and accuracy curves between Renplex and Tensorflow for the MNIST dataset with equivalent convolutional-based neural network models. The μ represents the average curve of all learning curves concerning 6 seeds, and 2σ two times the standard deviation related to every loss and accuracy value for each epoch.

When the process of training these models was finished, we extracted the output feature maps from the first convolutional layer (input layer) and collected them in Figures 4.9, 4.10 and 4.11. In the first, we show output feature maps from the RVNN modeled by Tensorflow where one can see some high-level features like image smoothing, contrasts and edge detection. In the second and third results, we show the complex output features maps from two different perspectives in the same CVNN.

It might be confusing to see MNIST images colored, however, since inside the CVNN features have complex-valued pixels, we decided to map on Figure 4.10, the real part of the pixel to shades of red and the imaginary part in shades of green. In Figure 4.10, the absolute values of the pixel are mapped to shades of red and their phases in shades of green. One can see from these results, that from both perspectives, the CVNN tried to capture some high-level features and transitions. Sometimes bringing the input image completely to the imaginary axis, or sometimes encoding sections of the input image with different phase values.

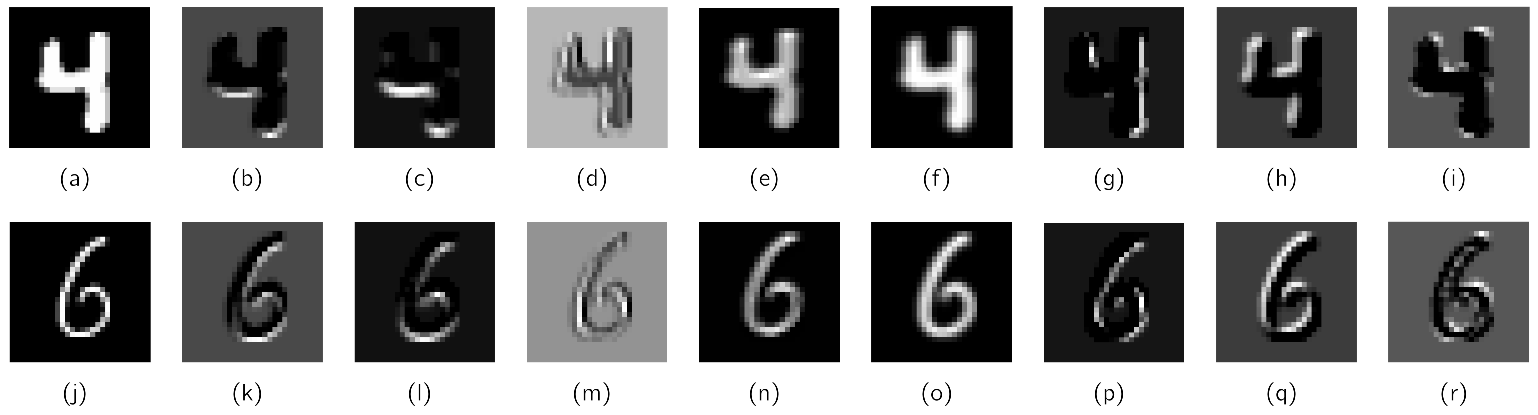


Figure 4.9: Feature maps extracted from the first convolutional layer of a (real-valued) convolutional neural network with 8 filters using Tensorflow concerning the MNIST dataset. Original images are (a) and (j) for different handwritten numbers of "4" and "6" respectively.

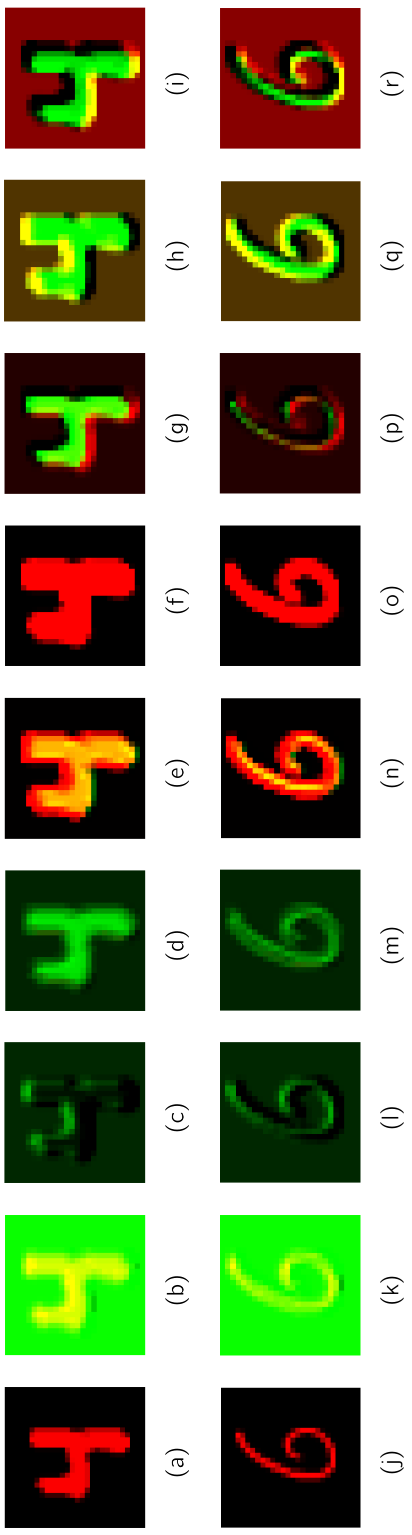


Figure 4.10: Feature maps extracted from the first convolutional layer of a complex-valued convolutional neural network with 8 filters using Tensorflow concerning the MNIST dataset. Original images are (a) and (j) for different handwritten numbers of "4" and "6" respectively. In this set, real part of the pixels is mapped in red scale and imaginary in green scale.

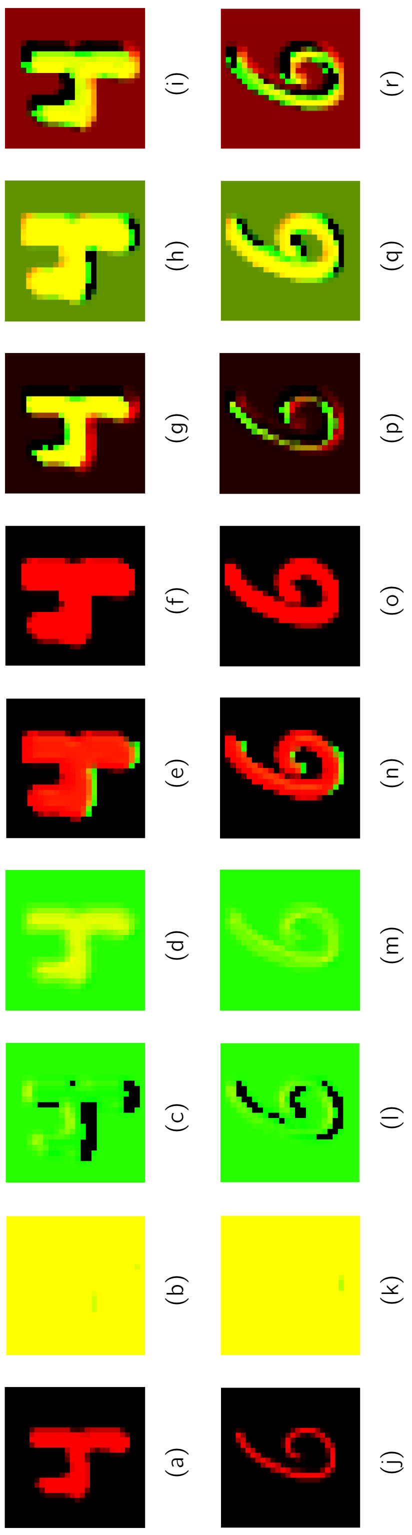


Figure 4.11: Feature maps extracted from the first convolutional layer of a complex-valued convolutional neural network with 8 filters using Tensorflow concerning the MNIST dataset. Original images are (a) and (j) for different handwritten numbers of "4" and "6" respectively. In this set, absolute value of the pixels is mapped in red scale and phase in green scale.

Chapter 5

Conclusions & Future Work

This Master Thesis delivers an open-source library for modulating Complex-Valued Neural Networks, with an overview on its inner-workings, how to interact with the library and a few simple case studies. It was provided a contextualization in Chapter 1 giving a reason why these types of neural networks are relevant to be explored and its potential. Then, in Chapter 2, a brief exploration of the State-of-the-art to see what are the most common functionalities, components and ways to modulate a CVNN, as well as, some major application where CVNN have a promising impact. It was also performed a search on already existent libraries that modulate CVNN and concluded that is still an incomplete field in need of tools for exploration of these networks. In Chapter 3, was established the mathematical foundation necessary for CVNN moduling together with some core algorithms and procedures developed. Lastly, the library was evaluated in Chapter 4 against the MNIST and a Synthetic Signal Reconstruction datasets, with multiple architectures namely Multi-Layer Perceptron, Auto-Encoder and Convolutional Neural Networks.

Despite being one author involved in the development of this library, this Master Thesis, pinpoints a solid start for CVNN exploration at a small scale with scaling capabilities. The reason being is the fact that the library was scaffold using a fast and secure system's programing language (Rust (Klabnik and Nichols 2018)) and easy interaction at the development level for improving core algorithms until a comparable performance with already existent RVNN libraries is reached. Instances of new layers and even learning algorithms unique to CVNN, can be easily added in future versions. Nevertheless, this library is of difficult use to inexperienced/unfamiliar users with ANN/Rust.

Results obtained by CVNN modulated in the library, typically outperform in loss and/or accuracy, have better generalization capabilities, take less epochs to train and offer more stability on local minima when compared to RVNN just as described in the literature (Chapter 2). Nevertheless, due to the additional computations and memory usages involve in operations with complex numbers, CVNN experience more performance overhead and compared to RVNN. For this purposes, regularization, was not needed so far.

Regarding Future Work, in spite of this library being ready to be used for some applications or research, there is still a many of missing features. The next steps of development would be to implement the non-gradient based approach Multi-Valued Neuron to optimize a CVNN, more layer instances like 1D convolution and recurrent layers, and more activation functions like the ones described in Section 2.2.1 with equal to greater potential as the core ones already implemented. Nevertheless, the most urgent steps to take is to optimize critical and core procedures in the library. This can be done, for instance, by the creation of thread pools when initiating the library to manage specific tasks associated with the forwarding and back-propagating a signal with concurrency, having an interface of predefined arrays to be stored in the stack memory (for quick memory access) or memory optimization in basic complex operations.

A very important study which could be conducted, would be a fair evaluation of runtime performance between RVNN and CVNN[1]. Having Renplex fully optimized in every possible aspect, it would be of upmost importance to check if a CVNN is effectivelly faster than a RVNN since the former takes less epochs to train a model, despite the performance over-head introduced by complex computations. Also, comparing over-fitting thresholds between CVNN and RVNN would be equally interesting.

The library presented in this document is available and open for contributions on the GitHub address Pxdr0-A/renplex.git, and the author has the intention to give some further improvements on the library's core functionalities and documentation, where at the time, the latter can still be improved.

[1]At the time of writing this thesis, such study was not yet addressed in the literature to the best of the authors knowledge.

Bibliography

Abadi, Martín et al. (2023). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. `https://www.tensorflow.org/`.

Abdalla, Rayyan (2023). "Complex-valued Neural Networks–Theory and Analysis". In: *arXiv preprint arXiv:2312.06087*.

Aizenberg, Igor, Shane Alexander, and Jacob Jackson (2011). "Recognition of Blurred Images Using Multilayer Neural Network Based on Multi-valued Neurons". In: *2011 41st IEEE International Symposium on Multiple-Valued Logic*, pp. 282–287. doi: `10.1109/ISMVL.2011.24`.

Aizenberg, Igor, Dmitriy V. Paliy, et al. (2008). "Blur Identification by Multilayer Neural Network Based on Multivalued Neurons". In: *IEEE Transactions on Neural Networks* 19.5, pp. 883–898. doi: `10.1109/TNN.2007.914158`.

Aizenberg, NN et al. (1973). "Multivalued threshold functions: II. Synthesis of multivalued threshold elements". In: *Cybernetics* 9.1, pp. 61–77.

Amari, Shun-ichi (1995). "Information geometry of the EM and em algorithms for neural networks". In: *Neural networks* 8.9, pp. 1379–1408.

Amilia, Sindi, Mahmud Dwi Sulistiyo, and Retno Novi Dayawati (2015). "Face image-based gender recognition using complex-valued neural network". In: *2015 3rd International Conference on Information and Communication Technology (ICoICT)*. IEEE, pp. 201–206.

Arjovsky, Martin, Amar Shah, and Yoshua Bengio (2016). "Unitary evolution recurrent neural networks". In: *International conference on machine learning*. PMLR, pp. 1120–1128.

Barrachina, J Agustin (Nov. 2022). *NEGU93/cvnn: Complex-Valued Neural Networks*. Version v2.0. doi: `10.5281/zenodo.7303587`. url: `https://doi.org/10.5281/zenodo.7303587`.

Barrachina, Jose Agustin et al. (2023). "Theory and Implementation of Complex-Valued Neural Networks". In: *arXiv preprint arXiv:2302.08286*.

Bassey, Joshua, Lijun Qian, and Xianfang Li (2021). "A survey of complex-valued neural networks". In: *arXiv preprint arXiv:2101.12249*.

Benvenuto, N. and F. Piazza (1992). "On the complex backpropagation algorithm". In: *IEEE Transactions on Signal Processing* 40.4, pp. 967–969. doi: `10.1109/78.127967`.

Cauchy, Augustin L. (1814). *Mémoire sur les intégrales définies*. Vol. 1. Oeuvres complètes. Published in 1882. Paris, pp. 319–506.

Cauchy, Augustin-Louis (1847). "Méthode générale pour la résolution des systèmes d'équations simultanées". In: *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences* 25, pp. 536–538.

Çevik, Hasan Hüseyin, Yunus Emre Acar, and Mehmet Çunkaş (2018). "Day ahead wind power forecasting using complex valued neural network". In: *2018 International Conference on Smart Energy Systems and Technologies (SEST)*. IEEE, pp. 1–6.

Ceylan, Murat and Hüseyin Yaçar (2013). "Blood vessel extraction from retinal images using complex wavelet transform and complex-valued artificial neural network". In: *2013 36th International Conference on Telecommunications and Signal Processing (TSP)*. IEEE, pp. 822–825.

Chatterjee, Soumick et al. (2022). “Complex Network for Complex Problems: A comparative study of CNN and Complex-valued CNN”. In: *2022 IEEE 5th International Conference on Image Processing Applications and Systems (IPAS)*. Vol. Five, pp. 1–5. doi: `10.1109/IPAS55744.2022.10053060`.

Chiheb, Trabelsi, O Bilaniuk, D Serdyuk, et al. (2017). “Deep complex networks”. In: *International Conference on Learning Representations*.

Chiheb Trabelsi, et al. (2017). “Deep Complex Networks”. In: *arXiv preprint arXiv:1705.09792*.

Clarke, T.L. (1990). “Generalization of neural networks to the complex plane”. In: *1990 IJCNN International Joint Conference on Neural Networks*, 435–440 vol.2. doi: `10.1109/IJCNN.1990.137751`.

Cole, Elizabeth et al. (2021). “Analysis of deep complex-valued convolutional neural networks for MRI reconstruction and phase-focused applications”. In: *Magnetic resonance in medicine* 86.2, pp. 1093–1109.

Cole, Elizabeth K et al. (2020). “Analysis of deep complex-valued convolutional neural networks for MRI reconstruction”. In: *arXiv preprint arXiv:2004.01738*.

Cox, David R. (1958). “The Regression Analysis of Binary Sequences”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 20.2, pp. 215–242. url: `http://www.jstor.org/stable/2983890`.

Cruz, Ariadne Arrais, Kayol Soares Mayer, and Dalton Soares Arantes (n.d.). “RosenPy: An Open Source Python Framework for Complex-Valued Neural Networks”. In: *Available at SSRN 4252610* ().

Dramsch, Jesper Sören and Contributors (2019). *Complex-Valued Neural Networks in Keras with Tensorflow*. doi: `10.6084/m9.figshare.9783773`. url: `https://figshare.com/articles/Complex-Valued_Neural_Networks_in_Keras_with_Tensorflow/9783773/1`.

Du, Hang, Rebecca Pillai Riddell, and Xiaogang Wang (2023). “A hybrid complex-valued neural network framework with applications to electroencephalogram (EEG)”. In: *Biomedical Signal Processing and Control* 85, p. 104862.

Eberhart, R. and J. Kennedy (1995). “A new optimizer using particle swarm theory”. In: *MHS’95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pp. 39–43. doi: `10.1109/MHS.1995.494215`.

European Commission (2021). *Proposal for a Regulation of the European Parliament and of the Council laying down harmonised rules on Artificial Intelligence (Artificial Intelligence Act) and amending certain Union legislative acts*. Official Journal of the European Union. COM/2021/206 final. url: `https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:52021PC0206`.

Fisher, Ronald A. (1936). “The Use of Multiple Measurements in Taxonomic Problems”. In: *Annals of Eugenics* 7.2, pp. 179–188. doi: `10.1111/j.1469-1809.1936.tb02137.x`.

Georgiou, G.M. and C. Koutsougeras (1992). “Complex domain backpropagation”. In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 39.5, pp. 330–334. doi: `10.1109/82.142037`.

Gleich, Dusan and Danijel Sipos (2018). “Complex valued convolutional neural network for TerraSAR-X patch categorization”. In: *EUSAR 2018; 12th European Conference on Synthetic Aperture Radar*. VDE, pp. 1–4.

Glorot, Xavier and Yoshua Bengio (2010). “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS) 2010*. Vol. 9. JMLR Workshop and Conference Proceedings, pp. 249–256.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. PMLR, pp. 315–323. url: `http://proceedings.mlr.press/v15/glorot11a.html`.

Gu, Shenshen and Lu Ding (2018). “A Complex-Valued VGG Network Based Deep Learing Algorithm for Image Recognition”. In: *2018 Ninth International Conference on Intelligent Control and Information Processing (ICICIP)*, pp. 340–343. doi: `10.1109/ICICIP.2018.8606702`.

Guberman, Nitzan (2016). “On complex valued convolutional neural networks”. In: *arXiv preprint arXiv:1602.09046*.

Gürüler, Hüseyin and Musa Peker (2015). “A software tool for complex-valued neural network: CV-ANN”. In: *2015 23nd Signal Processing and Communications Applications Conference (SIU)*, pp. 2054–2057. doi: `10.1109/SIU.2015.7130272`.

Hafiz, Abdul Rahman, Md. Faijul Amin, and Kazuyuki Murase (2011). “Real-Time Hand Gesture Recognition Using Complex-Valued Neural Network (CVNN)”. In: *Neural Information Processing*. Ed. by Bao-Liang Lu, Liqing Zhang, and James Kwok. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 541–549. isbn: 978-3-642-24955-6.

Hayakawa, Daichi, Takashi Masuko, and Hiroshi Fujimura (2018). “Applying Complex-Valued Neural Networks to Acoustic Modeling for Speech Recognition”. In: *2018 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pp. 1725–1731. doi: `10.23919/APSIPA.2018.8659610`.

He, Kaiming et al. (2015). “Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification”. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034.

Hirose, Akira (2012). *Complex-valued neural networks*. Vol. 400. Springer Science & Business Media.

Hirose, Akira and Shotaro Yoshida (2012a). “Generalization Characteristics of Complex-Valued Feedforward Neural Networks in Relation to Signal Coherence”. In: *IEEE Transactions on Neural Networks and Learning Systems* 23.4, pp. 541–551. doi: `10.1109/TNNLS.2012.2183613`.

– (2012b). “Generalization Characteristics of Complex-Valued Feedforward Neural Networks in Relation to Signal Coherence”. In: *IEEE Transactions on Neural Networks and Learning Systems* 23.4, pp. 541–551. doi: `10.1109/TNNLS.2012.2183613`.

Hofmann, Thomas, Bernhard Schölkopf, and Alexander J Smola (2008). “Kernel methods in machine learning”. In.

Hong, Xia et al. (2014). “Single-Carrier Frequency Domain Equalization for Hammerstein Communication Systems Using Complex-Valued Neural Networks”. In: *IEEE Transactions on Signal Processing* 62.17, pp. 4467–4478. doi: `10.1109/TSP.2014.2333555`.

Hu, Yanxin et al. (2020). “DCCRN: Deep complex convolution recurrent network for phase-aware speech enhancement”. In: *arXiv preprint arXiv:2008.00264*.

Jia, Lina, Bin Yang, and Wei Zhang (2018). “Research on Stock Forecasting Based on GPU and Complex-Valued Neural Network”. In: *Intelligent Computing Theories and Application*. Ed. by De-Shuang Huang, Kang-Hyun Jo, and Xiao-Long Zhang. Cham: Springer International Publishing, pp. 120–128. isbn: 978-3-319-95933-7.

Kataoka, Maki, Makoto Kinouchi, and Masafumi Hagiwara (1998). “Music information retrieval system using complex-valued recurrent neural networks”. In: *SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218)*. Vol. 5. IEEE, pp. 4290–4295.

Kim, Taehwan and T. Adali (2000). “Fully complex backpropagation for constant envelope signal processing”. In: *Neural Networks for Signal Processing X. Proceedings of the 2000 IEEE Signal Processing Society Workshop (Cat. No.00TH8501)*. Vol. 1, 231–240 vol.1. doi: 10.1109/NNSP.2000.889414.

Kim, Taehwan and Tülay Adali (2002). “Fully complex multi-layer perceptron network for nonlinear signal processing”. In: *Journal of VLSI signal processing systems for signal, image and video technology* 32, pp. 29–43.

Kim, Taehwan and Tulay Adali (2000). “Fully complex backpropagation for constant envelope signal processing”. In: *Neural Networks for Signal Processing X. Proceedings of the 2000 IEEE Signal Processing Society Workshop (Cat. No. 00TH8501)*. Vol. 1. IEEE, pp. 231–240.

Kingma, Diederik P and Jimmy Ba (2014). “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980*.

Kinouchi, Makoto and Masafumi Hagiwara (1996). “Memorization of melodies by complex-valued recurrent network”. In: *Proceedings of International Conference on Neural Networks (ICNN’96)*. Vol. 2. IEEE, pp. 1324–1328.

Klabnik, Steve and Carol Nichols (2018). *The Rust Programming Language*. No Starch Press. isbn: 978-1593278281. url: https://doc.rust-lang.org/book/.

Kotsovsky, Vladyslav, Anatoliy Batyuk, and Maksym Yurchenko (2020). “New approaches in the learning of complex-valued neural networks”. In: *2020 IEEE Third International Conference on Data Stream Mining & Processing (DSMP)*. IEEE, pp. 50–54.

LeCun, Yann, Corinna Cortes, and Christopher J.C. Burges (1998). *THE MNIST DATABASE of handwritten digits*. http://yann.lecun.com/exdb/mnist/.

Lee, ChiYan, Hideyuki Hasegawa, and Shangce Gao (2022). “Complex-Valued Neural Networks: A Comprehensive Survey”. In: *IEEE/CAA Journal of Automatica Sinica* 9.8, pp. 1406–1426. doi: 10.1109/JAS.2022.105743.

Li, Songsong et al. (2006). “An Individual Adaptive Gain Parameter Backpropagation Algorithm for Complex-Valued Neural Networks”. In: *Advances in Neural Networks - ISNN 2006*. Ed. by Jun Wang et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 551–557. isbn: 978-3-540-34440-7.

Liu, Qin et al. (2017). “An Efficient Algorithm for Complex-Valued Neural Networks Through Training Input Weights”. In: *Neural Information Processing*. Ed. by Derong Liu et al. Cham: Springer International Publishing, pp. 150–159. isbn: 978-3-319-70093-9.

Liu, Siming et al. (June 2017). “A Multilevel Artificial Neural Network Nonlinear Equalizer for Millimeter-Wave Mobile Fronthaul Systems”. In: *Journal of Lightwave Technology* PP, pp. 1–1. doi: 10.1109/JLT.2017.2717778.

Liu, Weifeng, Jose C Principe, and Simon Haykin (2011). *Kernel adaptive filtering: a comprehensive introduction*. John Wiley & Sons.

Liu, Yuanshan, He Huang, and Tingwen Huang (2014). “Gain parameters based complex-valued backpropagation algorithm for learning and recognizing hand gestures”. In: *2014 International Joint Conference on Neural Networks (IJCNN)*, pp. 2162–2166. doi: 10.1109/IJCNN.2014.6889685.

Luo, Can et al. (2024). “A Complex-Valued Neural Network Based Robust Image Compression”. In: *Pattern Recognition and Computer Vision*. Ed. by Qingshan Liu et al. Singapore: Springer Nature Singapore, pp. 53–64. isbn: 978-981-99-8549-4.

Mandic, Danilo P and Vanessa Su Lee Goh (2009). *Complex valued nonlinear adaptive filters: noncircularity, widely linear and neural models*. John Wiley & Sons.

Marseet, Akram and Ferat Sahin (2017). “Application of complex-valued convolutional neural network for next generation wireless networks”. In: *2017 IEEE Western New York Image* *and Signal Processing Workshop (WNYISPW)*, pp. 1–5. doi: 10.1109/WNYIPW.2017.8356260.

Matlacz, Marcin and Grzegorz Sarwas (2018). “Crowd counting using complex convolutional neural network”. In: *2018 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, pp. 88–92. doi: 10.23919/SPA.2018.8563383.

Mönning, Nils (2019). “Deep Complex-Valued Neural Networks for Natural Language Processing”. PhD thesis. University of York.

Murata, Tetsuya, Tianben Ding, and Akira Hirose (2015). “Proposal of Channel Prediction by Complex-Valued Neural Networks that Deals with Polarization as a Transverse Wave Entity”. In: *Neural Information Processing*. Ed. by Sabri Arik et al. Cham: Springer International Publishing, pp. 541–549. isbn: 978-3-319-26555-1.

Nafisah, Zumrotun, Febrian Rachmadi, and Elly Matul Imah (2018). “Face recognition using complex valued propagation”. In: *Jurnal Ilmu Komputer dan Informasi* 11.2, pp. 103–109.

Neacșu, Ana et al. (2022). “Design of Robust Complex-Valued Feed-Forward Neural Networks”. In: *2022 30th European Signal Processing Conference (EUSIPCO)*, pp. 1596–1600. doi: 10.23919/EUSIPCO55093.2022.9909696.

Al-Nuaimi, A.Y.H., Md. Faijul Amin, and Kazuyuki Murase (2012). “Enhancing MP3 encoding by utilizing a predictive Complex-Valued Neural Network”. In: *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–6. doi: 10.1109/IJCNN.2012.6252535.

Olanrewaju, R. F. et al. (2011). “Detection of alterations in watermarked medical images using Fast Fourier Transform and Complex-Valued Neural Network”. In: *2011 4th International Conference on Mechatronics (ICOM)*, pp. 1–6. doi: 10.1109/ICOM.2011.5937131.

Peker, Musa, Baha Sen, and Dursun Delen (2016). “A Novel Method for Automated Diagnosis of Epilepsy Using Complex-Valued Classifiers”. In: *IEEE Journal of Biomedical and Health Informatics* 20.1, pp. 108–118. doi: 10.1109/JBHI.2014.2387795.

Pillai, Jyoti and Michael R. Sperling (2006). “Interictal EEG and the Diagnosis of Epilepsy”. In: *Epilepsia* 47.s1, pp. 14–22. doi: https://doi.org/10.1111/j.1528-1167.2006.00654.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1528-1167.2006.00654.x. url: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1528-1167.2006.00654.x.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). *Learning Internal Representations by Error Propagation*. Ed. by David E. Rumelhart, James L. McClelland, and the PDP Research Group. Cambridge, MA, USA: MIT Press, pp. 318–362.

Saraswathi, D and E Srinivasan (2014). “An ensemble approach to diagnose breast cancer using fully complex-valued relaxation neural network classifier”. In: *International Journal of Biomedical Engineering and Technology* 15.3, pp. 243–260.

Scardapane, Simone et al. (2018). “Complex-valued neural networks with nonparametric activation functions”. In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 4.2, pp. 140–150.

– (2020). “Complex-Valued Neural Networks With Nonparametric Activation Functions”. In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 4.2, pp. 140–150. doi: 10.1109/TETCI.2018.2872600.

Sobel, Irwin (Feb. 2014). “An Isotropic 3x3 Image Gradient Operator”. In: *Presentation at Stanford A.I. Project 1968*.

Sobel, Irwin and Gary Feldman (1968). *A 3x3 Isotropic Gradient Operator for Image Processing*. Technical Report S127. Presented at a talk at the Stanford Artificial Intelligence Project. Stanford Artificial Intelligence Project (SAIL).

Stack Overflow (2023). *Stack Overflow Developer Survey 2023*. Accessed: 2024-06-24. url: `https://survey.stackoverflow.co/2023/`.

Stroustrup, Bjarne (2013). *The C++ Programming Language*. 4th. Addison-Wesley. isbn: 978-0321563842.

Sunaga, Yuki, Ryo Natsuaki, and Akira Hirose (2019). "Land Form Classification and Similar Land-Shape Discovery by Using Complex-Valued Convolutional Neural Networks". In: *IEEE Transactions on Geoscience and Remote Sensing* 57.10, pp. 7907–7917. doi: `10.1109/TGRS.2019.2917214`.

Tiba, Isayiyas Nigatu and Mao youhong (2023). "Deep Complex-Valued Neural Networks for Massive MIMO Signal Detection". In: *Artificial Intelligence and Digitalization for Sustainable Development*. Ed. by Bereket H. Woldegiorgis et al. Cham: Springer Nature Switzerland, pp. 239–251. isbn: 978-3-031-28725-1.

Tsuzuki, Hirofumi et al. (2013). "An approach for sound source localization by complex-valued neural network". In: *IEICE TRANSACTIONS on Information and Systems* 96.10, pp. 2257–2265.

Uncini, A. et al. (1999). "Complex-valued neural networks with adaptive spline activation function for digital-radio-links nonlinear equalization". In: *IEEE Transactions on Signal Processing* 47.2, pp. 505–514. doi: `10.1109/78.740133`.

Van der Walt, Stéfan et al. (2014). "Scikit-image: image processing in Python". In: *PeerJ* 2, e453. doi: `10.7717/peerj.453`. url: `https://doi.org/10.7717/peerj.453`.

Virtue, Patrick, Stella X. Yu, and Michael Lustig (2017). "Better than real: Complex-valued neural nets for MRI fingerprinting". In: *2017 IEEE International Conference on Image Processing (ICIP)*, pp. 3953–3957. doi: `10.1109/ICIP.2017.8297024`.

Wang, Haifeng, Bin Yang, and Jiaguo Lv (2017). "Complex-Valued Neural Network Model and Its Application to Stock Prediction". In: *Proceedings of the 16th International Conference on Hybrid Intelligent Systems (HIS 2016)*. Ed. by Ajith Abraham et al. Cham: Springer International Publishing, pp. 21–28. isbn: 978-3-319-52941-7.

Wang, Yu et al. (2021). "An Efficient Specific Emitter Identification Method Based on Complex-Valued Neural Networks and Network Compression". In: *IEEE Journal on Selected Areas in Communications* 39.8, pp. 2305–2317. doi: `10.1109/JSAC.2021.3087243`.

Widrow, Bernard, John McCool, and Michael Ball (1975). "The complex LMS algorithm". In: *Proceedings of the IEEE* 63.4, pp. 719–720.

Wirtinger, Wilhelm (1927). "Zur formalen theorie der funktionen von mehr komplexen veränderlichen". In: *Mathematische Annalen* 97.1, pp. 357–375.

Wu, Rongrong, He Huang, and Tingwen Huang (2017). "Learning of Phase-Amplitude-Type Complex-Valued Neural Networks with Application to Signal Coherence". In: *Neural Information Processing*. Ed. by Derong Liu et al. Cham: Springer International Publishing, pp. 91–99. isbn: 978-3-319-70087-8.

Yang, Xin-She and Suash Deb (2014). "Cuckoo search: recent advances and applications". In: *Neural Computing and applications* 24, pp. 169–174.

You, Cheolwoo and Daesik Hong (1998). "Nonlinear blind equalization schemes using complex-valued multilayer feedforward neural networks". In: *IEEE Transactions on Neural Networks* 9.6, pp. 1442–1455. doi: `10.1109/72.728394`.

Yuan, Quan et al. (2019). "Channel Estimation and Pilot Design for Uplink Sparse Code Multiple Access System based on Complex-Valued Sparse Autoencoder". In: *IEEE Access*, pp. 1–1. doi: `10.1109/ACCESS.2019.2904990`.

Zhang, Hui et al. (2021). "An optical neural chip for implementing complex-valued neural network". In: *Nature communications* 12.1, p. 457.

Zhang, Huisheng and Danilo P Mandic (2015). “Is a complex-valued stepsize advantageous in complex-valued gradient learning algorithms?” In: *IEEE transactions on neural networks and learning systems* 27.12, pp. 2730–2735.

Zhang, Junming and Yan Wu (2017). “A New Method for Automatic Sleep Stage Classification”. In: *IEEE Transactions on Biomedical Circuits and Systems* 11.5, pp. 1097–1110. doi: `10.1109/TBCAS.2017.2719631`.

Zhao, Weijing and He Huang (2023). “Adaptive orthogonal gradient descent algorithm for fully complex-valued neural networks”. In: *Neurocomputing* 546, p. 126358. issn: 0925-2312. doi: `https://doi.org/10.1016/j.neucom.2023.126358`. url: `https://www.sciencedirect.com/science/article/pii/S0925231223004812`.

– (2024). “Adaptive stepsize estimation based accelerated gradient descent algorithm for fully complex-valued neural networks”. In: *Expert Systems with Applications* 236, p. 121166. issn: 0957-4174. doi: `https://doi.org/10.1016/j.eswa.2023.121166`. url: `https://www.sciencedirect.com/science/article/pii/S0957417423016688`.