Chapter 1

Extent

From Wirtinger calculus backpropagation to specific software implementation challenges, in this chapter are described the fundamental details of complex-valued neural network components and how they are related to existing real-valued network implementations. We will show how existing layers and functionalities can be extended to work also with a complex-valued input and which of them needs to be completely redefined.
We address the problem of re-adapting the training process building a complex backpropagation algorithm on top of many prior works, that allows for an optimization when the loss function is real-valued, thanks to Wirtinger calculus.
Furthermore, we will discuss in details the problem of building complex-valued activation functions, that was one of the main obstacles in the development of deep learning in this direction.
In the end, we will provide a brief presentation of the high level library, built on top of `JAX`, that we have realized in order simplify the setup and train of those kind of networks. Nowadays, in fact, the internet is full of deep learning libraries implementing basically every kind of known model, with different optimizations, parallelizations, etc. However, for some reason, many of them still does not provide support to complex data types: a huge obstacle in the growth of complex-valued deep learning.

1.1 Problems in the extent

Considering just their fundamental structure, complex-valued neural networks work exactly like their real counterpart, and are again constituted by neurons connected among each other (figure 1.1): the only difference is that now those neurons are complex-valued.

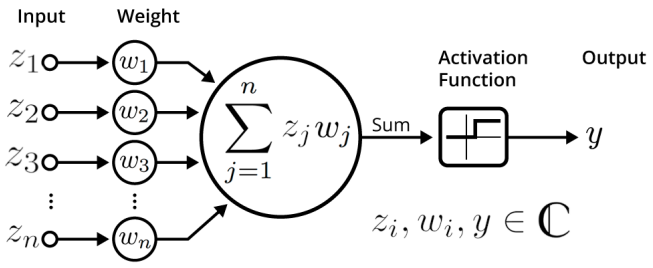

Figure 1.1: Fundamental unit (neuron) of a complex-valued neural network.

Each neuron receives a weighted input signal $\mathbf{z}$, that this time is complex valued (as the weights $\mathbf{w_i}$); this signal in summed up and added to a bias $\mathbf{b}$, and then passed through an activation function $f : \mathbb{C} \to \mathbb{C}$, that most of the times is non-linear. If we denote with the subscript ℓ the forward pass of

a neuron in the ℓ-th layer, then the output can be expressed with the following formula:

$$\mathbf{y}_\ell = f_\ell \left(\sum_{i=1}^{N} \mathbf{w_i} z_i + \mathbf{b}_\ell \right)$$

where N are the neurons in layer ℓ, M the neurons in layer $(\ell - 1)$, $\mathbf{z_{\ell-1}} \in \mathbb{C}^M$ was the output of the previous layer, $\mathbf{w}_\ell \in \mathbb{C}^{N \times M}$ and $\mathbf{b}_\ell \in \mathbb{C}^N$ are the learnable parameters of this level, f_ℓ the activation function and $\mathbf{y}_\ell \in \mathbb{C}^N$ the effective output.
However, when considering a possible extension from $\mathbb{R}$ to $\mathbb{C}$, we need to take into account a few inconveniences, since we look for a coherent and rigorous framework.

Max operator is undefined

As explained also in the introductory mathematical section, $\mathbb{C}$ is not an ordered field, in the sense that we cannot define a comparison relation among complex numbers that makes everybody agree. In principle you can define one, like the lexycographical ordering, that compares first the real part and only after the imaginary, or relying on establishing this relation among the magnitudes of those numbers. The latter is actually the preferred approach. This brief overview is important, since many non-linear functions in deep learning, like `ReLU` and `Max-Pooling` necessitate of a *maximum* operation in order to fulfill their purpose of increasing numerical stability and dimensionality reduction, respectively.

Unstable Activations

As we will see in a dedicated section, the problem of defining stable and coherent activation functions is one of the main issues that limited the development of complex-valued deep learning during the years. Complex functions, in fact, necessitate of further limitations to be suitable as activations: because of the Liouville's theorem **??**, for example, they can't be limited, and neither grow too slow, otherwise their derivative would always vanish during the backpropagation. So, simply re-adapting existing activations to support complex-valued inputs, maybe redefining ambiguous operations like `max`, is not enough, especially because you need to care about the eventual loss of complex correlations if the activation is applied independently on the real and imaginary components.

Lost Probabilistic Interpretation

One nice property of real-valued neural network classifiers is the probabilistic interpretation that we can associate to its final layer, mainly due to the normalization in the range $[0, 1]$ provided by sigmoid/softmax activation functions. But now, the final output of the network will be a set of complex numbers, that we cannot interpret anymore as a probability distribution over a set of probabilistic outcomes. This nice property can be partially recovered if we add a *magnitude* layer just before the last activation: in this way we drop all the phase information but we move back to a real-valued problem. Anyway, it always depends on the final objective of the model.

Optimization of a Complex-Valued Output

Another problem related to having a complex-valued output, beside its interpretation, is the loss associated. If you have a complex final loss, how can you effectively minimize it? In the first chapter we have defined the minimum of a function defined in $\mathbb{C}$ as the point $z_0 \in \mathbb{C}$ in which its modulus is minimized. But this definition, provided by the author of that complex analysis book, is referred to a total ordering in which we refer first to the magnitudes, and so also this is just a convention. Notice that, at the end, minimizing the modulus of a complex loss is equivalent to defining a real-valued loss, i.e. $\mathscr{L} : \mathbb{C} \to \mathbb{R}$, and minimizing it. And that's exactly what we are going to do when setup a backpropagation.

After all this steps, it is clear that we cannot simply reuse deep learning architectures designed for real values without first understanding the complex mathematics in neural networks forward and backward.

1.2 Complex Backpropagation

As anticipated in the introductory section, the interest of researchers in this complex-valued deep learning area started arising many years ago; in fact, the first trial to setup a complex backpropagation algorithm even dates back to 1991 [**?**]. According to the author, not only its algorithm turns out to have an higher convergence speed (and the same generalization performances) with respect to its real counterpart, but also that is able to learn an entire class of transformations (e.g. rotations, similarities, parallel displacements, etc.) that the real method cannot. However, having read the work of Nitta [**?**], I feel it is better to remark a couple of things, mainly because many years have passed from its publication. First of all, the author used a suboptimal setup:

- the network followed one of the "conventional" approaches that we are proving to be unefficient, i.e. treating real and imaginary parts of the data as independent random variables;
- he computed the derivatives $\partial f/\partial x$ and $\partial f/\partial y$, instead of relying on Wirtinger calculus **??**; even if this is a working alternative to ours, we will see that it is suboptimal;
- he relied on "bad" activation functions, since, as told by he himself, many times the algorithm failed to converge.

I decided to report his work because it was still one of the first and working attempts to develop a complex backpropagation algorithm, but also because of the purely theoretical analysis realized on the transformations that a complex network can learn. Nitta, managed to teach its networks several transformations in $\mathbb{R}^2$, like rotations, reductions and parallel displacements, that the corresponding real-valued model didn't make. He understood first that this was possible thanks to the higher degrees of freedom offered by complex multiplication (discussed in section **??**). But what I believe it is even more interesting, is the relation that Nitta have found among complex-valued networks and the **Identity theorem ??**:
"We believe that Complex-BP networks satisfy the Identity Theorem, that is, Complex-BP networks can approximate complex functions just by training them only over a part of the domain of the complex functions."
This means that exploiting holomorphic functions when building a complex-valued network can sometimes impact on its generalization capabilities (since its shape will be rigidly determined by its characteristics on a small local region of its domain) [**?**]. Unfortunately, no additional work have been realized on this statement during the years, but I think it is an aspect deserving further attention.
In section **??** we have discussed about complex differentiability, and we also said that holomorphicity is not a property assured for most functions, and even simple ones, like the square modulus, can be not differentiable in the complex sense. In our architectures we have mainly two sources of *nonholomorphicity*: the loss and the activations. For reasons that will be clearer later on, boundedness and analiticity cannot be achieved simultaneously in the complex domain, and the first feature is often preferred [**?**].
An elegant approach that can save computational labor is the usage of Wirtinger calculus to setup optimization problems, solvable via gradient descent, for functions that are not holomorphic but at least differentiable with respect to their real and imaginary components.
Also for neural network functions we have a similar problem: requiring them to have complex differentiable properties can be quite limiting, and we should again rely on $\mathbb{CR}$-calculus. To complicate matters, the chain rule requires now to compute two terms rather than one (both the $\mathbb{R}$-derivative and the conjugate $\mathbb{R}$-derivative), causing more than $4x$ calculations during the backward pass.
"Fortunately", we can significantly reduce both memory and computation time during complex backpropagation by assuming that our final network loss function is *real-valued*. This is a strong but valid assumption since, as already discussed in section **??**, minimizing a complex-valued function is an **ambiguous** operation. Also, minimizing the modulus of a complex loss, as suggested by the magnitude ordering introduced, in the end is exactly like optimizing a real-valued function.

But why does it turn out to be more efficient?

1.2.1 Steepest Complex Gradient Descent

In the previous sections we have widely discussed about the fact that complex-valued functions cannot be minimized, and so we ended up with the conclusion that we must use a real-valued loss in order to formulate the training process of a complex model as an optimization problem, in continuation to what happens inside real models.
Let's then consider a function $f(z) : \mathbb{C} \to \mathbb{R}$: if we decide to proceed with a gradient descent algorithm, which direction are we supposed to take since there are two different derivatives that one can compute ($\partial/\partial z$ and $\partial/\partial \bar{z}$)?
It can be proved ([**?**, **?**] and appendix **??**) that the direction of the *steepest gradient descent* is the **complex cogradient**, $\nabla_{\bar{\mathbf{z}}} f$. So, given a function f that depends on a complex random variable $z \in \mathbb{C}$, the update rule that minimizes it is:

$$\mathbf{z} \leftarrow \mathbf{z} - \alpha \nabla_{\bar{\mathbf{z}}} f \tag{1.1}$$

where $\alpha \in \mathbb{R}$ is the learning rate.

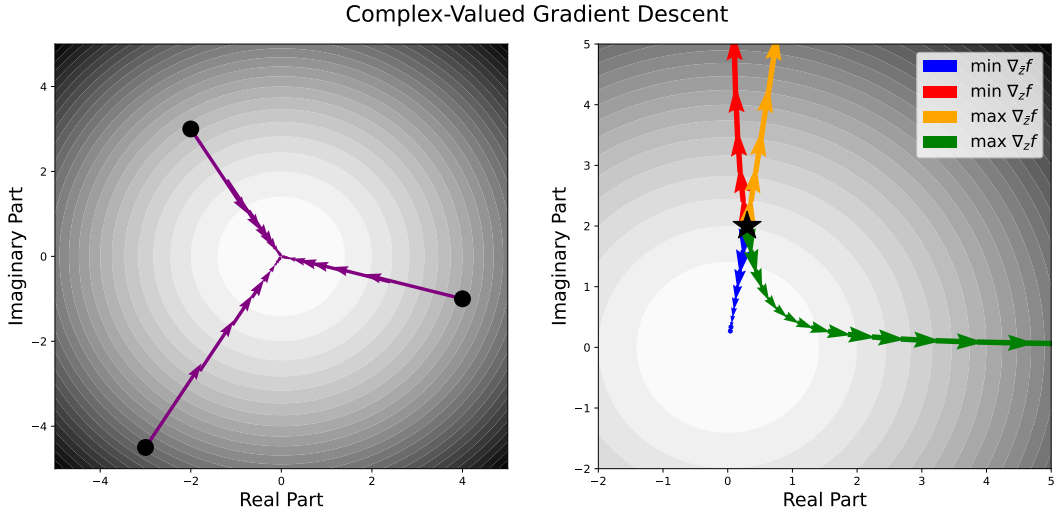

Figure 1.2: Complex Gradient Descent.

In order to provide also a visual representation, in figure 1.2 we have considered a simple, non holomorphic, real-valued function like $f(z) = z\bar{z} = \|z\|^2$, that has a unique global minimum at $z = 0 + 0j$. We have then applied the gradient descent and ascent rules in both directions of the gradient, $\nabla_{\mathbf{z}} f$, and the cogradient $\nabla_{\bar{\mathbf{z}}} f$, in order to verify what said above. In the plot we clearly see that the only direction that approaches the true minimum (starting from a random point in the dominium of f) is exactly the one determined by the complex cogradient, while the complex gradient moves in a completely wrong direction. Also considering the ascent rules we observe that the steepest direction maximizing f is again the one determined by the cogradient.

1.2.2 Backpropagation with a Real-valued Loss

With the real-valued loss assumption (proposed in 1.2) and the update rule 1.1, complex backpropagation turns out to be quite efficient and not so computationally expensive as we thought in section 1.2. The situation can be summarized with table 1.1 and figure 1.3.

Now the algorithm needs to pass only one of the two $\mathbb{CR}$ derivatives back to the earlier layers, even tho, because of the chain rule **??**, it must compute both of them, at least for the final layer. In figure 1.3 it is effectively illustrated how input and derivatives flow through the layers during the forward and backward pass, respectively.

Standard Real Calculus	**Complex Calculus**	**Complex Calculus, assuming real-valued loss**
Input to layer $\ell+1$:		
$\frac{\partial f_L}{\partial x_\ell}$	$\frac{\partial f_L}{\partial z_\ell}$ and $\frac{\partial f_L}{\partial \bar{z}_\ell}$	$\frac{\partial f_L}{\partial \bar{z}_\ell}$
Output from layer ℓ:		
$\frac{\partial f_L}{\partial x_{\ell-1}} = \frac{\partial f_L}{\partial x_\ell}\frac{\partial f_\ell}{\partial x_{\ell-1}}$	$\frac{\partial f_L}{\partial z_{\ell-1}} = \frac{\partial f_L}{\partial z_\ell}\frac{\partial f_\ell}{\partial z_{\ell-1}} + \frac{\partial f_L}{\partial \bar{z}_\ell}\overline{\left(\frac{\partial f_\ell}{\partial \bar{z}_{\ell-1}}\right)}$	
	$\frac{\partial f_L}{\partial \bar{z}_{\ell-1}} = \frac{\partial f_L}{\partial z_\ell}\frac{\partial f_\ell}{\partial \bar{z}_{\ell-1}} + \frac{\partial f_L}{\partial \bar{z}_\ell}\overline{\left(\frac{\partial f_\ell}{\partial z_{\ell-1}}\right)}$	$\frac{\partial f_L}{\partial \bar{z}_{\ell-1}} = \overline{\left(\frac{\partial f_L}{\partial \bar{z}_\ell}\right)}\frac{\partial f_\ell}{\partial \bar{z}_{\ell-1}} + \frac{\partial f_L}{\partial \bar{z}_\ell}\overline{\frac{\partial f_\ell}{\partial z_{\ell-1}}}$

Table 1.1: Comparison of backpropagation calculus. (source: [**?**])

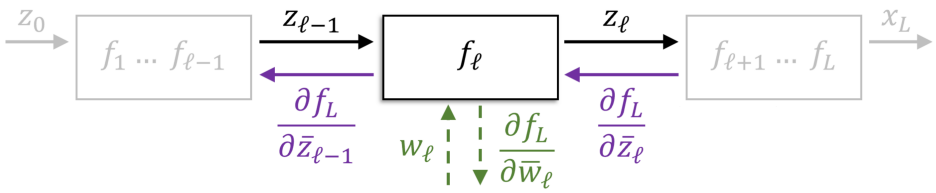

Figure 1.3: Forward and backward pass through a complex layer. (source [**?**])

Given the steepest complex gradient descent rule 1.1, we can then write down the equivalent expression for our network function depending on a set of parameters $\mathbf{w}$:

$$\mathbf{w}_n \leftarrow \mathbf{w}_n - \alpha \frac{\partial f}{\partial \bar{\mathbf{w}}_n}$$

1.3 Re-definition of the main neural network layers

We have started writing down this chapter with the purpose of building a working and coherent framework for complex-valued machine learning. And we also In section 1.1 we have analyzed the main obstacles that we encounter during this extension, while in 1.2 we

Fully-Connected Layers

For the fundamental layers of a neural network the extension is quite trivial. Consider a layer with K complex-valued units, a weight vector $\mathbf{w} \in \mathbb{C}^K$ and a bias term $\beta \in \mathbb{C}$; then, the response to a certain input vector $\mathbf{z} \in \mathbb{C}^N$ (given also the activation function f) is:

$$\mathbf{y} = f(\mathbf{z}, \{\mathbf{w}, \beta\}) = f\left(\sum_k z_k w_k + \beta\right) = f\left(\mathbf{z}^T\mathbf{w} + \beta\right)$$

So it is just like the real case, with the only difference that now the multiplication is in $\mathbb{C}$ and not in $\mathbb{R}$, with all the consequences already discussed in **??**.

What we should care about is, instead, the initial values of the networks' parameters. Proper initialization can, in principle, help reducing the risk of vanishing or exploding gradients. Conventionally, researchers follow the approaches proposed by Glorot [**?**] or by He [**?**] (the latter designed specifically for `ReLU` activations), with the final objective of ensuring that the variance of input, output and their gradients are the same. According to these two methods, weights should be initialized with a normal distribution (or truncated-normal) with zero mean and standard deviation that depends on the number of units in that specific layer. Thanks to Trabelsi [**?**] (derivation in appendix **??**) we could provide two equivalent procedures, but in the complex domain.
To put in place them, we need first to consider the weights in polar form, i.e. $\|\mathbf{w}\|e^{i\theta}$, and then:

- random sampling the magnitude according to a **Rayleigh distribution** with parameter $\sigma = 1/\sqrt{n_{in} + n_{out}}$ (`Complex Xavier` initialization) or $\sigma = 1/\sqrt{n_{in}}$ (`Complex He` initialization);
- random sampling the phases according to a **uniform distribution** in $[-\pi, \pi`]$.

The biases β, instead, can all be initialized simply to 0, or uniformly in a small interval $[-\varepsilon, \varepsilon]$.

Convolutional Layers

In order to perform the equivalent of a traditional real-valued 2D convolution in the complex domain, we convolve a complex filter matrix $\mathbf{W} = \mathbf{A} + i\mathbf{B}$ by a complex vector $\mathbf{h} = \mathbf{x} + i\mathbf{y}$, where $\mathbf{A}, \mathbf{B}$ are real matrices and $\mathbf{x}, \mathbf{y}$ are real vectors, since we are simulating complex arithmetic using real-valued entries. As the convolution operator is distributive, we have:

$$\mathbf{W} * \mathbf{h} = (\mathbf{A} * \mathbf{x} - \mathbf{B} * \mathbf{y}) + i\,(\mathbf{B} * \mathbf{x} + \mathbf{A} * \mathbf{y}) \tag{1.2}$$

This is a very nice behavior, since the a complex convolution can be decomposed into two real-valued independent operations. And this means that we can exploit already existing algorithms (like this one, from Gauss [1]) to perform efficiently this (already expensive) computation.
Notice also that complex convolutional layers' weights basically, can learn to rotate the phase of desirable data toward the positive real axis; in fact, if we rewrite 1.2 in a matrix form:

$$\begin{bmatrix} \Re(\mathbf{W} * \mathbf{h}) \\ \Im(\mathbf{W} * \mathbf{h}) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & -\mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix} * \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}$$

In figure 1.4 we can observe a visual representation of the complex convolution proposed by [**?**].

In principle, however, even tho complex convolution should have the same computational cost as its real-valued counterpart, in practice it is four times more expensive, just because complex multiplication in the worst case requires four products and two additions.

Pooling Layers

The pooling operation involves sliding a n-dimensional filter over each channel of the feature map and summarizing the features lying within the region covered by the filter. Pooling layers are essentially dimensionality reduction levels inside the network, with the purpose of making the model more robust to positional variations in the input.
There are mainly two kinds of pooling operations that we want to extend to the complex domain:

- `Average Pooling`: replaces the elements in the filtered region of the feature map with their mean. The average of a set of complex number is a well defined operation and so no further work is needed in this case.
- `Max Pooling`: replaces the elements in the filtered region of the feature map with their maximum. In this case, instead, we have that the maximum operation is ambiguous in $\mathbb{C}$, and so we must establish an ordering to re-define this layer. Our choice that was to setup the max pooling operation to return the complex number with the highest magnitude, inside the region covered by the filter. Namely:

$$\text{MaxPool}(\{z_k\}) = z_n \qquad \text{where } n = \underset{k}{argmax}\ \|z_k\|$$

[1] As explained in this brief Wikipedia section, we can reduce the cost of complex multiplication, in some cases.

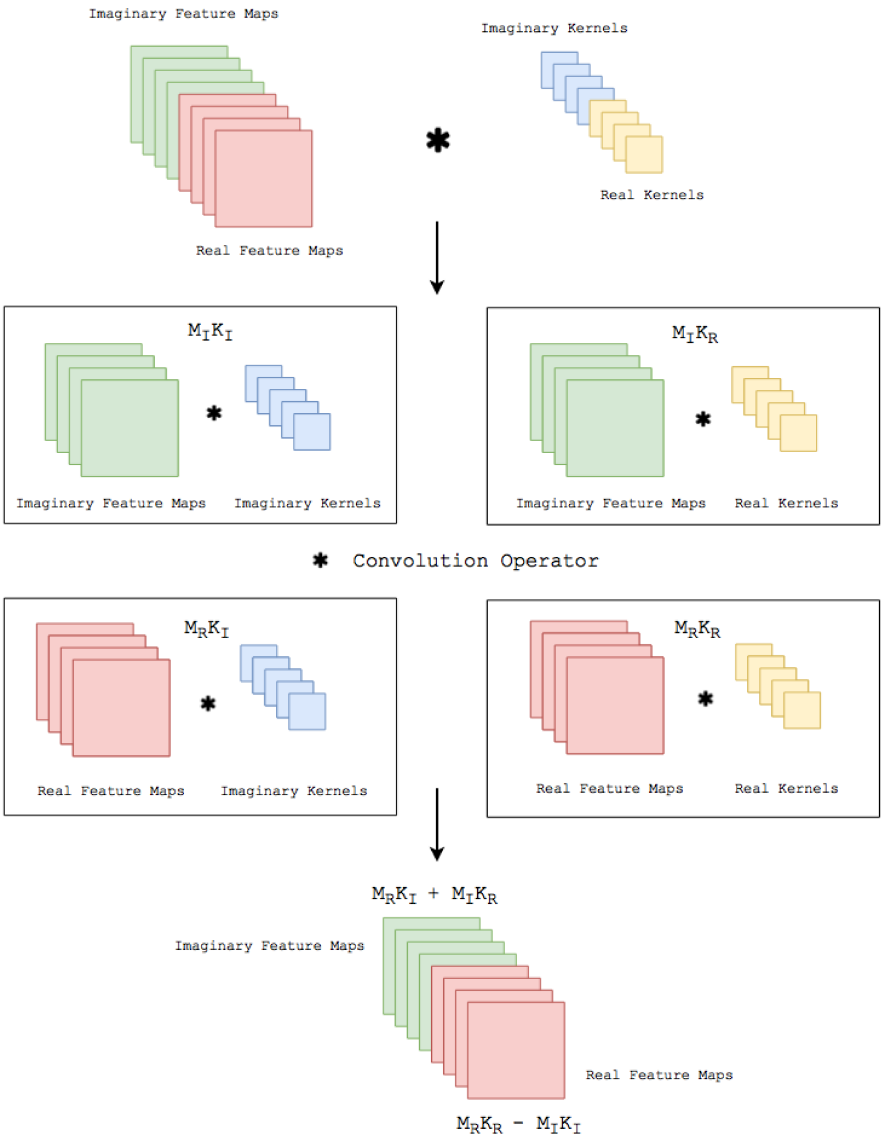

Figure 1.4: Implementation details of the Complex Convolution (by [**?**]).

This choice is not unique, since, as repeated several times, there are more than one total ordering for $\mathbb{C}$.

Normalization Layers

Normalization layers are usually inserted inside the architecture of a neural network with the purpose of improving the stability of the network optimization, especially in cases of an high depth, and for a better control of the gradients. In this case the extension not so straightforward, since you need to re-define several statistical objects.
In standard `Batch-Normalization` we train two internal parameters, a scale and an offset, such that each batch of data has zero mean and standard deviation one. This approach, however, is valid only for real data, since it does not guarantee equal variance for both the real and imaginary components, with a resulting distribution turning out to be non-circular. Following the method proposed by [**?**], the idea is to normalize data to obtain a standard complex normal distribution (basically the one in figure **??**), achieved by multiplying the zero-centered data $(\mathbf{z} - \mathbb{E}[\mathbf{z}])$ by the inverse square root of

their $2x2$ covariance matrix $\mathbf{V}$ [2]:

$$\tilde{\mathbf{z}} = (\mathbf{V})^{-\frac{1}{2}} (\mathbf{z} - \mathbb{E}[\mathbf{z}]) \qquad \text{where} \quad \mathbf{V} = \begin{pmatrix} \text{Cov}(\text{Re}\{\mathbf{z}\}, \text{Re}\{\mathbf{z}\}) & \text{Cov}(\text{Re}\{\mathbf{z}\}, \text{Im}\{\mathbf{z}\}) \\ \text{Cov}(\text{Im}\{\mathbf{z}\}, \text{Re}\{\mathbf{z}\}) & \text{Cov}(\text{Im}\{\mathbf{z}\}, \text{Im}\{\mathbf{z}\}) \end{pmatrix} \tag{1.3}$$

From a practical point of view, the implementation is the same of the real case. You have again two trainable parameters: $\beta \in \mathbb{C}$, i.e. the complex mean, and $\gamma \in \mathbb{C}^2$, the complex-valued positive-defined covariance matrix. The `complex batchnorm` operation is then defined as

$$BN(\mathbf{z}) = \gamma \mathbf{z} + \beta$$

We need, however, to be careful when relying on batchnormalization. This procedure allows, in fact, to avoid co-adaptation between real and imaginary parts of data, effectively reducing the risk of overfitting. But there is a cost to this, since you are basically decorrelating your complex data, partially losing the advantage over two-channels networks [**?**].
For this reason, we sometimes prefer to rely on other kind of normalization layers that, instead, allows to preserve complex data correlations.
In simple `Complex Normalization` we scales a complex scalar input $\mathbf{z}$ such that its magnitude is set to one, while the phase remains unchanged. In practice we project $\mathbf{z}$ onto the unit circle. The forward pass is then

$$\hat{\mathbf{z}} = e^{i\angle \mathbf{z}} = \frac{\mathbf{z}}{\|\mathbf{z}\|} = \frac{\mathbf{z}}{(\mathbf{z}\bar{\mathbf{z}})^{1/2}}$$

Other Layers

There are many layers that do not need any further re-definition to work also in the complex domain: `Dropout`, Pad or Attention layer, for example. There are also many other structures that should be re-derived (e.g Recurrent layers, LSTM, etc.), but that were out of our scope and so we haven't examined. This should be interpreted just as a starting point in the development of an higher level complex-valued deep learning framework.

1.4 Complex-Valued Activation Functions

One of the main issues encountered in the last 30 years in the developing of a complex-valued deep learning framework was exactly the definition of reliable activation functions. The extension from the real-valued domain turned out to be quite challenging: because of the Liouville's theorem **??**, in fact, stating that the only complex-valued functions that are bounded and analytic everywhere are constants, one have necessarily to choose between boundedness and analyticity, in the design of those activations. Furthermore, before the introduction of ReLU, almost all the activation functions known in the real case were bounded. And for the same ReLU the extent was not trivial, since operations like *max* are not defined in the complex domain, Additionally, with complex-valued outputs, we have lost the probabilistic interpretations that functions like `sigmoid` and `softmax` used to provide.
We have to say, however, that most of the candidate functions that have been proposed, have been developed in a split fashion, i.e. by considering the real and imaginary parts of the activation separately. But, as discussed also in the previous chapter, this approach should be abandoned, since you risk losing the complex correlations stored in those variables.
In this section, we will explore a few complex-valued activations proposed during the years: first with the ones that are direct extensions of their real counterparts, and then with more "abstract" candidates, that have more reasons to live and work in the complex domain.

[2] The existence of the inverse matrix is guaranteed by the positive (semi-) definiteness of $\mathbf{V}$. Eventually, you can enforce this condition by adding a small quantity $+\varepsilon\mathbb{I}$ to the matrix (Tikhonov regularization).

Extent from the real case

The first class of viable approaches consists into barely extending real-valued activations in the complex domain, like the `sigmoid` and the `hyperbolic tangent`:

$$\sigma_{\mathbb{C}}(z) = \frac{1}{1+e^{-z}} \qquad \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

These functions are fully-complex, and also analytic and bounded almost everywhere, at the cost of introducing a set of singular points: both function, in fact, seems to diverge periodically in the imaginary axis of the complex plane. Limiting and scaling carefully the input values seems to help avoiding then those singularities and so partially containing the instability. However, I believe that there are simpler and more efficient alternatives.

Separable Activations

As already explained, the main tendency in the development of complex-valued activation functions was basically getting back the "old" designs for real-valued models and using them independently on the real and imaginary components of the input.
This can be done easily with both the sigmoid and the hyperbolic tangent, mapping real and imaginary parts among input and output as they were independent channels:

$$f(\mathbf{z}) = g\left(\mathrm{Re}(\mathbf{z})\right) + ig\left(\mathrm{Im}(\mathbf{z})\right), \qquad \text{where} \quad g(x) = \frac{1}{1+e^{-x}} \quad \text{or} \quad g(x) = \tanh(x)$$

Notice that this approach maps the phase of the signal into $[0, 2\pi]$, since the function g returns always a positive value.
There are also interesting variations to the separable sigmoid, properly designed to work using a complex-valued network on real-valued data. But, for this reason, they are functions with values in $\mathbb{R}$ and not in $\mathbb{C}$, and so we won't go through them in this work.
After the advent of the ReLU activation functions, two designs where developed in this fashion, the $\mathbb{C}$`ReLU` and the z`ReLU`:

$$\mathbb{C}ReLU(z) = ReLU(\mathrm{Re}(z)) + iReLU(\mathrm{Im}(z)) \qquad zReLU = \begin{cases} z & \text{if } \theta_z \in [0, \pi/2] \\ 0 & \text{otherwise} \end{cases}$$

These functions also share the nice property of being holomorphic in some regions of the complex plane: $\mathbb{C}$`ReLU` in the first and third quadrants, while z`ReLU` everywhere but the set of points $\{\mathrm{Re}(z) > 0,\, \mathrm{Im}(z) = 0\} \cup \{\mathrm{Re}(z) = 0,\, \mathrm{Im}(z) > 0\}$.

Phase-preserving Activations

Phase-preserving complex-valued activations are those functions that usually act only on the magnitude of input data, preserving the pre-activated phase during the forward pass. Those are usually non-holomorphic, but at least bounded in the magnitude. They are all based on the intuition that, altering the phase could severely impact the complex representation.
The first proposal is the so called `siglog` activation function: It is called in this way because it is equivalent to applying the sigmoid to the log of the input magnitude and restoring the phase:

$$siglog(z) = g\left(\log \|z\|\right) e^{-i\angle z} = \frac{z}{1+\|z\|}, \qquad \text{where} \quad g(x) = \frac{1}{1+e^{-x}}$$

Unlike the sigmoid and its separable version, the siglog projects the magnitude of the input from the interval $[0, \infty)$ to $[0, 1)$. The authors of this proposal suggested also the addition of a couple of parameters to adjust the *scale*, r, and the *steepness*, c, of the function:

$$siglog(z; r, c) = \frac{z}{c + \frac{1}{r}\|z\|}$$

The main problem with *siglog* is that the function has a nonzero gradient in the neighborhood of the origin of the complex plane, which can lead to gradient descent optimization algorithms to continuously stepping past the origin rather then approaching the point and staying here.
For this reason, an alternative version have been proposed, this time with a better gradient behavior (approaching zero as the input approaches zero), that goes under the name of `iGaussian`.

$$iGauss(z;\sigma^2) = g(z;\sigma^2)n(z) \qquad \text{where} \quad g(z;\sigma^2) = 1 - e^{-\frac{z\bar{z}}{2\sigma^2}}, \quad n(z) = \frac{z}{\|z\|}$$

This activation is basically an inverted gaussian (and this is the reason for which it is more smooth around the origin) and so depends only on one parameter, i.e. its standard deviation σ.
The last activation that we want to consider for this class is another variation of the rectified linear unit, this time called `modReLU`:

$$modReLU(z) = ReLU(\|z\| + b)e^{i\theta_z} = \begin{cases} (\|z\| + b)\,\frac{z}{\|z\|} & \text{if } \|z\| + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $z \in \mathbb{C}$, θ_z is the phase of z, and $b \in \mathbb{R}$ is a learnable parameter. The idea of this activation is to create a "dead zone" of radius b around the origin, where the neuron will be inactive, while it will be active outside. We decided to consider this function, even if apparently was designed for Recurrent neural networks.

Complex Cardioid

Even if the `iGaussian` has nice gradients properties around the origin in the complex plane, the same cannot be said when in input are given large values: in that situation there is high risk of vanishing gradients, the same that have historically hindered the performances of sigmoid-like activations. Because of this, recently a new complex activation function have been proposed: the `Complex Cardioid` [**?**]. The cardioid acts as an extension of the ReLU function to the complex domain, rescaling from one to zero all the values with non-zero imaginary component, based on how much the same input is rotated in phase with respect to the real axis. Also, the cardioid is sensitive to the input phase, rather than the modulus: the output magnitude is, in fact, attenuated based on the input phase, while the output phase remains equal to the original one.
The analytical expression for this activation function is:

$$cardioid(z) = \frac{1}{2}\left(1 + \cos(\angle z)\right) z$$

Another very nice property is that when the inputs are restricted to real values, this function becomes simply the ReLU activation.
We will see, in our applications, that the cardioid effectively allows complex networks to converge in a fast and also stable way.

1.5 JAX Implementation

From a practical perspective, in order to setup and realized all the studies and analysis we are going to present, I had to realize a dedicated `Python` library.
In the previous sections we have analyzed all the theoretical obstacles that researchers had to overcome in order to develop a working complex-valued deep learning framework. But, in reality, there are many drawbacks also at the implementation level: first of all the most popular hardware acceleration architectures, `CUDA` and `CuDNN` doesn't own a native support for complex-valued data types. And this is by itself a huge limitation, since we cannot train our networks efficiently, and we will have to rely on simple models with few parameters.
Regarding existing deep learning libraries, like `Keras`, `TensorFlow` and `Pytorch`, we have to say that they provide a lot of interesting high-level architectures to setup deep learning algorithms, and also they "officially" support complex inputs. Even better, they support complex derivatives.

Activation	Analytic Form	Reference
Sigmoid	$\sigma(z)$	//
Hyperbolic Tangent	$\tanh(z)$	//
Split-Sigmoid	$\sigma(\text{Re}(z)) + i\sigma(\text{Im}(z))$	[?]
Split-Tanh	$\tanh(\text{Re}(z)) + i\tanh(\text{Im}(z))$	[?]
$\mathbb{C}$ReLU	$ReLU(\text{Re}(z)) + iReLU(\text{Im}(z))$	[?]
zReLU	z if $z \in [0, \pi/2]$, 0 otherwise	[?]
Siglog	$\sigma(\log \|\|z\|\|)e^{-i\theta_z}$	[?]
iGauss	$\left(1 - e^{-\frac{z\bar{z}}{2\sigma^2}}\right)\frac{z}{\|\|z\|\|}$	[?]
modReLU	$ReLU(\|\|z\|\| + b)e^{i\theta_z}$	[?]
Cardioid	$\frac{1}{2}(1 + \cos(\angle z))\, z$	[?]

Table 1.2: Recap of the most popular complex-valued activation functions.

But when you try to implement an effective complex-valued network, you understand how much are they far from an effective comprehensive support (just some known issues: 1, 2, 3): regardless of the many errors you can get from structures that should work, from an accurate analysis of the source code of the main layers, you notice that many operations are ambiguous or at least not compatible with the reasoning we adopted to develop the layers extents in this chapter. We could have probably found a way to redefine or override those structures, but we felt that modifying a so large and complex library, without getting undesired side effects, would have been too much work. For this reason we had to rely on a more niche library, called `JAX`, and recently developed by `Google DeepMind`. JAX is **Autograd** and **XLA** (a domain-specific compiler for linear algebra designed for TensorFlow models), brought together for high-performance numerical computing and machine learning research. It provides composable transformations of Python and NumPy programs: differentiate, vectorize, parallelize, Just-In-Time compile to GPU/TPU, and more. The main advantages that we found using JAX were:

- it supports and optimize complex differentiation, for holomorphic and non functions;
- it is extremely optimized, with XLA + JIT that partially compensate the lack of native hardware acceleration;
- many complex operations/layers are already supported and well defined.

Figure 1.5: JAX logo.

More specifically, for building our complex-valued neural networks we will rely on `dm-haiku`, another library built on top of JAX with the purpose of covering the same role that `Sonnet` (widely used ad DeepMind) has for Tensorflow, and to simplify the approach of users that are familiar with object oriented programming.
Since these are quite recent libraries, and also a definitive approach to complex-valued deep learning does not exists yet, I made a careful and complete analysis of the source code of Haiku and of the most important JAX functions. Thanks to this I effectively noticed that much work is still needed: even if in a small quantity respect to Tensorflow/Pytorch, also in this case many operations turns out to be ambiguous or bad-defined for complex-valued data types (e.g. the `square` or the `max` operators). Many of them are also completely undefined (e.g. initialization or, more in general, random complex distributions). We still decided to proceed with JAX mainly because of its flexibility, and many new functions/operations can be redefined without caring of implicit undesired side effects. This is possible also thanks to the design of the training loop, that is quite "explicit" and customizable.

From a practical point of view, I had basically to realize small a '`complex_nn`' library on top of Haiku, containing the definitions (and re-definitions) of all the necessary components of a complex-valued neural network:

- `layers`: the adaptations derived before for linear, convolutional, pooling and normalization operations;
- `activations`: all functions listed in table 1.2;
- `initializers`: weights initializers following uniform or truncated random normal distributions, together with the modified approaches described above by Xavier and He;
- `metrics`: categorical accuracy and categorical cross-entropy, useful for the successive classifiers designs;
- `optimizers`: a *complex-Adam* algorithm;
- a `classifier wrapper` with the purpose of collecting all the necessary functions to setup a training loop with JAX and Haiku;
- for completeness, also some utility functions, mainly to realize plots or wrapping more complex structures.

This code has been tested over several datasets. There are in fact a few `Jupyter Notebooks` providing a detailed explanation of analysis we are going to see, together with some basic setup of some learning procedures.
All the implementation and complete analysis is actually available at my gitlab page [3].

[3] https://gitlab.fbk.eu/mpujatti/complex-valued-deep-learning-for-condition-monitoring