Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi Civita"

Master Thesis in Mathematics

Numerical methods for neural networks and applications

Supervisor
Prof. Fabio Marcuzzi

Author
Pierfrancesco Piotto
1156412

Academic Year 2018/2019

22/02/2019

Contents

Introduction

One trending topic of the last years is with no doubt Neural Networks. Applications of this branch of Machine Learning can range over a wide variety of subjects, just think of automatic translators, voice assistants, object recognition, financial previsions and much more. But what are Neural Networks? They are a family of models that, as the name suggests, are inspired by biology and try to reach a task through a network; more precisely it is a collection of algorithms aiming to approximate a function f^*, through the composition of a bunch of functions $f_1, f_2, ...f_n$, simulating the concept of the network observed in human brain. Roughly speaking, Neural Networks can be considered as a big optimization problem.

The historical birth of Neural Network goes back to 1943 thanks to Warren McCulloch and Walter Pitts. Since that a lot of research has worked out and the majority of its theory has been developed within $1990s$. Nevertheless the biggest applicative results have been found only in the latest years. This can be explained by two factors: a tremendous quantity of data is now available in any sector and the computational power has increased exponentially in the latest years.

Indeed the biggest challenges that have been faced are that these models require a lot of data in order to *learn* and they are quite generic and complex, thus, in the most general case, a big quantity of parameters is needed. For this reason classical optimization algorithms are considered too computationally expensive, so they are not used. Then much simpler ones, like classic Gradient Descent, are used. But it is a matter of fact that they have slow convergence, thus a lot of heuristic and intuitions are used to speed up the process.

This brought to one more problem, that is the utilization of Neural Networks in a black-box way. Namely this means that, in general, there is no way to determine what property a model should have to achieve good approximation capability, and most of the literature about this topic is made by considering many different architectures with no precise criteria.

With that in mind, the purpose of this thesis is to investigate whether the heuristics behind this popular family of models has mathematical confirmation. In particular well known mathematical systems are used to generate data, in order to be able to compare the results obtained through Neural Networks with the true dynamic.

In the first chapter Neural Networks and its *training* procedure is presented in the most general setting. In particular the three main classes of model are going to be explained, both heuristically and mathematically: Multi Layer Perceptrons, Convolutional Neural Networks and Recursive Neural Networks. Beside that, it is going to be explained how the learning process works.

In the second chapter the most significant theoretical results are shown. In par-

ticular the Universal Approximation Theorem, that justify the use of these models, and a mathematical based discussion on some of the most used optimization algorithms. Moreover some accurate investigation is made on some relevant aspect in the stage of building a Neural Networks.

Finally, in the last chapter, some task related to DLTI systems, whose mathematical behaviour is well known, is considered. This allows to take a rigorous look at how each of the techniques in this topic is able to influence convergence and at what happens inside the models. In this chapter it is going to be investigated whether the empirical intuition behind Neural Networks has a practical and mathematical confirmation or not.

Chapter 1

Neural Networks

In this chapter the general frame of Deep Learning is going to be showed. In particular the three main structures for Neural Networks will be presented, namely Multi Layer Perceptron (section 1.1), Convolutional in 1.2 and Recurrent Neural Networks in 1.4. Moreover it is going to be introduced the so called *learning process*, that is the optimization strategy used in Deep Learning, in section 1.3.

But the first step is to build a reference structure for a generic Neural Network. The founding concept at the basis of this family of models is to approximate an objective function f^* through the composition of a bunch of functions $f_1, f_2, ...f_n$ as shown in figure (1.1). Each of them is called *layer*, and in particular the last function, f_n, is called *output layer*, while the others will be referred as *hidden layers*. Pay attention: in most of the literature the term *layer* is often referred to the output of these functions.

To make the frame of these functions f_i clearer, let's take a practical consideration: every dataset is made of a bunch of examples and each of them is made by a collection of observation. For this reason a single example can be considered as a vector and any other layer can be endowed of the same structure. The same exact representation can be noticed in human brain as well, if every component of hidden layers' (called *units*) vectors was considered as a neuron. Then the graph of Neural Networks can be represented as in figure (1.1).

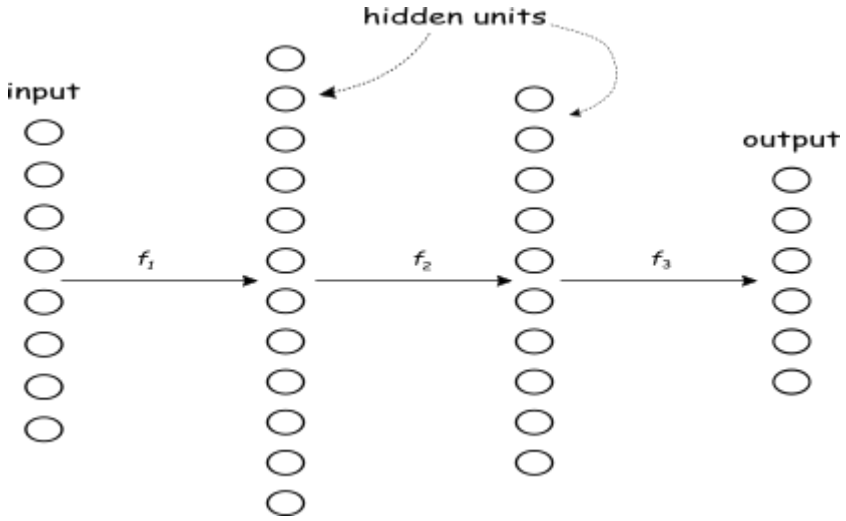

Figure 1.1

1.1 Multi-Layer Perceptron

In this section the most intuitive Neural Network model will be built. Since every layer's output is none other than a vector, the simplest functions that can be implemented are affine functions. Thus the output of this family of models is going to be a function of an affine transformation of the input.

Consider for example the case of a classification task, in particular let's suppose that the dataset is made of two families of samples. One simple case is given by the following figure, where the data are sampled by two distributions, centered in $(0, 0)$ and in $(3, 2)$ as in the following image:

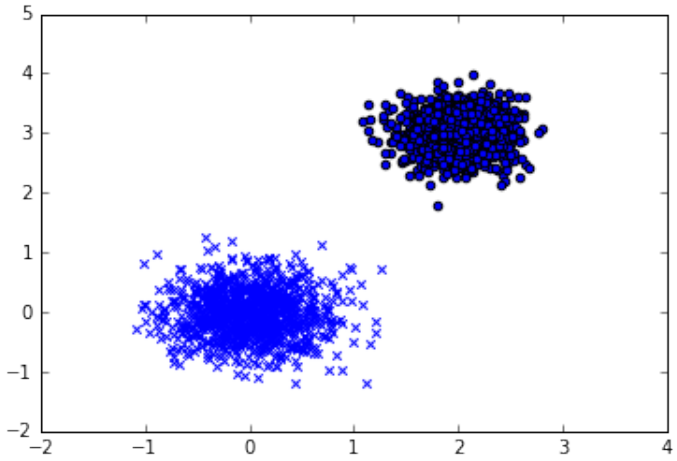

Figure 1.2

The partition of the dataset is clear, and the model can find an affine transformation splitting the input in the two families. In other words there exists a line, parametrized by (w_1, w_2, b) that divides the two groups, as figure (1.2) can suggest. This argument can be generalized for multi dimensional input as follows:

$$\begin{cases} \text{Family 1} & \text{if } w_1x_1 + ... + w_nx_n + b > 0 \\ \text{Family 2} & \text{if } w_1x_1 + ... + w_nx_n + b < 0. \end{cases}$$

However not every function can be approximated by this kind of models: let's take as an example a simple function, named XOR (exclusive or), that takes as input two binary values x_1 and x_2 and outputs 1 if and only if exactly one among x_1 and x_2 equal to 1. Graphically it could be represented as in figure (1.3).

Clearly, no such model can split the two families. The reason is quite intuitive: these data can not be divided by a line. In particular two key elements are missing in this example: non linearity and, accordingly, composition, since composing affine functions is the same of computing a single affine transformation, and thus it is useless. On the other hand, if each layer would be provided with a nonlinear function, then composition becomes significant. Then, implementing these components plays a key role in Neural Networks.

Each layer is then computed as $g(W^Tx+b)$, where g is a fixed non linear function applied element wise. Intuitively this representation means that the layers do not

x_1	x_2	output
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.1: XOR

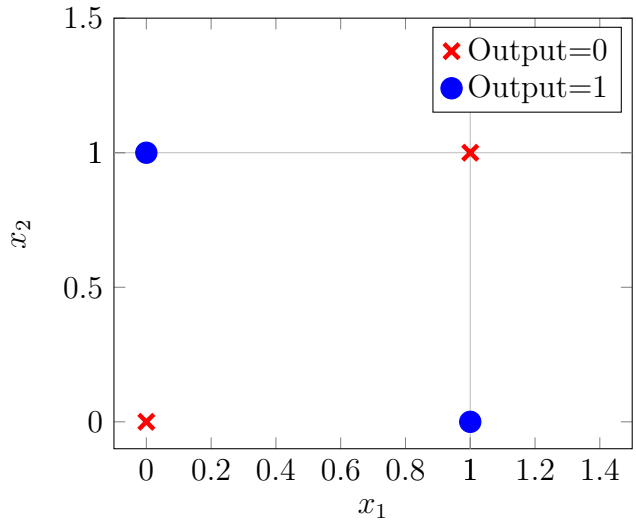

Figure 1.3

consist of applying the affine model directly to the input x, but to a transformed input $\Phi(x)$, where Φ is a nonlinear transformation, and eventually it allows to combine more than one layer. Φ is called *activation function*. In this way the linear separation is not applied to the data represented in the classic Cartesian plane, instead it is applied to the data represented in a new shape, that may be helpful to find the key features.

In particular, in the example above, one possible solution may be considering Φ as:

$$\begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \max\{0, W^T x + c\} \tag{1.1}$$

where:

$$W = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \qquad , \qquad c = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

Notice that the activation function is not applied to all the vector, but actually to each unit.

Graphically speaking the data in the new space are represented by figure 1.4.

In this representation, there is a clear separation line between the two families of data.

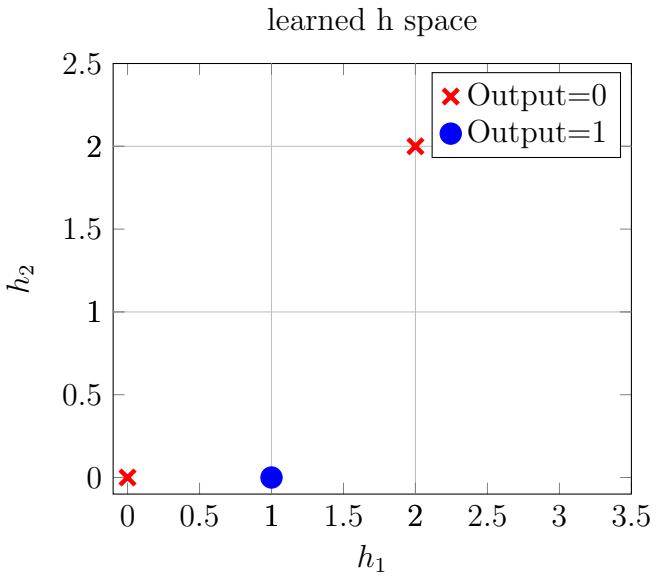

Figure 1.4

Activation Functions

The last building block of Multi Layer Perceptron is then the choice of the non linear activation function g. One of the most common as well as the first class of function considered for this task, is the *sigmoid* one, which are monotonically increasing functions that asymptotes at some finite value as is approached $\pm\infty$. The most common examples are the standard logistic sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ and hyperbolic tangent $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. This class of functions is used because of a simple reason: they allows to represent binary target values (e.g. $\{0, 1\}$ for the logistic sigmoid or $\{\pm 1\}$ for hyperbolic tangent). This is due to an important biological consideration: in human brain neurons interact with each other through electricity, in the sense that every neuron can either give an impulse or not and, roughly speaking, the decision is took by considering what units have given such an impulse. Thus *tanh* and σ, and more in general sigmoid functions, are implemented to replicate this argument: they try to *squash* the input in two different classes. However the functions that achieved the main results in Neural Networks are the so called *rectifiers* or *ReLu units*, that are the functions used to solve the XOR problem, i.e. $f(x) = \max\{0, x\}$. The reason why these functions are preferred will be explained with more details in section 2.3.3.

1.2 Convolutional Neural Networks

Multi Layer Perceptrons may seem perfect models for analysing almost any kind of real data. However there are many drawbacks in using them. First of all they are very expensive both in computational and in memory cost. Indeed, let's suppose that a certain layer is given n inputs and has to calculate m outputs. The operation it has to compute is a matrix-vector product, and it needs $n \cdot m + m$ parameters, one

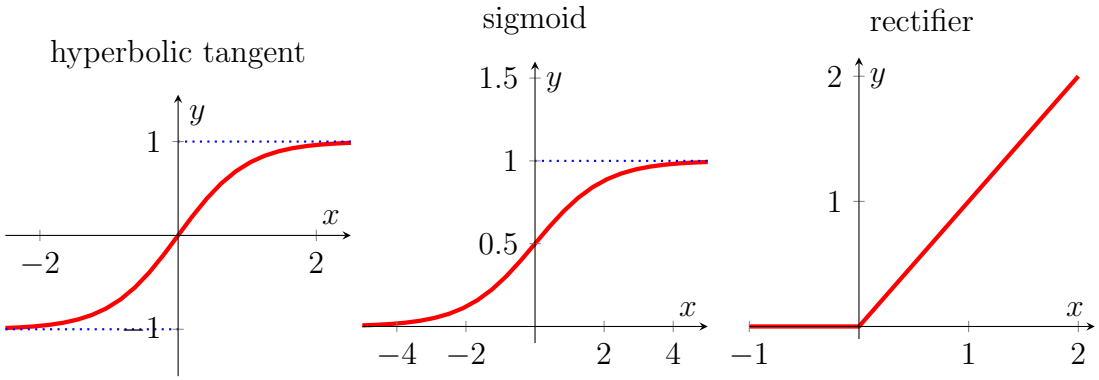

Figure 1.5

for each entry of the matrix and one for each bias term. This also means that there is an high risk of getting an overfitting model. Moreover the computational cost is of the order of $O(n \cdot m)$ for the same reason. And this is not even the main drawback: the principal problem is due to the fact that they do not consider the structure of the data at all. Consider, for example, the task of recognizing an object given a certain image, after that the training has been done: there is no way to have the same exact settings for two different images, even though they represent the same subject. Real world data have lot of details that can change, for example it may be the case of a slight change in the lighting or in the position of the object considered in the classification, since the location of the key features, like an eye or the mouth, can be translated by some pixel. These differences are meaningless for human being, but actually they are important for models like Multi Layer Perceptron. The reason is quite simple: once the training is completed, the weights and the biases of each layer are fixed. This means that small changes in the input will cause a change in the next layers and this variation will propagate through all the layers. Thus if two similar images representing the same object are taken as samples, the Network may output different values. This problem is attributable to a lack of invariance for little translations.

The last problem is once again related to structured inputs and it is completely heuristic. Multi Layer Perceptrons do not care about the topology of the data. This concept could be better understood by a question: does considering the interaction between two far inputs (in time or in space, for example) make sense? More practically, in most cases, it is unnecessary to consider a feature given by the value of the pixel in the high-left corner and the one in the down-right corner, because there is no characteristic of the image that can be deduced in real world data, with an high probability. On the contrary, it makes sense to consider a sparse interaction network, instead of a fully connected network, like Multi Layer Perceptron.

One of the ways that have been suggested to solve all these problems is to consider a model that is able to preprocess the raw data in order to get a vector of features. This vector can then be processed by a Multi Layer Perceptron since it is just a small list of values with no intrinsic meaning. The state of the art of this kind of models is represented by *Convolutional Neural Networks* (CNNs): they are made by a series of so called convolutional layers (used to pre process the data) and, in the final stage, by a small of Multi Layer Perceptron, also called, in CNNs' literature,

Fully Connected Layers.

Convolutional layers's structure is quite similar to the one of the Perceptrons, in the sense that the input and the output are still vectors (or, more in general, tensors). However these layers implement two new functions that are able to solve all the problems described above: convolution and pooling.

The purpose of this section is then to show the benefits given by convolutional layers and to examine rigorously these functions.

1.2.1 Convolution

Convolution is the main component of CNNs. The empirical meaning of this operation is essential, if one wants to understand why convolution is so important. Convolving a vector with one other, called *kernel*, has to be considered not only as the mathematical operation, but also as the correlation between the two inputs (in fact they the two operators are equivalent) or, more intuitively, as filtering the data through the kernel. Let's make this concept more clear and, in particular, consider the case of processing a black and white image. In image analysis convolving a small piece of the matrix given by the greyscale value of each pixels with a specific kernel allows to find a value associated with the characteristic correlated with the kernel. The higher this value is, the higher the probability of having this characteristic in that small piece of image. Repeating this procedure for all the pixels gives a new image (that will be called *feature map*) that empirically can be thought as an heat map for the sought characteristic. For example the following kernel is used to seek the edges:

$$M = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Then the result of convolving an image with this kernel is a new figure composed only by the edges of the starting image.

Advantage of using convolution

Clearly convolution is based on sparse interaction and this solves the third problem presented in the first part of the section. Empirically the algorithm looks for local features that will be assembled in the next layers to get overall features, whereas Perceptrons try to find overall features immediately. The second advantage is given by the fact that the input may be very large, virtually millions of values, but the kernel is usually restrained. Let's suppose to consider a layer whose input's size is n and it outputs a vector of dimension m. These quantities may be very big. Then, as stated before, a Multi Layer Perceptron needs a matrix of size $n \cdot m$ (and eventually m biases), and the computational cost will be $O(n \cdot m)$. On the other hand, if the size of the kernel is k ($<< n, m$), then the parameter required are only k, since the convolution is computed using always the same kernel, and the computational cost will be $O(k \cdot m)$. This means that the memory and the computational cost required are way less than the ones required by Perceptrons. Moreover, as a consequence, the capacity of the model is reduced and then this operations prevents overfitting.

Mathematics and other types of convolution

The starting point of convolution is the classic mathematical operation between two real valued functions f and g:

$$f * g(t) = \int_{-\infty}^{+\infty} f(t-\tau)g(\tau)d\tau$$

However in data analysis the convolution is computed between two vectors x and w with a finite dimension. In this case w is the kernel and its dimension, indicated by k, is supposed to be inferior with respect to the dimension of x. Then the operation is discretized, and the result is:

$$x * w(n) = \sum_{i=1}^{k} x(n-i)w(i) \tag{1.2}$$

This operation is called *discrete convolution*. Since in Neural Networks the kernel is not pre defined, it can be "flipped", in the sense that its last component becomes the the first, the last but one component becomes the second one and so on. Notice that this procedure is not a crime because the kernel has to be learned, but it gives a more suitable shape to equation (1.2):

$$x * w(n) = \sum_{i=1}^{k} x(n+i)w(i)$$

This operation can be extended to two dimension cases in a straightforward way. If the input matrix is y, the kernel is v and its dimension is $k_1 \times k_2$, then:

$$x * w(n,m) = \sum_{i_1=1}^{k_1}\sum_{i_2=1}^{k_2} y(n+i, m+j)v(i,j)$$

This is the classic equation used in image processing. However in this work the notation will be slightly different and it will be based on Python's notation. Thus a vector of dimension k will have components running from 0 to $k-1$ and the equations will be rewritten as:

$$\begin{aligned} x * w(n) &= \sum_{i=0}^{k-1} x(n+i)w(i) \\ y * v(n,m) &= \sum_{i_1=0}^{k_1-1}\sum_{i_2=0}^{k_2-1} y(n+i, m+j)v(i,j) \end{aligned} \tag{1.3}$$

This notation is useful since it allows to use more compact formula in the next lines.

Nevertheless, calculations in CNNs are a bit different. Indeed, if the input is an image it may be that its components are not just a matrix, but a 3-D array, since it may be given with a RGB representation (with colours). Moreover the process described above works if the model is interested in only one feature, because the output of equation (1.4) is a feature map corresponding only to one characteristic, but in general more features are required. In the literature these multiple feature maps are called *channels*, both for the input and the output. The result of these considerations is that the kernel has to be a 4-D array, let's say v_{mijk} where:

- $m \equiv$ output channel;
- $i \equiv$ input channel;
- $j, k \equiv$ raw and column *distance* of the output element with the input's one. Namely, for any a, b, the element in the row a and column b of the $m-$th output channel is influenced with a weight equal to v_{mijk} by the input of channel i of the $(a+j)-$th row and $(b+k)-$th column.

Then the equation of convolution used in CNNs (for images) is:

$$O(m,a,b) = \sum_{i=0}^{M-1} \sum_{j=0}^{k_1-1} \sum_{k=0}^{k_2-1} y(i, a+j, b+k) v(m,i,j,k)$$

Now, suppose the image considered has a resolution of 1000×1000 pixels, that is a realistic size (it could be even higher). The dimension of the kernel will naturally grow as well, since considering a portion of 3×3 pixels is quite insignificant. The same reasoning could be applied in one other sense, bringing to the question: does applying convolution centered in *every* pixel make sense? A priori, having a big feature map is not wrong, but it may result pretty useless. Thus one may think to translate the center of the n-th convolution by more than one pixel with respect to the $(n-1)$-th convolution. In order to define mathematically this argument, a new parameter (eventually an array) is introduced: the *stride* s.

$$O(m,a,b) = \sum_{i=0}^{M-1} \sum_{j=0}^{k_1-1} \sum_{k=0}^{k_2-1} y(i, a \cdot s_1 + j, b \cdot s_2 + k) v(m,i,j,k) \tag{1.4}$$

Actually this is not the only operation that is allowed in a convolutional layer, but there are other ones that can be used according to the particular model one may wish to use.

The first case is the so called *unshared convolution*. Even though this is classified as convolution, it computes each entry of the output with an affine transformation of a portion of the input. Conceptually it is like a mix between Perceptron's and convolutional layers, since there is sparse interaction among input's neurons, but each output's unit is calculated with new parameters (in convolution the parameters are shared). This kind of layers are used if the model is thought to look for different feature for each location of the input. The computational cost of such a layer is $O(\frac{n}{k \cdot s} \cdot m)$ and the number of parameters is about $\frac{n}{s}$. The equation of this operation is given by:

$$O(m,a,b) = \sum_{i=0}^{M-1} \sum_{j=0}^{k_1-1} \sum_{k=0}^{k_2-1} y(i, a+j, b+k) v(m,a,b,i,j,k)$$

A middle way between classic and unshared convolution is the so called *tiled convolution*. The idea is to consider a certain number of kernels that are alternated, in the sense that, if there are 5 kernels, the first one is used to compute the first, the sixth,... convolution. In most application this kind of layer is used in order to consider different transformation of the same feature. Let's make this concept more clear: instead of seeking vertical edges with a single kernel, in some applications one

may be interested in looking for edges inclined with a fixed slope of, let's say, $\frac{\pi}{3}$ and $-\frac{\pi}{3}$ as well. Then tiled convolution uses three kernels in the same feature map. The idea is to make a summary of these computations, for example taking the maximum among these values, in order to get the value that could be read as: there is an edge with slope either $\frac{\pi}{3}$ or $-\frac{\pi}{3}$ or a vertical with probability λ.

Then, if the number of kernels is t, the equation is:

$$O(m,a,b) = \sum_{i=0}^{M-1}\sum_{j=0}^{k_1-1}\sum_{k=0}^{k_2-1} y(i, a\%t + j, b\%t + k)v(m, a\%t, b\%t, i, j, k)$$

Finally, the last algorithm that is going to be mentioned is dilated convolution. It is a classic convolution, but for every term of the summation the step in the input is not 1, but d (called *dilation rate*). It is the same of considering a sparse kernel. In formula:

$$O(m,a,b) = \sum_{i=0}^{M-1}\sum_{j=0}^{k_1-1}\sum_{k=0}^{k_2-1} y(i, a\cdot s + j\cdot d_1, b\cdot s + k\cdot d_2)v(m, i, j, k)$$

Zero padding and output dimension

One natural question that arises when considering convolutional layers is: what is the dimension of the feature map? Well, let's consider equation (1.4), with an input $y \in \mathbb{R}^{N_1\times N_2}$ it is easy to see that the answer, in this case, is:

$$\begin{cases} \text{width} = \lfloor \frac{N_1-k_1}{s} \rfloor + 1 \\ \text{height} = \lfloor \frac{N_2-k_2}{s} \rfloor + 1 \end{cases} \tag{1.5}$$

This formula shows a problem of such models: the dimension of the output is fixed once all the parameters are fixed. Since this dimension is smaller than the input one, the number of layers is limited. For this reason convolutional layers implicitly provides a zero padding for the input in order to control the output size. Thus, defining the padding parameter as P, that symmetrically surround the input, this dimension is going to be:

$$\begin{cases} \text{width} = \lfloor \frac{N_1-k_1+2P_1}{s} \rfloor + 1 \\ \text{height} = \lfloor \frac{N_2-k_2++2P_2}{s} \rfloor + 1 \end{cases}$$

There are two special cases for zero padding. The first is reached when P is simply set to 0, and in this case the convolution is called *valid convolution*. The second one is the natural case, that is P is choosen in order to maintain the dimension of the output equal to the one of the input. In this case the convolution is called *same convolution*. These are the most used padding strategies, and most of the libraries implementing Neural Networks uses, like *TensorFlow* uses command "same" and "valid". However, since these are the extreme cases, in most applications P may be choosen higher than 0 and lower than the "same" value.

1.2.2 Pooling

Pooling is the second main function used in convolutional layers. It is a generic statistics used to summarize the results obtained in a certain region. Intuitively convolution looks for features all over the input and pooling summarizes the result, telling that a certain characteristic is present in a wider zone or not. The two most popular pooling functions are the arithmetic mean (*average pooling*) and the maximum function (*max pooling*).

Advantage of using pooling

The obvious advantage of using a pooling function is that it reduces the size of the feature map. This means that the computational cost is reduced as well as the risk of overfitting. Moreover, this function is helpful in handling input of varying size, together with equation (1.5). However the most sharp effect is that is permits to learn the transformation of the input, together with tiled and unshared convolution. As stated in the last section, using different kernels to compute the same feature map can be thought as the research of different characteristics in any different place of the image. Now, let's suppose that the difference is just the fact that every kernel looks for the same characteristic, but transformed. For example a straightforward transformation could be rotation. Using max-pooling among all the values calculated by computing the convolution for every single kernel allows the model to understand if that characteristic is in that particular zone, no matter if it is rotated (or more in general transformed). This process is called *learned transformation*.

1.2.3 Construction of a CNN

Now that all the ingredients have been presented, it is possible to present to build a Convolutional Neural Network. First of all, a certain number of convolutional layers has has to be implemented. The first step is to calculate the feature map through convolution (on of the different types presented in this work), whose entries are typically transformed by the activation nonlinear function (the most used one is ReLu). Then, the final stage of the convolutional layer is the pooling function, that is not always necessary. A bunch of these layers is composed and the output of the CNN is obtained by applying by the Fully Connected model to the features spotted in the first stage. Pay attention: in general, the feature map computed by convolutional layers is a tensor, whereas the input required by a Multi Layer Perceptron is a vector. It is then needed a process whose purpose is to vectorize this tensor. This intermediate step is called *flattening*.

1.3 Learning Process

Recall the structure of a general Neural Network: it is a composition of parametric functions, whose purpose is to approximate a certain function f^*. The learning problem consists of finding the optimal combination of weights so that the network function approximates f^* as closely as possible. However, f^* is not given explicitly, but only implicitly through some examples.

Now, let's consider a general model and suppose that it receives as input x and a set of weights θ, it combines them and it obtains an output y. The idea is to evaluate the goodness of such a model through a loss function, that measures the performance. It could be considered as the error function to make the concept more clear. Then the aim is to minimize this error.

Clearly, the branch of the mathematics that studies this problems is optimization. However there are a lot of drawbacks for applying the precise mathematical algorithms, like the fact that they would bring computational cost extremely high. Then in the following subsections, some meta heuristic methods used in Neural Network learning process will be presented.

1.3.1 Learning Algorithms

Gradient Descent

Gradient descent is one of the most popular algorithms to perform optimization and most primitive way to optimize Neural Networks. This is an iterative method that makes use of a classical mathematical result that states that the opposite of the gradient of the loss function, calculated in the current point θ, is a descent direction, in the sense that there exists a (sufficiently small) step size η such that the loss function has a lower value, if it is calculated in $\theta + \eta \cdot d$. Then the frame of the algorithm is to iteratively calculate the parameters following the rule:

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta L(\theta_k)$$

This approach has a lot of drawbacks. The first one is the computational cost: the loss function is the sample mean of the loss functions calculated on each single example of the dataset. Moreover gradient is a linear operator, then calculating $\nabla_\theta L(\theta)$ requires calculating the gradient of the loss function for each sample. Moreover, in complex NN there may be lot of parameters to be determined. Virtually, combining all these factors makes the computational cost very high. Then one may wonder if considering just a portion of the dataset would cause an high distortion for the accuracy of the computation. This observation deals with a simple probabilistic proposition:

Proposition 1.3.1. *The variance of the sample mean of a family of i.i.d. random variables* $\{X_n\}_{n\in 1,..,N}$ *is* $\frac{1}{N}Var(X_1)$

Proof. Since $Var(aX) = a^2 Var(X)$ and $Var(X_1 + X_2) = Var(X_1) + Var(X_2)$

$$\begin{aligned} Var(\frac{1}{N}\sum_{n=1}^{N} X_n) &= \frac{1}{N^2} Var(\sum_{n=1}^{N} X_n) \\ &= \frac{1}{N} Var(X_1) \end{aligned}$$

□

This means that, since standard deviation can be computed as the square root of the variance, the estimated loss of accuracy decreases like $\sqrt{N}$ as the number

of samples increases. More practically, this means that considering N rather than $10000 \cdot N$ samples means losing two significant figures only. This is the peculiarity of *Mini Batch Gradient Descent* (MBGD). It works exactly as GD, but it uses only a portion of the dataset to compute the gradient. Common mini batch sizes range between 32 and 256, since power of 2 sized batches offer better runtime [1]. This algorithm is guaranteed to converge and it is much faster than simple Gradient Descent. Moreover, since its variance is higher, it provides a simple way to avoid getting stuck in a non optimal stationary point. However, for the same reason, the solution found by MBGD may oscillate around the optimal solution. In addition to these problems, there are also other problems related to the structure of Gradient Descent: the parameter η, called *learning rate*, in optimization theory is usually computed using some technique such as the *Armijo method*. However these are quite expensive methods, then they can not be implemented in Neural Network, thus η is a fixed hyperparameter for Gradient Descent. How can be determined a good value for η? This is not an easy task, since choosing η too small brings to a painfully slow convergence; on the other hand, if η is too big, convergence is not guaranteed any more, since the update of parameters may not decrease the value of the loss function. In order to have an algorithm that adapts the learning rate to the specific task *Momentum* algorithm will be presented.

Additionally, the same learning rate applies to all parameter updates. If the gradient is sparse or the features have very different frequencies, one might not want to update all of them to the same extent, but perform a larger update for rarely occurring features. Let's make an example to clarify this example: consider the case in which the loss function has an high value due to only one parameter w. However, suppose that, around the particular θ considered in a particular iteration, the function, considered only as a function of w is almost flat. This means the partial derivative with respect to that particular parameter is very small or even zero and that the update of w will be insignificant. Then the loss function will not reach a much lower value. One may want to make a bigger step on w in order to move on a different field. On the other hand, considering such a fixed high value for the learning rate may bring convergence problems, as stated above. The idea is to consider the learning rate not as a fixed number, but rather a vector with different values for each parameter. The algorithms that try to attempt this purpose are called *Adaptive*.

Momentum and Nesterov Algorithm

MBGD has troubles navigating areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. In these scenarios, MBGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum. This behaviour can be explained by a simple example: consider a ball rolling down an hill. The ball rolls down following a dominant direction, even if it is contrasted by friction and it is slightly influenced by hill's morphology. In this example MBGD makes a small step and then stops at each iteration. Then it will neither follow the *correct* direction, since it will be influenced by the slope of the hill in that particular point, nor have an high velocity. Following this argument, Momentum algorithm is implemented.

The update rule is:

$$\begin{cases} v_{k+1} = \gamma v_k + \eta \nabla_\theta L(\theta_k) \\ \theta_{k+1} = \theta_k - v_{k+1} \end{cases}$$

where γ is a coefficient, usually set to 0.9.

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. The idea is to have a smarter ball, that has a notion of where it is going so that it knows to slow down before the hill slopes up again. This purpose is reached by considering the fact that the dominant factor of $\gamma v_k + \eta \nabla_\theta L(\theta_k)$ is γv_k since η usually is about 10^{-3}. Then θ_{k+1} will be in the neighborhood of $\theta_k - \gamma v_k$, so the evaluating the gradient in that point is a good approximation to the real gradient calculated in θ_{k+1} and can give an indication to the algorithm in advance. Thus usually Momentum algorithm is boosted by Nesterov algorithm:

$$\begin{cases} v_{k+1} = \gamma v_k + \eta \nabla_\theta L(\theta_k - \gamma v_k) \\ \theta_{k+1} = \theta_k - v_{k+1} \end{cases}$$

Adaptive Algorithms

Adaptive algorithms for gradient-based optimization does just this: they adapt the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. For this reason, it is well-suited for dealing with sparse data. Previously, the update was performed for all parameters θ at once as every parameter θ_i used the same learning rate η. As the considered algorithm uses a different learning rate for every parameter θ_i at every step k, the algorithm will be shown per-parameter update:

$$\begin{cases} m_{k+1,i} = m_{k,i} + \left(\frac{\partial}{\partial \theta_i} L(\theta_k)\right)^2 \\ \theta_{k+1,i} = \theta_{k,i} - \frac{\eta}{\sqrt{m_{k+1,i}+\epsilon}} \frac{\partial}{\partial \theta_i} L(\theta_k) \end{cases}$$

This describes the procedure of *Adagrad*. The main deficiency of this approach is that m_k is a sum of positive terms, thus for some parameters it could become very big. So the new learning rate can be so small that the parameter will not be update significantly.

This is the reason why a new algorithm is preferably used: *RMSProp*. It works exactly as Adagrad, but the vector m has an exponential decay:

$$\begin{cases} m_{k+1,i} = \beta m_{k,i} + (1-\beta)\left(\frac{\partial}{\partial \theta_i} L(\theta_k)\right)^2 \\ \theta_{k+1,i} = \theta_{k,i} - \frac{\eta}{\sqrt{m_{k+1,i}+\epsilon}} \frac{\partial}{\partial \theta_i} L(\theta_k) \end{cases}$$

Adam

The last optimization algorithm used in Neural Networks that is going to be showed is the so called *Adam* (Adaptive Moments). It has been presented for the first time in 2014, in the article [12] by D.P. Kingma and J.L. Ba. As the name may suggest,

it combines both Adaptive methods and Momentum algorithm. The algorithm is given by:

$$\begin{cases} v_{k+1} = \beta_1 v_k + (1-\beta_1)\nabla_\theta L(\theta_k) \\ m_{k+1,i} = \beta_2 m_{k,i} + (1-\beta_2)\Big(\frac{\partial}{\partial\theta_i} L(\theta_k)\Big)^2 \\ \hat{v}_{k+1} = \frac{v_{k+1}}{1-\beta_1^{k+1}} \\ \hat{m}_{k+1} = \frac{m_{k+1}}{1-\beta_2^{k+1}} \\ \theta_{k+1,i} = \theta_{k,i} - \frac{\eta}{\sqrt{\hat{m}_{k+1,i}}+\epsilon}\hat{v}_{k+1,i} \end{cases}$$

Moreover the authors suggest to use the following values for the hyperparameters of this algorithm:

$$\begin{cases} \eta = 0.001 \\ \beta_1 = 0.9 \\ \beta_2 = 0.999 \\ \epsilon = 10^{-8} \end{cases} \tag{1.6}$$

m_k and v_k are estimates of the first moment (the mean) and the second moment (the variance) of the gradients respectively. As m_k and v_k are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero. To counteract this effect, v and m are rescaled in order to get bias-corrected terms, $\hat{v}$ and $\hat{m}$. Indeed they are unbiased estimators of the first and the second moment of the gradient:

Proof. Suppose that the gradients are independent and identically distributed variables. Let's call the partial derivative with respect to the i-th variable g_i. Then:

$$\begin{aligned} \mathbb{E}[m_{k,i}] &= \mathbb{E}[\beta_1 m_{k-1} + (1-\beta_1)g_i^2] = (1-\beta_1)\sum_{j=1}^{k}\beta_1^j \mathbb{E}[g_{j,i}] \\ &= (1-\beta_1)\frac{1-\beta_1^k}{1-\beta_1}\mathbb{E}[g_{i,1}] \\ &= (1-\beta_1^k)\mathbb{E}[g_{i,1}] \end{aligned}$$

The computations for v are analogous. □

Finally Adam combines the techniques described above in computing the new parameters.

1.3.2 Back Propagation

The final step of learning consists of actually computing the gradient with respect to the weights of the loss function. However computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. Back Propagation is a simple algorithm that trades off memory for computational cost.

Let's consider a simple portion of a layer, in particular consider the procedure used to calculate a certain unit u_i in $(n+1)$-th layer. It will be the result of a certain computation of some input $x_1, ...x_k$ and weights $w_1, ...w_h$, that is $u_i = f_i(x, w)$. Now,

let's make this example very simple and suppose that these weights and are used only in the calculation of u and no other neurons. Finally, imagine the loss function as a particular function of the $(n+1)$-th layer: $L = F(u)$. Thus the gradient of L with respect to x_i and w_j will be:

$$\begin{cases} \frac{\partial L(x,w)}{\partial x_1} = \frac{\partial L(x,w)}{\partial u_i}\frac{\partial u_i(x,w)}{\partial x_1} \\ \vdots \\ \frac{\partial L(x,w)}{\partial x_n} = \frac{\partial L(x,w)}{\partial u_i}\frac{\partial u_i(x,w)}{\partial x_n} \\ \frac{\partial L(x,w)}{\partial w_1} = \frac{\partial L(x,w)}{\partial u_i}\frac{\partial u_i(x,w)}{\partial w_1} \\ \vdots \\ \frac{\partial L(x,w)}{\partial w_h} = \frac{\partial L(x,w)}{\partial u_i}\frac{\partial u_i(x,w)}{\partial w_h} \end{cases}$$

Here is the point: the coefficient $\frac{\partial L(x,w)}{\partial u_i}$ is calculated for every variable that contribute in calculating u_i. This procedure can be generalized: y and v are the inputs and the weights used to calculate (x, w), then $\nabla_{(y,v)} L$ employs $\frac{\partial L(x,w)}{\partial u_i}$ once again, for every variable, because of the chain rule. If this argument was repeated for each layer it is easy to understand that the number of times in which $\frac{\partial L(x,w)}{\partial u_i}$ is calculated grows exponentially with the number of layers before u, that is n. For this reason computing the analytical expression of the gradient is infeasible.

Nevertheless the same reasoning can be used to find a solution. The main problem is that a lot of coefficient has to be calculated many times per Gradient Descent iteration. What if these numbers were stocked in memory? This is the way Back Propagation works: starting from the output it moves back and for each step it calculates a gradient with respect to the weights and one with respect to the inputs. The latter is used to calculate the gradients of the previous layer, whereas the former is stocked to compute optimization steps.

Now it is easy to understand the name *Back Propagation*: this method flow backward through the computational graph of the NN and computes a derivative for each connection between two consecutive layers, adding all the contributes as stated by the chain rule.

Back Propagation in CNNs

Let's consider a particular convolutional layer that, given the matrices of inputs x and of the weights $w \in \mathbb{R}^{k_1 \times k_2}$, computes the output $y \in \mathbb{R}^{N_1 \times N_2}$ as follows:

$$y(a,b) = \sum_{i=0}^{k_1-1} \sum_{j=0}^{k_2-1} x(a+i, b+j) w(i,j) \tag{1.7}$$

After that, y is used to compute the loss function L through a generic function F. Considering the operation of Back Propagation, the aim is to calculate $\frac{\partial L}{\partial x(a,b)}$ and $\frac{\partial L}{\partial w(i,j)}$ for all indices, given the matrices of partial derivatives of L with respect to y, that is $\frac{\partial L}{\partial y(a,b)}$ for all $(a,b) \in \mathbb{R}^{N_1 \times N_2}$. Let's calculate the matrices of derivatives with respect to the weight, and the one of the inputs can be calculated analogously:

$$\frac{\partial L}{\partial w(i,j)} = \sum_{a=0}^{N_1-1} \sum_{b=0}^{N_2-1} \frac{\partial L}{\partial y(a,b)} \frac{\partial y(a,b)}{\partial w(i,j)}$$

Now, equation (1.7) brings to:

$$\frac{\partial L}{\partial w(i,j)} = \sum_{a=0}^{N_1-1} \sum_{b=0}^{N_2-1} \frac{\partial L}{\partial y(a,b)} x(a+i, b+j)$$

This is a classic convolution, where the kernel is given by the matrix of the derivatives of L with respect to the next layer y.

1.4 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of Neural Network that is generally used to study sequences. The peculiarity of this data type is the importance of the order of the records, due to the fact that each sample is not independent from the others anymore. Intuitively RNNs come into picture when the context is needed in order to provide the output based on the input and they are widely used in *next word prediction, image captioning, speech recognition, time series analysis, stock market prediction.*

1.4.1 Are CNNs perfect?

It has already been showed that every function can be approximated by Multi Layer Perceptrons (and Convolutional Neural Networks), but nothing has been said about the efficiency. In the following, some of the drawbacks related to MLPs and CNNs are going to be exposed.

- **Input and output size:** the first problem that is faced in these models is the length of the input. For example, let's consider an algorithm that tries to analyse a word text. It can assume several meanings according from the context. And how many word is a *context*? Clearly it is not possible to give an answer. A more rigorous example is a model whose purpose is to predict the time series generated by the following dynamical system:

$$y(k) + y(k-1) = u(k) \quad \forall k \leq \text{time series length} \tag{1.8}$$

  where y is the response, whereas u is the input. Clearly a traditional Neural Network can take into account N(=time series length) elements and return N values for the output. However it is obvious that the system (1.8) can generate sequences of any length, provided that the input is large enough! And that is not all, because sometimes the dimension of the output can be variable as well. For example the system:

$$y(k) = f(y(k-1), ..., y(k-m)) + g(u(0), ..., u(t)) \tag{1.9}$$

  can generate a time series of any length, provided that the input size is t. But classical models can not, since the input and the output sizes are fixed. The only shrewdness that can be used is to set a big enough size for the input and the output, eventually through zero padding. However this is not clever,

since it would require much higher capacity than what is needed and thus overfitting risk and much more parameters to train. And this is obvious since both examples compute recursively the same operation on the input and on the past output, but the model is looking for a number of functions that will be equal to the output size.

- **Information flow:** as it has already been said in the introduction of this section, RNNs rose in order to study sequences, whose peculiarity is that its elements are related to each others. This means that in order to predict the next element of a sequence, the knowledge of what happened before is needed. For example, let's think about prediction of the next word in a sentence: in order to get the best output, the context is needed. Furthermore, the systems (1.8) and (1.9) need the past outputs to compute the present one. But classical networks can only take as input $(u(0), ..., u(k), y(0), ..., y(k-1))$ and output $y(k)$, but this is not enough, since this operation has to be repeated for every k (that is the prototype of an RNN). Eventually it could take the input $(u(0), ..., u(k))$ and predict the whole sequence of output y, but this would be a simple map connecting u and y, without getting the real structure of the model (the map should involve a connection between y and itself).
- **The importance of the order:** let's say that a model trying to find the meaning of a text, a very important task nowadays, Google Translator or any voice assistant are some of the exponents of this. One simple idea is to consider the words of the dictionary as the components of the input. Then the vector associated to the sentence has coordinates that indicate the number of times that a particular word has been wrote in it. For example the phrase:

$$\text{"The food was good, not bad at all"} \tag{1.10}$$

would correspond to the vector with all 1 components, if the dictionary is $\{the, food, was, good, not, bad, at, all\}$. But then, also:

$$\text{"The food was bad, not good at all"} \tag{1.11}$$

would have the same representation, even though the meaning is the opposite! The problem of this architecture is that it associates a weight to each word, without considering the order at all. One possible way to remedy this situation is to pair each word with its position. This make the order matter within the model. Then the input would be a matrix. One problem is that this model is hard to train. This is because the same meaning can be expressed by different sentences, for example:

$$\text{"It was snowing on monday"} \tag{1.12}$$

$$\text{"On monday it was snowing"} \tag{1.13}$$

whose two input matrices have very different representations. Moreover it requires a lot of parameters, since the dimension of the input is

$$\text{(dimension of the dictionary)} \cdot \text{(number of possible positions)}$$

a number that may be very large.

1.4.2 Construction of an RNN

The purpose is to build a Network that can remedies all the inefficiencies of standard networks when the problem is related to a sequence. Thus, in particular, one of the properties needed is parameter sharing for different times, that enables the generalization of the model to input/output of different shape and introduces the concept of time/order. One obvious and intuitive way to achieve that is the introduction of a loop in the computational graph, in the sense that the current output is computed as a function of the input and one or more elements of the previous time steps, eventually more than one, as shown in figure (1.6). Here, three simple examples from *Deep Learning* [1] are given:

- **Hidden-to-hidden connection:** for this group of models the propagation of the information is given by considering the current hidden unit as a function of the previous one and of the input. Mathematically:

$$\begin{aligned} h(t) &= ActivationFunction(Uu(t) + Wh(t-1) + b) \\ o(t) &= Vh(t) + c \end{aligned}$$

  This representation is usually referred as the classical Recurrent Neural Network;
- **Output-to-hidden connection:** in this case the hidden unit is given by:

$$\begin{aligned} h(t) &= A.F.(Uu(t) + Wo(t-1) + b) \\ o(t) &= Vh(t) + c \end{aligned}$$

  Empirically this representation is weaker, since the output can be considered as a function of the information h, thus in this case not every piece of information can flow through the graph. However this allows to use a less expensive training technique called *teacher forcing*, that will be explained in section 1.4.3;
- **Hidden-to-hidden connection with single output:** now the process is the same of the first case, but there is a single output.

$$\begin{aligned} h(\tau) &= A.F.(Uu(\tau) + Wh(\tau-1) + b) \quad \forall \tau \in \{t-m, ..., t\} \\ o(t) &= Vh(t) + c \end{aligned}$$

  such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing.

Recall that one of the main innovations given by Neural Networks is the so called *deep* structure and notice that these models may seem not so deep. But these are just simple examples used to introduce the structure of an RNN. Consider the classical RNN: it can be described by three blocks of parameters,

1. from the input to the hidden state
2. from the previous hidden state to the present one
3. from the hidden state to the output

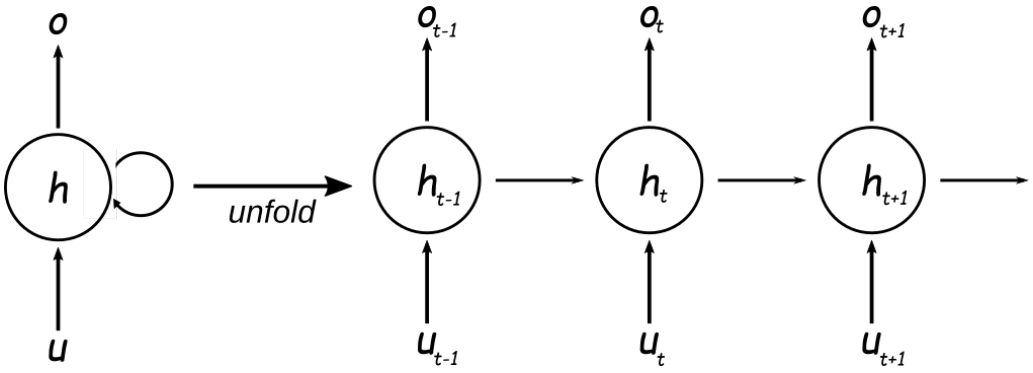

Figure 1.6

The depth of Recurrent Neural Networks is given by expanding the central block, implementing more than one layer. The experimental evidence is in agreement with the idea that enough depth is needed in order to perform the required mappings [1].

The key feature that has been introduced in RNNs is the sharing of a single model shared throughout different instants. This allows to use just few parameters and the sharing of information (in the shape of hidden units, outputs, etcetera...) between subsequent times.

Notice that the concept of parameter sharing has already been developed in CNNs' structure, but it is shallower. The reason is that the result is a sequence whose members depend on some neighboring elements of the input. By contrast the operation made by RNNs is stronger since it involves every previous time step and allows a very deep computational graph.

One other architectural difference between Feed Forward Neural Networks and Recurrent Networks is that the former is only able to map one input to one output, whereas the latter can map more input to one output (classification of a voice), or more outputs (translation), or even one input to more outputs (time series prediction).

These differences can be seen from a different point of view, involving the computational graph of the Network and in particular the way through which the hidden layer is obtained. Indeed it can be calculated as:

$$h(t) = g(u(t), u(t-1), ...; \theta) \tag{1.14}$$
$$= f(h(t-1), u(t); \theta) \tag{1.15}$$

The formulation (1.14) is the process that is made by Feed Forward Neural Networks to process sequences, taking the whole past sequence as input and producing the current state $h(t)$. On the other hand (1.15) is RNNs' one, and it is the mathematical implementation of the unfolded graph, like in figure (1.6). Clearly this notation introduces two major advantages:

1. Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states;

2. It is possible to use the same transition function f with the same parameters at every time step.

These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths, rather than needing to learn a separate model g_t for all possible time steps.

1.4.3 Back Propagation of RNNs

Consider a classic RNN, with input u and output o with size equal to t. Let's say that the label for this model is the sequence y. As usual a loss function is needed in order to compute the optimization step. In particular the total loss will be given by:

$$L(\{y(t), ..., y(1), u(t), ..., u(1)\}) = \sum_{\tau=1}^{t} L_\tau \tag{1.16}$$

that clearly involves involves the predicted outputs o and the vector y. Moreover the calculation of the gradient is expensive. In fact suppose, for example, that the model is a classifier, whose output is given by:

$$\begin{aligned} h(\tau) &= \tanh(Uu(\tau) + Wh(\tau - 1) + b) \\ o(\tau) &= Vh(\tau) + c \\ \hat{h}(\tau) &= softmax(o(\tau)) \end{aligned}$$

and the loss function is given by the negative log likelihood. Thus equation (1.16) becomes:

$$L(t) = \sum_{\tau=1}^{t} -p_{model}(y(\tau) \mid u(\tau), ..., u(1)) \tag{1.17}$$

where p_{model} is given by reading the entry corresponding to $y(t)$ given by the output of the model $\hat{y}(t)$. The gradient computation involves performing a forward propagation pass, in order to obtain the vector $\hat{y}(t)$ followed by a backward propagation pass, that computes the actual gradients. Then the computational cost of the back propagation is $O(t)$, as well as the memory cost and it can not be reduced as done before because of the sequential structure given to the model, since both procedures require the computation of the previous $(t-1)$ steps.

However there are some techniques that allows computing the gradients separately for different time steps. For example, as described in *Deep Learning* [1], when considering the *output-to-hidden* model. Since in supervised learning problems the real output is given, the gradient step for a fixed time τ does not require the knowledge of the previous ones, since in the back propagation step the model can be considered as:

$$\begin{aligned} h(\tau) &= \tanh(Uu(\tau) + Wy(\tau - 1) + b) \\ o(\tau) &= Vh(\tau) + c \end{aligned}$$

which differs from the computation that is actually done in the layer. However this separates the operation at time step τ from the results obtained at time $\tau - 1$. This technique is called *teacher forcing*. However it must be brought to attention the fact that these models are less powerful than the *hidden-to-hidden* ones.

Chapter 2

Theoretical Results

2.1 Universal Approximation Theorem

Initially Neural Networks have been thought as a collection of models built in order to replicate the behaviour of human brain. The question is: are these models reliable or they are just an heuristic set of algorithms? This section is written with the purpose of investigating the theoretical convergence properties of Neural Networks.

First of all a rigorous definition of squashing function is needed, since the first convergence results are associated with them:

Definition 2.1.1.
- $\mathscr{C}^r = \{f : \mathbb{R}^r \to \mathbb{R} \mid \text{f is continous}\}$
- $\mathscr{M}^r = \{f : \mathbb{R}^r \to \mathbb{R} \mid \text{f is Borel mesurable}\}$
- $\mathscr{B}^r$ is the Borel $\sigma-$algebra in $\mathbb{R}^r$

Definition 2.1.2. $\psi : \mathbb{R} \to [0,1]$ is called squashing function is it is monotone non decreasing and:

$$\begin{cases} \lim_{x\to-\infty} \psi(x) = 0 \\ \lim_{x\to\infty} \psi(x) = 1 \end{cases}$$

The class of squashing functions includes the Heaviside step function $psi(x) = \mathbb{1}_{x\geq\lambda}$ and the logistic sigmoid, for example.

The first significant result has been achieved by George Cybenco in 1988, [9]:

Theorem 2.1.3. *Let σ be a squashing function. Then finite sums of the form*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j^T x + b_j)$$

are dense in $\mathscr{C}(I^n)$, where $I^n = [0,1]^n$.

Independently from Cybenko, Kurt Hornik, Maxwell Stinchcombe and Halbert White in 1989, [7], proved a more general statement for the approximation properties of Neural Networks. Before getting into the theorem and its proof, some notation is needed.

Definition 2.1.4. A subset S of a metric space (X,ρ) is $\rho-$dense in $T \subset X$ if $\forall\epsilon > 0, t \in T, \exists s \in S$ such that $\rho(s,t) < \epsilon$.

Definition 2.1.5. A subset $S \in \mathscr{C}^r$ is said to be uniformly dense on compacts in $\mathscr{C}^r$ if $\forall K \subset\subset \mathbb{R}^r$, S is ρ_K-dense in $\mathscr{C}^r$, where ρ_K is defined by:

$$\rho_K(f,g) = \sup_{x\in K} \mid f(x) - g(x) \mid \qquad \forall f, g \in \mathscr{C}^r$$

Moreover a sequence of functions $\{f_n\}_n \in \mathbb{N}$ converges to a function f uniformly on compacts if $\forall K \subset\subset \mathbb{R}^r$, $\rho_K(f_n, f) \xrightarrow[n\to\infty]{} 0$.

Definition 2.1.6. Let μ be a probability measure on $(\mathbb{R}^r, \mathscr{B}^r)$. Then $f, g \in \mathscr{M}^r$ are $\mu-$equivalent if $\mu(\{x : f(x) = g(x)\}) = 1$.

This definition is given by convenience. Actually all the following results can be equivalently stated for any finite measure μ, that is $\mu(\mathbb{R}^r) = \alpha < \infty$.

Definition 2.1.7. Any probability measure μ defines a metrics $\rho_\mu : \mathscr{M}^r \times \mathscr{M}^r \to \mathbb{R}$. In particular:

$$\rho_\mu(f, g) = \inf\{\epsilon > 0 : \mu(\{x :\mid f(x) - g(x) \mid > \epsilon) < \epsilon\}$$

That is $\rho_\mu(f, g)$ is small if and only if the measure of the set in which f and g differs significantly is small. In particular $\rho_\mu(f, f) = 0$. The final step is represented by constructing the function spaces that are dense in $\mathscr{C}^r$ and $\mathscr{M}^r$. This means that any function $f \in {}^r$ or $f \in \mathscr{M}^r$ can be approximated for every rate of accuracy by a function in that space.

Definition 2.1.8. Given a Borel measurable function $F : \mathbb{R}^r \to \mathbb{R}$

- $\Sigma^r(F) = \{f : \mathbb{R}^r \to \mathbb{R} \mid f(x) = \sum_{i=1}^{N} \beta_i F(A_i(x)),\ \beta_i \in \mathbb{R}, A_i \text{ affine function}\}$
- $\Sigma\Pi^r(F) = \{f : \mathbb{R}^r \to \mathbb{R} \mid f(x) = \sum_{i=1}^{N} \beta_i \prod_{k=1}^{l_i} F(A_{i,k}(x)),\ \beta_i \in \mathbb{R}, A_{i,k} \text{ affine function}\}$

Now the main result can be stated:

Theorem 2.1.9. *Let σ be a squashing function. Then $\Sigma^r(\sigma)$ is uniformly dense on compacts in $\mathscr{C}^r$ and it is ρ_μ dense in $\mathscr{M}^r$.*

This result means that every single-layer perceptron can approximate arbitrarily well any measurable function and it can be said *universal approximator*. Before getting into the proof of theorem (2.1.9), a series of preliminary results are needed.

Theorem 2.1.10. *For any $F : \mathbb{R} \to \mathbb{R}$ continuous and non constant function, $\Sigma\Pi^r(F)$ is uniformly dense on compacts in $\mathscr{C}^r$.*

Proof. The reasoning makes use of an important result of functional analysis, that is the Stone-Weiestrass theorem. Thus, first of all, let's recall its statement. Suppose $\mathscr{A}$ is an algebra of real continuous functions on a compact set K, that means that it is closed under addition, multiplication and scalar multiplication. Moreover suppose that $\mathscr{A}$ separates points on K (that is, $\forall x, y \in K, x \neq y, \exists f \in \mathscr{A}$ such that $f(x) \neq f(y)$) and $\mathscr{A}$ vanishes at no point of K, meaning that there exists a function $f \in \mathscr{A}$ such that $f(x) \neq 0\ \forall x \in K$. Then the uniform closure of $\mathscr{A}$ consists of all real continuous functions on K. Equivalently $\mathscr{A}$ is ρ_K-dense in the space of $\mathscr{C}(K)$.
Therefore, if $K \subset\subset \mathbb{R}^r$, proving that $\Sigma\Pi^r(F)$ satisfies the hypothesis of Stone-Weiestrass theorem is enough to prove the result.

- $K \subset\subset \mathbb{R}^r$ is clearly an algebra of continuous functions on K;

- suppose $x, y \in K, x \neq y$. Since F is not constant, there exists $a, b \in \mathbb{R}$ such that $F(a) \neq F(b)$. It is then sufficient to find an affine transformation $A(\cdot)$ satisfying $A(x) = a, A(y) = b$. This ensures $K \subset\subset \mathbb{R}^r$ separates points on K;
- again, using the fact that F is not constant, there exists a point $c \in \mathbb{R}$ such that $F(c) \neq 0$. Then, picking $A(x) = 0^T x + b$ shows that $K \subset\subset \mathbb{R}^r$ vanishes at no point of K.

Since K is arbitrary, the proof is obtained through Stone-Weiestrass theorem. □

Lemma 2.1.11. *The following are equivalent:*

(a) $\rho_\mu(f_n, f) \xrightarrow{n\to\infty} 0$*;*

(b) $\forall \epsilon > 0$, $\mu(\{x :\mid f_n(x) - f(x) \mid > \epsilon\}) \xrightarrow{n\to\infty} 0$*;*

(c) $\int \min\{\mid f_n(x) - f(x) \mid, 1\} d\mu(x) \xrightarrow{n\to\infty} 0$.

Proof. *(a)⇔(b)* is trivial;
(b)⇒(c): by hypothesis, there exists $\hat{n}$ such that $\forall n \geq \hat{n}$, $\mu(R) < \frac{\epsilon}{2}$, where $R = \{\mid f_n(x) - f(x) \mid > \epsilon\}$. Then, $\forall n \geq \hat{n}$,

$$\int_{\mathbb{R}^r} \min\{\mid f_n(x) - f(x) \mid, 1\} d\mu(x) < \int_{\mathbb{R}^r \setminus R} d\mu(x) + \int_R \frac{\epsilon}{2} d\mu(x) < \epsilon \tag{2.1}$$

(c)⇒(b): suppose *(b)* is not valid. Then there exists $\epsilon > 0$ such that $\mu(R) > \epsilon$, where $R = \{\mid f_n(x) - f(x) \mid > \epsilon\}$. This means that:

$$\int_{\mathbb{R}^r} \min\{\mid f_n(x) - f(x) \mid, 1\} d\mu(x) > \int_R \epsilon d\mu(x) > \epsilon^2 > 0 \tag{2.2}$$

Then *(c)* is not verified as well. □

Lemma 2.1.12. *If* $\{f_n\}_{n\in\mathbb{N}}$ *converges uniformly on compacts to* f*, then* $\rho_\mu(f_n, f) \xrightarrow{n\to\infty} 0$.

Proof. From lemma (2.1.11), it is enough to show that

$$\int \min\{\mid f_n(x) - f(x) \mid, 1\} d\mu(x) \xrightarrow{n\to\infty} 0$$

Consider $\epsilon > 0$ and suppose $\mu(\mathbb{R}^r) = 1$. Since $\mathbb{R}^r$ is a locally compact metric space, μ is a regular measure. Thus there exists $K \subset\subset \mathbb{R}^r$ with the property that $\mu(K) > 1 - \frac{\epsilon}{2}$. Since f_n converges uniformly on every compact to f, there exists $\hat{n} \in \mathbb{N}$ such that $\forall n \geq \hat{n}$, $\sup_{x\in K} \mid f_n(x) - f(x) \mid < \frac{\epsilon}{2}$. Now:

$$\int \min\{\mid f_n(x) - f(x) \mid, 1\} d\mu(x) = \int_{\mathbb{R}^r \setminus K} d\mu(x) + \int_R \frac{\epsilon}{2} d\mu(x) < \epsilon \tag{2.3}$$

□

Lemma 2.1.13. *For any finite measure μ, $\mathscr{C}^r$ is $\rho_\mu-$dense in $\mathscr{M}^r$.*

Proof. The aim of the following reasoning is to prove that, given any $f \in \mathscr{M}^r$ and $\epsilon > 0$, there exists $g \in \mathscr{C}^r$ such that $\rho_\mu(f, g) < \epsilon$.
For sufficiently large M, $\int \min\{| f\mathbb{1}_{\{|f|<M\}} - f |, 1\}d\mu < \frac{\epsilon}{2}$. Moreover there exists a continuous function g such that $\int \min\{| f\mathbb{1}_{\{|f|<M\}} - g |, 1\}d\mu < \frac{\epsilon}{2}$. Then the result is obtained by applying triangular inequality. □

Theorem 2.1.14. *For any $F : \mathbb{R} \to \mathbb{R}$ continuous and non constant function, and for any μ finite measure on $(\mathbb{R}^r, \mathscr{B}^r)$, $\Sigma\Pi^r(F)$ is $\rho_\mu-$dense on compacts in $\mathscr{M}^r$.*

Proof. From theorem (2.1.10) it follows that for any given continuous and non constant function F, $\Sigma\Pi^r(F)$ is uniformly dense on compacts in $\mathscr{C}^r$. Then lemma (2.1.12) shows that $\Sigma\Pi^r(F)$ is $\rho_\mu-$dense in $\mathscr{C}^r$. Now, since $\mathscr{C}^r$ is $\rho_\mu-$dense in $\mathscr{M}^r$ by lemma (2.1.13), applying triangular inequality brings to the result. □

Lemma 2.1.15. *Let F be a continuous squashing function and ψ an arbitrary squashing function. Then, $\forall\epsilon > 0$ there exists $H_\epsilon \in \Sigma^r(\psi)$ such that*

$$\sup_{\lambda\in\mathbb{R}} | F(\lambda) - H_\epsilon(\lambda) | < \epsilon$$

Proof. The purpose is to find such an $H_\epsilon(x) = \sum_{j=1}^{Q-1} \beta_j\psi(A_j(x))$. These Q, β_j and A_j has to be found.

Pick an arbitrary $\epsilon > 0$ and suppose, without loss of generality, that $\epsilon < 1$. Now pick Q such that $\frac{1}{Q} < \frac{\epsilon}{3}$. Since ψ is a squashing function, there exists $M > 0$ such that $\psi(-M) < \frac{\epsilon}{2Q}$ and $\psi(M) > 1 - \frac{\epsilon}{2Q}$. Finally, for $j \in \{1, ..., Q-1\}$, let's define

$$r_j = sup\{\lambda : F(\lambda) = \frac{j}{Q}\}$$

and $r_Q = sup\{\lambda : F(\lambda) = 1 - \frac{1}{2Q}$. It is possible to define such r_js since F is continuous. Moreover, for any $r < s$ it is possible to find an affine transformation $A_{r,s}$ such that $A_{r,s}(r) = M$ and $A_{r,s}(s) = -M$.

Finally it is possible to build H_ϵ:

- $\beta_j = \frac{1}{Q}$
- $A_j = A_{r_j,r_{j+1}}$

Let's prove the desired inequality. First of all consider $x \in (r_i, r_{i+1}]$. In this case:

$$H_\epsilon(x) = \sum_{j=1}^{Q-1} \frac{1}{Q}\psi(A_{r_j,r_{j+1}}(x)) \tag{2.4}$$

$$= \sum_{j=1}^{i-1} \frac{1}{Q}\psi(A_{r_j,r_{j+1}}(x)) + \frac{1}{Q}\psi(A_{r_i,r_{i+1}}(x)) + \sum_{j=i+1}^{Q-1} \frac{1}{Q}\psi(A_{r_j,r_{j+1}}(x)) \tag{2.5}$$

Thus this quantity belongs to the interval:

$$\Big(\frac{(i-1)}{Q}(1 - \frac{\epsilon}{2Q}) + \frac{\epsilon}{2Q^2}, \frac{i-1}{Q} + \frac{1}{Q}(1 - \frac{\epsilon}{2Q}) + \frac{\epsilon}{2Q^2}(Q - i - 1)\Big)$$

That can be rewritten as:

$$\Big(\frac{i-1}{Q}+\frac{\epsilon}{2Q^2}(2-i),\frac{i}{Q}-\frac{\epsilon}{2Q^2}+\frac{\epsilon}{2Q^2}(Q-i-1)\Big)$$

and, since the following are surely true:

$$\epsilon\frac{2-i}{2Q^2}>\epsilon\frac{-Q}{2Q^2}>-\frac{1}{Q} \tag{2.6}$$

$$-\frac{\epsilon}{2Q^2}+\frac{\epsilon}{2Q^2}(Q-i-1)<\frac{\epsilon}{2Q}<\frac{1}{Q} \tag{2.7}$$

it is possible to say that:

$$H_\epsilon(x)\in\Big(\frac{i-2}{Q},\frac{i+1}{Q}\Big)$$

Hence $\mid H_\epsilon(x)-F(x)\mid<\frac{3}{Q}<1$.

If $x\in(-\infty,r_1]$, then $F(x)\in[0,\frac{1}{Q}$. And in this case:

$$H_\epsilon(x)<\frac{\epsilon}{2Q^2}Q<\frac{\epsilon}{2Q}<\frac{1}{2Q} \tag{2.8}$$

and the inequality is easily verified.

The final case is $x\in(r_Q,\infty)$, but it is trivial. □

Theorem 2.1.16. *For any squashing function ψ, and for any μ finite measure on $(\mathbb{R}^r,\mathscr{B}^r)$, $\Sigma\Pi^r(\psi)$ is dense on compacts in $\mathscr{C}^r$ and it is $\rho_\mu-$dense on compacts in $\mathscr{M}^r$.*

Proof. By theorem (2.1.14), it follows that for every F continuous squashing function, $\Sigma\Pi^r(F)$ is uniformly dense on compacts in $\mathscr{C}^r$ and $\rho_\mu-$dense on comacts in $\mathscr{M}^r$. Thus, given any squashing function ψ, by lemma (2.1.12) it is enough to show that $\Sigma\Pi^r(\psi)$ is uniformly dense on compacts in $\Sigma\Pi^r(F)$ for some continuous squashing function F. To this purpose, it is sufficient to show that every function of the form $\prod_{k=1}^{l}F(A_k(\cdot))$ can be approximated by an element of $\Sigma\Pi^r(\psi)$.

Hence, given $\epsilon>0$, there exists $\delta>0$ such that

$$\mid a_k-b_k\mid<\delta\quad 0\le a_k,b_k\le 1\quad\forall k\in\{1,...,l\}\Rightarrow\mid\prod_{k=1}^{l}a_k-\prod_{k=1}^{l}b_k\mid<\epsilon$$

Thus, by lemma (2.1.15) there exists a function $H_\delta(\cdot)=\sum_{l=1}^{r}\beta_l\psi(A_\delta^l(\cdot))$ such that:

$$\sup_{x\in\mathbb{R}}\mid F(x)-H_\delta(x)\mid<\delta$$

and then:

$$\sum_{x\in\mathbb{R}^r}\mid\prod_{k=1}^{l}F(A_k(x))-\prod_{k=1}^{l}H_\delta(A_k(x))\mid<\epsilon$$

□

The proof of theorem (2.1.9) is derived from the following three lemmas, whose proof is omitted (it can be found in [6]).

Lemma 2.1.17. *For any squashing function ψ, every $\epsilon > 0$ and every $M > 0$ there is a function $cos_{Mx} \in \Sigma\Pi^r(\psi)$ such that:*

$$\sup_{\lambda \in [-M,M]} \mid cos_{Mx}(\lambda) - cos(\lambda) \mid < \epsilon$$

Lemma 2.1.18. *Let $g(\cdot) = \sum_{i=1}^{N} \beta_i cos(A_i(\cdot))$, with A_i affine function. For arbitrary squashing function ψ, for arbitrary compact $K \subset\subset \mathbb{R}^r$ and for arbitrary $\epsilon > 0$ there is an $f \in \Sigma^r(\psi)$ such that:*

$$\sup_{x \in K} \mid g(x) - f(x) \mid < \epsilon$$

Lemma 2.1.19. *For every squashing function ψ, $\Sigma^r(\psi)$ is uniformly dense on compacts in $\mathscr{C}^r$.*

Then the proof of theorem (2.1.9) is easy:

Proof. By lemma (2.1.19), $\Sigma^r(\psi)$ is uniformly densse on compacts in $\mathscr{C}^r$. Thus lemma (2.1.12) implies that $\Sigma^r(\psi)$ is $\rho_\mu-$dense in $\mathscr{C}^r$. Triangular inequality and lemma (2.1.13) imply that $\Sigma^r(\psi)$ is $\rho_\mu-$dense in $\mathscr{M}^r$. □

In 2015 the same result was proved for ReLu activation function, [7].

These theorems show that Single Layer Perceptron can approximate all functions that are relevant in applications. However no theoretical result has been found to determine how many units are needed to reach a certain accuracy yet. Moreover, experimental results show that adding layers is more efficient than adding units on existing layers. This intuition is justified by heuristic, since every layer allows to represent its input in a new shape, as encoders do, like it has been seen in chapter 1.1 and, in particular, in figure (1.4). Thus, adding more layers intuitively means that the input's representation is way more suitable to make the input (encoded)-output relationship simpler.

From the application point of view each layer outputs a series of *features*, thus the more layers are considered, the more sophisticated features can be attained, allowing to have a smoother transition from the input to the output.

2.2 On The Convergence and Reliability of Adaptive Algorithms

Adaptive algorithms are the most used, and, maybe abused, optimization techniques for Neural Networks. As it has already been seen this is due to two factors: first of all, more sophisticated optimization algorithms, like Gauss-Newton or Levenberg-Marquardt, are too expensive as far as computation and memory are concerned. For example, the complexity of those algorithms is about $O(N^3)$, where N stands for the number of parameters, and Neural Networks are built with a lot of parameters.

On the other hand, simple Gradient Descent is a slow optimization algorithm and it is subject to many issues described in section 1.3.1. However, if Adaptive algorithms do not converge, it is quite useless to train Neural Networks with them. Thus the question is: what are the convergence properties of Adaptive algorithms?

There are not so many theoretical results to answer with certainty to this question, but in the following few lines, the latest researches are going to be reported.

First of all, it has not be proved that any Adaptive method is convergent to an optimal solution. On the contrary, in [10] it has been showed the following result:

Theorem 2.2.1. *For any constant $\beta_1, \beta_2 \in [0, 1)$ such that $\beta_1 < \sqrt{\beta_2}$ there is a stochastic convex optimization problem for which ADAM does not converge to the optimal solution.*

Notice that the values of β_1 and β_2 are compatible to the value suggested by the creators of Adam, as seen in (1.6).

Moreover, recent researches showed that Adaptive algorithms are not always better than Gradient Descent based ones. In particular it has been experimentally seen that sometimes the accuracy achieved through Stochastic Gradient Descent is higher than what is obtained through Adam or AdaGrad, or, that, even if the training accuracy achieved by Adam or AdaGrad are higher than SGD, the solution found by the former generalize worse than the one found with the latter, [11],[13].

These results are not surprising, since there is no reference that can prove the convergence of any adaptive algorithms. For example, as far as Adam is concerned, the validity of its proof has been argued recently. The main problem is that the convergence can be proved only in a very particular setting and provided that a non proved conjecture is valid, [14].

To conclude this section, an intuitive mathematical based comparison between the two algorithms is made. In particular two main elements have to be analysed: computational and memory cost and adaptive learning rates. As far as the latter is concerned, it has the undoubted advantage that it speeds up the learning and avoids a lot of *vanishing* gradient problems. Nevertheless there are some drawbacks. For example in the neighborhood of a stationary point the value of the gradient of the loss function is quite small, but the adaptive learning rate makes the updating step higher. On the contrary gradient descent let it to be small, as it should be. For a certain point of view this is not good, since it does not allow to move from a stationary point with an high value for the loss function. On the other hand in many cases Adam might oscillate around the optimal stationary point, when many iterations are with a small gradient. This is due to the fact that the moving average of the squared partial derivative is smaller and smaller, thus at a certain iteration the algorithm is forced to make bigger steps than the gradient is, bringing the loss function to an high value. This has been seen experimentally in many occasions during this work.

As far as the cost is concerned, Gradient Descent is surely less demanding. If N is the number of parameters, for each step, apart from computing the gradient, it requires only N multiplications and N summations. On the other hand Adam algorithm requires to compute the moving average of the squared gradient ($O(N + N)$ operations), and the moving average of the gradient as well ($O(N + N + N)$). Moreover Adam it requires also resources in memory, since it has to store the moving

average of past gradients (N elements) and of the squared gradients (one more element).

It must be said that this is not such a big difference, since both the algorithm have computational cost linear in N.

2.3 Empirical Advices

The purpose of this section is to analyse some simple situation that brings to non convergence algorithms, in order to obtain some advice that is universally valid.

First of all, recall that all the optimization steps are made through a gradient descent based or an adaptive method. Both these classes are made of iterative algorithms that need the computation of the gradient of a loss function for each epoch. Thus, given a certain loss function L it is important to recall how its derivatives with respect to the parameters are calculated in Neural Networks, that is Back Propagation, as explained in 1.3.2:

$$\begin{cases} \frac{\partial L}{\partial u_j^{(n)}} = \sum_i \frac{\partial L}{\partial u_i^{(n+1)}} \frac{\partial u_i^{(n+1)}}{\partial u_j^{(n)}} \\ \frac{\partial L}{\partial w_j^{(n)}} = \sum_i \frac{\partial L}{\partial u_i^{(n+1)}} \frac{\partial u_i^{(n+1)}}{\partial w_j^{(n)}} \end{cases} \tag{2.9}$$

Since a general equation for the forward computation of the $j-$th unit of the $(n+1)-$th layer is:

$$u_j^{(n+1)} = \sigma(\sum_i u_i^{(n)} w_{i,j}^{(n)} + b_j^{(n)}) \tag{2.10}$$

The following results can be easily deduced:

$$\frac{\partial u_j^{(n+1)}}{\partial u_i^{(n)}} = \sigma_i'(...) w_{i,j}^{(n)} \tag{2.11}$$

$$\frac{\partial L}{\partial u_i^{(n)}} = \sum_j \frac{\partial L}{\partial u_j^{(n+1)}} \frac{\partial u_j^{(n+1)}}{\partial u_i^{(n)}} = \sum_j \frac{\partial L}{\partial u_j^{(n+1)}} \sigma_j'(...) w_{i,j}^{(n)} \tag{2.12}$$

$$\frac{\partial L}{\partial w_{i,j}^{(n)}} = \sum_s \frac{\partial L}{\partial u_s^{(n+1)}} \frac{\partial u_s^{(n+1)}}{\partial w_{i,j}^{(n)}} = \frac{\partial L}{\partial u_j^{(n+1)}} \sigma_j'(...) u_i^{(n)} \tag{2.13}$$

$$\frac{\partial L}{\partial b_i^{(n)}} = \sum_j \frac{\partial L}{\partial u_j^{(n+1)}} \frac{\partial u_j^{(n+1)}}{\partial b_i^{(n)}} = \frac{\partial L}{\partial u_i^{(n+1)}} \sigma_i'(...) \tag{2.14}$$

where, with an abuse of notation, $\sigma_i'(...)$ stands for the derivative of σ calculated in the i-th affine transformation, that is $\sum_i w_{i,j}^{(n)} u_i^{(n)} + b_i^{(n)}$. Notice that these equations make sense only if the n is does not denote the last layer, since in this case there is no $(n+1)$-th layer.

Once the gradient has been computed, the update of the parameters follows by the definition of the optimization algorithm. In particular, as an example element of gradient descent methods, the classic (stochastic) gradient descent is considered:

$$\theta_{k+1} = \theta_k - \eta \Delta_\theta L(\theta_k) \tag{2.15}$$

whereas Adam is taken as one exponent of the adaptive algorithms' class:

$$\begin{cases} v_{k+1} = \beta_1 v_k + (1-\beta_1)\nabla_\theta L(\theta_k) \\ m_{k+1,i} = \beta_2 m_{k,i} + (1-\beta_2)\left(\frac{\partial}{\partial\theta_i} L(\theta_k)\right)^2 \\ \hat{v}_{k+1} = \frac{v_{k+1}}{1-\beta_1^{k+1}} \\ \hat{m}_{k+1} = \frac{m_{k+1}}{1-\beta_2^{k+1}} \\ \theta_{k+1,i} = \theta_{k,i} - \frac{\eta}{\sqrt{\hat{m}_{k+1,i}}+\epsilon}\hat{v}_{k+1,i} \end{cases} \tag{2.16}$$

Notice that in this case the update of parameters is not given by following the opposite of the gradient direction any more, but every component of the update direction is the result of the combination of an exponential moving average of the gradients computed during all previous iterations whose components are individually scaled by a quantity that can be thought as the inverse of the square root of an exponential moving average of the squared partial derivative of the loss function with respect to the corresponding parameter. However a small coefficient ϵ is inserted in order to prevent the denominator to be too small.

2.3.1 Weights and Bias Initialization

The first issue bringing to a non convergent model is not updating the weights. This means that the dependency of the output on the input is not adjusted during the training. In particular, not updating a certain parameter $w_{i,j}^{(n)}$ can be caused by three factors, as it can easily be deduced by (2.13): a zero value either for $\frac{\partial L}{\partial u_j^{(n+1)}}$ or $\sigma'(\sum_i u_i^{(n)} w_{i,j}^{(n)} + b_j^{(n)})$ or $u_i^{(n)}$. On the other hand, not updating a bias term $b_i^{(n)}$ is due to $\frac{\partial L}{\partial u_i^{(n+1)}}$ or $\sigma'(\sum_s u_s^{(n)} w_{s,i}^{(n)} + b_i^{(n)})$ only.

If the activation function has the property $\sigma(0) = 0$, then initializing weights and bias to 0 is the most dangerous thing to do. In fact in this case all units of every layer is zero, and, as a consequence, the partial derivatives with respect to every weight is zero as well. Thus neither weights are going to be updated, because of (2.13), nor biases of every but the last layer, since $\frac{\partial L}{\partial u_i^{(n+1)}}$ is zero, by (2.12) and the fact that all weights are zero. The only element that is going to be updated is the vector of biases of the last layer, since neither its derivative depends on any unit, nor the derivative of the loss function with respect to the output units is null. Thus the output is going to be constant (it is going to be the last bias). This argument may suggest the use of an activation function that has not the property $\sigma(0) = 0$. However this is not a good choice, as it is going to be explained in 2.3.3.

Now let's look at Adam algorithm (2.16): without the term ϵ the numerator and the denominator can be thought as with the same order of magnitude, then the length of the updating step follows the value η. However ϵ is necessary in order to avoid the nullification of the denominator, and since it is set by default to the value of 10^{-8}, the denominator is always greater than 10^{-8}. This means that if the magnitude of the numerator is inferior, the updating step can be significantly smaller than η.

Example 2.3.1. Consider a generic Multi Layer Perceptron made of N layers, with k_i units each, whose weights and biases are initialized to a fixed value μ. Suppose there is no parameter sharing. The idea is to get an intuition about the magnitude of the derivatives based on weights and biases. To this purpose, suppose that the module of each partial derivative of the loss functions with respect to the output units can be upper bounded by a finite value L and the derivative of the activation function can be limited up to D. Then:

$$\frac{\partial L}{\partial u_i^{(N)}} \leq L \tag{N}$$

$$\begin{cases} \frac{\partial L}{\partial u_i^{(N-1)}} = \sum_j \frac{\partial L}{\partial u_j^{(N)}} \sigma_j'(...) w_{i,j}^{(N-1)} \leq k_{N-1} L D \mu \\ \frac{\partial L}{\partial w_{i,j}^{(N-1)}} = \frac{\partial L}{\partial u_j^{(N)}} \sigma_j'(...) u_i^{(N-1)} \leq L D u_i^{(N-1)} \\ \frac{\partial L}{\partial b_i^{(N-1)}} = \frac{\partial L}{\partial u_i^{(N)}} \sigma_j'(...) \leq L D \end{cases} \tag{N-1}$$

$$\begin{cases} \frac{\partial L}{\partial u_i^{(N-2)}} = \sum_j \frac{\partial L}{\partial u_j^{(N-1)}} \sigma_j'(...) w_{i,j}^{(N-2)} \leq k_{N-2} k_{N-1} L D^2 \mu \\ \frac{\partial L}{\partial w_{i,j}^{(N-2)}} = \frac{\partial L}{\partial u_j^{(N-1)}} \sigma_j'(...) u_i^{(N-2)} \leq k_{N-1} L D^2 u_i^{(N-2)} \mu \\ \frac{\partial L}{\partial b_i^{(N-2)}} = \frac{\partial L}{\partial u_i^{(N-1)}} \sigma_j'(...) \leq L D^2 \mu \end{cases} \tag{N-2}$$

And so on. Then $\frac{\partial L}{\partial w^{(N-n)}} = O((\prod_{1 \leq i \leq n} k_{N-i}) \mu^{n-1} u^{(N-n)})$. On the other hand $\frac{\partial L}{\partial b^{(N-n)}}$ is $O((\prod_{1 \leq i \leq n} k_{N-i}) \mu^{n-1})$.

This example is quite exhaustive, however it must be said that the most general case deals with both parameter sharing and non constant values for the initialization of the parameters.

Thus if the magnitude of the weights and biases is very small, deep networks are more likely to have very small gradients for the first layers. In particular, supposing that the input and the output are normalized, smart initialization values should be smaller than 1, otherwise the predicted output magnitude would be different from the actual labels' one. But they should not be too small, since they may cause the convergence problem just analysed. One more problem of small value for the parameters is that if the learning rate is significantly bigger and the activation function is centered in zero, one optimization step would bring to biases much bigger than weights, since the additional term u in (N-1), (N-2) and all previous layers bring the partial derivative of the weights to be much lower than the one of the biases. If the activation function is ReLu this could bring to a point in the parameter space that can not be updated with an optimization step.

One other issue is due to high magnitude parameters. Forward propagation brings deep layers to high value units and this gives rise to the problem of saturation of sigmoids or dead ReLu, but this will be investigated with more details in 2.3.3.

2.3.2 Preprocessing the Data

Neural Networks can be considered as made of two main elements: the dataset and the model itself. Thus, one may wonder whether there exists a Neural Network

capable to catch efficiently the relationship in any input-output couple or not. Otherwise, is it possible to represent a given dataset in a different shape in order to attain better approximation properties?.

A key role in convergence properties of Neural Networks is played by its input (and eventually its labels), [8]. In particular, it is a standard practice to rescale the input and the output in order to have zero mean elements. The reason why this is recommended is quite intuitive: suppose that all the inputs are positive. Thus, by equation (2.13), if a Multi Layer Perceptron is considered, all the weights concurring on the same unit can only all decrease or all increase together. Thus, if the optimal value for the vector can be reached through a direction having components of different sign, it can be reached only by zigzagging. Thus, having input with different sign elements is preferred and this concept is generalized in having zero mean input.

One more commonly used practice is to normalize the input. In particular it is convenient to give all the input the same standard deviation, that should not be too high. This is due to the fact that, if all the weights are initialized with the same process as it has been seen in section 2.3.1, different magnitude in the input could bring to output of different shape, giving more importance to some input's component with respect to others that in the computation of the loss function. Moreover the magnitude of the dataset is meaningful as well. In particular high or small valued inputs or labels could bring to high or small valued parameters, as it can be seen in equations (2.13).

The solution proposed by Yann LeCun in [8] is to set the standard deviation to 1, apart from the case in which it is assumed that some component is more significant than the others. In general, the more a component is significant, the higher standard deviation it should have, with respect to the others components.

Thus it is widely accepted that compacting all features, input and output in order to make the most part of them range in a certain limited interval is a good strategy. *Normalization* and *Standardization* are the most used ones. The former consists of compacting all features in a certain interval that usually is $[0, 1]$. In this case the formula used to transform the data is:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2.17}$$

The former is used to make the dataset with 0 mean and standard deviation unitary. In particular the transformation is:

$$x_{std} = \frac{x - \mu_x}{\sigma_x} \tag{2.18}$$

where μ_x and σ_x are the mean and the standard deviation of the feature among all the dataset.

2.3.3 Activation Functions

From the first chapter it is clear that activation functions are one of the most important part of Neural Networks, since they give non linearity and allow stacking more than one layer in the same structure without redundancy. However, it is not really

clear whether to use a certain non linear function or one other. Thus the purpose of this section is to analyse these functions.

The most intuitive properties an activation function should have are clearly non linearity and differentiability, because it has to be derived in order to compute Back Propagation. Actually differentiability has been expanded to functions with a finite number of non differentiability, by choosing default values for non regular points. For example ReLu is not differentiable in 0, as it can be easily seen in figure (2.1), but it is set, by default $f'(0) = 0$.

Moreover there is one other crucial observation related to the derivatives of a potential activation function: *saturation problem*. All Gradient Descent based optimization algorithms, like Adam or Gradient Descent itself, update parameters according to the value of the gradient. However the derivative of the functions that are usually considered as activations is very small or even zero for large portions of the real domain. This is due to the fact that, in general, using functions with limited derivative is necessary, because if it does not happen, the gradient might be very big, making the optimization step not stable. The same could happen if the function is not limited, because of equation (2.13). However, using limited functions brings the derivatives to be very small in a large portion of the domain. This can be noticed in figure (2.1), in the black and the blue lines, that represent the derivative of two popular sigmoid functions. Equivalently there exists an infinite region in the parameters' space in which gradient is too small to get a significant update of the parameter considered. Furthermore since the value of a partial derivative with respect to a particular unit plays a key role in computing the derivative of all the parameters that are connected with that unit, as can be understood by (2.11), this "small derivative" makes impossible to update a big part of the weights.

One related problem is that it is preferred to use functions such that $\sigma(0) = 0$. To understand why, let's take logistic sigmoid as an example. By figure (1.5) it can be deduced that $\sigma(x) > x$ for x near the origin. Thus all those inputs are biased toward an higher value. Combining many units and many layers would bring the Network to have very high magnitude units, making the model unable to update significantly the parameters, because of vanishing gradient problems.

Considering these observations and that it is usual to have both weights and inputs with 0 mean, as described in sections 2.3.1 and 2.3.2, it is preferable to use function similar to the identity functions near the origin. In particular it is required that

$$\sigma(0) = 0 \tag{2.19}$$

$$\sigma'(0) = 1 \tag{2.20}$$

As far as the three activation functions that have been considered until now, the only one that observe any condition is the hyperbolic tangent. Nevertheless experimental results showed that ReLu achieves much better results as an activation function, [**?**]. It could seem quite strange, since some of the criteria above are not respected. In particular it is non differentiable and non limited. Moreover its derivative is exactly zero for half of its domain. On the other hand there are also many advantages given by using ReLu. The first and simplest reason is that it has a much easier representation than many other functions and its derivative does as well. It is quite

intuitive, since:

$$ReLu(x) = \begin{cases} x & if\ x > 0 \\ 0 & if\ x \leq 0 \end{cases} \qquad ReLu'(x) = \begin{cases} 1 & if\ x > 0 \\ 0 & if\ x \leq 0 \end{cases} \tag{2.21}$$

For example, hyperbolic tangent is much more complex:

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \qquad tanh'(x) = \left(\frac{2e^{-x}}{1 + e^{-2x}}\right)^2 \tag{2.22}$$

One more advantage of ReLu is related to the vanishing gradient problem. By looking at figure (2.1) it can be noticed that the derivative of many of the most popular activation functions is near zero in almost every point. On the contrary ReLu derivative is zero *just* in half of its domain. Pay attention: this does not mean that this problem does not exist. As a matter of fact it could even be more dangerous, since in the other half of the domain the derivative is zero, causing the so called *dead ReLu* issue. For this reason attention must be paid on how many layers are given ReLu activation function, since the more of this kind of layers there are, the higher the risk of having zero gradient is.

Finally the last e maybe the most important incentive to use this function is that it allows a sparse representation. In particular if the parameters of the Neural Network are initialized randomly, about 50% of the units having ReLu is set to zero. This allows to have a much less expensive model, computationally speaking, but it is not the only advantage. It also brings to a lower risk of overfitting, due to the fact that it implicitly drops all the non significant components of the Network.

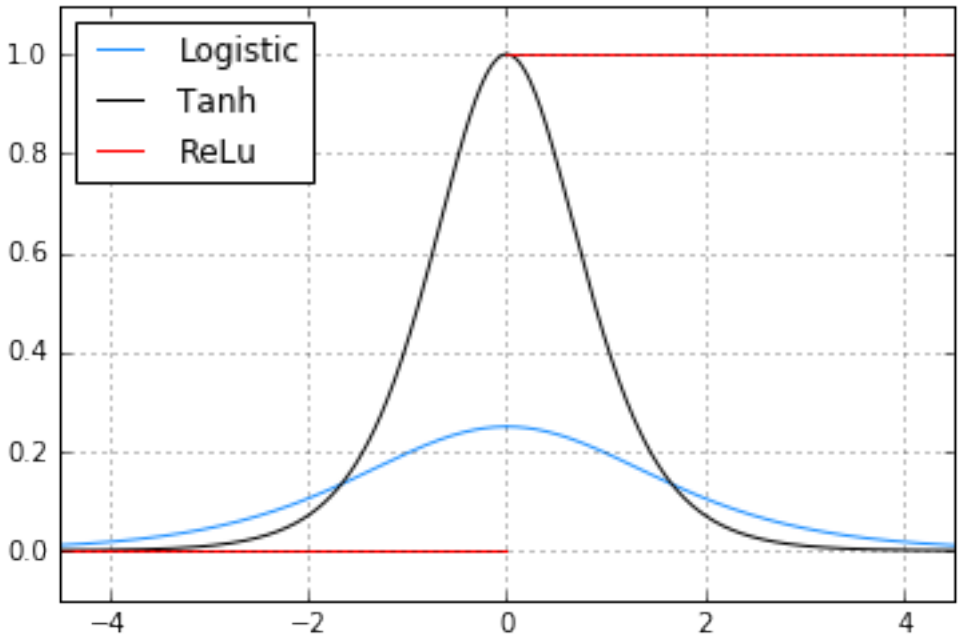

Figure 2.1

2.4 An applicative example

One of the most classical examples one may encounter when approaching Neural Networks is the so called *MNIST 10*. The dataset is made of black and white images representing handwritten digits and the goal is to build a Network that is able to associate the correct number to as many images as possible.

This experiment consists of comparing the results achieved with the same model, but with different initialization values of the parameters. In particular five scenarios have been tested. In all these cases the parameters have been initialized through a truncated normal distribution with 0 mean, with different variance: 0.05, 0.01, 0.0001, 10^{-6}, 10^{-8}.

The input: since these images are made of 28×28 pixels each, the input is considered as a 28×28 matrix, whose components are in the interval $[0, 1]$ and they represent the greyscale value of the corresponding pixels.

The architecture of the model: in this case it is not possible to give a mathematical meaning to the process that brings to the decision of which digit is represented in an image. Many different examples of architectures can be found in the web. The structure that has been considered in this case is made of two convolutional layers and two fully connected ones. In particular:

$$\begin{cases} \hat{C}_1 = ReLu[(U * W_1) + b_1] \\ C_1 = P(\hat{C}_1) \\ \hat{C}_2 = ReLu[(C_1 * W_2) + b_2] \\ C_2 = P(\hat{C}_2) \\ F = \text{vectorization of } C_2 \\ H_1 = ReLu[W_3 F + b_3] \\ O = W_4 H_1 + b_4 \end{cases} , \quad \begin{cases} W_1 \in \mathbb{R}^{5\times5\times1\times16} \\ W_2 \in \mathbb{R}^{5\times5\times16\times36} \\ W_3 \in \mathbb{R}^{1764\times128} \\ W_4 \in \mathbb{R}^{128\times10} \\ b_1 \in \mathbb{R}^{16} \\ b_2 \in \mathbb{R}^{36} \\ b_3 \in \mathbb{R}^{128} \\ b_4 \in \mathbb{R}^{10} \end{cases} \tag{2.23}$$

where $ReLu$ stands for the rectifier function applied to each unit and P is the pooling function with kernel size equal to 2×2 and stride equal to 2. Thus:

$$\begin{cases} U \in \mathbb{R}^{28\times28\times1} \\ C_1 \in \mathbb{R}^{14\times14\times16} \\ C_2 \in \mathbb{R}^{7\times7\times36} \\ F \in \mathbb{R}^{225792} \\ H_1 \in \mathbb{R}^{128} \\ O \in \mathbb{R}^{10} \end{cases} \tag{2.24}$$

The loss function is the cross entropy and the optimization algorithm is Adam, with learning rate set as 0.01, using mini batches of size 64.

Results: it must be said that, since the initialization of the parameters and the choice of the optimization batch are stochastic, results can slightly vary if the same algorithm is run different times.

After 1000 epochs the algorithms with 0.01 and 0.05 variance in the data reach an accuracy of about $98,5\%$, whereas the ones with very small variance $(10^{-6}, 10^{-8})$

	Conv. 1	**Conv. 2**	**F.C. 1**	**F.C. 2**
Pesi> 0	77	2782	81409	579
Pesi< 0	323	11618	144383	701
Bias> 0	8	23	72	7
Bias< 0	8	13	56	3

	Conv. 1	**Conv. 2**	**F.C. 1**
Unità> 0	1229	63	24
Unità$= 0$	1907	1701	104

Table 2.1: On the left a scheme representing how positive and negative parameters are distributed in the standard case. On the right the Networks activation values for a random sample of the test set

are stuck at about 10%. On the other hand the results achieved with parameters' variance of 10^{-4} are not sure, meaning that it could either converge or not.

Moreover the speed of the convergence is slightly different for different initialization variances. In particular if the variance is 0.05% the accuracy after 100 epochs is 5% more accurate than the case of 0.01 variance, on average. And this difference is considerably higher when the algorithm with variance 10^{-4} converges, since it reach 90% of accuracy after at least 9000 iterations.

However the most interesting observation made for these experiments is the number of the so called *dead units*, that are the units with 0 value. As it can be seen from table (2.1), after 1000 epochs the number of positive neurons of the output of the first fully connected layer is not negligible in the case of 0.05 variance, but it decreases when the variance is smaller and in the extreme cases, there is no non zero units, as table (2.2) may suggest. The average number of non zero units decrease considerably even in the case of 0.01 variance.

	Conv. 1	**Conv. 2**	**F.C. 1**	**F.C. 2**
Pesi> 0	73	9010	99930	671
Pesi< 0	327	5390	125862	609
Bias> 0	2	5	1	3
Bias< 0	14	31	127	7

	Conv. 1	**Conv. 2**	**F.C. 1**
Unità> 0	349	248	0
Unità$= 0$	2787	1516	128

Table 2.2: On the left a scheme shows that there is not a significant difference for the distribution of positive and negative weights between the standard case and the cases with smaller inputs. The main difference is given by a significantly higher number of negative biases and mostly by the fact that there is no non zero unit in the output of the fully connected layer, when evaluated for a generic input

Chapter 3

DLTI systems and Neural Networks

It should be clear from chapter 2 that a lot of heuristics is at the basis of Neural Networks. As an example, adaptive optimization algorithms and superposition of more than one non linear layer effectiveness have no rigorous mathematical explanation.

In this chapter all the tasks are going to be related to a well known class of mathematical systems: the so called DLTI systems. So results and parameters obtained through Deep Learning can be compared with the true physical one and a more accurate study can be accomplished.

At first it is going to be investigated whether Convolutional Neural Networks are able to catch the constitutive laws that generate DLTI systems, by analysing many input-output relationships.

The results show that there exists an actual equivalence between the two representations. Thus in the last section a more *complex* task. Pay attention, the world *complex* is used in order to underlie that the input-response relationship is not given explicitly any more, but it affects implicitly the output of the dataset.

In this case results show that the model is not able to catch the inner structure of DLTI system, and this brings to generalization problems.

3.1 Mathematical environment

A linear time-invariant system (LTI systems) is a transformation T that is both linear and time-invariant. Linearity means that, given any two input sequences u_1, u_2, the response of any linear combination of them is the same linear combination of the individual responses, y_1, y_2, that is:

$$\begin{cases} T(u_1) = y_1 \\ T(u_2) = y_2 \end{cases} \Rightarrow T(\alpha_1 u_1 + \alpha_2 u_2) = \alpha_1 y_1 + \alpha_2 y_2 \quad \forall \alpha_1, \alpha_2 \in \mathbb{R} \tag{3.1}$$

On the other hand the property of time invariance means that the output does not depend on the instant when the system is applied. In formulas:

$$T(u(t)) = y(t) \Rightarrow T(u(t+\tau)) = y(t+\tau) \tag{3.2}$$

By rewriting the input signal u as:

$$u(t) = \int_{-\inf}^{+\inf} u(\tau)\delta(t-\tau)d\tau \tag{3.3}$$

and, more in general:

$$u(\cdot) = \int_{-\inf}^{+\inf} u(\tau)\delta(\cdot - \tau)d\tau \tag{3.4}$$

it easily follows that:

$$y(\cdot) = T\left(\int_{-\inf}^{+\inf} u(\tau)\delta(\cdot - \tau)d\tau\right) \tag{3.5}$$

$$= \int_{-\inf}^{+\inf} u(\tau)T(\cdot - \tau)d\tau \tag{3.6}$$

$$= \int_{-\inf}^{+\inf} u(\tau)h(\cdot - \tau)d\tau \tag{3.7}$$

thus, given an input u, the corresponding output y will be:

$$y(t) = \Big(u * h\Big)(t) \tag{3.8}$$

where $*$ stands for the convolution operation and h is a linear function that will be called response to the unitary sample, that is the response of the system to Kroenecker delta, namely the function:

$$\delta(t) = \begin{cases} 1 & \textit{if } t = 0 \\ 0 & \textit{otherwise} \end{cases} \tag{3.9}$$

These models can be equivalently represented by a system of differential equation, thanks to the introduction of an *internal state* vector x. This allows to write the following:

$$\begin{cases} \dot{x}(t) = A_c x(t) + B_c u(t) \\ y(t) = C_c x(t) + D_c u(t) \end{cases} \tag{3.10}$$

where $x \in \mathbb{R}^k$, $A_c \in \mathbb{R}^{k\times k}$, $B_c \in \mathbb{R}^{k\times m}$, $C_c \in \mathbb{R}^{n\times k}$, $D_c \in \mathbb{R}^{n\times m}$.

As it has already been stated, the representation of the system through (3.8) is equivalent to (3.10), with the right choice of h, A_c, B_c, C_c, D_c. This fact will be proved soon in a particular setting.

Since in the sections below these systems have to be simulated and it is not possible to simulate the exact motion of a differential equation, a discretized version of the LTI systems has to be considered: the so called DLTI systems.

The corresponding DLTI representation of (3.10) is given by the following system for the state-space form

$$\begin{cases} x(k+1) = Ax(k) + Bu(k) \\ y(k) = Cx(k) + Du(k) \end{cases} \tag{3.11}$$

that can be obtained from (3.10) through a discretization method for differential equations, as a generic θ-method.

On the other hand, the corresponding discrete convolutional structure (3.8) is:

$$y(k) = \sum_{j=1}^{k} h(k-j)u(j) \tag{3.12}$$

where h is the discrete response to:

$$h(k) = \begin{cases} 1 & if\ k = 0 \\ 0 & otherwise \end{cases} \tag{3.13}$$

To make the concept of the equivalence between the two representations a little more rigorous, let's consider the particular case of $C = \mathbb{1}$ and $D = \mathbb{0}$. This means:

$$\begin{aligned} y(1) = x(1) &= Ax(0) + Bu(1) \\ y(2) = x(2) &= Ax(1) + Bu(2) = A^2x(0) + ABu(1) + Bu(2) \\ y(3) = x(3) &= ... = A^3x(0) + A^2Bu(1) + ABu(2) + Bu(3) \\ &\vdots \\ y(k) = x(k) &= A^kx(0) + \sum_{j=1}^{k} A^{k-j}Bu(j) \end{aligned}$$

from this series of equations, it can be easily noticed that:

$$Y(k) = A^kx(0) + \Big(u * h\Big)(k) \quad \forall k \in \mathbb{N} \tag{3.14}$$

and, in particular, that, given $u \in \mathbb{R}$, the unitary response or, equivalently, the kernel of the convolution (3.12) can be rewritten as:

$$h(j) = A^jB \tag{3.15}$$

It must be said that this is just a particular case of equation (3.11). The most general unitary response is given by the so called *Markov coefficients*:

$$h(j) = CA^jB \tag{3.16}$$

3.2 Finite Response State Space Model

Let's consider the following DLTI system, represented in the State Space form:

$$\begin{cases} x(k+1) = Ax(k) + Bu(k) \\ y(k) = Cx(k) + Du(k) \end{cases} \tag{3.17}$$

where

$$y \in \mathbb{R}^3, \quad A = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \quad C = \mathbb{1}, \quad D = \mathbb{0} \tag{3.18}$$

This is one of the simplest DLTI systems that can be considered. Namely, all parameters have a restrained dimension and thus it is easier to analyse in this particular setting A is chosen in order to be nilpotent. This fact, together with (3.15), means that the unitary response is non trivial if and only if $j = 0, 1, 2$. In particular:

$$h(0) = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad h(1) = \begin{pmatrix} 2 \\ 3 \\ 0 \end{pmatrix} \quad h(2) = \begin{pmatrix} 3 \\ 0 \\ 0 \end{pmatrix} \tag{3.19}$$

The first experiment that will be considered is to try to approximate the input output relationship in the system that has just been presented with a basic convolutional Network in order to verify both the equivalence of this DLTI system and the convolutional Layer, and the ability of these models to approximate the actual kernel.

In particular suppose that an input sequence $U \in \mathbb{R}^{20\times 1}$ generated by a random normal distribution is given. For every mode-1 element of U (in this case it corresponds to the component of the vector) the system described in (3.17) raises a response of dimension 3. Then the whole output sequence is $Y \in \mathbb{R}^{20\times 3}$, meaning that it is made of 3 output channels, with 20 evaluations each.

Therefore the simplest architecture of a network considered able to model this system is a basic CNN:

$$O = (U * W) + b \quad , \quad \begin{cases} O \in \mathbb{R}^{20\times 3} \\ U \in \mathbb{R}^{20\times 1} \\ W \in \mathbb{R}^{3\times 1\times 3} \\ b \in \mathbb{R}^{3} \end{cases} \tag{3.20}$$

Where O is the output predicted by the model, that has to be as close to the true response of the system as possible. The loss function that has been chosen is the mean squared error and the optimization algorithm is Adam with learning rate equal to 0.01.

As already observed model (3.20) is equivalent to the State Space system when weights corresponds to the value presented in (3.19) and biases are zero. In particular, notation introduced in chapter 2 means that the correct matrix representing the only slice of the weight tensor obtained by fixing the input channel is:

$$W[:, 0, :] = \begin{pmatrix} h(0)^T \\ h(1)^T \\ h(2)^T \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 2 & 3 & 0 \\ 3 & 0 & 0 \end{pmatrix} \tag{3.21}$$

The input output generation is repeated 20000 times, thus the dataset that is going to be considered is made of 20000 couples $(U, Y) \in \mathbb{R}^{20\times 1} \times \mathbb{R}^{20\times 3}$.

The first step is to check the effective equivalence between the two representations by initializing the weights and biases with the correct values in order to verify that optimization steps do not change significantly the parameters. This is actually what happens, since just 100 iterations bring the slice of the weight tensor to the value:

$$W[:, 0, :] = \begin{pmatrix} -0.0000000 & 0.0000000 & 1.0000000 \\ 2.0000000 & 3.0000000 & -0.0000000 \\ 3.0000000 & -0.0000000 & 0.0000000 \end{pmatrix} \tag{3.22}$$

After that, one may wonder if this Network is able to find the correct parameters, even when the initialization is not correct. For this purpose weights and biases are initialized as constants equal to 0. After 3000 epochs the parameters obtained through the training reach a great accuracy, since all biases, that should be zero, are lower than 10^{-5} and the slice of the weight tensor are:

$$W[:,0,:] = \begin{pmatrix} -0.0000164 & -0.0000002 & 0.9999998 \\ 1.9999838 & 2.9999955 & 0.0000000 \\ 2.9999850 & 0.0000002 & 0.0000000 \end{pmatrix} \tag{3.23}$$

And the accuracy is even better if the optimization step is run 10000 times. In this case the predicted 0 parameters are about 10^{-10}.

3.2.1 Increasing Capacity

Two Layers Models

The results and the models built in the previous examples have been obtained since the structure of the system was well known, so the parsimony criterion could be applied perfectly.

On the contrary suppose now that the structure is unknown. Therefore a bigger architecture is considered, so that a big family of systems, including the one that has to be approximated, can be represented with high accuracy from this model.

Thus two different enlargements of the basic network are considered to achieve the convergence.

The first model is similar to (3.20):

$$O = (U * W) + b \quad , \quad \begin{cases} O \in \mathbb{R}^{20\times 3} \\ U \in \mathbb{R}^{20\times 1} \\ W \in \mathbb{R}^{20\times 1\times 3} \\ b \in \mathbb{R}^{3} \end{cases} \tag{3.24}$$

In particular it is made of a single convolutional layer, with kernel size equal to 20 (in the previous case it was equal to 3), dimension of the output 3 and no final activation function. In this case all parameters are initialized as 0 and the optimization step is made by considering mean squared error as the loss function and Adam as the optimization algorithm with learning rate 0.01. Notice that there is no risk of not updating parameters since there is only one layer. After 3000 iterations every parameters that is supposed to be zero has magnitude lower than 10^{-7}. On the other hand the first three rows of the slice of the weight tensor obtained by fixing the input channel are:

$$W[:2,0,:] = \begin{pmatrix} -0.0000001 & 0.0000000 & 0.9999998 \\ 1.9999983 & 2.9999962 & 0.0000000 \\ 2.9999952 & 0.0000002 & 0.0000000 \end{pmatrix} \tag{3.25}$$

This is exactly what was expected, since all weights with offset index (the first one) from the fourth to the last are near 0, whereas the first three are the real values, as described in (3.21).

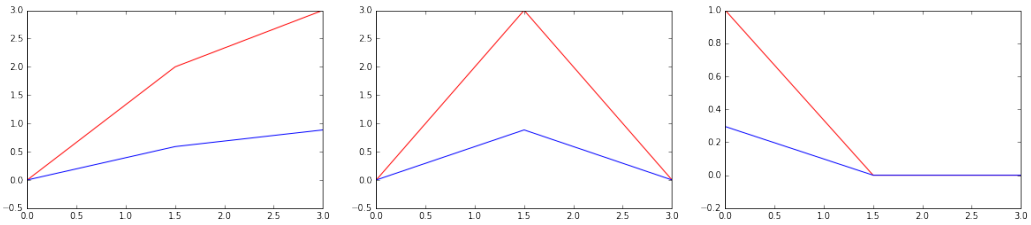

Figure 3.1: The red plot represents the weights of the dynamic system's kernel, whereas the blue one is the learned one. There is a figure for each output channel.

The second model is made of two convolutional layers. We made the original intuition that the first convolutional layer represents the real transformation and the second one is just the identity. Thus the dimensions of the parameters follows easily:

$$\begin{cases} H_1 = (U * W_1) + b_1 \\ O = (H_1 * W_2) + b_2 \end{cases} \quad , \quad \begin{cases} O \in \mathbb{R}^{20\times 3} \\ U \in \mathbb{R}^{20\times 1} \\ H_1 \in \mathbb{R}^{20\times 3} \\ W_1 \in \mathbb{R}^{3\times 1\times 3} \\ W_2 \in \mathbb{R}^{3\times 3\times 3} \\ b_1 \in \mathbb{R}^3 \\ b_2 \in \mathbb{R}^3 \end{cases} \tag{3.26}$$

All parameters are initialized to 0.1. After 5000 iterations of Adam optimization algorithm with learning rate 0.01, the results found are quite surprising. The biases are set to a very small value (about $O(10^{-8})$, and weight are as follows. For the first convolutional layer:

$$W_1[:,0,:] = \begin{pmatrix} 1.1303016 & 1.1303016 & 1.1303010 \\ -0.0000004 & -0.0000004 & 0.0000006 \\ 0.0000001 & 0.0000001 & -0.0000003 \end{pmatrix} \tag{3.27}$$

This matrix has to be read as: the element of position i, j is the contribute of the $(k+i)-th$ input the k-th element of the j-th channel of the output H_1, for every k. In particular 3.27 means that

$$H_1[i,j] = 1.131106 * U[i] + b_j \tag{3.28}$$

approximately.

On the other hand the output of the second convolutional layer is computed through the following matrices:

$$W_2[:,:,0] = \begin{pmatrix} 0.0000000 & 0.0000000 & -0.0000001 \\ 0.5898130 & 0.5898130 & 0.5898135 \\ 0.8847203 & 0.8847203 & 0.8847190 \end{pmatrix} \tag{3.29}$$

$$W_2[:,:,1] = \begin{pmatrix} -0.0000000 & -0.0000000 & 0.0000000 \\ 0.8847198 & 0.8847198 & 0.8847197 \\ -0.0000002 & -0.0000002 & 0.0000004 \end{pmatrix} \tag{3.30}$$

$$W_2[:,:,2] = \begin{pmatrix} 0.2949066 & 0.2949066 & 0.2949066 \\ -0.0000000 & -0.0000000 & 0.0000001 \\ 0.0000001 & 0.0000001 & -0.0000002 \end{pmatrix} \tag{3.31}$$

Each matrix represents the contribution of the corresponding component of H_1 to the channel of the output, that is the slices of the weight tensor obtained by fixing the index corresponding to the output channel. This means that the element $W_2[i,j,k]$ is the scaling factor to the output channel k of the element of the channel j of the input with an offset equal to i.

They could seem quite unreasonable, since one would expect that the intuitive representation of the transformation in this environment is the first kernel set as the actual convolutional kernel and the second one made only of 1 for the first element and 0 for all the others. However this is not a bad result at all! The transformation is learned in a different representation, but with the same properties. In particular the model scales the input through the factor 1.1303016 (equation (3.28)) and the second layer recreates three times the true transformation by applying it to each output of the first hidden layer. This might be clearer by looking at figure (3.1) and by noticing that:

$$3 * 1.1303016 * 0.2949066 = 1.00000020549168 \tag{3.32}$$
$$3 * 1.1303016 * 0.5898130 = 1.9999997328023997 \tag{3.33}$$
$$3 * 1.1303016 * 0.8847198 = 3.0000006164750395 \tag{3.34}$$

This means that the matrices of the second layer can approximately be seen as:

$$W_2[:,:,0] = \frac{1}{1.1303016} \cdot \frac{1}{3} \begin{pmatrix} 0 & 0 & 0 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix} \tag{3.35}$$

$$W_2[:,:,1] = \frac{1}{1.1303016} \cdot \frac{1}{3} \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{3.36}$$

$$W_2[:,:,2] = \frac{1}{1.1303016} \cdot \frac{1}{3} \begin{pmatrix} 3 & 0 & 0 \\ 3 & 0 & 0 \\ 3 & 0 & 0 \end{pmatrix} \tag{3.37}$$

Three Layers Model

It is a common belief that increasing the capacity of a certain model can improve convergence results. Thus, what happens if the model is enlarged? Can convergence results, obtained in the previous section, be affected by any particular choice in the architecture of the Network?

In the following the model considered is:

$$H_1 = (U * W_1) + b_1 \quad , \quad \begin{cases} U \in \mathbb{R}^{20\times 1} \\ H_1 \in \mathbb{R}^{20\times 3} \\ W_1 \in \mathbb{R}^{3\times 1\times 3} \\ b_1 \in \mathbb{R}^3 \end{cases} \tag{3.38}$$

$$H_2 = (H_1 * W_2) + b_2 \quad , \quad \begin{cases} H_2 \in \mathbb{R}^{20\times 3} \\ W_2 \in \mathbb{R}^{3\times 3\times 3} \\ b_2 \in \mathbb{R}^3 \end{cases} \tag{3.39}$$

$$O = (H_2 * W_3) + b_3 \quad , \quad \begin{cases} O \in \mathbb{R}^{20\times 3} \\ W_3 \in \mathbb{R}^{3\times 3\times 3} \\ b_3 \in \mathbb{R}^3 \end{cases} \tag{3.40}$$

$$\tag{3.41}$$

Basically the difference between this model and (3.26) consists of an extra hidden convolutional layer. This architecture is still compatible with the original system, since one possible representation could be the following:

- the first and the second convolutional layers can convey the input, eventually rescaled, to the tensor H_2;
- the third layer (3.39) recreates the real transformation.

This intuition is given by recreating the behaviour of the trained model obtained in (3.26). One other possible generalization can be:

- the parameters of the first and the second layer behave exactly as in the 2-layers model;
- the third layer can convey H_2 to the output, and eventually rescaling its components.

The results obtained after 8000 epochs of Adam algorithm with learning rate and initialization values for the parameters are 0.001 are not as expected. In particular the MSE is about 10^{-7} and the model obtained is not considerable as linear any more, since the biases that have been found are:

$$b_1 = \begin{pmatrix} -0.0000032 & -0.0000032 & -0.0000091 \end{pmatrix} \tag{3.42}$$

$$b_2 = \begin{pmatrix} -0.0001234 & -0.0001234 & 0.0002484 \end{pmatrix} \tag{3.43}$$

$$b_3 = \begin{pmatrix} -0.0000099 & -0.0000153 & 0.0003193 \end{pmatrix} \tag{3.44}$$

Moreover weights are neither sparse nor *shared* as in (3.27)-(3.31). For example the weight tensor for the first layer is:

$$W_1[:, 0, :] = \begin{pmatrix} 0.6463890 & 0.6463890 & 0.6462395 \\ 0.0663007 & 0.0663007 & 0.0667040 \\ 0.0025836 & 0.0025836 & 0.0090527 \end{pmatrix} \tag{3.45}$$

And it is even more evident for deeper layers:

$$W_3[:,:,0] = \begin{pmatrix} 0.3632962 & 0.3632962 & 0.3945965 \\ 0.5775110 & 0.5775110 & 0.5791647 \\ -0.0441417 & -0.0441417 & 0.0588008 \end{pmatrix} \tag{3.46}$$

In order to gain better convergence properties the model has been upgraded, by increasing the kernel size of each layer:

$$H_1 = (U * W_1) + b_1 \quad , \quad \begin{cases} U \in \mathbb{R}^{20\times 1} \\ H_1 \in \mathbb{R}^{20\times 3} \\ W_1 \in \mathbb{R}^{20\times 1\times 3} \\ b_1 \in \mathbb{R}^3 \end{cases} \tag{3.47}$$

$$H_2 = (H_1 * W_2) + b_2 \quad , \quad \begin{cases} H_2 \in \mathbb{R}^{20\times 3} \\ W_2 \in \mathbb{R}^{20\times 3\times 3} \\ b_2 \in \mathbb{R}^3 \end{cases} \tag{3.48}$$

$$O = (H_2 * W_3) + b_3 \quad , \quad \begin{cases} O \in \mathbb{R}^{20\times 3} \\ W_3 \in \mathbb{R}^{20\times 3\times 3} \\ b_3 \in \mathbb{R}^3 \end{cases} \tag{3.49}$$

$$\tag{3.50}$$

This representation is not natural, since it is basically the representation that is used in an infinite length unitary response. However the results found after 5000 iterations of Adam optimization algorithm with learning rate equal to 0.001 and initialization values for the parameters equal to 0.001 as well are much better than before. In particular all biases are near zero (they are about 10^{-9}) and the final MSE is about 10^{-13}. In this case there are two interesting factors.

First of all in this case it can be observed that weights are replicated among all

the output channels. For example:

$$W_1[:,0,:] = \begin{pmatrix} 0.5921481 & 0.5921481 & 0.5921481 \\ 0.2660780 & 0.2660780 & 0.2660780 \\ 0.0056468 & 0.0056468 & 0.0056468 \\ -0.0185987 & -0.0185987 & -0.0185987 \\ 0.0220667 & 0.0220667 & 0.0220667 \\ 0.0031530 & 0.0031530 & 0.0031530 \\ -0.0027568 & -0.0027568 & -0.0027568 \\ 0.0024285 & 0.0024285 & 0.0024285 \\ 0.0021107 & 0.0021107 & 0.0021107 \\ 0.0013231 & 0.0013231 & 0.0013231 \\ 0.0024848 & 0.0024848 & 0.0024848 \\ 0.0028737 & 0.0028737 & 0.0028737 \\ 0.0017980 & 0.0017980 & 0.0017980 \\ 0.0012938 & 0.0012938 & 0.0012938 \\ 0.0014946 & 0.0014946 & 0.0014946 \\ 0.0008530 & 0.0008530 & 0.0008530 \\ -0.0015292 & -0.0015292 & -0.0015292 \\ -0.0032948 & -0.0032948 & -0.0032948 \\ -0.0023037 & -0.0023037 & -0.0023037 \\ -0.0031917 & -0.0031917 & -0.0031917 \end{pmatrix} \tag{3.51}$$

Furthermore as far as the weight tensor of the first two layers are concerned, the parameters are shared among the three slices obtained by fixing the index corresponding to the output channel. For example the first three elements of the weight tensor of the second layer are:

$$W_2[:3,:,0] = \begin{pmatrix} 0.5920714 & 0.5920714 & 0.5920714 \\ 0.2659931 & 0.2659931 & 0.2659931 \\ 0.0041708 & 0.0041708 & 0.0041708 \end{pmatrix} \tag{3.52}$$

$$W_2[:3,:,1] = \begin{pmatrix} 0.5920714 & 0.5920714 & 0.5920714 \\ 0.2659931 & 0.2659931 & 0.2659931 \\ 0.0041708 & 0.0041708 & 0.0041708 \end{pmatrix} \tag{3.53}$$

$$W_2[:3,:,2] = \begin{pmatrix} 0.5920713 & 0.5920713 & 0.5920713 \\ 0.2659931 & 0.2659931 & 0.2659931 \\ 0.0041708 & 0.0041708 & 0.0041708 \end{pmatrix} \tag{3.54}$$

This does not happen in the third layer and this is not surprising: the aim of the third layer is the same of the second one in (3.26).

$$W_3[:3,:,0] = \begin{pmatrix} -0.0000009 & -0.0000009 & -0.0000008 \\ 0.6338437 & 0.6338437 & 0.6338437 \\ 0.3811884 & 0.3811884 & 0.3811884 \end{pmatrix} \tag{3.55}$$

$$W_3[:3,:,1] = \begin{pmatrix} -0.0000027 & -0.0000027 & -0.0000026 \\ 0.9507663 & 0.9507663 & 0.9507663 \\ -0.8543857 & -0.8543857 & -0.8543857 \end{pmatrix} \tag{3.56}$$

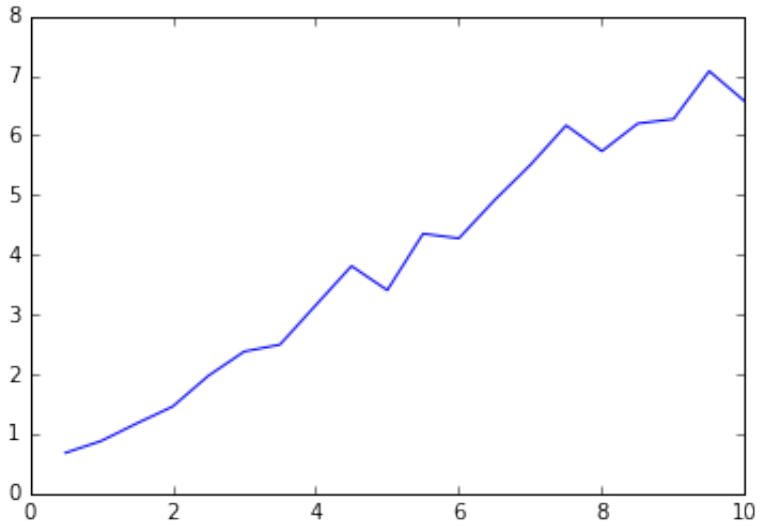

Figure 3.2: In this graph the variance of the random variable used to initialize the weights is set on the abscissa. On the ordinate the average of the variance of the trained parameters. This is referred to the second layer of model (3.26).

$$W_3[:3,:,2] = \begin{pmatrix} 0.3169168 & 0.3169168 & 0.3169168 \\ -0.2848194 & -0.2848194 & -0.2848194 \\ 0.1866540 & 0.1866540 & 0.1866540 \end{pmatrix} \tag{3.57}$$

One more interesting consideration is that the representation is not sparse any more.

Random initializations

One simplification of models in this section is the basic initialization of the weights. In all cases it is set to a tensor with equally valued elements. What happens when this hypothesis falls? In particular it is interesting to understand whether the properties found above (like sparsity and repetition of the weights) are still reached.

Consider, as an example, model (3.26). When it is trained with all the same hyperparameters described before, but with parameters initialized with random normal of variance 0.1, the result are quite different. In particular:

$$W_1[:,0,:] = \begin{pmatrix} 0.37731665 & -0.36477697 & 0.5041107 \\ 1.1142182 & 0.91963494 & -0.99504423 \\ 0.91104233 & 0.17498589 & 0.03419494 \end{pmatrix} \tag{3.58}$$

And also the second layer loses the sparsity and repetition properties. However the composition of the two kernels bring to an equivalent representation of the physical transformation.

More in general, in figure (3.2) it is possible to see that optimization step do not change the variance of the weights: the one of the initialization random variable is the same of the trained parameters.

3.3 Infinite response State Space Model

One of the many simplifications in the process of building the system (3.17-3.18) consists of considering a nilpotent matrix A, so that the unitary response has a finite length. Thus in that case, it was easy to build a model with the *right* architecture and to compare the weights obtained with the real unitary response (3.21).

However this occurrence is quite anomalous and, more in general, the unitary response has infinite length, due to a non nilpotent matrix A in the state space representation of the system. Thus the purpose of this section is to analyse whether or not the results previously obtained can be extended to a *non finite* response DLTI system.

Therefore consider the DLTI system

$$\begin{cases} x(k+1) = Ax(k) + Bu(k) \\ y(k) = Cx(k) + Du(k) \end{cases} \tag{3.59}$$

where

$$y \in \mathbb{R}^3, \quad A = \begin{pmatrix} 0.6 & 0.6 & 0.6 \\ 0.6 & 0 & 0.6 \\ 0 & 0.6 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \quad C = \mathbb{1}, \quad D = \mathbb{0} \tag{3.60}$$

(the values in A are chosen in order to not have an output with too high magnitude).

A is not nilpotent anymore, thus the unitary response is not zero for every evaluation, as equation (3.15) may suggest.

First of all let's build the samples: as already done, consider an input sequence $U \in \mathbb{R}^{20\times 1}$, so that the response of the system is the output sequence $Y \in \mathbb{R}^{20\times 3}$. The dataset is made by 20000 of these couples $(U, Y) \in \mathbb{R}^{20\times 1} \times \mathbb{R}^{20\times 3}$.

Now consider the Network already seen in (3.24):

$$O = (U * W) + b \quad , \quad \begin{cases} O \in \mathbb{R}^{20\times 3} \\ U \in \mathbb{R}^{20\times 1} \\ W \in \mathbb{R}^{20\times 1\times 3} \\ b \in \mathbb{R}^3 \end{cases} \tag{3.61}$$

where O is the output of the Neural Network given the input U. Thus the aim of the loss function that has to be considered is any norm representing the difference between O and Y. As done before, the chosen loss function is the mean squared error.

Since there is a strong relationship between the parameters of the model and the unitary response, the interesting part of the experiment consists of comparing these two elements. In particular, it can be deduced from (3.60) that the biases should be zero, while weights should be:

$$W[:, 0, :] \sim \begin{pmatrix} B^T \\ (AB)^T \\ (A^2B)^T \\ \vdots \\ (A^{19}B)^T \end{pmatrix} \tag{3.62}$$

All parameters are initialized with 0 (there is no risk of stopping the training since there is only one layer) and Adam is considered as the optimization algorithm, with learning rate equal to 0.01.

After 10000 epochs the mean of the difference between

$$\mathbb{E}[W[:,0,:] - \begin{pmatrix} B^T \\ (AB)^T \\ (A^2B)^T \\ \vdots \\ (A^{19}B)^T \end{pmatrix}] \sim 10^{-6} \tag{3.63}$$

where the operation $\mathbb{E}[\cdot]$ is the mean of the module of the elements in the object considered. Biases are $O(10^{-7})$, thus the model obtained is actually equivalent to the true system.

3.4 Time Variant Systems

Now suppose that a the dataset is generated by a time variant system, like the following:

$$\begin{cases} x(k+1) = A(k)x(k) + u(k) \\ y(k) = Cx(k) + Du(k) \end{cases} \tag{3.64}$$

where

$$y \in \mathbb{R}^3, \quad A = \begin{cases} \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{pmatrix} & 0 \le k \le 9 \\ \begin{pmatrix} 0 & 2 & 3 \\ 0 & 0 & 4 \\ 0 & 0 & 0 \end{pmatrix} & 10 \le k \le 19 \end{cases}, \quad B = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \quad C = \mathbb{1}, \quad D = \mathbb{0} \tag{3.65}$$

The only difference with the system described in (3.17)-(3.18) is that dynamic of the data is controlled by a DLTI system for the first 10 steps, by one other for the last ones. Thus the process is not time invariant any more and the equivalence between the system and convolution is not valid for the whole input output relationship. However $u[:9]$ generates $x[:9]$ through a convolution, and so does $u[9:]$ with respect to $x[9:]$. In particular the kernel of the the first convolution is given by:

$$h_1(0) = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad h_1(1) = \begin{pmatrix} 2 \\ 3 \\ 0 \end{pmatrix} \quad h_1(2) = \begin{pmatrix} 3 \\ 0 \\ 0 \end{pmatrix} \tag{3.66}$$

whereas the second one is:

$$h_2(0) = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad h_2(1) = \begin{pmatrix} 3 \\ 4 \\ 0 \end{pmatrix} \quad h_2(2) = \begin{pmatrix} 8 \\ 0 \\ 0 \end{pmatrix} \tag{3.67}$$

Consider a dataset made by applying the system (3.64)-(3.65) to 20000 different inputs $U \in \mathbb{R}^{20\times 1}$, each of whom generated by a random normal distribution.

The first model that is going to be considered is the following:

$$O = (U * W) + b \quad , \quad \begin{cases} O \in \mathbb{R}^{20\times 3} \\ U \in \mathbb{R}^{20\times 1} \\ W \in \mathbb{R}^{2\times 3\times 1} \\ b \in \mathbb{R} \end{cases} \tag{3.68}$$

The output of the model O has to be considered as an approximation of the dynamic response Y to the input U, thus the Network is trained by using mean squared error as the loss function and Adam algorithm with learning rate set to 0.01.

However two separate convolutions generate the response in the dynamic system (3.64), thus good approximation properties of this model are rather unlikely, because of only one kernel is considered. Nevertheless, it would be interesting to understand what happens to the parameters of the Network.

The error is quite high, the biases are:

$$\begin{pmatrix} 0.01437 & 0.00258 & 5\cdot 10^{-9} \end{pmatrix} \tag{3.69}$$

and the weights are:

$$W[:,0,:] = \begin{pmatrix} -0.0043934 & -0.0128955 & 1.0000001 \\ 2.5247779 & 3.5258610 & 0.0000000 \\ 5.6695366 & -0.0146008 & 0.0000000 \end{pmatrix} \tag{3.70}$$

Thus, as expected, the estimates are not very good, but it is quite interesting to notice that the slice obtained by fixing the output channel index is similar to the mean of the unitary response of the two DLTI system, that is equal to:

$$\begin{pmatrix} (\frac{h_1(0)+h_2(0)}{2})^T \\ (\frac{h_1(1)+h_2(1)}{2})^T \\ (\frac{h_1(2)+h_2(2)}{2})^T \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 2.5 & 3.5 & 0 \\ 5.5 & 0 & 0 \end{pmatrix} \tag{3.71}$$

It is not surprising that the smallest bias corresponds to the third coordinate, that is generated with the same dynamic by both systems (3.66)-(3.67) and the difference between the third column of the mean of the responses and the learned one is almost zero.

One more surprising fact is that, if the model (3.68) is given more layers, it does not improve the accuracy. On the contrary, it works exactly as described in section 3.2.1, meaning that the output does not change. This observation can be deduced by comparing the outputs of the two different Networks and by looking at the fact that the mean of the absolute value of their difference is about 0.09. Even the test error is always about 1.9.

Moreover, if it is given non linearity, the accuracy get worse, even if this is not surprising, since the system is linear.

One different approach is to consider a fully connected Neural Network. Previous model could not converge because it reuses the same transformation for each time

(parameter sharing), even though the real dynamic is generated by different convolutions. This operation has to be generalized, and a linear model can be suitable in this situation:

$$O = U \cdot W + b \quad , \quad \begin{cases} O \in \mathbb{R}^{60} \\ U \in \mathbb{R}^{20\times 1} \\ W \in \mathbb{R}^{20\times 60} \\ b \in \mathbb{R}^{60} \end{cases} \tag{3.72}$$

where O represents an approximation of Y in a different shape. Thus, after having reshaped O to the dimension $(20, 3)$, the loss function is, again the mean squared error and the parameters are trained through gradient descent optimizer with learning rate equal to 0.05 and batch size equal to 50.

In this case, after 15000 epochs, convergence has been achieved, as the mean squared error on the test set is about $O(10^{-9})$. Moreover the real kernel has been found and the biases are all $O(10^{-7})$. It is not possible to report all the learned weights (they are 1200), thus to give a proof of the actual convergence, the mean of the absolute value of difference between the weights obtained and and the expected ones is lower than $O(10^{-5})$.

3.5 Inverse estimates

In this section, it is going to be investigated whether CNNs are able to represent the inverse estimates (output-input) of a DLTI system. From the theoretical point of view, this is plausible, since from (3.11) it can be deduced:

$$Bu(k) = y(k+1) - Ay(k) \tag{3.73}$$

that resemble a convolution. The only difference is that if u is scalar, B a vector and A a matrix, as considered until now, the relationship between $u[:]$ and $y[:]$ is not clear, since B can not be inverted.

The simplest case is to consider B made of all zeros with the exception of one component. For example:

$$A \in \mathbb{R}^{3\times 3}, B = \begin{pmatrix} 0 \\ 0 \\ \alpha \end{pmatrix}, C = \mathbb{1}, D = \mathbb{0}, y(k) = \begin{pmatrix} y_0(k) \\ y_1(k) \\ y_2(k) \end{pmatrix} \tag{3.74}$$

In this scenario u can be computed easily:

$$u(k) = \frac{1}{\alpha}(y_2(k+1) - (Ay(k)))_2 \tag{3.75}$$

Hence the following system is used to generate the dataset:

$$\begin{cases} x(k+1) = Ax(k) + Bu(k) \\ y(k) = Cx(k) + Du(k) \end{cases} \tag{3.76}$$

where

$$y \in \mathbb{R}^3, \quad , \quad B = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad C = \mathbb{1}, \quad D = \mathbb{0} \tag{3.77}$$

and A is a 3×3 matrix generated through a random normal $N(0; 0.1)$. The input to the system $U \in \mathbb{R}^{20\times 1}$ is generated randomly; in particular all of its elements are taken as the module of a random standard normal. 20000 input output couples are generated through system (3.76)-(3.77) and this constitutes the dataset.

The particular CNN considered is:

$$O = (Y * W) + b \quad , \quad \begin{cases} O \in \mathbb{R}^{20\mathbb{0}\times \mathbb{1}} \\ Y \in \mathbb{R}^{20\times 3} \\ W \in \mathbb{R}^{2\times 3\times 1} \\ b \in \mathbb{R} \end{cases} \tag{3.78}$$

The dimension of W are deduced from (3.75). The purpose is to approximate U with O, thus the loss function that has been considered is the mean squared error.

The Network is initialized with zero valued parameters and it is trained through Adam with learning rate 0.001 and batch size 64. After about 15000 iterations the loss function reaches a very low value and the weights and biases found are similar, but not exactly equal, to what was expected. For example, the process of generating of the matrix A gives rise to:

$$A = \begin{pmatrix} 0.04203465 & -0.06176848 & 0.0063325 \\ 0.04226415 & -0.04879597 & -0.03071113 \\ 0.06982541 & 0.09223424 & 0.20546796 \end{pmatrix} \tag{3.79}$$

In this case, for the previous reasoning, it would be expected that the slices of the weight tensor, obtained through fixing the first index, correspond $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ and to the third row of matrix A. Even though the result is similar to the expected one, it can not be considered *equal to*. In fact, the bias is very small, about 10^{-6}, but the slices are:

$$W[0,:,0] = \begin{pmatrix} -0.1355553 \\ 0.2123551 \\ 1.0000001 \end{pmatrix} \quad W[1,:,0] = \begin{pmatrix} -0.0725815 \\ -0.0901222 \\ -0.1980880 \end{pmatrix} \tag{3.80}$$

In particular, neither the first two elements of $W[0,:,0]$ can be considered zero, nor $W[1,:,0]$ can be an approximation of the third row of A. However it can be noticed that:

$$A[0,:] \cdot W[0,0,0] + A[1:] \cdot W[0,1,0] + A[2,:] \cdot W[0,2,0] = \begin{pmatrix} 0.07310241 \\ 0.09024523 \\ 0.19808792 \end{pmatrix} \tag{3.81}$$

It means that the algorithm finds that the significant element for determining U is the third component of $y(k+1) - A \cdot y(k)$. However instead of nullifying the other two components of the weight vector corresponding to $y(k+1)$ and approximating $A[2,:]^T$ with $W[0,2,0]$, it reaches a point that satisfy equations (3.75) and the following:

$$\alpha_0 \left(y(k+1)_0 - (Ay(k))_0\right) = 0 \tag{3.82}$$

$$\alpha_1 \left(y(k+1)_1 - (Ay(k))_1\right) = 0 \tag{3.83}$$

$$\frac{1}{\alpha} \left(y(k+1)_1 - (Ay(k))_1\right) = u(k) \tag{3.84}$$

Thus, in this case, it is acceptable any solution such that:

$$\begin{cases} W[0,0,0] = a_0 \\ W[0,1,0] = a_1 \\ W[0,2,0] = a_2 \end{cases} \tag{3.85}$$

$$W[1,:,0] = \Big(A[0,:]\cdot a_0 + A[1:]\cdot a_1 + A[2,:]\cdot a_2\Big)^T \tag{3.86}$$

That is a generalization to the argument made in the beginning of the section, since the latter would be obtained by setting $\alpha_0, \alpha_1 = 0$.

Generalizing this experiment can be done, for example, by considering any vector $B = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$. Mathematically speaking the new scenario is the following:

$$\begin{aligned} (y(k+1)_0 - (Ay(k))_0) &= b_0 u(k) && (3.87)\\ (y(k+1)_1 - (Ay(k))_1) &= b_1 u(k) && (3.88)\\ (y(k+1)_1 - (Ay(k))_1) &= b_2 u(k) && (3.89) \end{aligned}$$

Now, suppose that each equation is rescaled through a coefficient α_i and then sum the three equations:

$$\begin{aligned} & \alpha_0 \left(y(k+1)_0 - (Ay(k))_0\right) \\ + \;& \alpha_1 \left(y(k+1)_1 - (Ay(k))_1\right) \\ + \;& \alpha_2 \left(y(k+1)_1 - (Ay(k))_1\right) \\ = \;& (\alpha_0 b_0 + \alpha_1 b_1 + \alpha_2 b_2) u(k) \end{aligned} \tag{3.90}$$

Thus the physical acceptable solutions are:

$$W[0,:,0] = a \qquad a \cdot B = 1 \tag{3.91}$$

$$W[1,:,0] = \Big(A[0,:]\cdot a_0 + A[1:]\cdot a_1 + A[2,:]\cdot a_2\Big)^T \tag{3.92}$$

This is actually what has been found. In particular the dataset is generated by 20000 couples (U, Y) with the dynamic described in (3.76), with A generated by a random normal $N(0; 0.1)$ and

$$B = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

The Network (3.78) has been trained with Adam optimizer with learning rate set to 0.001, batch size 50 and loss function mean squared error. After about 16000 iterations these are the results:

$$W[0,:,0] = \begin{pmatrix} 0.24229373 \\ 0.24563794 \\ 0.26977462 \end{pmatrix} \qquad W[0,:,0]\cdot B \sim 10^{-8} \tag{3.93}$$

Moreover:

$$A[0,:]\cdot W[0,0,0] + A[1,:]\cdot W[0,1,0] + A[2,:]\cdot W[0,2,0] + W[1,:,0] \sim 10^{-6} \tag{3.94}$$

3.6 DLTI systems classification

In this experiment, the input-output relationship is given by the following:

$$\begin{cases} x(k+1) = A_V x(k) + Bu(k) \\ y(k) = Cx(k) + Du(k) \end{cases} \tag{3.95}$$

where V is a discrete random variable such that:

$$V \in \{0, 1, 2, 3, 4\} \tag{3.96}$$

with the same probability, that is $\mathbb{P}(V = i) = \frac{1}{5}$. Moreover:

$$y \in \mathbb{R}^3, \quad , \quad B = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \quad C = \mathbb{1}, \quad D = \mathbb{0} \tag{3.97}$$

$$A_0 = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} A_1 = \begin{pmatrix} 0 & 2 & 2 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{pmatrix} A_2 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{pmatrix} \tag{3.98}$$

$$A_3 = \begin{pmatrix} 0 & 2 & 3 \\ 0 & 0 & 3 \\ 0 & 0 & 0 \end{pmatrix} A_4 = \begin{pmatrix} 0 & 1 & 3 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \tag{3.99}$$

These matrices are taken in order to be simple to analyse. In particular they all are nilpotent and all possible results of B,AB,A^2B are made of positive components.

Since the response of the system (3.95) depends on the value of V, the purpose of this experiment is to classify the input-output relationship in the 5 different classes. Moreover, it would be interesting to understand whether a multi layer Network is able to achieve lower level physical relevant information.

For example, after having built the dataset by considering 20000 samples generated by an input $U \in \mathbb{R}^{20\times1}$-output $Y \in \mathbb{R}^{20\times3}$ through the model (3.95)-(3.98), let's consider the following architecture:

$$\begin{cases} H_1 = (I * W_1) + b_1 \\ H_2 = flattening(H_1) \\ L = (H_2 \cdot W_2) + b_2 \end{cases} \quad , \quad \begin{cases} O \in \mathbb{R}^5 \\ I \in \mathbb{R}^{20\times4} \\ H_1 \in \mathbb{R}^{20\times15} \\ H_2 \in \mathbb{R}^{300} \\ W_1 \in \mathbb{R}^{3\times4\times15} \\ W_2 \in \mathbb{R}^{300\times5} \\ b_1 \in \mathbb{R}^{15} \\ b_2 \in \mathbb{R}^5 \end{cases} \tag{3.100}$$

This architecture has been inspired by a simple intuition: I is the superposition of U and Y and the 5 channels of H_1 should represent $\tilde{Y}_i - Y$, where $\tilde{Y}_i$ is the predicted output of each DLTI system and Y is the actual output signal.

For the sake of classification, L is a vector with 5 entries. Each of them can be considered as the non normalized log probability that I is an input-output couple generated through the DLTI system corresponding to that index. Thus softmax cross entropy is considered as loss function. Convergence is obtained even after just 100 iterations of Adam optimizer with learning rate set to 0.001 and batch size 50.

However the results show that the first layer does not give the physical relevance given by human intelligence. This is due to the fact that biases are of the order of 10^{-2} and thus the model is can not be considered linear, as intuition would suggest. Moreover, if the following architecture is considered:

$$\begin{cases} H_1 = flattening(Y) \\ L = (H_1 \cdot W_1) + b_1 \end{cases} \quad , \quad \begin{cases} O \in \mathbb{R}^5 \\ Y \in \mathbb{R}^{20\times 3} \\ H_1 \in \mathbb{R}^{60} \\ W_1 \in \mathbb{R}^{60\times 5} \\ b_1 \in \mathbb{R}^5 \end{cases} \tag{3.101}$$

This is a black-box model mapping the output Y directly to its class. Convergence is achieved rapidly as well.

The question is: how is it possible that the model is able to classify without knowing the structure of the system? Looking at the 5 systems, it can be noticed that they are quite different from each other. It is plausible that the network is able to catch the shape of the response instead of the physical frame. Intuitively, thus, the Network can learn by just *looking* at the magnitude of the signal.

Now one may wonder whether a more complex dynamics can force the Neural Network to learn the true kernels of the different DLTI systems. For example, consider again the model (3.95)-(3.97), but replace equation (3.98) with the following:

$$A_i = 0.1 * RandomNormal(3 \times 3) \quad B_i = 0.1 * RandomNormal(3 \times 1) \tag{3.102}$$

In this case, the dynamic is more complex, thus it is possible for the Network to catch the structure of the system instead of the shape of the signal.

In this case the Neural Network analogous to (3.100) is:

$$\begin{cases} H_1 = (I * W_1) + b_1 \\ H_2 = flattening(H_1) \\ L = (H_2 \cdot W_2) + b_2 \end{cases} \quad , \quad \begin{cases} O \in \mathbb{R}^5 \\ I \in \mathbb{R}^{20\times 4} \\ H_1 \in \mathbb{R}^{20\times 5} \\ H_2 \in \mathbb{R}^{100} \\ W_1 \in \mathbb{R}^{20\times 4\times 5} \\ W_2 \in \mathbb{R}^{100\times 3} \\ b_1 \in \mathbb{R}^5 \\ b_2 \in \mathbb{R}^5 \end{cases} \tag{3.103}$$

Again, the intuition is quite simple, since the convolutional layer is thought to catch the discrepancy between the predicted output and the true one. However in this case it is not possible to achieve convergence. In particular it has not been found any way to get more than 40% accuracy. If activation functions are added in the last

layer, as suggested by universal approximation theorem 2.1.9, the accuracy reached is at most 60%.

Thus, following one of the most popular and strong beliefs in Neural Networks' community, it has been built a model that could effectively been considered a multi layer network, that is, an activation function has been inserted between H_1 and H_2. In formulas:

$$\begin{cases} H_1 = ReLu[(I * W_1) + b_1] \\ H_2 = flattening(H_1) \\ L = (H_2 \cdot W_2) + b_2 \end{cases} , \quad \begin{cases} O \in \mathbb{R}^5 \\ I \in \mathbb{R}^{20\times 4} \\ H_1 \in \mathbb{R}^{20\times 5} \\ H_2 \in \mathbb{R}^{100} \\ W_1 \in \mathbb{R}^{20\times 4\times 5} \\ W_2 \in \mathbb{R}^{100\times 5} \\ b_1 \in \mathbb{R}^5 \\ b_2 \in \mathbb{R}^5 \end{cases} \tag{3.104}$$

The only difference between (3.103) and (3.104) is the ReLu activation function in the first layer. Pay attention, activation function is computed for each element of H_1! Thus the loss function is still softamx cross entropy. In this case, 100% accuracy is achieved in just 35 iterations of Adam optimizer with learning rate 0.01 and batch size 50. This is an experimental evidence of the power of multi layer networks with respect to single layer ones. Now, a key point is to understand whether the network utilizes physical information in H_1 or it just looks for positive or negative peaks that are characteristic properties of each system.

The answer is that the physical model can not be seen in the weights. In particular, the sign of the first three elements in the first kernel of the Neural Networks does not match the true one of the convolutional kernels, nor it is the exact opposite. This observation is crucial, since in the physical point of view, it is important to understand how different kernels affect the output, thus having W as convolutional kernel or $\alpha \cdot W$, $\alpha \in \mathbb{R}$ is the same. This means that at least a portion of the the physical information is not used in H_1.

Until now it has been seen that classification can be achieved successfully by using a non trivial Neural Network. It would be interesting to understand whether the network needs effectively all the pieces of information that it has been given. In particular, since it has been seen that the true convolution has not been learned, is it possible to classify successfully even if the input signal is not given to the Network? Thus the following architecture has been considered:

$$\begin{cases} H_1 = ReLu[(Y * W_1) + b_1] \\ H_2 = flattening(H_1) \\ L = ReLu[(H_2 \cdot W_2) + b_2] \end{cases} \quad , \quad \begin{cases} O \in \mathbb{R}^5 \\ Y \in \mathbb{R}^{20\times 3} \\ H_1 \in \mathbb{R}^{20\times 5} \\ H_2 \in \mathbb{R}^{100} \\ W_1 \in \mathbb{R}^{20\times 3\times 3} \\ W_2 \in \mathbb{R}^{60\times 5} \\ b_1 \in \mathbb{R}^5 \\ b_2 \in \mathbb{R}^5 \end{cases} \tag{3.105}$$

That is very similar to (3.104), with the only difference that the input is not the couple $(U, Y) \in \mathbb{R}^{20\times 4}$ any more, but just Y. However, classification's results are very good because 97% is reached after about 160 epochs of optimization step, even though 100% accuracy is never obtained.

But let's make it even more difficult and suppose that only one channel of the output of the dynamical system (3.95) is given to the Network. Thus the structure has to be the following:

$$\begin{cases} H_1 = ReLu[(I * W_1) + b_1] \\ H_2 = flattening(H_1) \\ L = ReLu[(H_2 \cdot W_2) + b_2] \end{cases} \quad , \quad \begin{cases} O \in \mathbb{R}^5 \\ Y \in \mathbb{R}^{20\times 1} \\ H_1 \in \mathbb{R}^{20\times 5} \\ H_2 \in \mathbb{R}^{100} \\ W_1 \in \mathbb{R}^{20\times 1\times 3} \\ W_2 \in \mathbb{R}^{60\times 5} \\ b_1 \in \mathbb{R}^5 \\ b_2 \in \mathbb{R}^5 \end{cases} \tag{3.106}$$

In this case convergence is much slower, since 90% accuracy is reached in more than 2300 epochs and to obtain the 99% 13000 and more iterations have to be waited.

Now, the question is: what happens when the dynamic that generates the data is forced into the Network? In other words, let's consider firstly a Network whose purpose is to approximate the responses on a fixed channel of the 5 DLTI systems. Figure (3.3) represents this argument. Thus the first step is to consider the model:

$$O = (U * W_1) + b_1 \quad , \quad \begin{cases} O \in \mathbb{R}^{20\times 15} \\ U \in \mathbb{R}^{20\times 1} \\ W_1 \in \mathbb{R}^{20\times 1\times 15} \\ b_1 \in \mathbb{R}15 \end{cases} \tag{3.107}$$

And the corresponding dataset is made by 20000 samples of the shape $(U, Y_0, Y_1, Y_2, Y_3, Y_4)$, where $U \in \mathbb{R}^{20\times 1}$ is the input signal and it is the input to the Network as well. On the other hand $Y_0, ..., Y_4$ are the components corresponding to the chosen channel of the response to the signal U of each DLTI system. It has been decided to consider only one channel because of the fact that, in general it is much

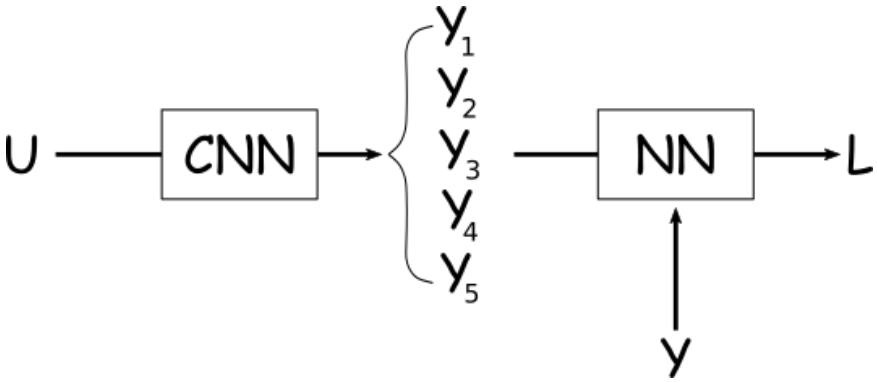

Figure 3.3

more difficult to achieve convergence in this case, as observed previously. Once the training is done, the trained weights are used in the following model:

$$\begin{cases} H_1 = (U * W_1) + b_1 \\ H_2 = stack(H_1, Y) \\ L = H_2 \cdot W_2 + b_2 \end{cases} \quad , \quad \begin{cases} L \in \mathbb{R}^5 \\ U \in \mathbb{R}^{20\times 1} \\ Y \in \mathbb{R}^{20\times 3} \\ H_1 \in \mathbb{R}^{20\times 5} \\ W_1 \in \mathbb{R}^{20\times 1\times 5} \\ W_2 \in \mathbb{R}^{160\times 5} \\ b_1 \in \mathbb{R}^5 \\ b_2 \in \mathbb{R}^5 \end{cases} \tag{3.108}$$

Three different experiments are made:

1. consider W_1 fixed with the value found in the training of model (3.107);
2. W_1 is variable and initialized with the values above found as above;
3. finally, initialize randomly W_1.

It must be said that results depend on the matrices A, on the vector B and on the data. The latter is due to the fact that sinusoidal input signals makes the data much more difficult to approximate, because of their frequency spectrum.

However repeating the experiment many times, it has been found that this kind of model can not reach any kind of convergence in the most general setting. Nevertheless, it is interesting to see the results in two separate settings: random or sinusoidal input. If the input is random, convergence is easily achieved with Network (3.108) in no more than 300 iterations through Adam optimization with learning rate 0.01. If Gradient Descent with the same learning rate is used, more than 1900 epochs are needed. In any case there are some curious differences arising when the convolutional part is set in the three ways just presented. In particular the Network with fixed convolutional weights has the slowest convergence, while the completely random one has the fastest. This is not so evident with Adam (the three cases needs about 250 iterations for the fastest, 300 for the slowest), but it is prominent

for Gradient Descent. In particular when the convolution is fixed, the optimization step has to be run about 4000 times in order to converge. On the contrary, if also the convolutional parameters are variable, 2500 epochs are needed when the initialization is the *correct* one, whereas only 1900 iterations are necessary for the random initialization.

In the case of sinusoidal input, the convergence is not so easy. In particular the Network (3.108) is not able to converge. However, if the network is given one more non linear fully connected layer, namely:

$$\begin{cases} H_1 = (U * W_1) + b_1 \\ H_2 = stack(H_1, Y) \\ H_3 = ReLu(H_2 \cdot W_2 + b_2) \\ L = H_3 \cdot W_3 + b_3 \end{cases}, \quad \begin{cases} L \in \mathbb{R}^5 \\ U \in \mathbb{R}^{20\times 1} \\ Y \in \mathbb{R}^{20\times 3} \\ H_1 \in \mathbb{R}^{20\times 5} \\ W_1 \in \mathbb{R}^{20\times 1\times 5} \\ W_2 \in \mathbb{R}^{160\times 5} \\ W_3 \in \mathbb{R}^{5\times 5} \\ b_1 \in \mathbb{R}^5 \\ b_2 \in \mathbb{R}^5 \\ b_3 \in \mathbb{R}^5 \end{cases} \tag{3.109}$$

it is very interesting to analyse the results. In particular, for any choice of the convolutional layer, the network is never able to reach the perfect accuracy with Adam optimizer. When the convolution is fixed 98% accuracy is reached in less epochs than the trainable case. And more interestingly in the latter the weights W_1 obtained are completely different from the true one. As an example the results of a particular experiment are going to be shown. In this case $\hat{W}_1$ are the correct weights and W_1 are the weight found by training all the Network (3.109). Thus:

$$W_1[:3,0,:] = \begin{pmatrix} -0.37233993 & -0.93339639 & -0.33790907 & 0.03506417 & -0.15369382 \\ -0.23304437 & -0.23825402 & -0.31590494 & 0.13513074 & 0.02095338 \\ -0.03579723 & 0.11440075 & -0.09640872 & -0.02512695 & 0.00057436 \end{pmatrix} \tag{3.110}$$

$$\hat{W}_1[:3,0,:] = \begin{pmatrix} -0.6040414 & -1.2870907 & -0.6166479 & 0.06861603 & -0.17842875 \\ 0.0399288 & -0.34307066 & -0.27957436 & 0.4304605 & 0.04269145 \\ -0.2789009 & 0.554305 & 0.05700739 & -0.14468141 & -0.05165162 \end{pmatrix} \tag{3.111}$$

It can be easily checked that there is no permutation of the indices, nor a rescaling that can match W_1 and $\hat{W}_1$. Moreover the weights found by the Network are all about 10^{-1}, but the true ones are even 10^{-6} for the latest indices. This can be seen in figure (3.4). This figure compares the first channel of the response of the first DLTI system with the parameters obtained in all the 5 corresponding channels of the CNN. As it can be seen, no significant relationship can be deduced.

But there is a more interesting fact about model (3.109). If the dataset is generated by a series of random inputs U, the trained parameters are completely different from the ones obtained with a dataset obtained by sinusoidal signals U. Figure (3.5) provides an intuitive qualitative comparison between those two classes. In

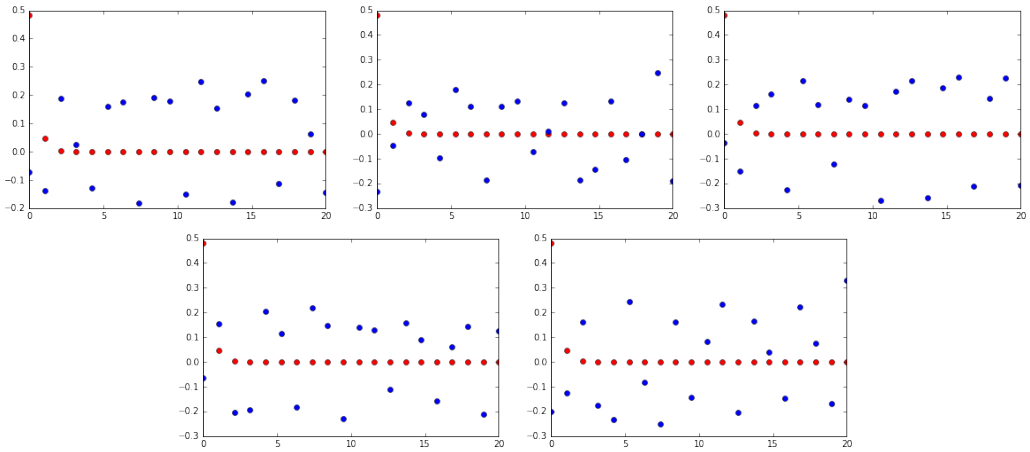

Figure 3.4: These figures compare each of the kernels of the 5 output channels of the Convolutional Neural Network obtained by training model (3.109) with the true convolutional kernel of the first DLTI system.

both layers the weights fluctuate around zero, but the set of weights generated by the random inputs is about 10 times lower than the others. The figure represents only 20 of each of them, so that they are significant, but this relationship can be seen for all the weights.

Moreover if the training set is generated by a sinusoidal set of different U, the test accuracy on a dataset generated by random signals U is about 20% and vice versa.

All these considerations mean that Deep Learning fails to get the true inner structure of the dynamic system. Thus the intuition that superposing more layers means getting more specific features, fails completely in this case.

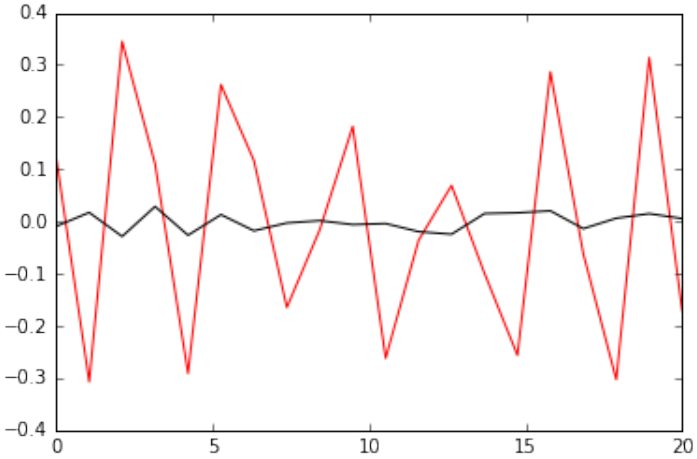

Figure 3.5: Comparison between the weights trained with different dataset. The red plot represents a set of weights corresponding to an output channel when the inputs are sinusoidal. The black one is the same weights obtained by considering random signals.

3.7 Conclusion

3.7.1 Neural Networks and DLTI systems

In sections 3.2-3.5 the relationship between DLTI systems and Neural Networks has been investigated. First of all it has been observed that the changes carried to Gradient Descent still preserve convergence of the optimization procedure, in general. More specifically, if the Network satisfies the parsimony criterion, the *trained* weights correspond to the true one (result obtained in section 3.20).

This means that, in the perfect scenario, Neural Networks can catch the inner structure of DLTI systems.

This happens even if the architecture of the model gets more complicated. In particular when the simple convolutional Network with only one layer is given more units nothing significant happens, whereas if the number of layers grows, the final parameters depend on the initial ones. This actually makes sense: the optimization algorithm brings the parameters to values that are *near*, even though they are not the most intuitive ones.

Sections 3.4 and 3.5 confirm the intuition that Neural Networks are able to catch the inner structure of the system if it has to approximate a certain explicit input output relationship, even when the task is more complicated.

3.7.2 Generalization

When the input-output relationship is not explicitly given to the Network, the results are not good any more. Conversely in the experiments of classification made in section 3.6 neither the actual physical kernel is reached, nor any of its rescaled versions does.

However a couple more observations are interesting. First of all, models (3.104)-(3.105) show that adding layers is effectively helpful in reaching better convergence properties. This confirms the intuition, but it can not be compared with any mathematical meaningful element of the kernel. Moreover, if the true structure of the system is given to the Network, as in experiments (3.108)-(3.109), the convergence is not faster. It does not even allow to use a less complex Network. On the contrary, giving that particular a priori knowledge slows down significantly the convergence.

But the most important result obtained in this chapter is reached with model (3.109) and it is the following. Even though the model that generates the responses does not depend on the inputs it is given, the Neural Network does. This is not surprising, since a similar observation arose also in section 3.2.1. The key point is that the *learned* models are valid only when similar samples are given to it. For example if the training set is constituted by responses to sinusoidal signals, the final parameters are not capable to classify a signal that is the response of a random input. In other words, Deep Learning does not catch the physical constitutive laws, even though it has been shown in sections 3.2 and 3.5 that it has the capability to do that. On the contrary the hidden layers, in this specific case, do not contain any cryptic feature.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, The MIT Press, 2016

[2] Shai Shalev-Shwartz, Shai Ben-David *Understanding Machine Learning: From Theory to Algorithms*, Cambridge University Press, 2014

[3] Yann LeCun, Yoshua Bengio, *Convolutional Networks for Images, Speech and Time-Series*, The MIT Press, 1998

[4] Tamara Kolda, Brett Bader, *Tensor Decompositions and Applications*, SIAM Review, 2009

[5] Nicholas Sidiropoulos, Lieven De Lathauewer, Xiao Fu, Kejun Huang, Evangelos Papalexakis, Christos Faloutsos, *Tensor Decomposition for Signal Processing and Machine Learning*, 2016

[6] Kurt Hornik, Maxwell Stinchcombe, Halbert White, *Multilayer feedforward networks are universal approximators*, Neural Networks, 1989

[7] Sho Sonoda, Noboru Murata, *Neural Network with Unbounded Activation Functions is Universal Approximator*, 2015

[8] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, Klaus-Robert Mülle, *Efficient BackProp*, Springer-Verlag Berlin Heidelberg, 2012

[9] George Cybenko, *Approximation by Superposition of a Sigmoidal Function*, Mathematics of Control Signal and Systems, 1989

[10] Sashank J. Reddi, Satyen Kale, Sanjiv Kumar, *On the convergence of ADAM and beyond*, https://openreview.net/pdf?id=ryQu7f-RZ, 2018

[11] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, Benjamin Recht, *The Marginal Value of Adaptive Gradient Methods in Machine Learning*, arXiv:1705.08292, 2017

[12] Diederik Kingma, Jimmy Lei Ba, *Adam: A Method for Stochastic Optimization*, arXiv:1412.6980, 2014

[13] *SGD or Adam?? Which One Is The Best Optimizer: Dogs-VS-Cats Toy Experiment*, https://shaoanlu.wordpress.com/2017/05/29/sgd-all-which-one-is-the-best-optimizer-dogs-vs-cats-toy-experiment/, 2018

[14] Sebastian Bock, Josef Goppold, Martin Weiss, *An improvement of the convergence proof of the ADAM-Optimizer*, arXiv:1804.10587, 2018

[15] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton,*ImageNet Classification with Deep Convolutional Neural Networks*, NIPS, 2012