Larissa Kawano Mori

Considerations on the architecture of multilayer neural networks

Brasília

2017

Larissa Kawano Mori

Considerations on the architecture of multilayer neural networks

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharela em Economia pela Faculdade de Administração Contabilidade e Economia - FACE da Universidade de Brasília - UnB.

Universidade de Brasília - UnB

Faculdade de Administração, Contabilidade e Economia - FACE

Graduação em Economia

Orientador: Daniel Oliveira Cajueiro

Brasília

2017

Larissa Kawano Mori

Considerations on the architecture of multilayer neural networks

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharela em Economia pela Faculdade de Administração Contabilidade e Economia - FACE da Universidade de Brasília - UnB.

Brasília, 6 de julho de 2017:

Monografia aprovada.

Brasília
2017

Abstract

We are interested in finding optimal multilayer neural network architectures in terms of the number of hidden layers and nodes in each layer. Since the behavior of a multilayer neural network is very complex due to the existence of many nonlinearities, high dimensionality and high connectivity between the nodes, most of the research in this topic is concentrated in understanding the behavior of neural networks with a single hidden layer. So, we intend to contribute to the understanding of neural networks with multiple hidden layers, which have until now been mostly studied to advance results in applications in complex tasks such as digital image processing and speech recognition. In this work we add to the scientific knowledge some insights on how the network architecture influences the generalization performance of the algorithm and how the initial conditions may influence the search for optimal solutions over the cost function hypersurface and indirectly provide information about the existence of multiple local minima or optimal solutions.

Keywords: multilayer perceptrons, neural networks, deep learning, machine learning, number of hidden layers, initial conditions.

Sumário

Introduction

This work started with the following question: "How does the architecture of the neural network affect its generalization performance?". We all know about the recent developments accompanying the resurging field of deep learning since the early 2000s, which have demonstrated the amazing ability of neural networks to process huge amounts of data and make accurate predictions in image processing and speech recognition tasks, for example. But most of the time, these algorithms are used as black boxes and not only the fine-tuning, but also the general design of the network's architecture is usually performed manually.

Even though many studies have been conducted on the structural properties of a neural network during the 1990s (Scarceli and Tsoi, 1997; Yoneyama and Camargo, 2001), most of the research in this topic was concentrated on finding an optimal architecture for neural networks with a single hidden layer. However, after the mid-1990s this topic lost popularity, along with the field of neural networks (Goodfellow, 2016, p.18). Though neural networks became popular again in the beginning of the 2000s, the attention has been devoted mostly in applications involving huge quantities of data, such as in image processing, recommendation systems, etc.

Thus, the theoretical understanding on how computations deliver such impressive results has not evolved in the same pace as the practical applications and many of the inner processes that allow the machines to learn are still seen as almost mystical.The purpose of this work is to make a tiny dent into this unexplored area by analyzing how the architecture of multilayer perceptrons influence the capacity of the algorithm to make better generalizations. Especially, we are interested in exploring the determination of the optimal number of hidden layers for a multilayer neural network to approximate a general set of functions.

This idea is related to the *Universal Approximation Theorem*, that states that there exists a number of hidden layers that, satisfied some conditions, can approximate any continuous function. However, the theorem does not tell us how to find the optimal number of layers. Also, Haykin (Haykin, 2009, p.167) states that "(...) the theorem states that a *single hidden layer is sufficient for a multilayer perceptron to compute a uniform* ϵ *approximation to a given training set represented by the set of inputs* x_1, x_2, ..., x_{m_0} and a desired (target) output $f(x_1, x_2, ..., x_{m_0})$". However the theorem does not say that a single hidden layer is optimum in learning time, ease of implementation and (most importantly) generalization."

Multilayer neural networks are very difficult to study because the nonlinearities and the high level of connectivity between the nodes make their behavior very complex. In this initial work, we decided to implement a neural network from scratch to learn a sine function, which is relatively simple, but nonlinear, so we could not learn it with simpler algorithms such as a linear regression. In order to experiment with different neural network structures in terms of number of hidden layers, nodes per layer, initial weights and different training sets, we made an object oriented program that takes as input a neural network structure and implements batch learning with vectorized computation.

Since putting together these pieces was not a simple task, we have dedicated a chapter to explain the matrix operations that are being performed in the back-propagation algorithm and tried to add intuition about the workings of a neural network and also to provide a brief explanation of our code and the basic ideas behind its implementation.

The basic structure of this work is organized as follows:

- Chapter 1 makes an introduction to multilayer neural networks to establish a common ground on basic machine learning and notation for the rest of the text and introduces the formal model of the multilayer neural network and the back-propagation algorithm used for training the neural network.
- Chapter 2 presents some of the results we obtained during training and testing our neural network for various different architectures.
- Chapter 3 shows how the initial weights and the set of training examples used may influence the training of the neural network.

Finally, we present a conclusion of our study and observations and indicate possible paths for future projects.

1 The multilayer neural network

1.1 Basic concepts

The model of a neural network (also called artificial neural network) is inspired by the biological structure of the neurons and synapses in the brain. It is composed by nodes (or neurons), where each node makes a linear combination of its inputs and then applies a nonlinear differentiable activation function (usually the sigmoid or the hyperbolic tangent). When a neural network contains one or more layers that are *hidden* from the input and output layers, it is called a multilayer neural network.

According to Haykin (Haykin, 2009, p. 123), the same reasons that make the neural network a powerful learning tool are the ones that make its behavior hard to understand, especially because of difficulties **(i)** in the theoretical analysis, due to the nonlinearities and the high level of connectivity; and **(ii)** in the visualization of the learning process because of the existence of hidden layers.

In Figure 1 below, we illustrate the operations that are performed in a single neuron:

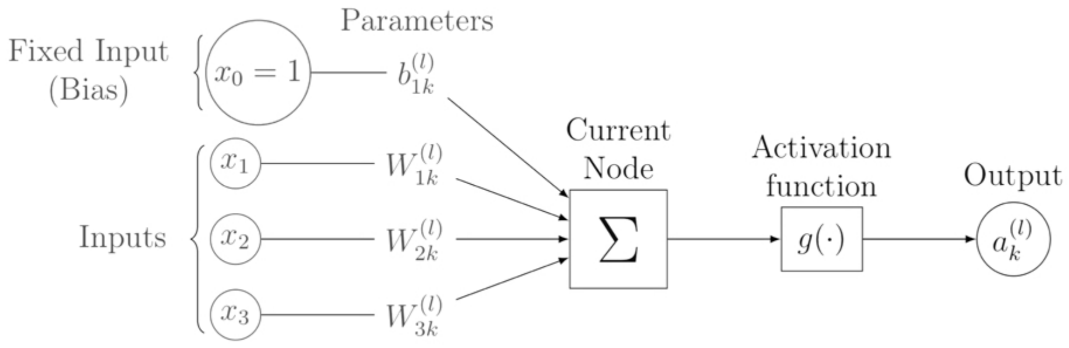


Figura 1: Detail of the operations performed in each node.

The node considered is the kth node of the (l)th hidden layer. Here, each input x_n, $n = 1, 2, 3, ..., n_0$, is pondered by its corresponding weight $W_{nk}^{(l)}$. The gray circle $x_0 = 1$ is only a computational element added to the input layer and to each hidden layer to be multiplied by the bias $b_{1k}^{(l)}$. Adding these elements, we get a linear combination of the inputs, to which we apply an activation function denoted by $g(\cdot)$ and then this output $a_k^{(l)}$ is fed to the next hidden layer or the output layer.

The mathematical model for these operations, where $z^{(l)}$ is the linear combination of the inputs plus the bias and $a_k^{(l)}$ is the result of the activation function applied to $z^{(l)}$, is the following:

$$z_k^{(l)} = b_{1k}^{(l)} + \sum_{n=1}^{n_0} W_{nk}^{(l)} x_n, \tag{1.1}$$

and

$$a_k^{(l)} = g(z^{(l)}) = g(b_{1k}^{(l)} + \sum_{n=1}^{n_0} W_{nk}^{(l)} x_n, \tag{1.2}$$

where we note that k and (l) are fixed for a specific node in a specific layer, respectively.

So, the output of a multilayer neural network is the result of the successive combination of the operations in various nodes over a set of inputs. In Figure 4, we show the general structure of a multilayer neural network, considering that the inputs of the network x_1, x_2, ..., x_{n_0} are the features used to calculate the estimate $\hat{y}_1$, $\hat{y}_2$, ..., $\hat{y}_{n_j}$.

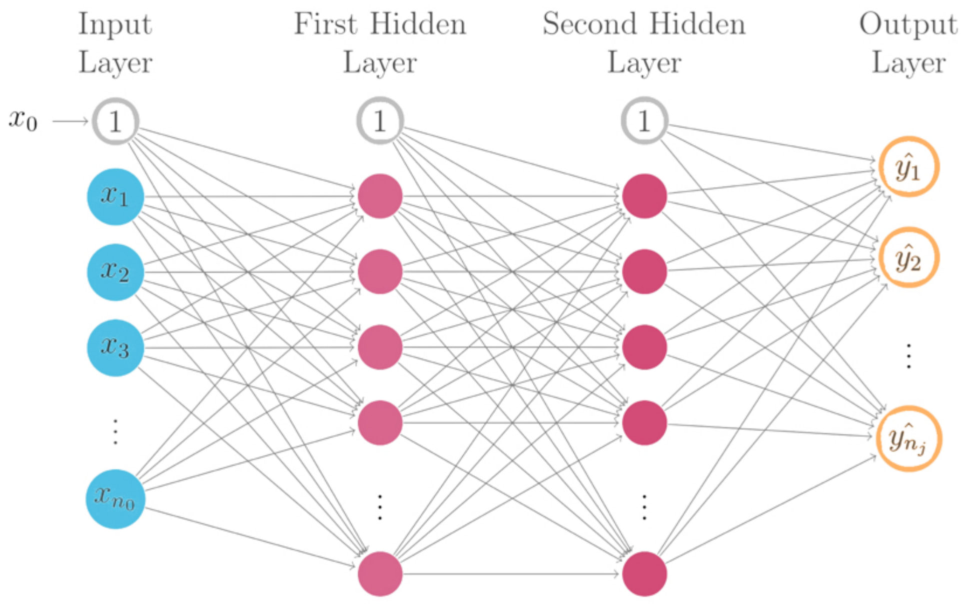


Figura 2: General structure of a multilayer neural network.

This is a supervised learning algorithm, which means that we have a teacher telling the algorithm what are the desired output values for a set of input values. Thus, we feed the algorithm with a sample with m examples with labeled target values for a set of input values. The objective is to minimize a cost function J, which in this case is given by the mean squared error (MSE)

$$J(x, W, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(\sum_{j=1}^{n_{output}} (\hat{y}_{ij} - y_{ij})^2 \right), \tag{1.3}$$

where i is the number of training examples and j the number of output nodes. Note that the cost function depends on the values of the variables x, W and b, which represent the inputs, weights and biases of the network, respectively.

Throughout the text, we use the notation $[1, n_1, n_2, ..., n_{hl}, 1]$ to indicate the architecture of the network in terms of the numbers of nodes in each layer. Note that the number of nodes in the first layer and the last layer are fixed to 1, because our input and output are one dimensional. Besides, the letters n_i, $i = 1, ..., hl$, indicate the number of nodes in the ith hidden layer, where hl represents the number of hidden layers.

The main algorithm used for training a neural network is the back-propagation algorithm, which is an iterative algorithm that uses the chain rule to update the parameters of the neural network by minimizing a cost function. Since this is not a very straightforward algorithm, especially in its backward phase, we will explore the topic in our next section.

1.2 The back-propagation algorithm

Because of our intention to dissect the inner workings of multilayer perceptrons, we would like to further analyze the vectorized implementation of the back-propagation algorithm. Our hope is that the reader acquires more insight on the matrix operations and becomes more acquainted with the many steps involved in this computation.

Thus, we present below all the computations for one iteration to train a simple neural network that learns the sine function. In this context, learning a function means estimating the parameters of the network from a set of labeled training examples to minimize the cost function and approximate the value of the function for any input values.

Consider we start with a data set of m training examples with their corresponding targets. The input matrix for a general network, as illustrated in Figure 2, has the following form:

$$\mathbf{x} = \begin{bmatrix} x_{1_{11}} & x_{2_{11}} & \dots & x_{n_{0 11}} \\ x_{1_{21}} & x_{2_{21}} & \dots & x_{n_{0 21}} \\ \vdots & \vdots & \dots & \vdots \\ x_{1_{m1}} & x_{2_{m1}} & \dots & x_{n_{0 m1}} \end{bmatrix}_{m \text{ x } n_0} , \tag{1.4}$$

where m equals the number of training examples and n_0 the number of features. So, each entry indicates the value of the mth example in the data set for the nth feature of the network.

Since we are learning the sine function, our neural network will have only one input feature x_1 $(n = 1)$, which we define as a real number in the interval $[0, 2\pi]$, and one output $\hat{y}_1$, which will be the approximation of the target value $y_1 = sin(x_1)$.

Suppose that $x_{1_{11}} = \dfrac{\pi}{6}$ and $x_{1_{21}} = \dfrac{3\pi}{2}$ are two training examples obtained from a

$[0, 2\pi]$ interval, then our input matrix is:

$$\mathbf{x} = \begin{bmatrix} x_{1_{11}} \\ x_{1_{21}} \end{bmatrix} = \begin{bmatrix} \frac{\pi}{6} \\ \frac{3\pi}{2} \end{bmatrix}_{2 \text{ x } 1} . \tag{1.5}$$

Using that $sin\left(\frac{\pi}{6}\right) = \frac{1}{2}$ and $sin\left(\frac{3\pi}{2}\right) = -1$, we will update the parameters of our neural network, so we can approximate the sine of any x in $[0, 2\pi]$ interval (since the algorithm is designed to work with interpolation) by minimizing the cost function J, which in this case is given by:

$$J(x_1, W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y_{i1}} - y_{i1})^2, \tag{1.6}$$

where we follow the notation used in Equation (1.3). However, differently from Equation (1.3), we do not sum the squared error over each output node j, because we only have one output $\hat{y}_1$, thus we calculate the cost only over the sum of the squared errors for each example i in the training set.

In the next sections, we use a neural network with an architecture [1, 2, 2, 1] (i.e., two hidden layers with two nodes in each layer) and show the computations for the forward and backward phases of the back-propagation algorithm.

1.2.1 The forward phase

It is important to note that, in a vectorized implementation, all the computations in the figure are done concomitantly to all examples $i = 1, 2, ..., m$. Besides, the biases and weights applied to each node are the same, independently of the example calculated. In this case, we use two examples ($i = 1, 2$) but we are keeping the matrix dimensions in an abstract notation to allow a posterior generalization for an arbitrary number of m examples. In Figure 3, we present an illustration of how our neural network is structured and establish the notation that will be used in the rest of the chapter.

The forward propagation algorithm is calculated following these steps:

First, we randomly initialize the elements of the matrices $b^{(1)}$ and $W^{(1)}$ from a normal distribution $N(0, 1)$. Note that the number of lines of the weight vectors $W^{(j)}$ are determined by the number of nodes in the previous layer $j - 1$ and the number of columns is determined by the number nodes in the current layer j, so its dimensions are n_{j-1} x n_j:

$$b^{(1)} = \begin{bmatrix} b_{11}^{(1)} & b_{12}^{(1)} \end{bmatrix}_{n_0 \text{ x } n_1} = \begin{bmatrix} -0.8044583 & 0.32093155 \end{bmatrix}_{n_0 \text{ x } n_1}, \tag{1.7}$$

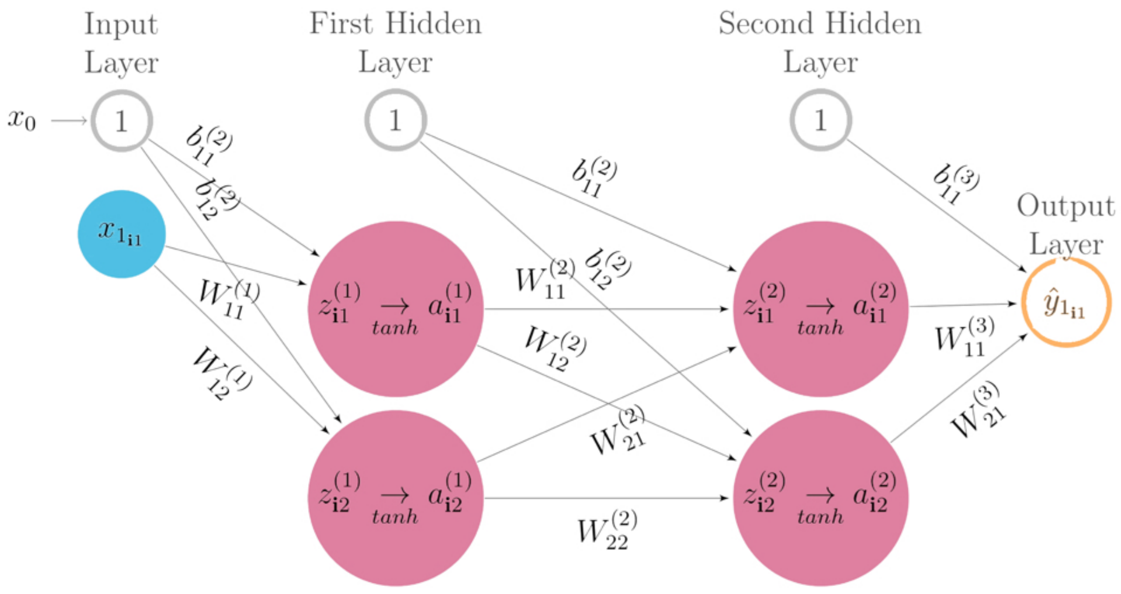


Figura 3: Illustration of the computations of the neural network, where i denotes the example in the training set that is being computed. Note that the biases and weights applied to each node are the same, independently of the example calculated.

$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} \end{bmatrix}_{n_0 \text{ x } n_1} = \begin{bmatrix} -0.02548288 & 0.64432383 \end{bmatrix}_{n_0 \text{ x } n_1}, \tag{1.8}$$

where n_j indicates the number of nodes in the jth hidden layer, so in this case n_1 indicates the number of nodes in the first hidden layer.

Then we obtain the matrix $z^{(1)}$, which is a matrix in which each element is a linear combination of the elements in x. To do this, we multiply the matrices x_{ones}, obtained by adding a column of ones to x, and $W_{temp}^{(1)}$, obtained by stacking the matrix $b^{(1)}$ over $W^{(1)}$, so the column of ones will be multiplied by the bias elements in $b^{(1)}$, as seen below:

$$z^{(1)} = x_{ones} \cdot W_{temp}^{(1)} = \begin{bmatrix} 1 & x_{11} \\ 1 & x_{21} \end{bmatrix}_{m \text{ x } (n_0+1)} \begin{bmatrix} b_{11}^{(1)} & b_{12}^{(1)} \\ W_{11}^{(1)} & W_{12}^{(1)} \end{bmatrix}_{(n_0+1) \text{ x } n_1} \tag{1.9}$$

$$= \begin{bmatrix} 1 & \frac{\pi}{6} \\ 1 & \frac{3\pi}{2} \end{bmatrix}_{2 \text{ x } 2} \begin{bmatrix} -0.8045 & 0.3209 \\ -0.0255 & 0.6443 \end{bmatrix}_{2 \text{ x } 2} \tag{1.10}$$

Multiplying the matrices we then have that:

$$z^{(1)} = \begin{bmatrix} b_{11}^{(1)} + W_{11}^{(1)} x_{11} & b_{12}^{(1)} + W_{12}^{(1)} x_{11} \\ b_{11}^{(1)} + W_{11}^{(1)} x_{21} & b_{12}^{(1)} + W_{12}^{(1)} x_{21} \end{bmatrix}_{m \text{ x } n_1} \tag{1.11}$$

$$= \begin{bmatrix} -0.8045 - 0.0255 \cdot \frac{\pi}{6} & 0.3209 + 0.6443 \cdot \frac{\pi}{6} \\ -0.8045 - 0.0255 \cdot \frac{3\pi}{2} & 0.3209 + 0.6443 \cdot \frac{3\pi}{2} \end{bmatrix}_{2 \times 2} \tag{1.12}$$

Finally, we get the output matrix $a^{(1)}$ of the first hidden layer by applying the activation function g to $z^{(1)}$:

$$a^{(1)} = \begin{bmatrix} g(z_{11}^{(1)}) & g(z_{12}^{(1)}) \\ g(z_{21}^{(1)}) & g(z_{22}^{(1)}) \end{bmatrix}_{m \times n_1}, \tag{1.13}$$

$$= \begin{bmatrix} g(-0.8178) & g(0.6583) \\ g(-0.9245) & g(3.3572) \end{bmatrix}_{2 \times 2} \approx \begin{bmatrix} -0.6739 & 0.5772 \\ -0.7280 & 0.9976 \end{bmatrix}_{2 \times 2} \tag{1.14}$$

where g represents the activation function. We apply the hyperbolic tangent as the activation function, so that $g = tanh$.

The same procedure is applied to the second hidden layer, noting that the dimensions of the weights matrix $W^{(2)}$ will change, since now we have two nodes providing the input to other two other nodes, whereas in the first hidden layer we had only one dimensional inputs. Remember that we are working with two examples, but they are both examples for the only feature we have in this case, which is x_1.

In order to better illustrate how we determine the dimension of our parameters, we will see how the bias matrix $b^{(2)}$ and the weights matrix $W^{(2)}$ are initialized so that their dimensions agree with the dimensions of $a^{(1)}$ for matrix multiplication:

$$b^{(2)} = \begin{bmatrix} b_{11}^{(2)} & b_{12}^{(2)} \end{bmatrix}_{1 \times n_2} = \begin{bmatrix} -0.30079667 & 0.38947455 \end{bmatrix}_{1 \times 2}, \tag{1.15}$$

$$W^{(2)} = \begin{bmatrix} W_{11}^{(2)} & W_{12}^{(2)} \\ W_{21}^{(2)} & W_{22}^{(2)} \end{bmatrix}_{n_1 \times n_2} = \begin{bmatrix} -0.1074373 & -0.47998308 \\ 0.5950355 & -0.46466753 \end{bmatrix}_{2 \times 2}. \tag{1.16}$$

Again, we stack $b^{(2)}$ over $W^{(2)}$ and multiply this new matrix by $a^{(1)}$ added by a column of ones to obtain the matrix $z^{(2)}$, which is the linear combination of the outputs of the first hidden layer:

$$z^{(2)} = a^{(1)}_{ones} \cdot W^{(2)}_{temp} = \begin{bmatrix} 1 & g(z^{(1)}_{11}) & g(z^{(1)}_{12}) \\ 1 & g(z^{(1)}_{21}) & g(z^{(1)}_{22}) \end{bmatrix}_{m \text{ x } (n_1+1)} \begin{bmatrix} b^{(2)}_{11} & b^{(2)}_{12} \\ W^{(2)}_{11} & W^{(2)}_{12} \\ W^{(2)}_{21} & W^{(2)}_{22} \end{bmatrix}_{(n_1+1) \text{ x } n_2} \tag{1.17}$$

$$\approx \begin{bmatrix} 1 & -0.67387 & 0.57723 \\ 1 & -0.72804 & 0.99758 \end{bmatrix}_{2 \text{ x } 2} \begin{bmatrix} -0.3008 & 0.3895 \\ -0.1074 & -0.4800 \\ 0.5950 & -0.4647 \end{bmatrix}_{3 \text{ x } 2} . \tag{1.18}$$

Calculating the matrix multiplication we get that:

$$z^{(2)} = \begin{bmatrix} b^{(2)}_{11} + W^{(2)}_{11}[g(z^{(1)}_{11})] + W^{(2)}_{21}[g(z^{(1)}_{12})] & b^{(2)}_{12} + W^{(2)}_{12}[g(z^{(1)}_{11})] + W^{(2)}_{22}[g(z^{(1)}_{12})] \\ b^{(2)}_{11} + W^{(2)}_{11}[g(z^{(1)}_{21})] + W^{(2)}_{21}[g(z^{(1)}_{22})] & b^{(2)}_{12} + W^{(2)}_{12}[g(z^{(1)}_{21})] + W^{(2)}_{22}[g(z^{(1)}_{22})] \end{bmatrix}_{m \text{ x } n_2} \tag{1.19}$$

$$= \begin{bmatrix} -0.301 - 0.107[-0.67387] + 0.595[0.57723] & 0.389 - 0.48[-0.67387] - 0.465[0.57723] \\ -0.301 - 0.107[-0.72804] + 0.595[0.99758] & 0.389 - 0.48[-0.72804] - 0.465[0.99758] \end{bmatrix}_{2 \text{ x } 2} . \tag{1.20}$$

$$= \begin{bmatrix} 0.11507464 & 0.44470128 \\ 0.37101538 & 0.27537991 \end{bmatrix}_{2 \text{ x } 2} \tag{1.21}$$

Applying the activation function g to $z^{(2)}$, we obtain $a^{(2)}$, which is the matrix output of the second hidden layer:

$$a^{(2)} = \begin{bmatrix} g(z^{(2)}_{11}) & g(z^{(2)}_{12}) \\ g(z^{(2)}_{21}) & g(z^{(2)}_{22}) \end{bmatrix}_{m \text{ x } n_2} = \begin{bmatrix} g(0.11507464) & g(0.44470128) \\ g(0.37101538) & g(0.27537991) \end{bmatrix}_{2 \text{ x } 2} \tag{1.22}$$

$$= \begin{bmatrix} 0.11456937 & 0.41753373 \\ 0.35487954 & 0.26862372 \end{bmatrix}_{2 \text{ x } 2} \tag{1.23}$$

Finally, we initialize the values of the matrices $b^{(3)}$ and $W^{(3)}$ from a normal distribution $N(0, 1)$ and doing the same procedure of the first and second hidden layers, we calculate $z^{(3)}$, which corresponds to our estimate $\hat{Y}$, since we are not applying the activation function to the output, as a convention:

$$z^{(3)} = \hat{Y} = \begin{bmatrix} 1 & g(z_{11}^{(2)}) & g(z_{12}^{(2)}) \\ 1 & g(z_{21}^{(2)}) & g(z_{22}^{(2)}) \end{bmatrix}_{m \text{ x } (n_2+1)} \begin{bmatrix} b_{11}^{(3)} \\ W_{11}^{(3)} \\ W_{21}^{(3)} \end{bmatrix}_{(n_2+1) \text{ x } n_{output}} \tag{1.24}$$

$$= \begin{bmatrix} 1 & 0.11456937 & 0.41753373 \\ 1 & 0.35487954 & 0.26862372 \end{bmatrix}_{2 \text{ x } 3} \begin{bmatrix} 0.66728131 \\ -0.80611561 \\ -1.19606983 \end{bmatrix}_{3 \text{ x } 1} \tag{1.25}$$

Multiplying the matrices above we get:

$$z^{(3)} = \begin{bmatrix} b_{11}^{(3)} + W_{11}^{(3)}[g(z_{11}^{(2)})] + W_{21}^{(3)}[g(z_{12}^{(2)})] \\ b_{11}^{(3)} + W_{11}^{(3)}[g(z_{21}^{(2)})] + W_{21}^{(3)}[g(z_{22}^{(2)})] \end{bmatrix}_{m \text{ x } n_{output}} \tag{1.26}$$

$$= \begin{bmatrix} 0.66728131 - 0.80611561[0.11456937] - 1.19606983[0.41753373] \\ 0.66728131 - 0.80611561[0.35487954] - 1.19606983[0.26862372] \end{bmatrix}_{2 \text{ x } 1} = \begin{bmatrix} 0.07552565 \\ 0.05991465 \end{bmatrix}_{2 \text{ x } 1} \tag{1.27}$$

where n_{output} representes the dimension of the output, which in this case equals 1.

This result means that we have estimated a value of 0.0755 for the sine of $\frac{\pi}{6}$ and 0.0599 for the sine of $\frac{3\pi}{2}$, which are clearly distant from the target values (0.5 and 1, respectively). We proceed to the next phase and update the parameters of the neural network (biases and weights) to improve the performance of the model in estimating the sine function.

1.2.2 The backward phase

In the backward phase of the back-propagation algorithm, we calculate the derivative of the cost function with respect to each parameter in the hidden layers of the neural network and update them to minimize cost. In this phase, we use the target values $sin\left(\frac{\pi}{6}\right) = 0.5$ and $sin\left(\frac{3\pi}{2}\right) = -1$ and the estimate $\hat{Y}$ obtained above to calculate the variables Δ_l, $l = 3, 2, 1$, which represent the derivative of the cost function with respect to the element $z^{(l)}$. The targets matrix corresponds to

$$Y = \begin{bmatrix} y_{11} \\ y_{21} \end{bmatrix}_{m \text{ x } n_f} = \begin{bmatrix} 0.5 \\ -1 \end{bmatrix}_{m \text{ x } n_f}, \tag{1.28}$$

and the error matrix is given by:

$$\text{error} = \begin{bmatrix} \hat{y}_{11} - y_{21} \\ \hat{y}_{21} - y_{21} \end{bmatrix}_{m \text{ x } n_f} = \begin{bmatrix} 0.07552565 - 0.5 \\ 0.05991465 - (-1) \end{bmatrix}_{2 \text{ x } 1} = \begin{bmatrix} -0.42447435 \\ 1.05991465 \end{bmatrix}_{2 \text{ x } 1} . \tag{1.29}$$

Since we did not apply the activation function to the output node in the forward phase, Δ_3 is only the difference between targets and estimated values.

$$\Delta_3 = \text{error} = \begin{bmatrix} e_{11} \\ e_{21} \end{bmatrix} = \begin{bmatrix} -0.42447435 \\ 1.05991465 \end{bmatrix}_{2 \text{ x } 1} \tag{1.30}$$

Because we only observe the output error, we calculate the derivatives of the cost function in the hidden layers by propagating the error values through the network in the backwards direction. In the calculation of Δ_1 and Δ_2 we apply the chain rule of calculus, pondering each element of the Δ's by the weights applied in the forward propagation.

$$\Delta_2 = \left[\Delta_3 \cdot W^{(3)^T}\right] .^* dg(z^{(2)} = \left[\begin{bmatrix} e_{11} \\ e_{21} \end{bmatrix}_{m \text{ x } n_f} \begin{bmatrix} W_{11}^{(3)} & W_{21}^{(3)} \end{bmatrix}_{n_f \text{ x } n_2}\right] .^* \begin{bmatrix} dg(z_{11}^{(2)}) & dg(z_{12}^{(2)}) \\ dg(z_{21}^{(2)}) & dg(z_{22}^{(2)}) \end{bmatrix}_{m \text{ x } n_2} \tag{1.31}$$

$$= \left[\begin{bmatrix} -0.42447435 \\ 1.05991465 \end{bmatrix}_{2 \text{ x } 1} \begin{bmatrix} -0.80611561 & -1.19606983 \end{bmatrix}_{n_f \text{ x } n_2}\right] .^* \begin{bmatrix} 0.99741028 & 0.73111831 \\ 0.37101538 & 0.92784129 \end{bmatrix}_{m \text{ x } n_2} , \tag{1.32}$$

where $.^*$ denotes element-wise multiplication.

Calculating the matrix multiplication and the element-wise multiplication in our example we get:

$$\Delta_2 = \begin{bmatrix} W_{11}^{(3)} \cdot e_{11} \cdot dg(z_{11}^{(2)}) & W_{21}^{(3)} \cdot e_{11} \cdot dg(z_{12}^{(2)}) \\ W_{11}^{(3)} \cdot e_{21} \cdot dg(z_{21}^{(2)}) & W_{21}^{(3)} \cdot e_{21} \cdot dg(z_{22}^{(2)}) \end{bmatrix}_{m \text{ x } n_2} \tag{1.33}$$

$$= \begin{bmatrix} -0.91340755 & -0.76420018 \\ -0.80899242 & -0.85876958 \end{bmatrix}_{m \text{ x } n_2} . \tag{1.34}$$

Finally, we calculate Δ_3 below, so we can proceed to the calculations of the derivatives of the cost function J with respect to the weights $W^{(3)}$, $W^{(2)}$ e $W^{(1)}$:

$$\Delta_1 = \left[\Delta_2 \cdot W^{(2)^T}\right] .^* dg(z^{(1)}) = \left[\begin{bmatrix} \Delta_{2_{11}} & \Delta_{2_{12}} \\ \Delta_{2_{21}} & \Delta_{2_{22}} \end{bmatrix}_{n_2 \text{ x } n_1} \begin{bmatrix} W_{11}^{(2)} & W_{21}^{(2)} \\ W_{12}^{(2)} & W_{22}^{(2)} \end{bmatrix}_{n_1 \text{ x } n_2}\right] .^* \begin{bmatrix} g(z_{11}^{(2)}) & g(z_{12}^{(2)}) \\ g(z_{21}^{(2)}) & g(z_{22}^{(2)}) \end{bmatrix}_{m \text{ x } n_2} \tag{1.35}$$

$$= \begin{bmatrix} [W_{11}^{(2)}\Delta_{2_{11}} + W_{12}^{(2)}\Delta_{2_{12}}]dg(z_{11}^{(1)}) & [W_{21}^{(2)}\Delta_{2_{11}} + W_{22}^{(2)}\Delta_{2_{12}}]dg(z_{12}^{(1)}) \\ [W_{11}^{(2)}\Delta_{2_{21}} + W_{12}^{(2)}\Delta_{2_{22}}]dg(z_{21}^{(1)}) & [W_{21}^{(2)}\Delta_{2_{21}} + W_{22}^{(2)}\Delta_{2_{22}}]dg(z_{22}^{(1)}) \end{bmatrix}_{m \text{ x } n_1} \tag{1.36}$$

$$= \begin{bmatrix} 0.25380805 & -0.12563341 \\ 0.23456118 & -0.00039861 \end{bmatrix}_{2 \text{ x } 2} \tag{1.37}$$

Finally we calculate the derivatives of the cost function with respect to the weights $W^{(3)}$, $W^{(2)}$ e $W^{(1)}$:

$$\frac{\partial J}{\partial W^{(3)}} = \frac{1}{m}\left[a_2^T \cdot \Delta_3\right] = \frac{1}{m} \begin{bmatrix} g(z_{11}^{(2)}) & g(z_{21}^{(2)}) \\ g(z_{12}^{(2)}) & g(z_{22}^{(2)}) \end{bmatrix}_{n_2 \text{ x } m} \begin{bmatrix} e_{11} \\ e_{21} \end{bmatrix}_{m \text{ x } n_{output}} \tag{1.38}$$

$$= \begin{bmatrix} e_{11} \cdot g(z_{11}^{(2)}) + e_{21} \cdot g(z_{21}^{(2)}) \\ e_{11} \cdot g(z_{12}^{(2)}) + e_{21} \cdot g(z_{22}^{(2)})] \end{bmatrix}_{n_2 \text{ x } n_{output}} \tag{1.39}$$

$$= \begin{bmatrix} 0.163755129547 \\ 0.0537429281873 \end{bmatrix}_{2 \text{ x } 1} \tag{1.40}$$

Note that the derivatives of the cost function with respect to W_3 are just the error values in the output layer pondered by the outputs for each node in the second hidden layer. Now we calculate the derivatives of the cost function with respect with W_2, which is the output of the first hidden layer pondered by

$$\frac{\partial J}{\partial W^{(2)}} = \frac{1}{m}\left[a_1^T \cdot \Delta_2\right] = \frac{1}{m} \begin{bmatrix} g(z_{11}^{(1)}) & g(z_{21}^{(1)}) \\ g(z_{12}^{(1)}) & g(z_{22}^{(1)}) \end{bmatrix}_{n_1 \text{ x } m} \begin{bmatrix} \Delta_{2_{11}} & \Delta_{2_{12}} \\ \Delta_{2_{21}} & \Delta_{2_{22}} \end{bmatrix}_{m \text{ x } n_2} \tag{1.41}$$

$$= \frac{1}{m} \begin{bmatrix} g(z_{11}^{(1)})\Delta_{2_{11}} + g(z_{21}^{(1)})\Delta_{2_{21}} & g(z_{11}^{(1)})\Delta_{2_{12}} + g(z_{21}^{(1)})\Delta_{2_{22}} \\ g(z_{12}^{(1)})\Delta_{2_{11}} + g(z_{22}^{(1)})\Delta_{2_{21}} & g(z_{12}^{(1)})\Delta_{2_{21}} + g(z_{22}^{(1)})\Delta_{2_{22}} \end{bmatrix}_{m \text{ x } n_2} \tag{1.42}$$

$$= \begin{bmatrix} 0.60224888 & 0.57009548 \\ -0.66713908 & -0.64890384 \end{bmatrix}_{2 \text{ x } 2} \tag{1.43}$$

Finally, we calculate the derivatives with respect to $W1$:

$$\frac{\partial J}{\partial W^{(1)}} = \frac{1}{m}\left[X^T \cdot \Delta_1)\right] = \frac{1}{m}\begin{bmatrix} x_{11} & x_{21} \end{bmatrix}_{n_0 \text{ x } m} \begin{bmatrix} \Delta_{1_{11}} & \Delta_{1_{12}} \\ \Delta_{1_{21}} & \Delta_{1_{22}} \end{bmatrix}_{m \text{ x } n_1} \tag{1.44}$$

$$= \frac{1}{m}\begin{bmatrix} [\Delta_{1_{11}} x_{11} + \Delta_{1_{21}} x_{21}] & [\Delta_{1_{12}} x_{11} + \Delta_{1_{22}} x_{21}] \end{bmatrix}_{n_0 \text{ x } n_1} \tag{1.45}$$

$$= \begin{bmatrix} 1.07001654888 & 0.247668537317 \end{bmatrix}. \tag{1.46}$$

This process is then repeated and iteratively the algorithm minimizes the cost function. In the next chapters, we make some observations about the stopping methods of the process and analyze how the initial conditions and the architecture of the neural network influence the learning time and the generalization capacity of the algorithm.

2 The architecture of a neural network

2.1 General observations on training a neural network

During the process of building and improving our neural network, we have acquired practical knowledge about the behavior of a neural network. So, we would like to introduce this chapter with some of the observations about non-zero mean inputs, the determination of the learning rate value, factors that influence learning time, etc.

First, one of the issues we confronted during the project was how to deal with cost functions whose graph presented plateaus and oscillations. In general, these phenomena do not represent a real threat to the learning task when they still allow the overall cost to decrease, but they can make the learning process longer. In the next figures, we show how the size of the learning rate α influences the effectiveness of the neural network.

In Figure 4, we see big oscillations in the cost function per iteration for $\alpha = 3$. This is a case of concern, since the algorithm is not capable to learn the function and the oscillations (zigzags) are not resulting in an overall smaller cost function in the end. The solution in a similar situation would be to reduce the learning rate α.

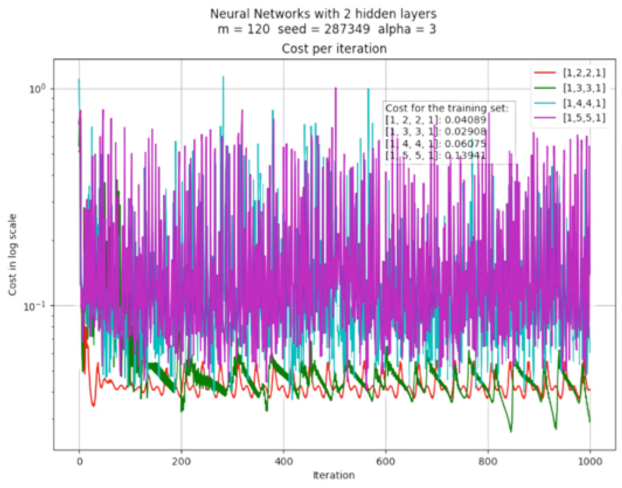


Figura 4: Neural Networks trained with learning rate $\alpha = 3$.

As we decrease the learning rate α, the graph of the cost function becomes more stable, as shown in Figures 5 and 6, where α is set equal to 0.5 and 0.05, respectively. In Figure 5, we zoom the behavior of the cost function between the 300-600 iterations in order to illustrate the oscillatory behavior of the the cost function from one iteration to another. One possible explanation is that the size of our learning step α is bigger than the distance of the current position to a local or the global minimum and we end up missing

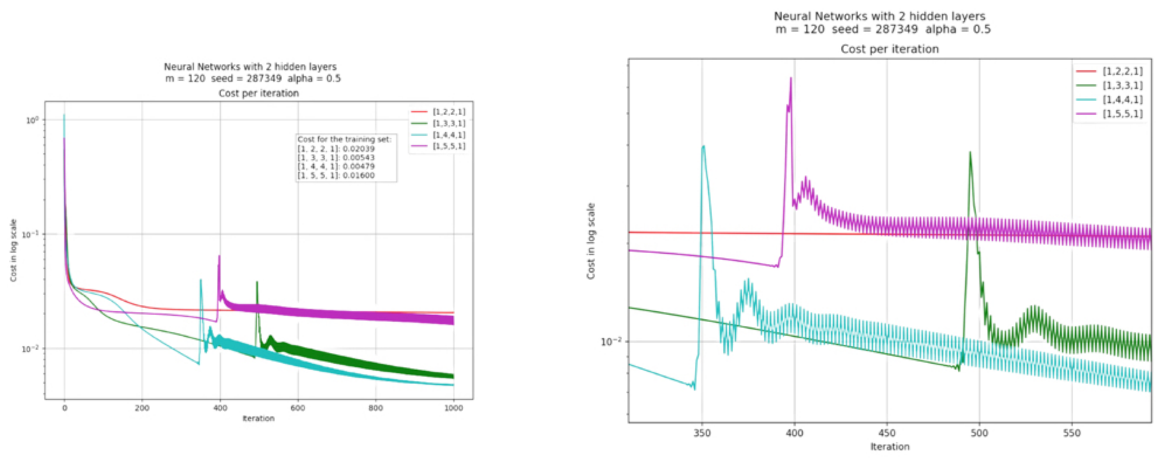


Figura 5: *Left*: Neural Networks trained with learning rate $\alpha = 0.5$. *Right*: Zoom in the region of 300-600 iterations.

the minimum and reaching a new position associated with a higher cost. However, since we are training our network with a gradient descent algorithm, the next step in the training will lead to a position with lower cost, which is why we can see this zigzag shape in Figure 5.

It is possible to reduce α and get smoother curves, though in general the network will require a greater number of iterations to achieve better results (and this can be costly). For example, for $\alpha = 0.05$, we have the graph in Figure 6.

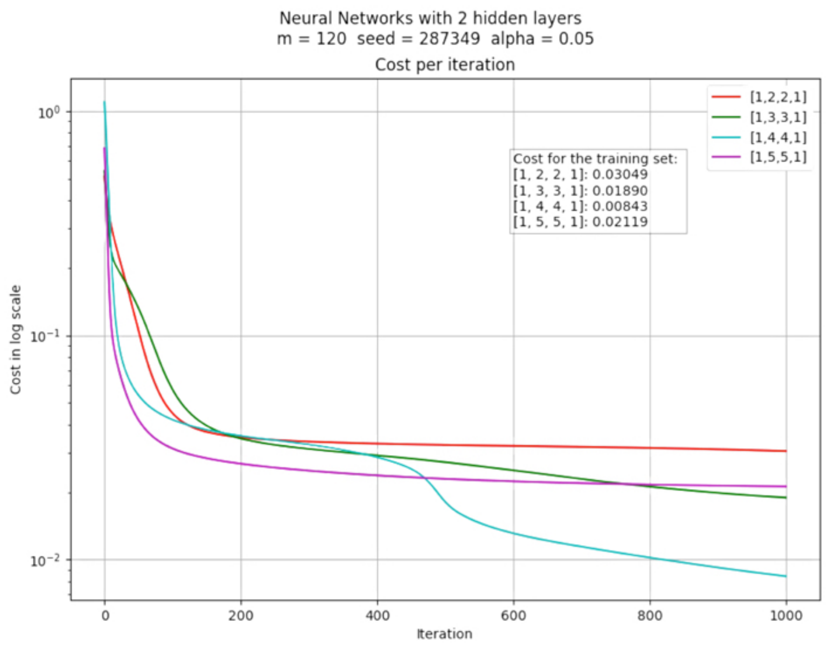


Figura 6: Neural networks trained with learning rate $\alpha = 0.05$.

However, there are no objective criteria to define the adequate value of α and the choices are usually made in a case-by-case analysis, by trying different values of α and analyzing the behavior of the cost function. In the Machine Learning MOOC in the Coursera platform, Andrew Ng advises to test different values of α in a range like 0.003, 0.01, 0.03, 0.1, ..., 0.33, 1 and maybe refine the search between the interval of the two best performing values. There are some algorithms to implement a dynamic learning rate but we are not developing this matter in this work (for more information, read Haykin, 2009, p. 158).

Another observation we made while training our neural network and testing for different scenarios was that using inputs with zero mean may improve significantly the time of training. As Haykin notes (Haykin, 2009, p. 156), this is a fact already established by empirical results, but we believe it would be interesting to show a concrete example here, since our results improved significantly after we have abided to it.

In Figures 7, we see that the same neural network takes more than 2000 times the number of iterations necessary to obtain similar training and test costs with inputs transformed to have zero-mean.

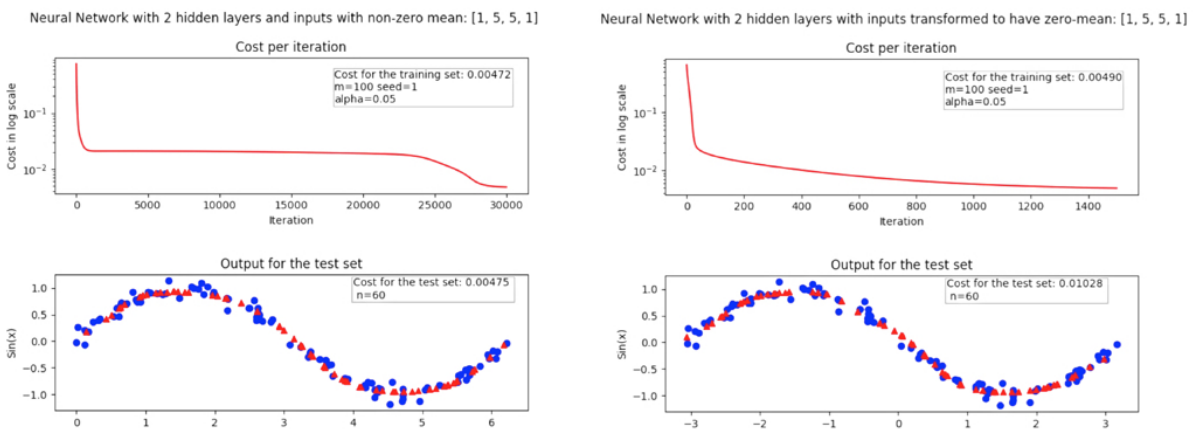


Figura 7: Comparison of the approximate number of iterations necessary to achieve training costs smaller than $\epsilon = 10^{-2}$ when $seed = 1$ and $\alpha = 0.05$. *Left:* Training set with non-zero mean. *Right:* Training set transformed to have zero mean.

Though this is a general recommendation, it does not mean that necessarily transforming the inputs to have zero mean will reduce the training time. In Figure 8, we show that both training sets (with and without zero mean) take the same number of iterations to reach a small training cost.

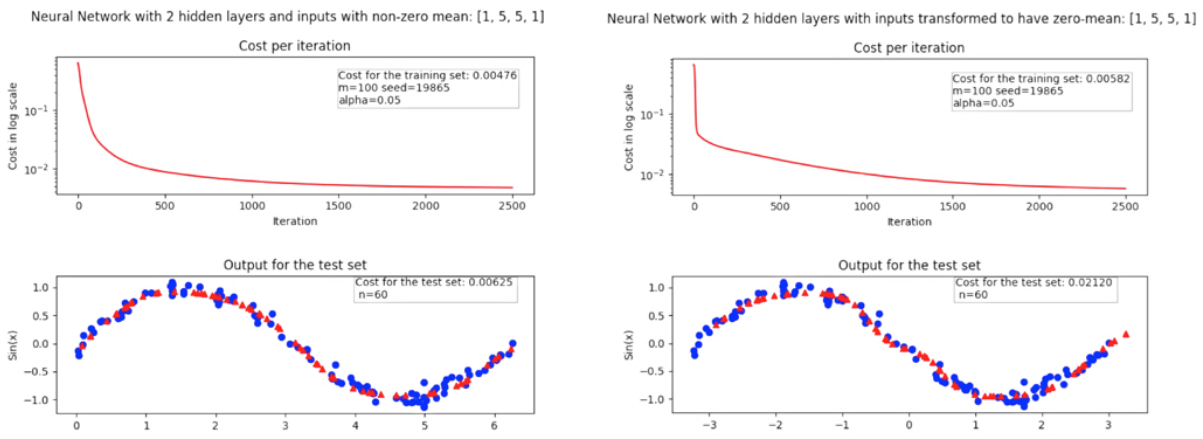
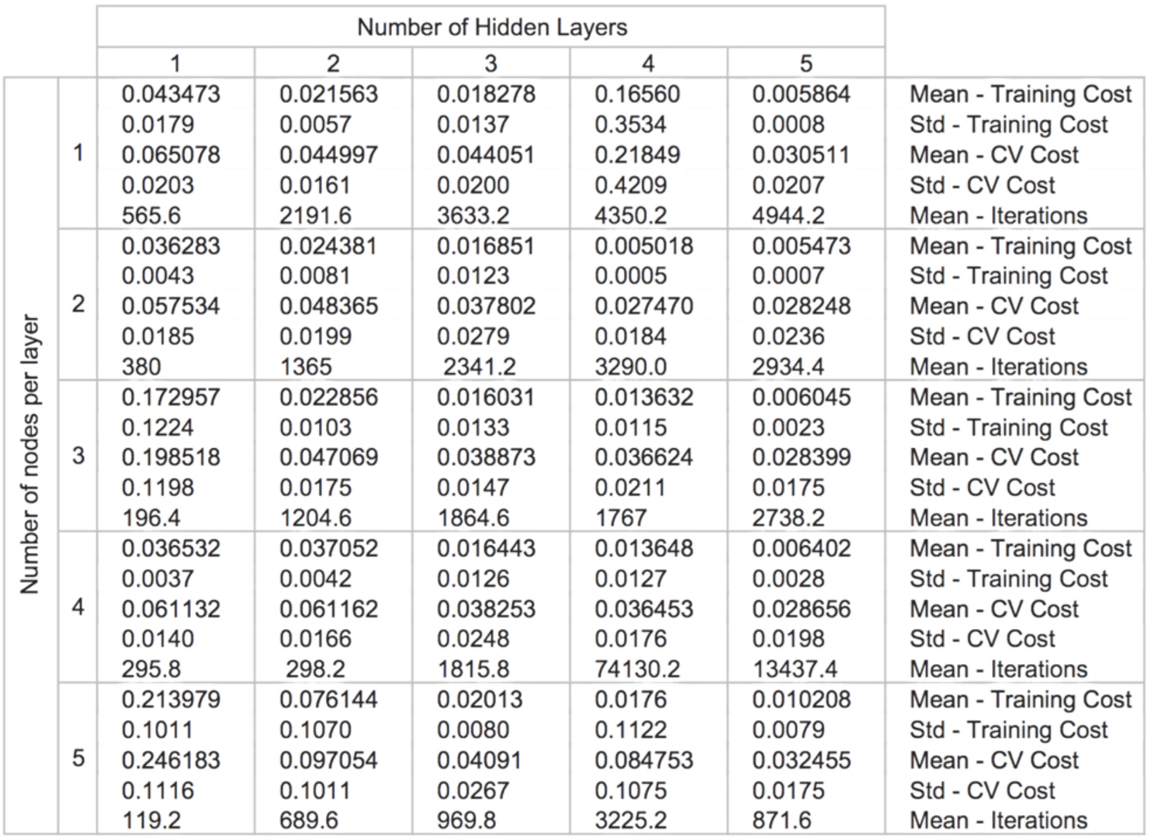


Figura 8: Comparison of the approximate number of iterations necessary to achieve training costs smaller than $\epsilon = 10^{-2}$ when $seed = 19865$ and $\alpha = 0.05$. *Left:* Training set with non-zero mean. *Right:* Training set transformed to have zero mean.

2.2 Number of hidden layers

According to Haykin (Haykin, 2009, p. 139), "In general the back-propagation algorithm cannot be shown to converge, and there are no well-defined criteria for stopping its iteration". He notes that there are some "reasonable criteria"to terminate the adjustments of the weights, which are mainly (Haykin, 2009 p.139): **(i)** when the norm of the derivatives of the cost function in relation to their weights is a sufficiently small value, **(ii)** when the change in the absolute value of the cost function per iteration is sufficiently small (typically in the range of 0.1 to 1% per iteration); and **(iii)** when the weights reach their maximum generalization capacity, tested by the cross-validation set.

We tested different network architectures varying from one to five hidden layers and one to five nodes per hidden layer using the stopping method when the percentual variation of the cost is less than 0.001% to see if there was a clear trend of an optimal architecture. We used five different sets of training examples and calculated the mean and the standard deviation of the final training costs and the final cross validation costs. Our results are presented in the table in Figure 9 below.



		Number of Hidden Layers					
		1	2	3	4	5	
Number of nodes per layer		0.043473	0.021563	0.018278	0.16560	0.005864	Mean - Training Cost
		0.0179	0.0057	0.0137	0.3534	0.0008	Std - Training Cost
	1	0.065078	0.044997	0.044051	0.21849	0.030511	Mean - CV Cost
		0.0203	0.0161	0.0200	0.4209	0.0207	Std - CV Cost
		565.6	2191.6	3633.2	4350.2	4944.2	Mean - Iterations
		0.036283	0.024381	0.016851	0.005018	0.005473	Mean - Training Cost
		0.0043	0.0081	0.0123	0.0005	0.0007	Std - Training Cost
	2	0.057534	0.048365	0.037802	0.027470	0.028248	Mean - CV Cost
		0.0185	0.0199	0.0279	0.0184	0.0236	Std - CV Cost
		380	1365	2341.2	3290.0	2934.4	Mean - Iterations
		0.172957	0.022856	0.016031	0.013632	0.006045	Mean - Training Cost
		0.1224	0.0103	0.0133	0.0115	0.0023	Std - Training Cost
	3	0.198518	0.047069	0.038873	0.036624	0.028399	Mean - CV Cost
		0.1198	0.0175	0.0147	0.0211	0.0175	Std - CV Cost
		196.4	1204.6	1864.6	1767	2738.2	Mean - Iterations
		0.036532	0.037052	0.016443	0.013648	0.006402	Mean - Training Cost
		0.0037	0.0042	0.0126	0.0127	0.0028	Std - Training Cost
	4	0.061132	0.061162	0.038253	0.036453	0.028656	Mean - CV Cost
		0.0140	0.0166	0.0248	0.0176	0.0198	Std - CV Cost
		295.8	298.2	1815.8	74130.2	13437.4	Mean - Iterations
		0.213979	0.076144	0.02013	0.0176	0.010208	Mean - Training Cost
		0.1011	0.1070	0.0080	0.1122	0.0079	Std - Training Cost
	5	0.246183	0.097054	0.04091	0.084753	0.032455	Mean - CV Cost
		0.1116	0.1011	0.0267	0.1075	0.0175	Std - CV Cost
		119.2	689.6	969.8	3225.2	871.6	Mean - Iterations

Figura 9: Comparison of the mean and the standard deviation of the final training and cross validation costs for neural network architectures with 1-5 hidden layers and 1-5 nodes per layer.

The table above gives us some hints on how the number of nodes in each hidden layer affect the final cross validation cost. We note that the best performing network had the structure [1, 2, 2, 2, 2, 1] (i.e., four hidden layers with two nodes in each layer) with a mean cross validation cost of approximately 0.027 and a mean of 3,290 iterations computed for training the network.

Even though we tend to believe that a greater number of nodes results in better approximations, independently of the number of layers, we could not extract a pattern from these results. Thus, we intend to further investigate this issue by using a larger number of data sets and also by applying the same tests for other stopping methods, so we can obtain more information and hopefully elaborate for more assertive conclusions.

3 Varying initial conditions

In this chapter we analyze how the initial values of the weights and the set of values used for training influence, *ceteris paribus* (i.e., maintaining all the other factors constant), the learning performance of the neural network. In the analysis, we fixed a learning rate $\alpha = 0.05$ and we stop the training process when the variation of the cost is less than 0.0001%. Also, we are using 120 training examples and 30 cross-validation examples, all of them consisting of pairs of (x, $sin(x)$+noise) where the $x's$ were obtained from an uniform distribution in the interval $[0, 2\pi]$ and the noise from a normal distribution $N(0, 0.1)$.

3.1 Different initial weights and a fixed set of training inputs

This is the main topic of interest of the study, since the behavior of the weights give an indirect indication on the structure of the cost hypersurface, which is multidimensional even for very small neural networks and thus its graph cannot be visualized. So, we need to search for other factors that may signalaze the existence (or not) of a global minimum, local minima or saddle points in the cost hypersurface, and their behavior, in case they exist.

For example, when the weights in the network are initialized with different values, for a fixed network architecture and fixed training inputs, what can we infer about the analysis of the final values of the weights obtained for the different sets? If the final values for each weight happen to be sufficiently close, we could further investigate the possibility that the weights converge to the same local minimum or to a global minimum, given a range of initial values. On the other hand, if the final values are very far from each other, that could be an indication of the existence of more than one local minimum.

To make valid comparisons, we have to make sure that we are stopping the training procedure when the weights are very close to their "asymptotic"value (in an informal sense) for each of the sets. Thus, we have also followed up the value of the derivatives of each of the weights during the training process to check if their absolute values were approximating zero.

In our first scenario, we have randomly initialized nine different sets of weights from a normal distribution $N(0, 1)$ and we have trained a [1, 4, 4, 1] neural network (i.e., with two hidden layers and four nodes in each hidden layer).

An important result we obtained with this experiment was that there may be as many local minima as there are different sets of initial weights obtained from a N(0,1). All the weights have different final values (Figure 10), which seem to be conditioned by their initial values, because weights that begin positive tend to remain positive and weights that begin negative tend to remain negative. Also, the final training and cross validation costs are very close for all nine sets, which means that all the initial values were adequate to train the neural network (Figure 11).

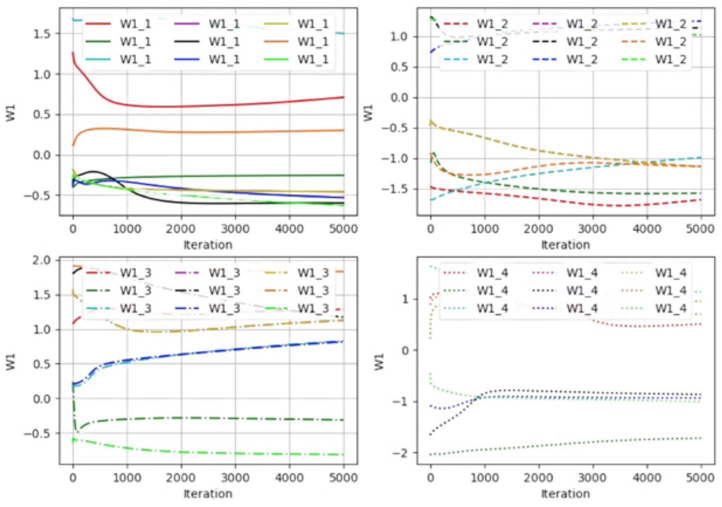


Figura 10: $W^{(1)}$'s values per iteration for the neural network for 9 different sets of initial parameters values. Neural network [1,4, 4, 1], $m = 120$, $seed = 7374$, $\alpha = 0.05$.

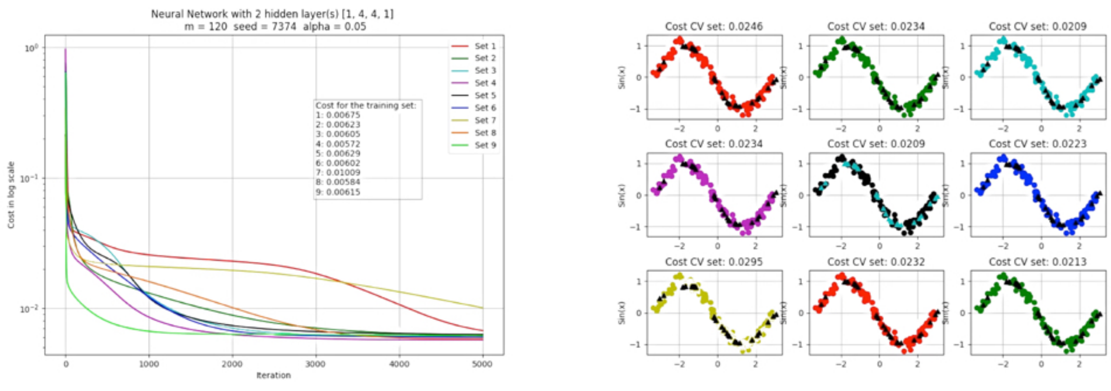


Figura 11: Final training and cross validation costs for 9 different sets of initial parameters values in Figure 10. Neural network [1,4, 4, 1], $m = 120$, $seed = 7374$, $\alpha = 0.05$.

Besides, the derivatives of the weights at the end of training are all very close to zero (within a distance of $\epsilon = 0.05$ from zero) for all the sets, passing our test regarding the stability of the behavior of the weights. In Figure 12, we see an example of the graphs of the derivatives of all the weights for Set 2.

With this observation in mind, we decided to train a neural network with an initial set *Set 1* of randomly picked values from a $N(0,1)$ distribution and then set the other

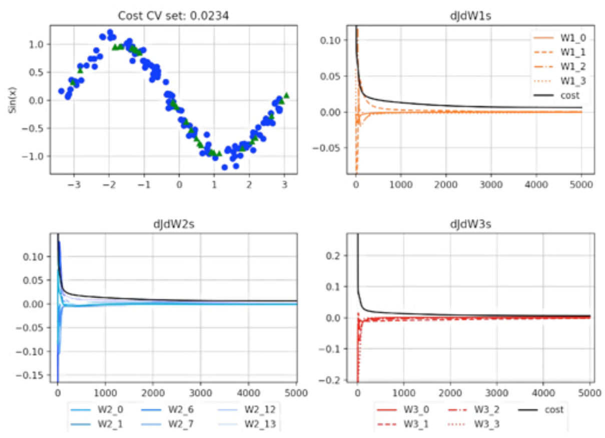


Figura 12: Derivatives with respect to the weights $W^{(3)}$, $W^{(2)}$ e $W^{(1)}$ per iteration for Set 2 of the neural network in Figure 10.

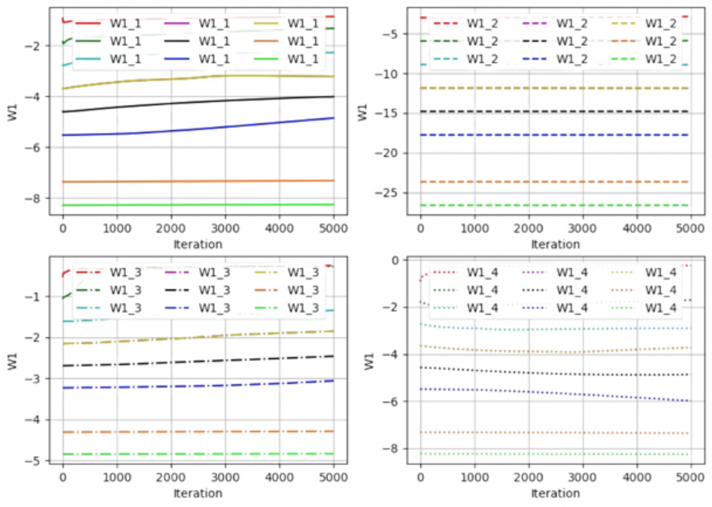


Figura 13: W_1's values per iteration for a neural network [1, 4, 4, 1], *seed* = 635473, $\alpha = 0.05$

sets of weights to be a positive integer multiple of the first set. So, to obtain the initial weights values for *Set 2* we multiplied each weight in *Set 1* by 2, for *Set 3* by 3, and so on. The results for this second scenario appear in Figures 13 and 14.

Note that the weights for W_1 (Figure 13) remain very close to their initial values and their sizes are kept, which is also and indication that the weights are sticky, as they tend to move very little from their initial values.

Another interesting observation is that both the final training cost increase as we increase the size of our initial weights (Figure 14(left)) and the generalization capacity is very poor for Sets 3-9 (Figure 14(right)). Note that the final shape for those sets is very close to the graph of the *tanh* function. This leads to the observation that big initial values of weights should be avoided, since the neural network may have difficulties in the

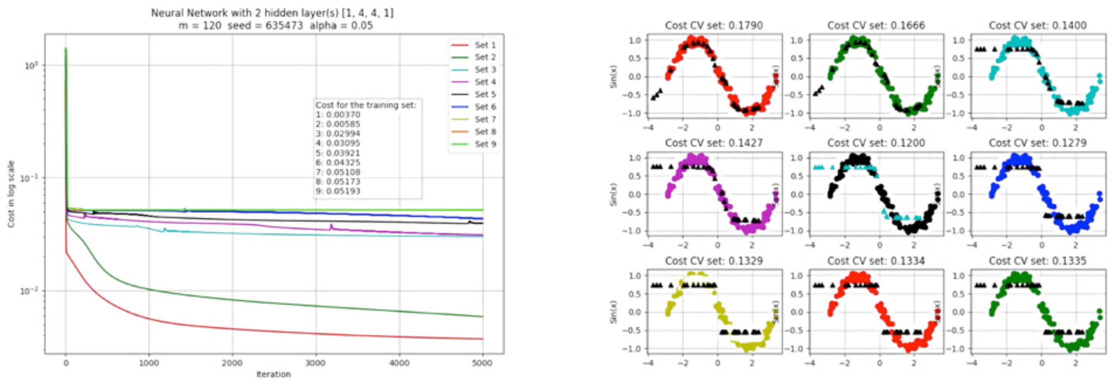


Figura 14: Final training and cross validation costs for a neural network [1, 4, 4, 1] for sets whose weights are multiples of the weights in Set 1, *seed* = 635473, $\alpha = 0.05$

learning process due to the saturation of the *tanh* function for arguments greater than 2.

In our next experiments, we intend to initialize the weights very close or very far from each other and analyze the effects on the learning process. It would be interesting to analyze if the absolute values of the initial weights are important to achieve good results or if there is any statistical property makes the most difference. Also, we would like to compare the results obtained for the final weights and costs when using other stopping methods.

3.2 Different sets of training examples and fixed initial weights

Changing the training examples has not significantly affected the behavior of the network for fixed initial weights or its learning capacity, since the final training costs and the generalization capacity were approximately close to zero for all the sets (Figure 15). Also, we note that the final weights remained relatively close to each other, as seen in Figure 16.

In one of our trials, we have noticed a different behavior for a specific training set. Although all the other sets behaved similarly and according to our general conclusions for this combination of initial conditions, the final weights have not followed the usual pattern of the same weights for other training examples (Figure 17(left). Besides, this training set had the lowest cost in comparison with the 8 other random sets of training examples (Figure 17(right).

There are many factors that could influence this behavior and one of the possibilities is that the training input may have a role in shaping the error surface influencing the existence of different local minima.

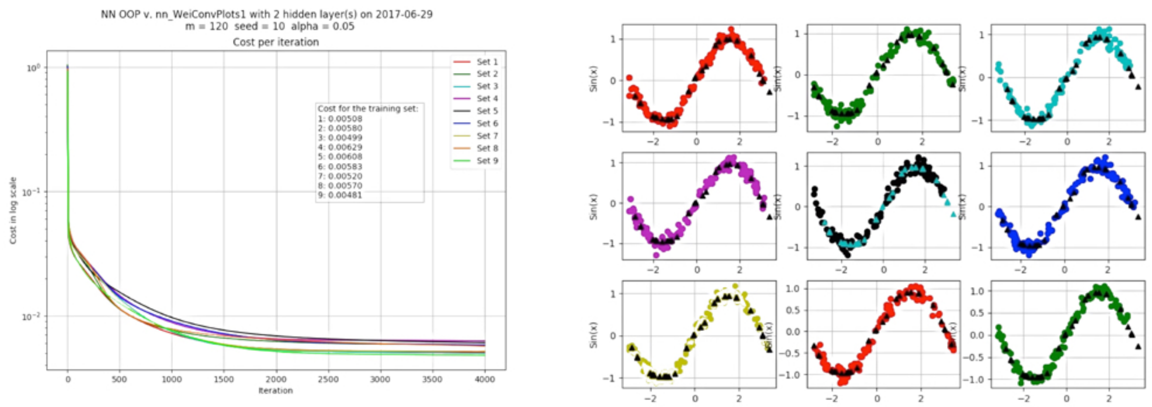


Figura 15: Final training and cross validation costs for a neural network [1, 4, 4, 1] for multiples of the weights in Set 1, $seed = 635473$, $\alpha = 0.05$.

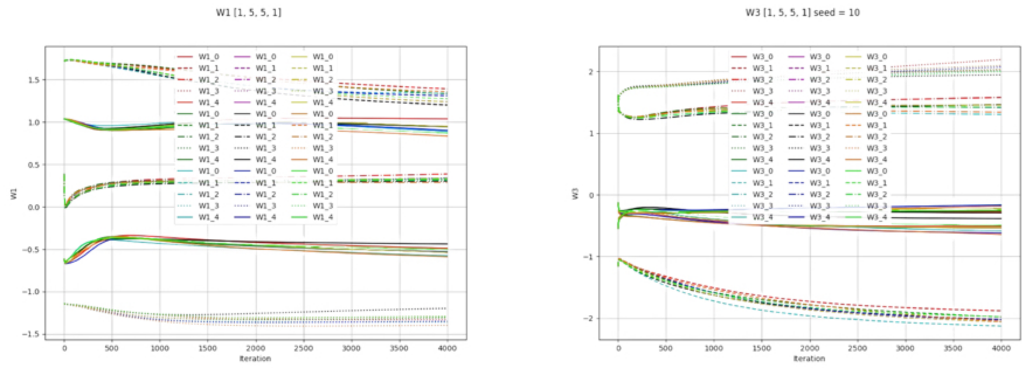


Figura 16: Final weights W_1 and W_3 for a neural network [1, 4, 4, 1] for multiples of the weights in Set 1, $seed = 635473$, $\alpha = 0.05$.

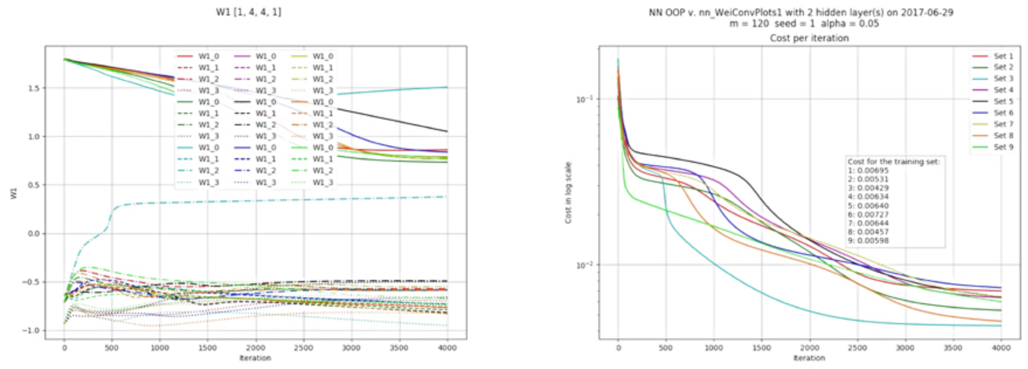


Figura 17: Final W_1's for a neural network [1, 4, 4, 1] for different training sets, $seed = 1$, $\alpha = 0.05$. Note how the light blue line diverges significantly from the rest.

4 Conclusion

Even in a simplified setting, we have observed that the multilayer neural network has a very complex behavior, even when controlled for the most relevant variables such as the initial weights and the set of training examples used. The use of the sine function was very convenient since we had full control of our inputs and outputs and had no limitation on the size of training examples.

One of our most interesting results was the analysis on the behavior of the weights given different initial conditions and the potential conclusions we can extract on local minima in the cost surface. We indicated many issues we would like to further investigate during the text and we think all of them are worth exploring in future work, but especially refining our analysis on the optimal combination of hidden layers and nodes per layer to learn effectively and efficiently the sine function would be an important study.

For future research, we think it would be very interesting to learn more nuanced functions, such as a sum of sines with different periods, and functions that present singularities. Also, as an extension of the study of optimal neural network architectures, a possible path for research would be to to mimic nature by analyzing the various possible combinations of a fixed number of neurons in different neural architectures.

5 References

HAYKIN, Simon. *Neural Networks and Learning Machines*. Pearson, 3rd edition, 2009.

GOODFELLOW, Ian *et al.* *Deep Learning*, 1st edition, 2016.

SCARSELLI, F., CHUNG Tsoi, A. *Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results*, Neural Networks, 11(1), 1998, 120.

CAMARGO, L.S., YONEYAMA, T. *Specification of training sets and the number of hidden neurons for multilayer perceptrons.* Neural computation 13 (12), 2673-2680, 2001.