Universidad de Córdoba
Escuela Politécnica Superior de Córdoba

Ingeniería Informática
Especialidad: Computación
Cuarto curso. Primer cuatrimestre

Introduction to computational models.

Lab Assignment 1: Implementation of the multilayer perceptron.

Javier Herrero Porras
i72hepoj@uco.es

Academic Course 2020-2021
Córdoba, October 11, 2020

Contents

List of Figures

List of Tables

Algorithm Index

NOTE: This is my first report developed with LaTeX. I want to apologize if there would be some errors on format or style. I will try to improve the next ones.

1 Neural network models

For this lab assignment, I have to implement an Artificial Neural Network model, which is a graph of neurons connected by links.

Every neuron is represented as a struct, which has the following attributes:

- out (Represents the output produced by each Neuron).
- delta (Represents the derivative of the output).
- w (Represents the input weights of a neuron).
- deltaW (Represents changes to apply to weights).
- lastDeltaW (Represents last changes applied to weights).
- wCopy (Represents a copy of the weights).

These neurons are organized in layers (input layer, hidden layers and output layer), where outputs of one layer are used as inputs for neurons of the next layer. All layers put together are known as **MultiLayer Perceptron (MLP)**.

As we learnt from theory lessons, there are 2 types of Artificial Neural Networks. In this practice, I had to implement a **Feed-forward** neural network, where neurons of one layer are only linked with neurons of the next layer.

The algorithm used to make the network learning is called **Back-propagation Algorithm**. The main idea is to adjust the weights of the connections between neurons to reduce the error of regression or classification on a given set. This algorithm has several steps:

- Forward propagation by feeding input layer and obtaining the output of the network in output layer.

- Calculate error and back-propagate it through the network.
- Obtain the weight update with errors.
- Apply the update and adjust weights.

Also, there are different ways to make the network learning. In this case, **online learning** has been implemented, that consists in applying weight adjustment for every pattern.

2 Pseudocode

In this section I will analyze the most important parts of the code implemented for the practice.

2.1 Back-propagation algorithm

This is the most relevant function implemented. It is a large function, so I will resume the part where validation dataset is created and I will focus on early stopping conditions.

Algorithm 1 shows learning process of the network. This process consists in two stop conditions:

- Standard version, which considers error of the training dataset.
- Validation version, which considers training dataset error and validation dataset error.

Algorithm 1 Online Back Propagation

```
1: procedure runOnlineBackPropagation
2:     [...]
3:     repeat
4:         if validationRatio is greater than 0 and less than 1 then
5:             trainOnline(trainDataset)
6:             lastValidationError ← validationError
7:             validationError ← test(validationDataset)
8:             trainError ← test(trainError)
9:             if validationError is less than lastValidationError then
10:                Copy weights in wCopy
11:                valItWithoutImprove ← 0
12:            else if validationError - lastValidationError is < 0.00001 then
13:                valItWithoutImprove ← 0
14:            else
15:                valItWithoutImprove + 1
16:            end if
17:            if trainError is less than minTrainError then
18:                minTrainError ← trainError
19:                Copy weights in wCopy
20:                ItWithoutImprove ← 0
21:            else if trainError - minTrainError is < 0.00001 then
22:                ItWithoutImprove ← 0
23:            else
24:                ItWithoutImprove + 1
25:            end if
26:            if ItWithoutImprove = 50 or valItWithoutImprove = 50
   then
27:                Restore wCopy in weights
28:                countTrain ← maxiter
29:            end if
30:        end if
31:        [...]
32:    until countTrain < maxiter
33:    Obtain predictions and errors.
34: end procedure
```

2.2 Accumulate Weight Changes

This algorithm save in deltaW[] the outputs from neurons of the previous layer multiplied by the correspondent delta. This operation is applied for layers to 1 from H-1, where H is the number of layers. Then deltaW[] will be used to update weights.

Algorithm 2 Accumulate Weight Changes

```
1: procedure FORWARDPROPAGATE
2:     for all layer l from 1 to H-1 do
3:         for all neuron n of layer l do
4:             for all neuron nPL of layer l − 1 do
5:                 delta ← layers[l].neurons[n].delta
6:                 output ← layers[l − 1].neurons[nPL].out
7:                 layers[l].neurons[n].deltaW[nPL] += delta ∗ output
8:             end for
9:             delta ← layers[l].neurons[n].delta
10:            layers[l].neurons[n].deltaW[nPL + 1] += delta
11:        end for
12:    end for
13: end procedure
```

2.3 Back Propagation Error

In this algorithm the error of the output layer is calculated with respect to a target or a desired output. The derivative is saved on delta and then, it goes through the layers to obtain each neuron's deltaW.

Algorithm 3 Back Propagation Error

```
1:  procedure backPropagateError
2:      target ← desiredOutput
3:      for all neurons n from layer H-1 do
4:          output ← layers[H − 1].neurons[n].out
5:          delta ← −1 ∗ (target[n] − output) ∗ (1 − output)
6:          layers[H − 1].neurons[n].delta ← delta
7:      end for
8:      for all layer l from H-2 to 1 do
9:          for all neuron n of layer l do
10:             WeightsDelta ← 0
11:             for all neuron nNL of layer l + 1 do
12:                 delta ← layers[l + 1].neurons[n].delta
13:                 weight ← layers[l + 1].neurons[nNL].w[n]
14:                 WeightsDelta += delta ∗ weight
15:             end for
16:             out ← layers[l].neurons[n].output
17:             layers[l].neurons[n].delta ← WeightsDelta ∗ out ∗ (1 − out)
18:         end for
19:     end for
20: end procedure
```

2.4 Adjust Weights

The last step consists of applying the changes in the weights, which are updated according the learning rate and momentum, which improves the convergence of weight updates.

Algorithm 4 Adjust Weight

```
1: procedure ADJUSTWEIGHTS
2:     for all layer l from 1 to H-1 do
3:         for all neuron n of layer l do
4:             for all neuron nPL of layer l − 1 do
5:                 deltaW ← layers[l].neurons[n].deltaW[nPL]
6:                 lastDeltaW ← layers[l].neurons[n].lastDeltaW[nPL]
7:                 layers[l].neurons[n].w[nPL] −= (learningRate∗deltaW+
   mu ∗ learningRate ∗ lastDeltaW)
8:             end for
9:             deltaW ← layers[l].neurons[n].deltaW[nPL + 1]
10:            lastDeltaW ← layers[l].neurons[n].lastDeltaW[nPL + 1]
11:            layers[l].neurons[n].w[nPL+1] −= (learningRate∗deltaW+
   mu ∗ learningRate ∗ lastDeltaW)
12:        end for
13:    end for
14: end procedure
```

3 Experiments and results discussion

3.1 Datasets used

For these experiments, 4 different datasets have been used to train and test the neural network:

- *XOR problem*, where training dataset consists of 4 patterns with 2 inputs and 1 output. Train and test dataset have the same patterns.
- *Sine function*, where training dataset consists of 120 patterns with 1 input and 1 output. Also, test dataset has 41 patterns.
- *Quake dataset*, where training dataset consists of 1633 patterns with 3 inputs and 1 outputs. Also, test dataset has 546 patterns.

- *Parkinson dataset*, where training dataset consists of 4406 patterns with 19 inputs and 2 outputs. Also, test dataset has 1469 patterns.

3.2 Parameters considered

In order to make the experiments analyzed in this report, these values have been considered:

- *Validation Ratio* to establish the number of patterns that will be used to validate neural network: {0.0; 0.15; 0.25}
- *Learning Rate*, which is a tuning parameter of the neural network, which determine the step size at each iteration: {1.0; 2.0}
- The number of *Hidden Layers* of the neural network, so the model can have a different number of layers to find the best number of layers for each dataset: {1, 2}.
- The number of *Neurons* to be in each hidden layer. This number together with hidden layers establish the *topology* of the neural network: {2; 4; 8; 32; 64; 100}
- *Eta*, which controls the update of the weights, allowing us to give more or less importance to the weight derivative: {0.1}
- *Mu* or momentum, to improve the convergence of the algorithm by controlling the direction of the weights: {0.9}

3.3 Results obtained

3.3.1 XOR Problem

In this section, I have to make a neural network for the XOR problem, that consists in predicting the outputs of an XOR logic gate with two binary inputs[1]. So, the network should return one if the inputs are not equal and zero if so. Thus, the function output is shown in Table 1.

X	Y	Output
-1	-1	0
-1	1	1
1	-1	1
1	1	0

Table 1: XOR Function

According to these values, the neural network should be able to learn from these patterns and make approximate outputs. This is a special problem because there are only 4 patterns, so train dataset and test dataset measures (MSE and std) will be the same.

First, I will make a little example of the neural network in this problem with the easiest architecture (1 hidden layer and 2 neurons) to demonstrate that the algorithm and the neural network are working correctly (Note: This configuration is not appropriate for the neural network, because errors are high in comparison with others, but it is just an example).

In the execution, the neural network has been obtained after training is shown in figure 1.

To test this network, I am going to follow the same process as the algorithm. First of all, inputs need to be fed and then, we should apply forward propagation to get the outputs.

For example with pattern <-1,1><1>, the algorithm follow this process:

$$net_1^1 = out_1^0 * w_{11}^1 + out_2^0 * w_{12}^1 + w_0^1 =$$
$$= -2.69985 * -1 + -2.70048 * 1 + 2.44915 = 2.44852$$
$$out_1^1 = sigmoid(net_1^1) = 0.92045$$

If we repeat the process for the remaining neurons, we obtain:

$$out_1^2 = sigmoid(net_1^2) = 0.86836$$

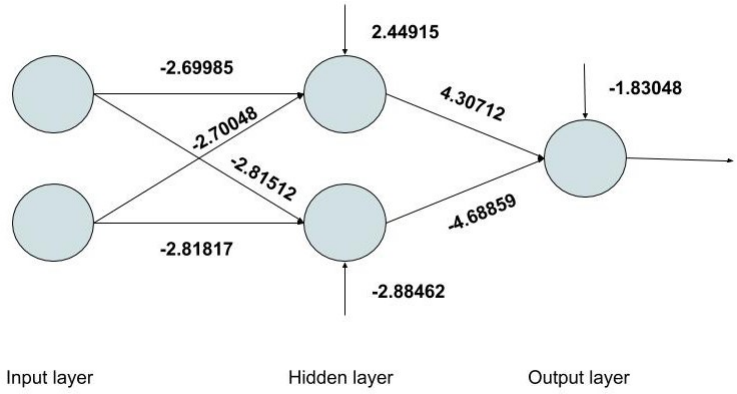

Figure 1: XOR Neural Network

The network produces an output with value 0.86836, which is close to the target output, 1. Then, the network and the algorithm seem to be working correctly for this problem. Also, the results obtained with every configuration are shown in table 2.

Hidden Layers	Neurons	Mean_{train}	σ_{train}	Mean_{test}	σ_{test}
1	2	0.0979274	0.0645687	0.0979274	0.0645687
1	4	0.0196991	0.0059445	0.0196991	0.0059445
1	8	0.0111259	0.00292624	0.0111259	0.00292624
1	32	0.0043782	0.0003530	0.0043782	0.0003530
1	64	0.0033485	0.0000435	0.0033485	0.0000435
1	100	0.0030856	0.0002047	0.0030856	0.0002047
2	2	0.2404630	0.0136966	0.2404630	0.0136966
2	4	0.1636540	0.0808994	0.1636540	0.0808994
2	8	0.0298095	0.0231569	0.0298095	0.0231569
2	32	0.0026801	0.0001442	0.0026801	0.0001442
2	64	0.0013056	0.0000772	0.0013056	0.0000772
2	**100**	**0.0008221**	**0.0000372**	**0.0008221**	**0.0000372**

Table 2: Result of experiment XOR

Moreover, we can see previous data represented on a bar chart in figure 2. This figure shows that despite being a simple problem with 4 patterns, the experiments get worse results when the value of neurons in hidden layers is low (2,4,8). The network which gave a better result in the experiment has 2 hidden layers and 100 neurons. As we can see in bar chart, it seems to be a trend independently of the number of layers, so, if we increase the number of network neurons the results will be better. This happens because the

network can fit better with the patterns as a result of making more accurate weight adjustments when it has more neurons.

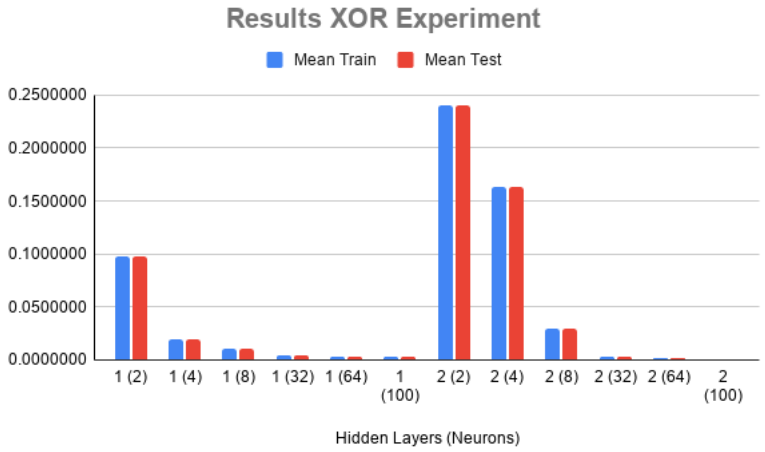

Figure 2: Bar chart experiment XOR

Once we find the best architecture for the problem, we should change some internal parameters of the algorithm in order to reach the optimal configuration. In this case there are 2 patterns, therefore, the problem does not allow a validation subset. Then, we can only work with decreasing factor which influences the learning rate of the network. If this decreasing factor is higher, then the learning rate of each layer will increase.

v	F	Mean_{train}	σ_{train}	Mean_{test}	σ_{test}
0.0	**1**	**0.0008221**	**0.0000372**	**0.0008221**	**0.0000372**
0.0	2	0.0139756	0.0645687	0.0139756	0.00303938

Table 3: Result of experiment for best arquitecture XOR

Table 3 show us that the network works better with a lower decreasing factor (F), so we can conclude that it has better results with a lower learning rate, which causes a lower update of the weights during the iterations.

3.3.2 Sine Function

In this section the neural network will deal with sine function (figure 4). But, there was added random noise, so the curve is not perfect.

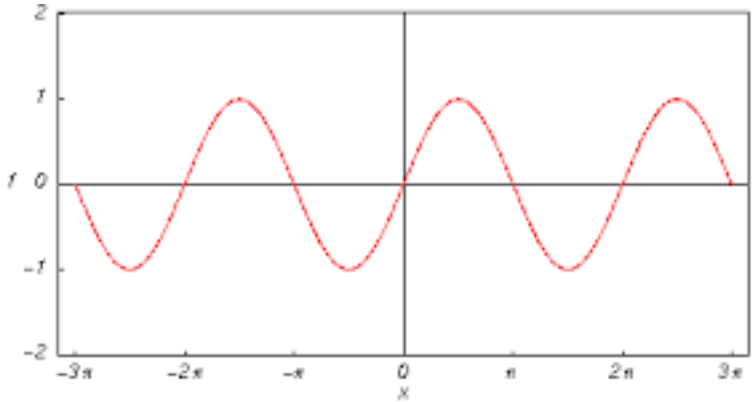

Figure 3: Sine function

For example if we take the pattern $<0.1875, 0.5234154808>$ we notice that *sin(0.1875)* = 0.186403, in consequence, there is a noise of *0.5234154808 - 0.186403 = 0.3370124808*. Therefore, the neural network should learn and work with these values.

The train dataset is more complex than XOR dataset, because it has 120 patterns. So we can firstly think that network will need more neurons and layers to work better, like as with XOR. If we analyse table 4, we conclude that the best architecture for Sine function has 2 hidden layers and 64 neurons.

Hidden Layers	Neurons	Mean_{train}	σ_{train}	Mean_{test}	σ_{test}
1	2	0.0294147	0.000499823	0.0364772	0.000340456
1	4	0.0295698	0.000419948	0.036578	0.000338089
1	8	0.0289501	0.000363621	0.0356082	0.000537065
1	32	0.027976	0.000240893	0.0346238	0.000759253
1	64	0.0292516	0.00126879	0.0369078	0.00143591
1	100	0.0309102	0.000704113	0.0380537	0.00101349
2	2	0.0298574	0.000055436	0.036041	0.000093847
2	4	0.02988	0.000107363	0.0361589	0.000139977
2	8	0.0298915	0.000374441	0.036306	0.000138532
2	32	0.0286485	0.00126128	0.0355821	0.00175719
2	**64**	**0.0198283**	**0.00375279**	**0.0269547**	**0.00388868**
2	100	0.0291437	0.0077889	0.0382179	0.00918714

Table 4: Result of experiment Sine function

In this case, results (test mean) with 2 hidden layers and 100 neurons are worse than obtained with respect to 64 neurons. It seems that this function

does not work well with high number of neurons as we can see also with 1 hidden layer. This fact can be easily seen in figure 4.

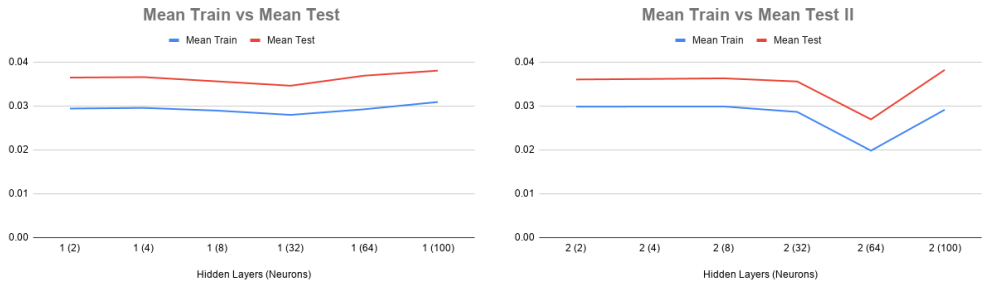

Figure 4: Sine function

Once the best architecture for the function is found, we should change some parameters of the network to check if we get better results. In this case we are going to work with validation ratio (which allow it to make a validation dataset) and decreasing factor.

v	F	Mean_{train}	σ_{train}	Mean_{test}	σ_{test}
0.0	1	**0.0198283**	**0.00375279**	**0.0269547**	**0.00388868**
0.0	2	0.0291888	0.000181147	0.0366121	0.000455103
0.15	1	0.022018	0.00526533	0.0310097	0.00826575
0.15	2	0.029796	0.00276166	0.0374013	0.00081878
0.25	1	0.0261117	0.00290839	0.0344473	0.00271953
0.25	2	0.0306023	0.00342122	0.0367225	0.000579431

Table 5: Result of experiment for best architecture Sine function

In table 5 we check that the best combination of these parameters is $v = 0.0$ and $F = 1$. But, why are the results worse with a validation dataset?

When we have a validation dataset we remove patterns from train dataset, in consequence, the neural network has not many patterns to learn from. If training dataset has few items then the learning is poor and results are worse. Otherwise, if training dataset has many items we can remove some patterns but learning process will be still good. In our case, there are few patterns (120) and it is better to assign them to learn dataset.

NOTE: Here are the results obtained by the network against the results given by the test dataset. We can check it is a good approximation of the

function, and the error is minimum. I think it would be a good idea to represent also the sine function, but we train network with values of sine function in range [-1,1] but the period of the function is $2*\pi$. Results are shown in figure 5.

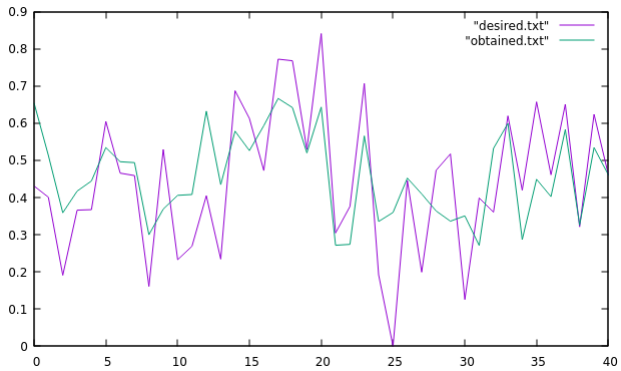

Figure 5: Sine function Gnuplot

3.3.3 Quake dataset

In this section the neural network has to learn from a dataset which describes main characteristics of quake data. Dataset has 3 inputs (focal depth, latitude and longitude) and 1 output (strength of the earthquake, in Richter)[2].

Some ideas I want to point out is that this dataset has a relative high number of patterns (more than 1500), so a validation dataset could be appropriate in this case. We can take even 200 or 300 patterns to validate the dataset and check if training is working correctly and avoiding over-fitting. As we can see in table 6 the best architecture obtained in this experiment has 2 layers and 8 neurons (according to mean obtained in test dataset). It is important to say that errors are very close independently the architecture, but we must take the one which has lower test error.

Hidden Layers	Neurons	Mean_{train}	σ_{train}	Mean_{test}	σ_{test}
1	2	0.0300748	0.00004652	0.0271408	0.00004804
1	4	0.0298411	0.00006011	0.0270049	0.00004483
1	8	0.0297152	0.00005496	0.0269837	0.00005407
1	32	0.029647	0.00007193	0.0269536	0.00003850
1	64	0.0295999	0.00008826	0.0270561	0.00004408
1	100	0.0295885	0.00004667	0.0270819	0.00002799
2	2	0.0299965	0.0001240	0.0271023	0.0001236
2	4	0.0299761	0.00006417	0.0270645	0.00006955
2	**8**	**0.0298237**	**0.00009871**	**0.026918**	**0.00006481**
2	32	0.0296115	0.00007308	0.0269865	0.00006847
2	64	0.0296026	0.00025379	0.0270794	0.00009982
2	100	0.0292135	0.00013026	0.0271317	0.00005058

Table 6: Result of experiment Quake dataset

Once we have selected the best architecture, we should work with some parameters of the network to check if we get better results. In this case we are going to work with validation ratio and decreasing factor. The idea of a validation dataset does not work yet, because the mean error with respect to train dataset ($Mean_{train}$) decreases when we have a validation dataset but the mean error with respect to test dataset ($Mean_{test}$) increases. So we can conclude that this network works better without a validation dataset, maybe because there are few patterns. Also, the best combination is $v = 0.0$ and $F = 1$.

v	F	Mean_{train}	σ_{train}	Mean_{test}	σ_{test}
0.0	**1**	**0.0298237**	**0.00009871**	**0.026918**	**0.00006481**
0.0	2	0.0301834	0.00000883	0.0272954	0.00001864
0.15	1	0.029364	0.00067395	0.0270494	0.00007048
0.15	2	0.029696	0.0006641	0.027309	0.00003928
0.25	1	0.0294395	0.0006805	0.0271828	0.0001359
0.25	2	0.0297313	0.0008441	0.0273466	0.00004504

Table 7: Result of experiment for best arquitecture Quake dataset

Now the best configuration has been found. In the figure 6 we can see and compare different errors (train, validation and test) during the different iterations (set to 1000) with the best architecture and the best configuration of the neural network. We notice that all the errors decrease during the execution because the network is training and adjusting the weights in a appropriate way. If training or validation error increase during 50 consecutive

iterations, the algorithm would stop to avoid over-fitting. This is known as **early stopping**.

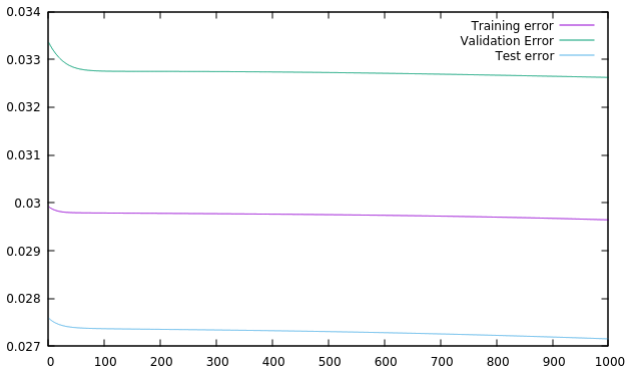

Figure 6: Errors comparison Quake dataset

3.3.4 Parkinson dataset

Last section is referred to Parkinson dataset. It contains some measures taken from people with early-stage Parkinson's disease, having 19 inputs and 2 outputs with 4406 patterns.

As we check in table 8, the best architecture for the model should give better results when having a high number of neurons, because dataset has more inputs than other studied (XOR, quake or sine function) and it needs more neurons and weights to get better predictions. According to the table 8 the best architecture has 2 layers and 64 neurons.

Hidden Layers	Neurons	Mean_{train}	σ_{train}	Mean_{test}	σ_{test}
1	2	0.034599	0.0001100	0.037128	0.0001361
1	4	0.0273436	0.0019086	0.0274779	0.0019903
1	8	0.0201363	0.0014425	0.0207065	0.0016783
1	32	0.0163166	0.0020715	0.0177134	0.0017907
1	64	0.0134555	0.0009974	0.0155163	0.0007876
1	100	0.0131422	0.00146776	0.0155962	0.00175907
2	2	0.0314947	0.0005988	0.0330642	0.0007789
2	4	0.0230822	0.0034659	0.0229783	0.0036423
2	8	0.0164145	0.0034582	0.0167412	0.0029731
2	32	0.00708539	0.0015713	0.00932782	0.0017461
2	**64**	**0.00370819**	**0.0010254**	**0.0060584**	**0.0012407**
2	100	0.00414681	0.0022709	0.00641803	0.0026788

Table 8: Result of experiment Parkinson dataset

This dataset has a big number of patterns (in connection with the others studied in this lab assignment). In consequence, the MSE error between experiments with validation dataset ($v > 0.0$) with respect to those who have not ($v = 0.0$) should be minimal. As we can check in table 9 this idea is correct because error differences are less than 0.0024.

v	F	Mean_{train}	σ_{train}	Mean_{test}	σ_{test}
0.0	**1**	**0.00370819**	**0.00102549**	**0.0060584**	**0.0012407**
0.0	2	0.0133248	0.0005089	0.0149895	0.0008788
0.15	1	0.00611242	0.0005089	0.00838148	0.0033472
0.15	2	0.0142312	0.0005975	0.0158444	0.0005323
0.25	1	0.00393473	0.0007504	0.00774042	0.0011246
0.25	2	0.0141801	0.0006609	0.0162285	0.0006475

Table 9: Result of experiment for best architecture Parkinson dataset

Table 9 results show us that no-validation experiments have better results than validation datasets. The reason is that if we delete patterns from training datasets to create a validation one, the learning will be worse. But having a validation dataset has some advantages too.

In table 10 and figure 7 we can check the time spend by the experiments. Those experiments consists in 5 repetitions of the execution of the algorithm with the best architecture for the problem (2 hidden layers, 64 neurons), and changing the value of patterns which belong to the validation dataset.

Execution	Validation Ratio	Time
1	0.0	822.947
1	0.15	694.642
1	0.25	718.741
2	0.0	810.975
2	0.15	666.281
2	0.25	682.02
3	0.0	824.899
3	0.15	694.106
3	0.15	713.319
4	0.0	829.955
4	0.15	682.993
4	0.25	716.933
5	0.0	785.819
5	0.15	659.141
5	0.25	686.549

Table 10: Execution times

In the figure we can see that execution times when the network does not have a validation dataset are higher than when it has. This is caused by early stopping, because learning algorithm stops if train or validation error increases and also, training dataset has less patterns, so learning is a bit faster.

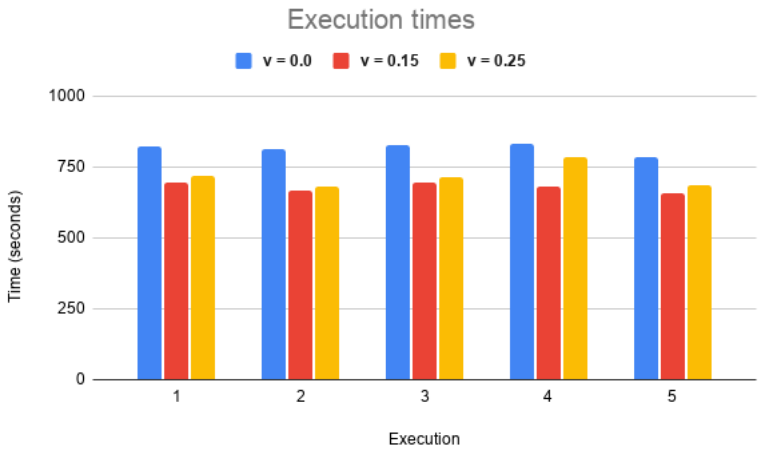

Figure 7: Execution times

3.4 Conclusions

To sum up, despite the dataset number of patterns, neural networks need a high number of layers and neurons to get better results. But it has a negative effect in computational time and resources. Although datasets studied have a relative low number of patterns, the execution times were high (close to 10-11 minutes in the last dataset). In real datasets which have hundred of thousands or millions of datasets these times would be prohibitive.

Here is when the concept of validation datasets make sense. Despite getting a bit worse MSE error in validation and train datasets, execution times are lower (around 15-17% better), so it is a big advantage in problems which have enormous amount of data.

It would be interesting working and making experiments with other values or changing other parameters (iterations of the outer loop, eta, mu...) and get better conclusions, but unfortunately, I have not time enough to make it.

References

[1] *The XOR Problem in Neural Networks*. URL: `https://medium.com/@jayeshbahire/the-xor-problem-in-neural-networks-50006411840b`. [Online. Last consultation: 9-10-2020].

[2] *Quake data set*. URL: `https://sci2s.ugr.es/keel/dataset.php?cod=75`. [Online. Last consultation: 10-10-2020].