Grado en Ingeniería Informática

Lab Assignment 1: Multilayer perceptron implementation

Universidad de Córdoba

Fourth year of "Grado en Ingeniería Informática"
Dpto. de Informática y análisis Numérico
Introduction to computational models
Course 2023-2024

Autor:

Antonio Llamas García i92llgaa@uco.es
31886320V

Docentes:

Pedro A. Gutiérrez Peña	pagutierrez@uco.es
Javier Sánchez Monedero	jsanchezm@uco.es
César Hervás Martínez	chervas@uco.es

Córdoba, 5 de febrero de 2024

Índice

1. Description of Neural Network Models

In this experiment, we explore a range of neural network architectures with varying hidden layers and neurons in each layer. The goal is to identify the optimal configuration for the given problem. A momentum factor is incorporated into the training process for enhanced convergence.

1.1. Single Hidden Layer Architectures

1. **{n : 2 : k}:** This architecture features a single hidden layer with 2 neurons.
2. **{n : 4 : k}:** Single hidden layer with 4 neurons.
3. **{n : 8 : k}:** Single hidden layer with 8 neurons.
4. **{n : 32 : k}:** Single hidden layer with 32 neurons.
5. **{n : 64 : k}:** Single hidden layer with 64 neurons.
6. **{n : 100 : k}:** Single hidden layer with 100 neurons.

1.2. Two Hidden Layers Architectures

1. **{n : 2 : 2 : k}:** This architecture involves two hidden layers, each with 2 neurons.
2. **{n : 4 : 4 : k}:** Two hidden layers with 4 neurons in each layer.
3. **{n : 8 : 8 : k}:** Two hidden layers with 8 neurons in each layer.
4. **{n : 32 : 32 : k}:** Two hidden layers with 32 neurons in each layer.
5. **{n : 64 : 64 : k}:** Two hidden layers with 64 neurons in each layer.
6. **{n : 100 : 100 : k}:** Two hidden layers with 100 neurons in each layer.

For each architecture, the neural network undergoes training and evaluation. The objective is to analyze the performance across different complexities and depths, aiming to find the most effective configuration for the specific problem.

Note: The notation used is {n : x : k}, where n represents the number of neurons in the input layer, x denotes the number of neurons in each hidden layer, and k signifies the number of neurons in the output layer.

2. Algorithms description

In this section, we will learn about the most important functions/algorithms of our program. We are going to know how they work and why are they so important.

2.1. BackPropagation Algorithm

The backpropagateError function computes error signals (deltas) for each neuron in the network during the training of a multilayer perceptron (MLP). It starts from the output layer, calculating deltas based on the difference between actual and target output, considering the sigmoid activation function's derivative. These deltas are then propagated backward through hidden layers, indicating each neuron's contribution to the overall error. The function is crucial for adjusting weights during training to minimize errors through gradient descent.

Algorithm 1 BackpropagateError(target)

```
Require: $target$: Array representing the desired output for the current input pattern
 1: for each neuron $i$ in the output layer ($this \to layers[this \to nOfLayers - 1]$) do
 2:     $out \leftarrow this \to layers[this \to nOfLayers - 1].neurons[i].out$    ▷ Get the output of the neuron
 3:     $this \to layers[this \to nOfLayers - 1].neurons[i].delta \leftarrow -(target[i] - out) \cdot out \cdot (1 - out)$    ▷ Calculate the delta for the output layer
 4: end for
 5: for $i \leftarrow this \to nOfLayers - 2$ down to 1 do    ▷ For each hidden layer from second last to the first
 6:     for each neuron $j$ in the current layer ($this \to layers[i]$) do
 7:         $out \leftarrow this \to layers[i].neurons[j].out$    ▷ Get the output of the neuron
 8:         $aux \leftarrow 0{,}0$    ▷ Reset aux
 9:         for each neuron $k$ in the next layer ($this \to layers[i + 1]$) do
10:             $aux \mathrel{+}= this \to layers[i + 1].neurons[k].w[j + 1] \cdot this \to layers[i + 1].neurons[k].delta$    ▷ Calculate the weighted sum in aux
11:         end for
12:         $this \to layers[i].neurons[j].delta \leftarrow aux \cdot out \cdot (1 - out)$    ▷ Calculate the delta for the current hidden layer neuron
13:     end for
14: end for
```

2.2. ForwardPropagation Algorithm

The forwardPropagate function executes the forward pass in a multilayer perceptron (MLP), computing the output of each neuron for a given input. It iterates through each layer and neuron, calculating a weighted sum of inputs (net) by considering the weights and outputs of neurons in the preceding layer. The sigmoid activation function is then applied to produce the final output.

Algorithm 2 ForwardPropagate()

```
 1: for $i \leftarrow 1$ to $this \to nOfLayers - 1$ do    ▷ For every layer
 2:     for $j \leftarrow 0$ to $this \to layers[i].nOfNeurons$ do    ▷ For every neuron
 3:         $net \leftarrow 0{,}0$    ▷ Reset net
 4:         for $k \leftarrow 1$ to $this \to layers[i - 1].nOfNeurons + 1$ do    ▷ For every weight
 5:             $net \mathrel{+}= this \to layers[i].neurons[j].w[k] \cdot this \to layers[i - 1].neurons[k - 1].out$    ▷ Calculate the net
 6:         end for
 7:         $net \mathrel{+}= this \to layers[i].neurons[j].w[0]$    ▷ Add the bias
 8:         $this \to layers[i].neurons[j].out \leftarrow \frac{1}{1+e^{-net}}$    ▷ Calculate the output using the sigmoid function
 9:     end for
10: end for
```

2.3. Accumulate Change Function

The purpose of this function is to accumulate the changes that need to be made to the weights of the neural network during the training process. These changes are determined during the backpropagation step, where the network's error is propagated backward, and the weights are adjusted to minimize the error. The accumulated changes (deltaW) will be used later in the weight adjustment step to update the weights of the network.

Algorithm 3 AccumulateChange()

```
1: for $i \leftarrow 1$ to $this \rightarrow nOfLayers - 1$ do                ▷ For every layer
2:     for $j \leftarrow 0$ to $this \rightarrow layers[i].nOfNeurons$ do      ▷ For every neuron
3:         for $k \leftarrow 1$ to $this \rightarrow layers[i-1].nOfNeurons + 1$ do      ▷ For every weight
4:             $this \rightarrow layers[i].neurons[j].deltaW[k] \mathrel{+}= this \rightarrow layers[i].neurons[j].delta \cdot this \rightarrow layers[i-1].neurons[k-1].out$      ▷ Calculate the deltaW
5:         end for
6:         $this \rightarrow layers[i].neurons[j].deltaW[0] \mathrel{+}= this \rightarrow layers[i].neurons[j].delta$      ▷ Update the bias term
7:     end for
8: end for
```

2.4. Weight Adjustment Function

The weightAdjustment function performs the update of weights in the multilayer perceptron (MLP) based on the calculated weight changes during backpropagation. It iterates through each layer, neuron, and weight, updating the weights using the delta rule with momentum. The learning rate (eta) and momentum factor (mu) control the magnitude of the weight adjustments, contributing to the convergence and stability of the training process.

Algorithm 4 WeightAdjustment()

```
1: for $i \leftarrow 1$ to $this \rightarrow nOfLayers - 1$ do                ▷ For every layer
2:     for $j \leftarrow 1$ to $this \rightarrow layers[i].nOfNeurons$ do      ▷ For every neuron
3:         for $k \leftarrow 1$ to $this \rightarrow layers[i-1].nOfNeurons + 1$ do      ▷ For every weight
4:             $this \rightarrow layers[i].neurons[j].w[k] \mathrel{-}= (this \rightarrow eta \cdot this \rightarrow layers[i].neurons[j].deltaW[k])$
                   $-(this \rightarrow mu \cdot this \rightarrow eta \cdot this \rightarrow layers[i].neurons[j].lastDeltaW[k])$      ▷ Update the weight
5:         end for
6:         $this \rightarrow layers[i].neurons[j].w[0] \mathrel{-}= (this \rightarrow eta \cdot this \rightarrow layers[i].neurons[j].deltaW[0])$
               $-(this \rightarrow mu \cdot this \rightarrow eta \cdot this \rightarrow layers[i].neurons[j].lastDeltaW[0])$      ▷ Update the bias
7:     end for
8: end for
```

2.5. Online BackPropagation Algorithm

The runOnlineBackPropagation function implements online backpropagation training for a multilayer perceptron. It iteratively trains the neural network using the provided training dataset, adjusting weights and biases after processing each individual training instance. The training process continues until the specified maximum number of iterations (maxiter) is reached or early stopping conditions are met, such as lack of improvement in training error.

Algorithm 5 runOnlineBackPropagation($trainDataset$, $pDatosTest$, $maxiter$, $errorTrain$, $errorTest$)

```
$countTrain \leftarrow 0$
$randomWeights()$                                        ▷ Randomly initialize weights
$minTrainError \leftarrow 0$
$iterWithoutImproving \leftarrow 0$
$testError \leftarrow 0$
repeat
    $trainOnline(trainDataset)$                          ▷ Train the network online
    $trainError \leftarrow test(trainDataset)$           ▷ Evaluate training error
    if $countTrain = 0$ or $trainError < minTrainError$ then
        $minTrainError \leftarrow trainError$
        $copyWeights()$
        $iterWithoutImproving \leftarrow 0$
    else if $(trainError - minTrainError) < 0{,}00001$ then
        $iterWithoutImproving \leftarrow 0$
    else
        $iterWithoutImproving \mathrel{+}= 1$
    end if
    if $iterWithoutImproving = 50$ then
        Print "We exit because the training is not improving!!"
        $restoreWeights()$
        $countTrain \leftarrow maxiter$
    end if
    $countTrain \mathrel{+}= 1$
    Print Ïteration" $countTrain$ and Training error:” $trainError$
until $countTrain < maxiter$
Print "NETWORK WEIGHTS"
Print -=============="
$printNetwork()$
Print "Desired output Vs Obtained output (test)"
Print -========================================"
for $i \leftarrow 0$ to $pDatosTest \rightarrow nOfPatterns - 1$ do
    $prediction \leftarrow$ new double[$pDatosTest \rightarrow nOfOutputs$]
    $feedInputs(pDatosTest \rightarrow inputs[i])$
    $forwardPropagate()$
    $getOutputs(prediction)$
    for $j \leftarrow 0$ to $pDatosTest \rightarrow nOfOutputs - 1$ do
        Print $pDatosTest \rightarrow outputs[i][j]\ \ --\ prediction[j]$
    end for
    Print                                                ▷ Print the desired and obtained outputs
    delete[]$prediction$
end for
$testError \leftarrow test(pDatosTest)$
$*errorTest \leftarrow testError$
$*errorTrain \leftarrow minTrainError$
```

3. Experiments

In this section, we will see the datasets and parameters that we have used, and the results obtained with a discussion about them.

3.1. Datasets Used

To assess the performance of the implemented algorithm, four datasets are utilized:

3.1.1. XOR Problem

Description: The XOR dataset represents the non-linear classification problem of XOR (exclusive OR). It is a classic problem in neural network training, challenging due to its non-linearity.

Usage: The same file is used for both training and testing.

3.1.2. Sine Function

Description: The Sine function dataset consists of 120 training patterns and 41 testing patterns. It is generated by adding random noise to the sine function, introducing variability and complexity.

Usage: Used to assess the algorithm's performance in capturing patterns from a function with noise.

3.1.3. Quake Dataset

Description: The Quake dataset comprises 1633 training patterns and 546 testing patterns. It is designed for earthquake strength prediction, measured on the Richter scale. Input variables include depth of focus, latitude, and longitude.

Usage: The goal is to predict earthquake strength based on geographic factors.

3.1.4. Auto MPG Dataset

Description: The Auto MPG dataset, sourced from the UCI Machine Learning Repository, is a well-known dataset with 398 samples and 7 attributes. It provides information about various car models, including attributes like cylinders, displacement, horsepower, weight, acceleration, model year, and origin (USA, Europe, Japan).

Usage: Commonly utilized for regression and predictive modeling tasks, especially to predict a car's miles per gallon (MPG) based on its features. Discrete attributes were binarized, and the model name was removed for the analysis.

3.2. Parameter Values

The algorithm's performance is evaluated with various parameter configurations:

- **Iterations:** 1000
- **Hidden Layers:** 1 or 2 for all datasets except XOR, and 1 or 2 or 5 for the XOR dataset.
- **Neurons:** 2, 4, 8, 32, 64, 100
- **Learning Rate (η):** 0.1
- **Momentum Factor (μ):** 0.9

These parameter configurations are systematically tested to assess the algorithm's robustness and performance across different scenarios.

3.3. Results obtained

This section is thought with the purpose of show the results obtain in each experiment done with each dataset.

3.3.1. Results for XOR Dataset

Layers	Neurons	Train MSE ± SD	Test MSE ± SD
1	2	0.283139 +- 0.0320771	0.283139 +- 0.0320771
	4	0.251745 +- 0.0443425	0.251745 +- 0.0443425
	8	0.145517 +- 0.0595993	0.145517 +- 0.0595993
	32	0.0192992 +- 0.0161049	0.0192992 +- 0.0161049
	64	0.0995657 +- 0.121508	0.0995657 +- 0.121508
	100	0.100026 +- 0.12242	0.100026 +- 0.12242
2	2	0.244155 +- 0.028406	0.244155 +- 0.028406
	4	0.23881 +- 0.0124342	0.23881 +- 0.0124342
	8	0.218292 +- 0.0514311	0.218292 +- 0.0514311
	32	0.00969784 +- 0.00645419	0.00969784 +- 0.00645419
	64	0.00165301 +- 0.00221251	0.00165301 +- 0.00221251
	100	0.000177182 +- 5.95404e-05	0.000177182 +- 5.95404e-05
5	2	0.261521 +- 0.0105792	0.261521 +- 0.0105792
	4	0.219043 +- 0.04844	0.219043 +- 0.04844
	8	0.249977 +- 3.7244e-05	0.249977 +- 3.7244e-05
	32	0.110823 +- 0.114067	0.110823 +- 0.114067
	64	0.0577208 +- 0.0467478	0.0577208 +- 0.0467478
	100	0.000115301 +- 0.000196396	0.000115301 +- 0.000196396

Tabla 1: Results for XOR Dataset

3.3.2. Results for Sine Dataset

Layers	Neurons	Train MSE ± SD	Test MSE ± SD
1	2	0.0406614 +- 0.0102932	0.0465517 +- 0.00823905
	4	0.0371684 +- 0.0065696	0.0437955 +- 0.008132
	8	0.0289577 +- 0.00253645	0.0382328 +- 0.00343976
	32	0.0217279 +- 0.00168838	0.0306966 +- 0.00237157
	64	0.0270687 +- 0.00371482	0.0337327 +- 0.00348898
	100	0.065082 +- 0.0889406	0.0698627 +- 0.0806091
2	2	0.0474134 +- 0.0236896	0.0559176 +- 0.0278075
	4	0.0354404 +- 0.011196	0.0433238 +- 0.0139477
	8	0.0481493 +- 0.0223393	0.0578935 +- 0.0268058
	32	0.0299701 +- 0.00196213	0.0360851 +- 0.00124617
	64	0.0779685 +- 0.101821	0.0881434 +- 0.110057
	100	0.0277953 +- 0.00296732	0.0353795 +- 0.00163168

Tabla 2: Results for Sine Dataset

3.3.3. Results for Quake Dataset

Layers	Neurons	Train MSE ± SD	Test MSE ± SD
1	2	0.165327 +- 0.075057	0.163568 +- 0.0754274
	4	0.0475008 +- 0.0225532	0.0448039 +- 0.0225956
	8	0.055017 +- 0.0350405	0.052587 +- 0.0356882
	32	0.0513887 +- 0.031012	0.0490565 +- 0.0318429
	64	0.0456043 +- 0.0121674	0.0424836 +- 0.0118869
	100	0.041641 +- 0.0120729	0.0388026 +- 0.0116336
2	2	0.0475693 +- 0.0143249	0.0450249 +- 0.0145282
	4	0.106999 +- 0.0329799	0.105027 +- 0.0331454
	8	0.033819 +- 0.00426385	0.0310616 +- 0.00438118
	32	0.0391623 +- 0.0110673	0.0361634 +- 0.0108072
	64	0.0353627 +- 0.0105226	0.0325254 +- 0.010257
	100	0.0354552 +- 0.0104848	0.0326212 +- 0.010217

Tabla 3: Results for Quake Dataset

3.3.4. Results for Auto MPG Dataset

Layers	Neurons	Train MSE ± SD	Test MSE ± SD
1	2	0.0503848 +- 0.0157352	0.0472035 +- 0.0148411
	4	0.0575833 +- 0.0200467	0.0587243 +- 0.0229748
	8	0.0184446 +- 0.00818181	0.0142829 +- 0.00590306
	32	0.0071128 +- 0.00151472	0.00766544 +- 0.00163126
	64	0.0267791 +- 0.027617	0.0178723 +- 0.0165191
	100	0.0301736 +- 0.0172557	0.0357913 +- 0.0299167
2	2	0.0421328 +- 0.0170314	0.0360756 +- 0.0179131
	4	0.0698431 +- 0.0251795	0.0661566 +- 0.0297423
	8	0.0324618 +- 0.0103745	0.027766 +- 0.00793947
	32	0.0179523 +- 0.00870276	0.0168803 +- 0.00647706
	64	0.0456452 +- 0.0745984	0.0424289 +- 0.0643001
	100	0.00765243 +- 0.000883788	0.00751826 +- 0.000955446

Tabla 4: Results for Auto MPG Dataset

3.4. Discussion

We are facing the most important section of the project carried out, in which the results obtained will be studied, reflecting on the reason for these results, on the parameters used, and even on changes that could produce significant differences.

In general terms, we can see a decrease in the average error obtained in the set of iterations as the number of hidden layers and neurons increases.

Even so, each dataset will be studied specifically since there are some significant variations, such as the fact that in certain datasets from a certain number of hidden layers or neurons the improvement is not significant or even does not exist.

It should be said that the project has focused on carrying out the experiments that were required, knowing that the improvements could have been exploited even further by studying in more detail the possible variations of parameters.

3.4.1. XOR results

This dataset is the simplest used. It should be said that the same data set has been used for both training and testing.

As we can see, the algorithm in this dataset makes a significant improvement in the change to 32 neurons, regardless of the number of hidden layers, although for the implementation with 1 hidden layer it begins to get worse if the number of neurons continues to increase.

Even so, the best error result obtained corresponds to the implementations with 2 and 5 hidden layers and 100 neurons, in which we see very low results for both the training set and the end of the test.

It is also worth saying that although the times have not been included, they are being taken into account, and therefore, clarify that the implementation with 2 hidden layers and 100 neurons does not vary much in time compared to the rest with worse results, while that that of 5 hidden layers and 100 neurons does increase significantly.

From these metrics obtained, we can conclude that if we want to obtain a good result and in just a couple of seconds, the most appropriate implementation is the one that is composed of 2 hidden layers and 100 neurons, obtaining a very low average error.

Finally, we are going to analyze the simplest implementation carried out with this datasaet, as was required of us. This implementation consists of a network with a hidden layer and 2 neurons, for which an MSE = 0.283139 and a SD = 0.0320771 have been obtained.

Although the MSE is not the lowest, it is still a small value, and a lower n MSE indicates that the model predictions are closer to the real classes.

Furthermore, in this case, the standard deviation of 0.0320771 indicates how much the individual errors vary from the mean (MSE). A low SD suggests that most individual errors are close to the mean, which can be interpreted as consistency in the quality of the predictions.

3.4.2. Sine results

The Sine dataset exhibits behaviors similar to the XOR dataset, suggesting that both datasets share some characteristics. One plausible explanation for this similarity could be the relatively small size of both datasets. In machine learning, smaller datasets often pose challenges due to limited diversity and variations. The neural network may quickly learn patterns in smaller datasets, potentially leading to similar behaviors across different datasets of similar size.

Upon closer examination of the metrics obtained for the Sine dataset, it becomes apparent that, regardless of the configuration of hidden layers, the neural network achieves its best performance when the number of neurons is set to 32. This consistency in optimal neuron count across different configurations indicates a robustness in the network's ability to capture underlying patterns in the Sine dataset.

Interestingly, when comparing the execution times for different configurations, it is noted that the differences are practically negligible. This implies that, in terms of computational efficiency, the various configurations perform similarly. Therefore, considering the obtained results, it can be concluded that the implementation featuring 1 hidden layer and 32 neurons stands out as the most effective in achieving a consistently low average error for the Sine dataset.

Drawing parallels between the behaviors observed in both the XOR and Sine datasets, it's noteworthy that the algorithm demonstrates similar performance characteristics for datasets of comparable sizes. This similarity suggests that an extensive computational effort may not be imperative for studying these datasets thoroughly. The effectiveness of the chosen neural network architecture in achieving low errors on both datasets reinforces the adaptability and generalization capability of the implemented algorithm to diverse small-sized datasets.

3.4.3. Quake results

In delving into the outcomes of the Quake dataset experiments, let's adopt a more serious technical perspective.

The influence of varying network architectures is evident. Notably, the 2-layer configuration outperforms its 1-layer counterpart across different neuron counts. Optimal performance is consistently observed around 32 neurons. Beyond this point, the marginal benefits diminish while computational overhead increases.

Efficiency is a key consideration. Interestingly, the 2-layer, 32-neuron model strikes a balance between accuracy and computational efficiency. Models with higher neuron counts incur computational costs without proportional accuracy gains.

The 2-layer, 100-neuron model stands out as the best performer, exhibiting low Test MSE. In contrast, the 2-layer, 4-neuron model struggles, resulting in a considerably higher Test MSE. An intriguing observation is the increase in computational time beyond 64 neurons, with a further increment for 100 neurons. This emphasizes the trade-off between model complexity and computational efficiency. Our efficiency champion emerges as the 2-layer, 32-neuron configuration. It demonstrates commendable accuracy while managing computational resources judiciously.

In conclusion, for the Quake dataset, the sweet spot lies in the 2-layer architecture with 32 neurons. This model delivers a commendable balance of accuracy and computational efficiency, a crucial consideration for real-world applications.

3.4.4. Auto MPG results

The impact of network architecture is discernible. The 2-layer configuration consistently outshines the 1-layer counterpart across varying neuron counts.

The optimal performance across different architectures aligns with a neuron count of 32. This is a recurring theme across both 1-layer and 2-layer configurations. Similar to the Quake dataset, the 2-layer, 32-neuron model demonstrates a commendable balance between accuracy and computational efficiency. The 2-layer, 100-neuron model again surfaces as a strong performer, showcasing lower Test MSE. On the contrary, the 2-layer, 4-neuron model exhibits relatively higher Test MSE.

A notable observation is the substantial increase in computational time beyond 64 neurons, with a further increment for 100 neurons. This emphasizes the trade-off between model complexity and computational efficiency. The 2-layer, 32-neuron configuration stands out as the efficiency champion. It strikes an optimal balance, providing noteworthy accuracy while managing computational resources effectively.

In conclusion, for the Sine dataset, the 2-layer architecture with 32 neurons emerges as the preferred model. This configuration excels in terms of accuracy and computational efficiency, reinforcing its suitability for practical applications. Also, that if we want the best result in terms of MSE referred, the best model is the 2-layer architectura with 100 neurons, that improves the best result with difference.

3.5. Final Conclusions

Once all these results and factors are taken into account, we can conclude that in general the algorithm obtains good results in terms of MSE in all cases, both for smaller datasets and for larger datasets. It is true that for larger datasets, from a certain number of neurons, the execution time increases noticeably, but it is not excessive either.

We can also observe that configurations with 32 neurons tend to be the ones that obtain the best results regarding execution time.

Finally, it should be added that when standardization is applied, as in the case of the last dataset, the results obtained are significantly better, which may imply that standardizing the data can help improve the results in some cases.

Even so, to truly analyze all these behaviors, a more in-depth study is necessary, including more parameter variations or the inclusion of new datasets, which would help us draw more accurate conclusions about all these hypotheses.

4. Bibliografía

- Curso Introducción a los Modelos Computacionales
- scratch-multilayer-perceptron
- davidalbertonogueira/MLP: Simple multilayer perceptron
- mLEARn: An Implementation of Multi-layer Perceptron in C++