Warsaw University of Technology

Neural Networks

Multilayered Perceptron

Authors:
Kołakowska Aleksandra
Litkowski Andrzej

April 14, 2022

Contents

1 Problem description

The goal of this project is creating a Multilayered Perceptron trained with Backpropagation and tested on two data sets from the UCI Machine Learning repository.

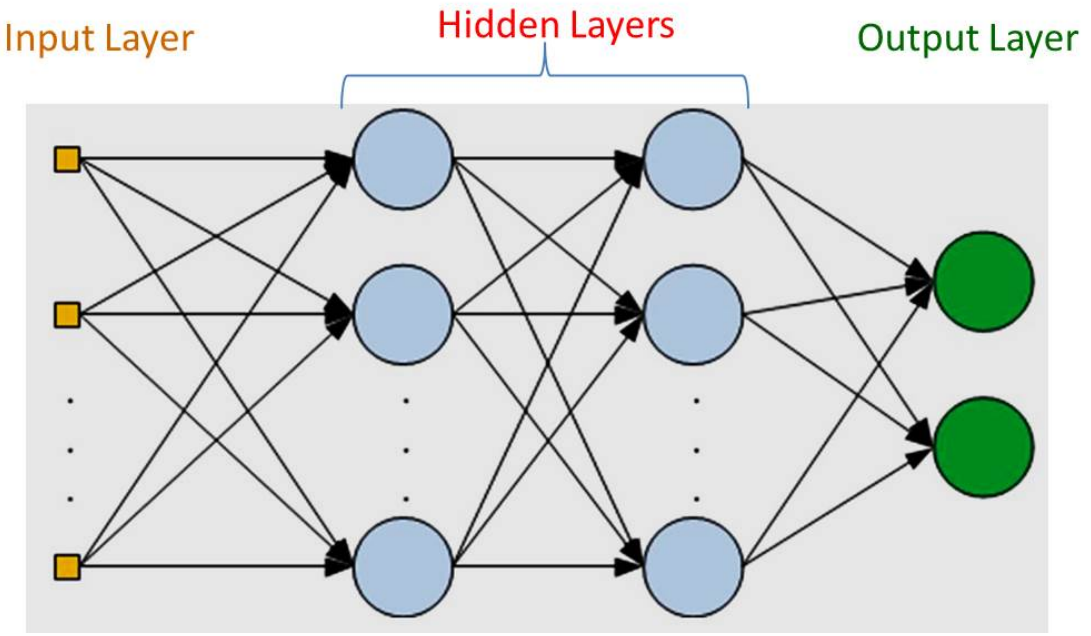

Figure 1: Basic design of multilayer perceptron[1]

The network consist of input layer - attributes of problem instance, output layer - where each neuron corresponds to one quantity we want to find, and several hidden layers. Each neuron in given layer (except input layer) receives input in form of weighted sum of outputs of all the neurons from previous layer with additional bias term, and then calculates its own output based on activation function. The process is repeated until the last layer finishes - then we can read outputs from the neurons and draw conclusions.

The main problem in such network is, how to choose all the weights so it works well. Apart from manual assignment, which in practice is impossible, there exists an algorithm that is supposed to do it - backpropagation. The main idea is to calculate error in the output, and change weights to make it better. Then, the process is repeated for the last hidden layers, but instead of output error, error calculated in next layer is used and combined with weight that contributed to that error - this way, if contribution to invalid output was small, the error will be small. The process is

repeated for the rest of hidden layers, until the first one is reached (hence the name, backpropagation). The whole process exists to minimize the error in the output.

We can improve the algorithm further, by adding two additional parameters. The first one is called learning rate - when the weight change is calculated based on the error, it is additionally multiplied by the rate. It should be kept small. This way, changes won't override all changes from previous iterations. The second parameter is momentum. Instead of simply changing the weights based on error, we additionally use scaled change from previous iteration. Normally backpropagation can lead to local minimum of the error, but it is not always the same as global minimum, but we would be stuck there. Momentum allows to jump over the local minimum to possibly find a better one (at the same time if this was a global minimum, we would come back there).

With algorithms established, we shall describe how the training works. We take one problem instance with know result, feed it to the network and receive output. Based on the output, we calculate error and use backpropagation algorithm. Then weights are updated. We can repeat it many times - we call one iteration of training using all instances in the training set an epoch. We can repeat training on the whole set several times, but we must remember, that each time we should randomize the order to avoid remembering specific pattern. However, it is not the most efficient option. It is better performance-wise, to accumulate changes for several instances and then apply them at once. The set of instances for one update is called a batch. So we divide one epoch into several batches and we update weights between them. The smaller the batch size, the better the results, but worse performance.

With all that, only two problems remain. Firstly, which weights should we use at the beginning? The answer is, small, random ones. We won't be able to guess what they should look like anyway, and small ones will be easier to adjust by the algorithm. The second one is about the activation function described at the beginning. It should have two properties - the domain should be, $(-\infty, +\infty)$, and the derivative should be easy to compute. Examples are $\tanh x$ with derivative $\tanh' x = 1 - \tanh^2 x$ or sigmoid (logistic) function - $\frac{1}{1+e^{-x}}$ with derivative $S'(x) = S(x) * (1 - S(x))$.

2 Solution description

Our solution is divided into two separate programs written in C# and .NET 5.0 (plus script for creating graphs written in R).

2.1 Training and classification

The first program is for training the network, and the output is file with trained network, the other one use trained network to get predictions. Both programs doesn't have GUI and are used from console. They both require json files with settings. We provide listings with example settings and description of parameters.

Let's start with listing for training program. It consists of network parameters. Two activation functions are supported. Output layer also use the same function, even though we focused on classification problems without multiple classes at once, where softmax function would work a little better.

```
{
  "DataSetPath": "./dataset.txt", // Path to the learning
     dataset
  "NumberOfInputParameters": 3, // How many parameters
     input have
  "OutputNetworkFilePath": "./network.nn", // Path for
     the output neural network that can be used in
     classification part
  "Layers": [ 4, 5, 6, 7, 8, 9, 10, 2 ], // Number of
     neurons in layers (not including input), so only
     hidden layers and ouput layer (the last one)
  "ActivationFunction": "Tanh", // "Sigmoid" or "Tanh"
  "Epochs": 20, // Number of epochs
  "Momentum": 0.9, // Momentum
  "BatchSize": 20, // If 0, there is only one batch with
     all inputs per epoch
  "LearningRate": 0.1 // Learing rate
}
```

For classification program, we don't have any parameters apart for input.

```
{
  "NetworkFilePath": "./network.nn", // Path to the
     network file from Training part
  "InputFilePath": "./data.txt", // Path to data
```

```
4   "OutputResultFilePath": "./output.txt" // Output of the
        classification
5 }
```

Obviously, input data has to be prepared. In both cases all the instances should be in one text file, with one instance per file. In each line we give all attributes of the instance, in order, separated by tabs. In case of training input, we additionally have to add the class at the end, separated from the last attribute by tab. Number of classes should match number of neurons in the last layer, and they should be counted from 1. Example of training dataset with 3 attributes and 2 classes:

94	134	183	1
95	132	182	1
92	132	181	1
94	131	181	1
198	198	158	2
198	198	158	2
198	198	158	2
198	198	158	2

And classification data (with the same 3 attributes) can look like that:

70	81	119
70	81	119
163	162	112
163	162	112

After datasets are prepared and settings files are created, program usage is as simple as opening the console and running executable with pah to settings file as parameter. Let's assume that training program is named "training.exe", settings for training are names "training-settings.json" and both are in current working directory, the command would be "./training.exe ./training-settings.json". Similarly, if classification program is named "classification.exe" and settings are "classification-settings.json", the command would be "./classification.exe ./classification-settings.json". Remember that all paths in settings can be given as full paths or relative paths. In case of relative ones, they will be relative to current working directory in CLI.

Additionally, it is worth to notice, that network file obtained from the training is in reality just a json file, so any internal information like weights and layers are easily obtainable by opening the file in any text editor.

2.2 Testing in R

In order to test the program the user should open the Tests folder and file training_error.R. For both of the functions, inputPath corresponds to a text file from folder Training and it should look similar to "/Training/iris_training_150.txt". The outputPath is always the same and equal "/Classification/output.txt". The functions in the file are:

1. getAccuracy <- function(inputPath, outputPath)
   returns accuracy as a percentage
2. plotDataSet <- function(inputPath, outputPath)
   graphs data set and returns error

All other *R* functions text files or test different hypotheses.

3 Description of test data sets

We decided to use two data sets of different sizes - Skin Segmentation Data Set [3] that has 245057 instances and Iris Data Set [2] that has 150 instances. In case of the Skin Segmentation Data Set most of the tests were performed on the subset of that data set - first 8098 instances.

Both of the data sets were normalized using R so each row of the data set uses the following format. Since these data sets contain class column and are used for classifications, so in our program we predict classes based on the attributes given in the data set.

$$\mathbf{attr_1 attr_2 ... attr_n y}$$

where $\mathbf{attr_n}$ is one of the attributes(inputs)
that we use in the Input Layer and $\mathbf{y}$ is a value that we predict

3.1 Skin Segmentation Data

The skin dataset is collected by randomly sampling blue, green, and blue values from face images of various age groups, race groups, and genders obtained from FERET database and PAL database. There are 245057 instances in this data set, out of which 50859 is the skin samples and 194198 is non-skin samples. The attributes of this data set consist of:

- B - represents blue values, ranges from 0 to 255
- G - represents blue values, ranges from 0 to 255
- R - represents blue values, ranges from 0 to 255
- class label - represents where the sample was taken from, 1 for skin sample and 2 for non-skin sample

During the implementation we use 8098 first instances to train the neural network.

3.2 Iris Data Set

This data set contains information about 3 different types of iris plant. It is worth noting that one of the types(classes) is not linearly separable from the others. There are 150 instances in this data set, with each type containing 50 instances.

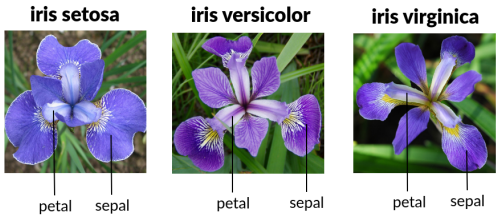

Figure 2: Iris types[1]

The attributes of this data set consist of:

- Sepal length - represents the length of the sepal, is a quite good predictor of a class, ranges from 5.0 to 7.4
- Sepal width - represents the width of the sepal, is a bad predictor of a class, ranges from 2.3 to 4.4
- Petal length - represents the length of the petal, is an extremely good predictor of a class, ranges from 1.0 to 6.1
- Petal width - represents the width of the petal, is an extremely predictor of a class, ranges from 0.2 to 2.5
- class label - represents what type of iris the sample flower is, 1 for Iris Setosa, 2 for Iris Versicolour, and 3 for Iris Virginica

4 Experimental setup and results

We performed various tests using R scripts and output generated by neural network in txt files. We wanted to find the answers to the following questions:

- How does the activation function selection affects the model's accuracy?
- How does the number of hidden layers and the size of that layers impact the model's accuracy?
- How can we calculate the training and test error?

Additionally, we will train and test the Iris Data with different number of attributes.

4.1 Activation function

In our program, we implemented two different activation functions **tanh** and **sigmoid**.

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} \text{ and } \sigma(x) = \frac{1}{1+e^{-x}}$$

The main disadvantage of the **sigmoid** function is that it returns only positive values from range 0 to 1 while the **tanh** function returns values from range -1 to 1. This makes **tanh** much faster and less likely to be stuck on one value.

Figure 3: Activation Functions, Sigmoid(left) and Tanh(right), tested on the subset of Skin Segmentation Data Set. The red dots represent values from the dataset, the green ones represent predicted values.

We tested both functions with different sets of parameters. On average, after 20 epochs **tanh** had 95% of accuracy while **sigmoid** had 92% of accuracy.

4.2 Hidden layers

We performed many manual tests to check how the number of hidden layers and the number of neurons in each layer would affect our model.

- 1 output neuron and simple relation between attributes in the input and expected output $\Longrightarrow$ we used 1 hidden layer and for the number of neurons, we tested between $\frac{2}{3} * \text{inputNeuronsCount}$ and inputNeuronsCount
- multiple output neurons or complex relation between attributes in the input and expected output $\Longrightarrow$ we used 1 layer with inputNeuronsCount + outputNeuronsCount neurons or split that 1 large layer into a few smaller ones
- if the previous setting did not work $\Longrightarrow$ we used 1 layer with $2 * \text{inputNeuronsCount}$ neurons or split that 1 large layer into a few smaller ones

4.3 Training and Testing Error

The training error is calculated when we use the same data set to train and predict values. We calculated training error for network trained on the subset of Skin Segmentation data set.

Figure 4: Skin Segmentation Data Set trained with different epoch settings

Both networks had the same settings except epoch value - network on the left was

trained for 20 epochs while the one the right was trained for 500 epochs. We run the network with settings that resulted in high training error on the entire Skin Segmentation Data Set. The data set was too large to display as a plot but we calculated that training error was quite high, around 0.21.

```
> plotDataSet("/Training/Skin_NonSkin.txt","/Classification/output.txt")
[1] 0.207539
```

Figure 5: Iris types[1]

The testing error can happen when we use one data set to train neural network and then predict values of a second data set. The second data set can have no rows in common with the first data set. For the Iris data set, we decided against dividing the data set into two parts - there are only 150 instances and 75 instances would not be enough to train the network. In order to calculate the testing error for Skin Segmentation Data Set, we trained the data set with the first 8098 values from that data set. Then, we predicted the values for another subset of Skin Data set of size 8098. The second subset was disjoined from the first subset and all expected class values were equal 1. The testing error was equal to 0 for these subsets.

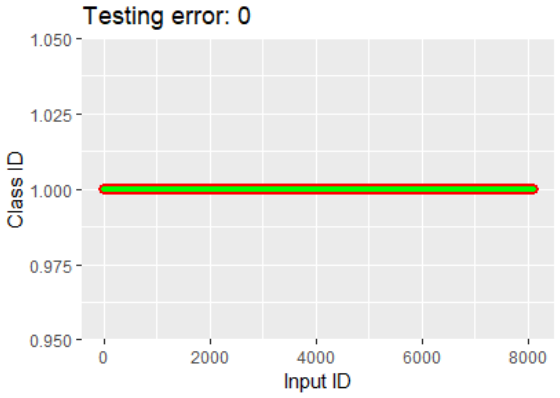

Figure 6: Testing error for Skin Data Set

4.4 Iris Data Set

We wanted to test if the information given about the data set was true. In the iris_README file, the authors stated that the worst prediction occurs for the sepal width. We can see the comparison between expected values(red) and predicted values(green) for sepal width on Figure 7.

Figure 7: Classification by Sepal Width of Iris

The training error was extremely high, around 0.67. The authors of the data set also stated the best class prediction occur when we use for petal length and petal width columns. We can see the comparison between expected values(red) and predicted values(green) on Figure 8.

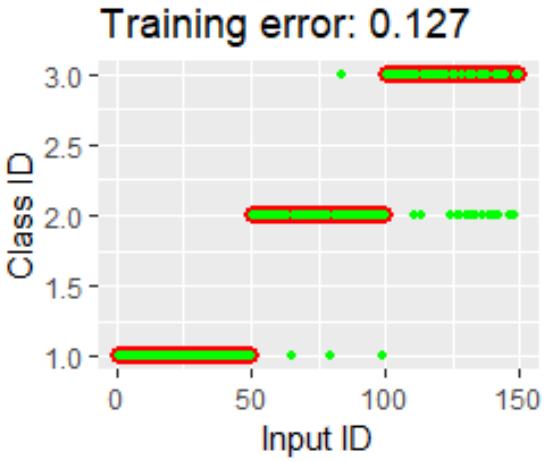

Figure 8: Classification by Petal length and Petal width of Iris

Now, we can look at the graph displaying predictions when we use all attributes of Iris data set as input and change the epoch. We used standard paramters for this data set - learning rate of 0.1, batch size equal 10, momentum equal 0.9, **sigmoid** as an activation function and 1 hidden layer with 4 neurons.

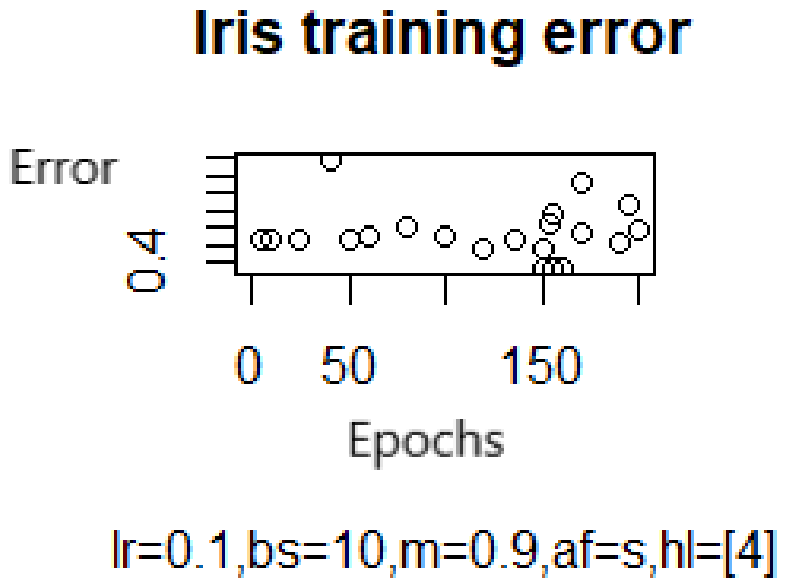

Figure 9: Training Error for Iris Data Set

As can be seen on Figure 7 and Figure 8, attributes of Iris data set lead to either minimal or maximal training errors. This results in a lack of reproducibility of results when we train the network using all attributes of Iris data set.

5 Conclusions

- Due to randomness during training and weights initialization, training with exactly the same parameters and dataset can give networks that perform differently. When we checked the difference in terms of training error, the error could be between 1 and about 45%.
- With our simple datasets, it was often the case where smaller network with lower amount of layers and neurons were performing better.
- Increasing the momentum was improving the outcomes.
- If the number of epochs was too big, although training error was small, on other data the network performance was bad.
- In case of flower dataset, two classes are linearly inseparable - and those two classes are recognized as one also in our networks.

We also have some conclusions about how to improve our solution.

- Since we focused our attention on classification problems with only one class, we could change the activation function to the softmax function.
- We could use adaptive learning rate instead of constant one.
- Instead of simple error we used during backpropagation, we could use better suited errors like mean squared error.

References

[1] Basics of multilayer perceptron. Accessed: 2022-03-12.

[2] Fisher R.A. Iris data set. Accessed: 2022-03-12.

[3] Abhinav Dhall Rajen Bhatt. Skin segmentation data set. Accessed: 2022-03-12.