Deep Learning

Multilayer perceptron employing Backpropagation Algorithm

Paulina Pacyna, Mateusz Wójcik
Warsaw University of Technology, Faculty of Mathematics and Information Science

Student ID:290600, 290639

Keywords: neural network, perceptron, backpropagation, stochastic gradient descent, classification, regression

16th March, 2021

1 Introduction

In the project, we were supposed to perform a low-level implementation of Artificial Neural Network, Multilayer Perceptron. To learn the network, we have used the backpropagation algorithm while calculating the gradient of the loss function with respect to the weights and biases of the network for the single input-output example. The application covers both the classification and regression problem depending on the learning data.

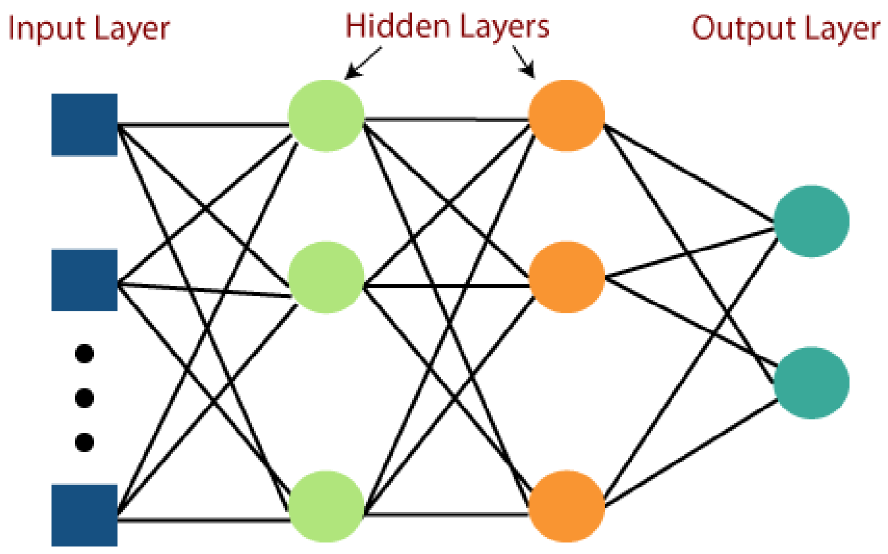

Fig. 1. MLP schema (source).

After implementing the structure of the network mostly using *NumPy* Python package, we tested different hyperparameters configurations.

2 Datasets

Together with the project description, the training and test sets - for either classification and regression problem - were included. Each dataset consisted of two types of the data - both for training and testing - having different sizes $(100, 500, 1000, 10\,000)$.

3 Methods

The source code of the project is in the form of Python repository consisting of a few scripts as follows:

- *network.py* - core Python file consisting of two main classes **Layer** and **Network**. It is the implementation of the network as well as its training algorithm.
- *classification.py* - tests performed on the classification problem data.
- *regression.py* - tests performed on the regression based data.
- *iris.py* - additional tests on the known Iris dataset focused on the classification (small dataset).
- *circle.py* - preparing of the benchmark data in the shape of circles and testing the network on such sets.

The **Layer** network consists of the activation function type for the layer, the dimensions of the input and output. The objects of this class also store weights, biases and a momentum. A *delta*, which is the rate of change of the cost with respect to any bias in the network and is propagated through the network, is also kept in the **Layer** class.

```python
class Layer:
  def __init__(
        self,
        n_input: int,
        n_output: int,
        activation_type: str = "sigmoid",
        init_sigma=1,
  ):
        self.weights = np.random.normal(0, init_sigma, n_output * n_input)
        self.bias = np.random.normal(0, init_sigma, n_output)
        self.activation_type = activation_type
        self.n_input = n_input
        self.n_output = n_output
        self.momentum = np.zeros(self.weights.shape)
        self.momentum_bias = np.zeros(self.bias.shape)
        self.outputs = np.zeros(n_output)
  def update_weights(self, W, learning_rate, momentum_rate):...
  def update_bias(self, b, learning_rate, momentum_rate):...
  def activation(self, x):...
  def fit(self, inputs: np.array):...
  def lin_comb(self, inputs):...
  def set_delta(self, delta):...
```

Listing 1: Network class

The **Network** class consists of the whole network, its hyperparameters and the training method. While building the network, we can set:

- architecture of the network (number of hidden layers and their neurons),
- the activation function,
- the standard deviance of the normal distribution used for assigning the weights for the first time,
- learning and momentum rate,
- number of epochs,
- the cost function,
- the size of batches used for learning (we update the weights after each batch),
- a printing progress parameter,
- if the network is applied to the regression or classification problem.

For the regression problem, we first preprocess the data for better convergence. The preprocessing consists of scaling training and test data before learning and prediction.

$$\text{scale}(X) = \frac{X - X_{\min}}{X_{\max} - X_{\min}}.$$

Variables are scaled independantly in the case of higher dimensions.

In the project, we focused on the linear algebra aspects of maintaining a neural network and tried to implement maths based approaches, e.g. by using matrix multiplication which is quite effective in *NumPy* library.

The steps of the training algorithm are as follows:

1. Preprocessing of the data if needed.
2. Shuffling of the training data and splitting the data into batches.
3. Using random weights to calculate the output of the network and calculating the gradient of the loss function using the output layer.
4. Going back to front and calculating the rates of change of the loss function in remaining layers.
5. Updating the weights and biases front to back.

4 Results

The result can be assessed by running the algorithm on sample datasets. At first we may discuss accuracy of the algorithm on datasets provided by the teacher, concerning classification and regression problems.

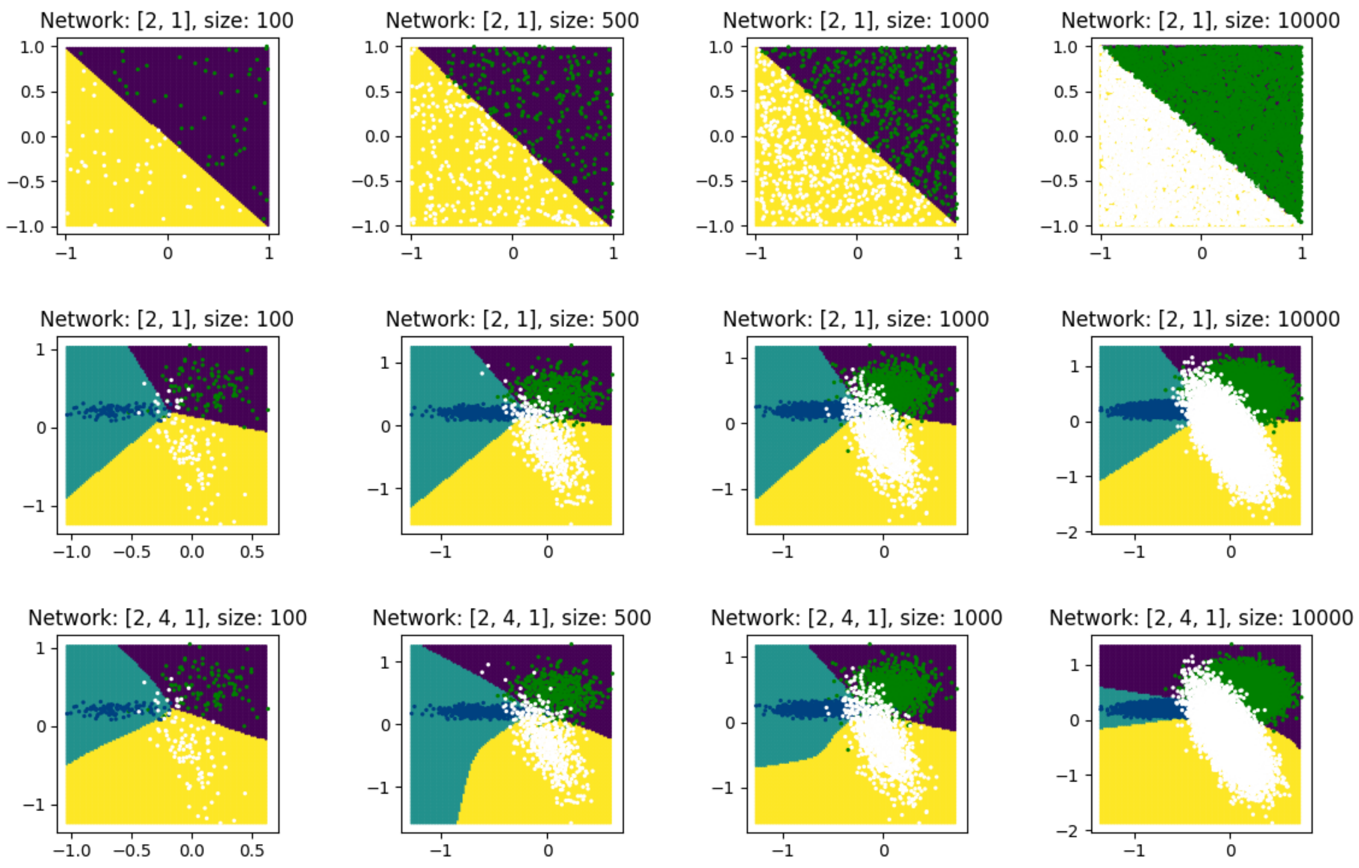

Fig. 2. Assessment on 'classification' dataset.

We checked if the network works corectly if it is reduced to one layer, using iris dataset. The accuracy of the classification equals about 96%.

To check if the model can fit to very nonlinear shapes, we generated a simple dataset for classification consisting of circles. The dataset and results of classification are shown below.

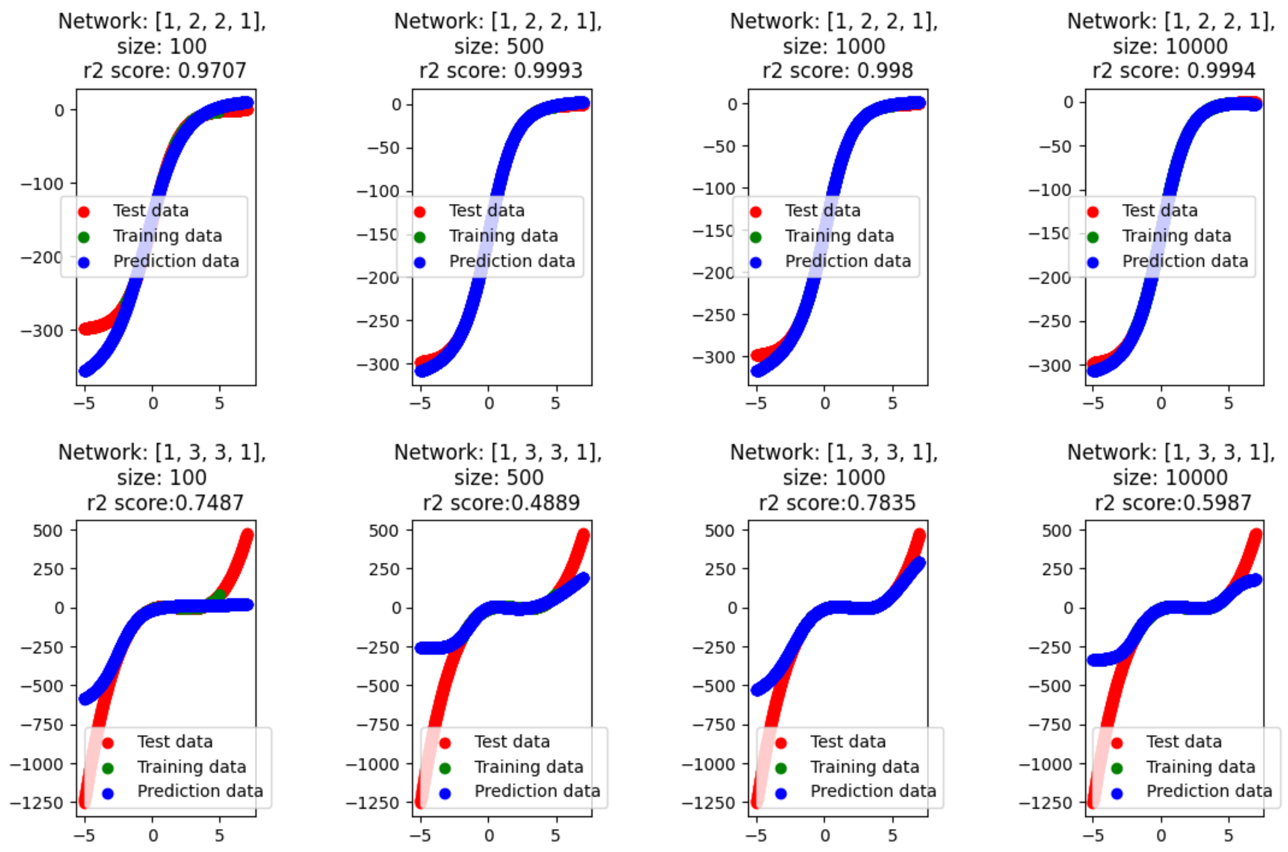

Fig. 3. Assessment on 'regression' dataset.

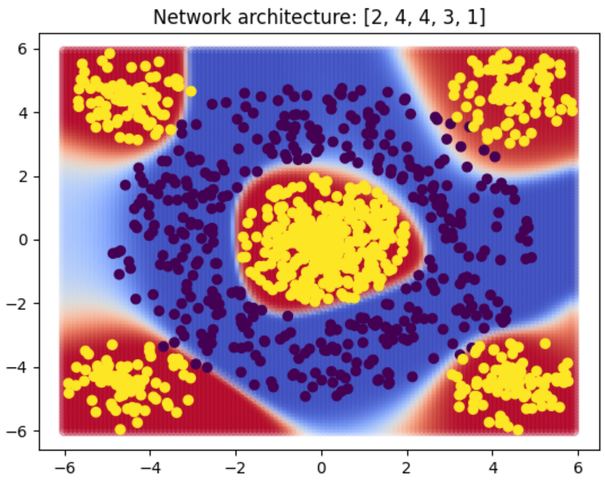

Fig. 4. Classification for the dataset consisting of circles.

4.1 Activation Functions

We wanted to compare the activation functions for the classification problem. For this case, we applied the architecture of one hidden layer with 4 neurons with learning and momentum rate equal to 0.1.

Number of observations	**Activation function**		
	sigmoid	*tanh*	*relu*
100	0.52	0.9167	0.9267
500	0.8473	0.9273	0.9207
1000	0.919	0.9246	0.9263
10000	0.9295	0.9318	0.9291

Table 1
Number of epochs - 1, batch size - 16.

Number of observations	**Activation function**		
	sigmoid	*tanh*	*relu*
100	0.9366	0.9266	0.9233
500	0.9333	0.9313	0.9326
1000	0.9273	0.9273	0.9313
10000	0.9335	0.9339	0.9328

Table 2
Number of epochs - 10, batch size - 16.

Number of observations	**Activation function**		
	sigmoid	*tanh*	*relu*
100	0.4233	0.59	0.8433
500	0.8393	0.9247	0.9166
1000	0.9177	0.9193	0.9213
10000	0.9304	0.9332	0.9333

Table 3
Number of epochs - 1, batch size - 50.

We can note that using tanh and reLU activation functions speeded up the learning process. Also tanh was more consistent than the sigmoid activation function. For higher size of the batch, applying reLU created the best results for 1 epoch.

4.2 Network Architecture

We tested how the architecture of the network incluences the model's accuracy for classification. We again used the given dataset with 3 groups. The number of epochs was set to 10, batch size to 50 and activation function to reLU.

Number of neurons	**Number of observations**			
	100	500	1000	10000
2	0.9233	0.8346	0.8306	0.8324
8	0.9333	0.9366	0.9276	0.9319
32	0.9266	0.936	0.931	0.9361
64	0.5333	0.64	0.9303	0.938
128	0.39	0.9313	0.9283	0.9366

Table 4
One hidden layer.

Note that, for only 2 neurons in a hidden layer, less observations created better accuracy. The model was not complicated enough for more data. On the other hand, as we increase the number of neurons, we see that we lack observations to teach the model. Nonetheless, for 1000 and 10000 observations, we obtain quite decent results, even for 128 neurons.

Let's now build the model with 2 hidden layers and the same number of neurons each.

Number of neurons	**Number of observations**			
	100	500	1000	10000
2	0.5433	0.9193	0.3333	0.5438
8	0.9166	0.9286	0.3333	0.9374
16	0.9233	0.688	0.9303	0.9345
32	0.3333	0.452	0.512	0.333
64	0.1966	0.3333	0.915	0.6472

Table 5
Two hidden layers.

Similarly, we can note that the more complicated the model is, the more observations the model needs to learn effectively. On the contrary, when we set more than 16 neurons per hidden layer, the accuracy drops down and the algorithm does not converge (for example all the predicted groups are 0).

In the following tests, we use *sigmoid* activation function and some different combinations of neurons in hidden layers (even for more than 2 layers).

Number of neurons in hidden layers	**Number of observations**			
	100	500	1000	10000
$8, 4$	0.6033	0.9266	0.9286	0.9325
$4, 4, 4$	0.3333	0.9213	0.9136	0.9354
$4, 8, 4$	0.61	0.928	0.926	0.9370
$4, 8, 8, 4$	0.4933	0.906	0.9223	0.9368
$2, 2, 2, 2$	0.6866	0.9406	0.9306	0.9338

Table 6
Two hidden layers.

4.3 Loss function

We have also tested three different loss functions: *quadratic*, *cross-entropy*, and *hellinger*.

$$\textbf{Quadratic: } C(a,y) = \frac{1}{2}\sum_j \left(a_j^L - y_j\right)^2$$

$$\textbf{Cross-Entropy: } C(a,y) = -\sum_j \left[y_j \ln a_j^L + (1-y_j)\ln\left(1-a_j^L\right)\right]$$

$$\textbf{Hellinger: } C(a,y) = \frac{1}{\sqrt{2}}\sum_j \left(\sqrt{a_j^L} - \sqrt{y_j}\right)^2$$

We performed the test on the architecture of two hidden layers with 8 and 4 neurons. First, we considered only 1 epoch for each dataset.

Number of observations	**Loss/Cost function**		
	hellinger	*cross-entropy*	*quadratic*
100	0.3333	0.3333	0.1
500	0.626	0.834	0.932
1000	0.606	0.886	0.6033
10000	0.9324	0.9309	0.9322

Table 7
Batch size - 50, learning and momentum rate - 0.1.

For more consistent and stable results we also considered only 10 epoch for each dataset.

Number of observations	**Loss/Cost function**		
	hellinger	*cross-entropy*	*quadratic*
100	0.8733	0.92	0.88
500	0.9266	0.9366	0.9266
1000	0.923	0.919	0.9236
10000	0.935	0.9307	0.9352

Table 8
Batch size - 50, learning and momentum rate - 0.1.

Looking at the previous tables, we can note that for small number of iterations *Cross-Entropy* loss function was the most stable and consistent. For 10000 learning observations, all of the loss functions performed very well. In the second table, we can see that again *Cross-Entropy* loss function obtained the best results, even for small amounts of data. On the other hand, the accuracies of the networks with *Quadratic* and *Hellinger* cost functions applied was also satisfactory and quite stable taking into account different sizes of the data.