Universitat Pompeu Fabra Barcelona

Deep Learning, laboratory 1

Basic MLP Implementation with Numpy/PyTorch

Using Optimizers and Autograd

Lida Calsamiglia, Patricia Castelijns, Pau Cobacho

u172787, u172949, u161616

May 8, 2023

1 Load, visualize and normalize data

We first load the train and test csv files into two datasets is staightforward using the pandas function `read_csv`. The datasets are 2-dimensional (features X and Y) with two different classes.

To better understand the thata we are working with, once the train and test data are loaded, we can visualize the data with matplotlib:

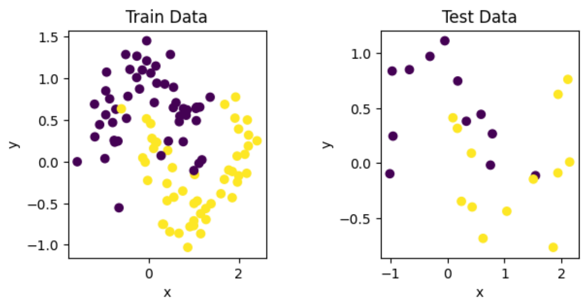


Figure 1: Raw data

1.1 Preprocessing of data: Normalization

For better training, data is usually normalized in order to unify the values so that they belong in similar scales and the model can be applied in the future to different datasets. It also helps us in further steps to know the domain we are working with when we set a hyperparameter (e.g., the learning rate).

Observe that the dimensions of the samples are not normalized. We normalize the data using z-score normalization: $x := \frac{x-\mu}{\sigma}, \forall x \in (X, Y)$ so that the data is centered in zero with a unit standard deviation.

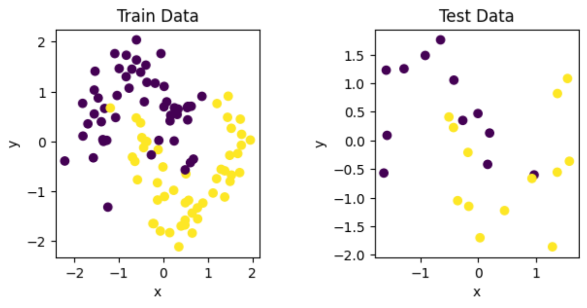


Figure 2: Normalized data

2 Exercise 1: Basic Multi Layer Perceptron

Multi-layer Peceptrons (MLP) are a type of Neural Networks that are able to classify between groups of data by being able to recognize non-linear relationships between groups.

A MLP will consist of an input layer where the preprocessed data will be fed, a number of hidden layers that will extract features of the data and an output layer with the estimated classification made.

2.1 Implementation and training of a Multi-Layer Perceptron (MLP)

We implemented a MLP with three layers: input layer (consisting of two inputs X,Y), the hidden layer (which has 3 neurons) and an output layer (constisting of one single neuron).

It has been implemented defining a class which initializes the parameters to be optimized (the weights), and defines the functions that will be used for the training of the MLP. For better training we initialized the weights using *Normal Xavier initialization* instead of random initialization:

$$W_l \sim Z\left[-\sqrt{\frac{2}{in+out}}, \sqrt{\frac{2}{in+out}}\right]$$

where *in* is the size of the previous layer and *out* is the current layer's (l) size.

Starting from the input layer, the data is propagated forward from the left to the right, where the output layer is i.e., *forward pass*. First, we apply the sigmoid activation function, σ, to the output neuron, $z = X \cdot W_1$, σ has to be a differentiable function so that afterwards the MLP can use gradient descent and learn the optimal weights:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Then, we calculate the loss between the predicted output and the *real* one. The aim of the training is to minimize the loss function, hence, at the next step we perform back-propagation, where we compute the derivatives of the loss with respect to each weight and bias and update them.

These steps are repeated a number of *epoches* (iterations) so the weights achieve the optimum value, i.e. the neural network *learns* the weights that minimize the loss function. We defined a training function, which computes the derivatives for the gradient descent and updates the weights at each iteration.

2.1.1 Choosing the Loss-Function

We first started defining the loss function as the L2-Loss function: $L = (y - \hat{y})^2$, where $\hat{y}$ is the output of the MLP and y is the *real* value. But the results were better using the Cross-Entropy Loss function:

$L = -ylog(\hat{y}) - (1-y)log(1-\hat{y}))$. See below the plot of the resulting cost function depending on the loss function used:

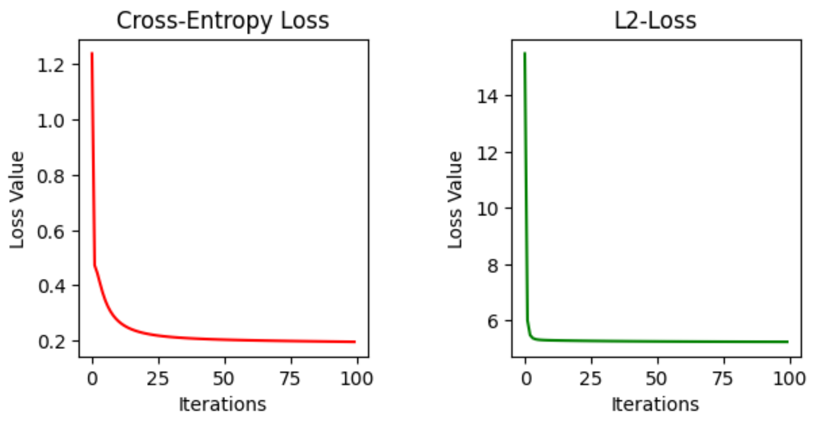


Figure 3: Plots for Cross-Entropy loss, L2-loss

Observe that the cost function converges to 0 when using the Cross-Entropy Loss function, whereas the minimum reached using the function is around 5. So we chose to use the Cross-Entropy function to improve our training phase.

2.2 Visualization of the decision boundary

When plotting the decision boundary in 2D classification we obtained the probabilities for each train sample of belonging to one class or another. The blue and red areas are the ones where the model is sure about the accuracy of the classification done for those samples belonging to those area, whereas in the areas with intermediate colours, the model is not so sure about it.

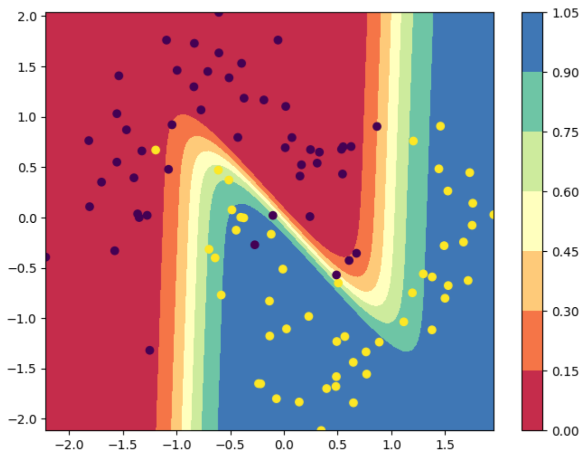


Figure 4: Decision boundary for our model

2.3 Estimation of the classes of the test data

To know how well our model works we computed the accuracy of our model using sklearn.metrics that compares each pair of y_predicted and y_true and computes what fraction is well predicted. In one of the runs, we obtained the following results:

- Train set accuracy: 0.97
- Test set accuracy: 0.92

These results are considerably good compared to the first results obtained at the beginning with random values and taking into account the not very high complexity of our network. [1]

3 Exercise 2: MLP using SDG with momentum

Momentum is an extension to the gradient descent optimization algorithm that has the effect of dampening down the step size with each new point, i.e. leads to faster convergence and reduced oscillation. Momentum accelerates the gradient descent using an exponential moving average, which assigns greater weight on the most recent values, still taking into account past gradients.

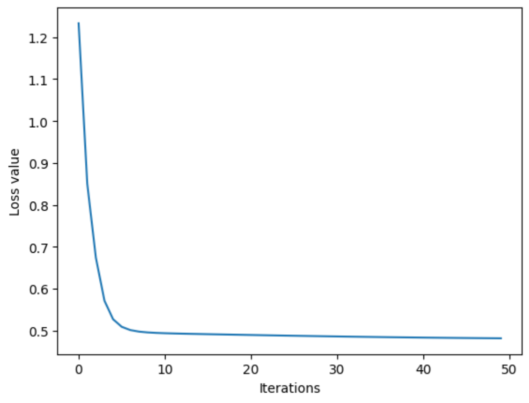
$$v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot \alpha \cdot grd_t$$

where $\beta \in [0, 1]$ is the momentum rate, α the learning rate and grd_t the gradient at time t.

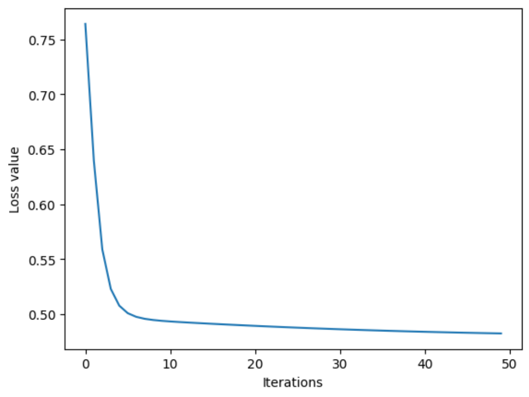
[1]Take into account that each time the code runs, these results will slightly vary.

The good point about momentum is that gradients from the past will push the cost further to move around a saddle point.

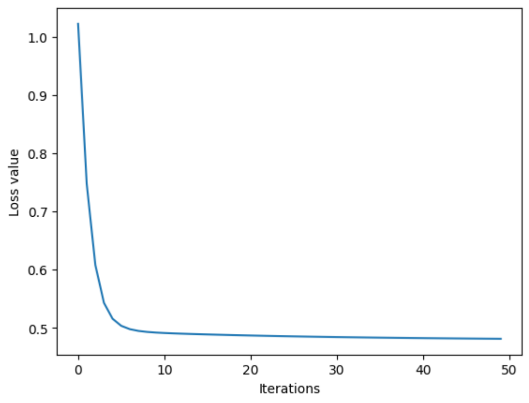
Below we can observe the effect of changing the momentum rate β at learning rate $\alpha = 0.001$.



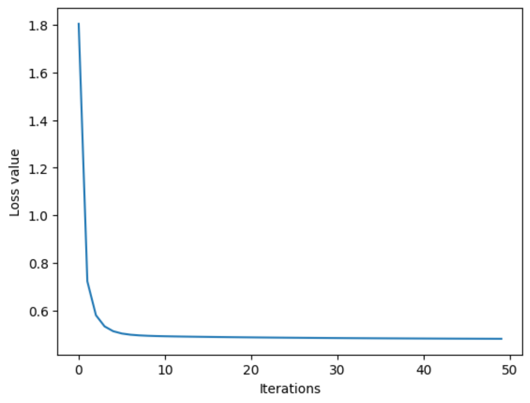
(a) $\beta = 0.0$



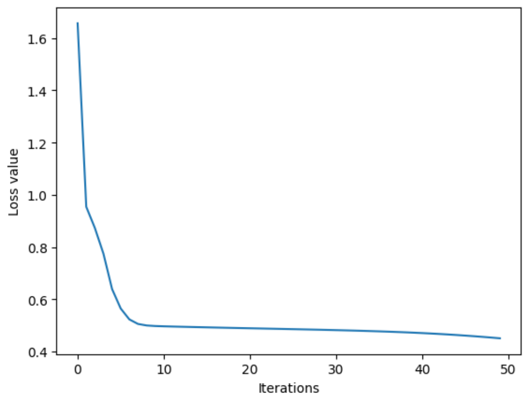
(b) $\beta = 0.3$



(c) $\beta = 0.6$



(d) $\beta = 0.9$



(e) $\beta = 0.99$

We can observe that increasing the momentum rate results in a faster decreasing loss, that is, the loss function converges earlier. However when $\beta > 0.9$ it does not go smoothly.

4 Exercise 3: MLP using PyTorch

PyTorch is an open-source Deep Learning framework that provides a way to build and train neural networks efficiently. The objective of this last exercise is to replicate the MLP constructed in exercise 1 now using the PyTorch framework and to analyze the performance of the neural network in terms of the number of hidden neurons and different learning rates.

4.1 Implementation

The tensor is the basic building block of Pytorch, this data structure encodes the inputs and outputs of a model as well as its parameters. Tensors can run on GPUs and other hardware accelerators and are optimized for automatic differentiation.

So before starting to implement the MLP we define tensors to represent the training data with which the model will be trained, for this we use our z-score normalised training data set. Once the tensors are defined we start building the MLP class in a similar fashion as in the exercise 1: that is, with a constructor method to initialize the network with a specific number of input (2 by default), hidden (3 by default) and output (1 by default) neurons and a concrete loss function and with a method that defines the forward pass.

For the forward pass implementation we just call a PyTorch method that takes care of the needed computations and the loss function is as well already defined inside PyTorch.

Therefore, this MLP class is exactly the same [2] as the one from the first exercise but applied using a framework that allowed us to go through the implementation process a little easier and with already proven and trusted methods.

4.2 Train MLP varying the number of hidden neurons and learning rate

The PyTorch framework allows for a simpler way of defining the training for the MLP as the back-propagation algorithm is already implemented in PyTorch.

To test the MLP we initialize different instances of the class, all of them with a different hidden neurons number and a different optimizer. The PyTorch optimizer class provides us with all the needed

[2]With the only exception that here we use the MSE loss function instead of the Cross Entropy.

algorithms to adjust the model and here we just use it just to modify the learning rate as we chose same optimization algorithm for each instance, the SGD.

We defined 3 different MLPs, each one of them characterized by a different number of hidden neurons (3, 6 and 11) and 3 optimizers assigned to each MLP (that is 9 in total) with different learning rates of values 0.01, 0.005 and 0.5. Following we present the results.

4.3 Analysis and discussion of the results for each case

Below there's the evolution of the loss function during the training stage for each of the 9 cases described above:

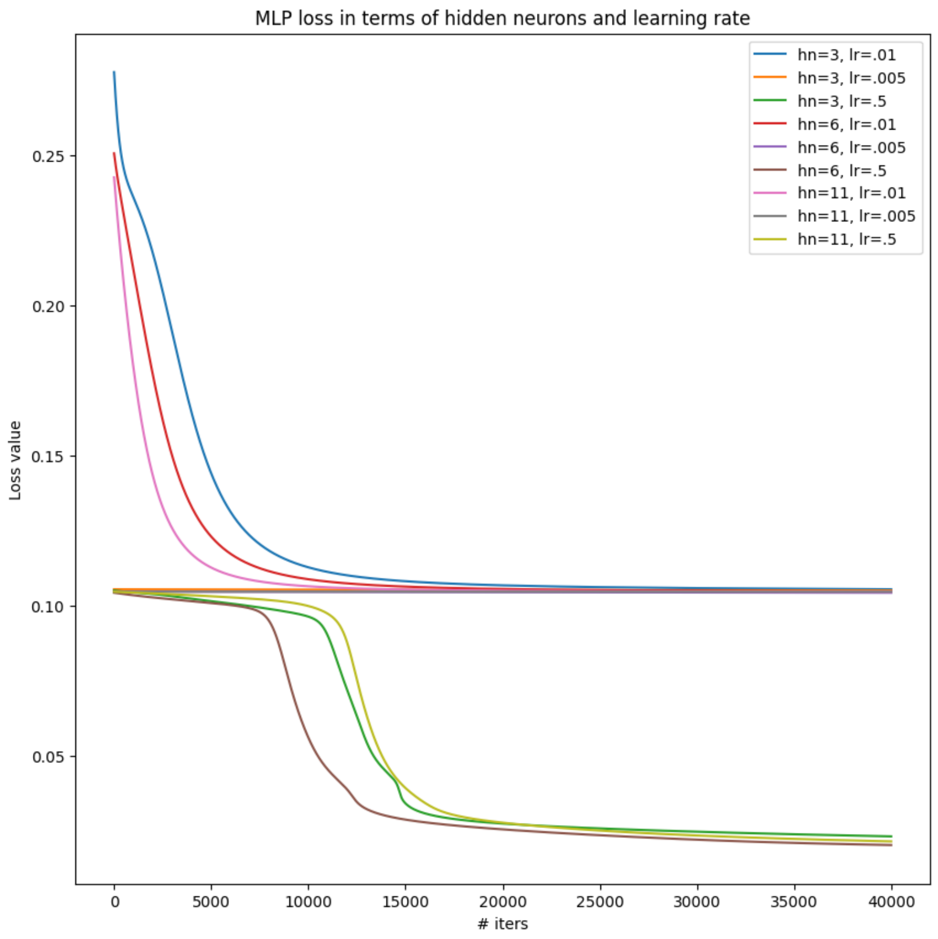


Figure 6: Analysis of the loss function value evolution through the iterations for different hidden neuron numbers (hn in the legend) and learning rates (lr in the legend).

Observing the figure above we can deduce how different parameters affect the network performance in terms of the loss function minimization. We want to stress the fact that this analysis applies for the data set used for this lab as different results could be obtained with the same parameters but different data sets.

From one hand, we see that the smallest learning rate (value of 0.005) seems to have found a local minimum right at the start and doesn't move from there, it is a bit odd to us as we were expecting the lowest learning rate to advance slowly to the local minimum and not find it just at the start. However, the interesting thing to notice here is that the biggest learning rate (of value 0.5) has been able to reach a much lower minimum than the other one of value 0.1. We think that the largest learning rate was able to discard the local minimum and that the learning rate of 0.1 was not able to see any further as the gradient descent did not advance enough to observe a better optimization. This observation allowed us to notice the importance of the learning rate in terms of the minimization of the loss function as the gradient descent can get stuck in a local minimum.

On the other hand we see that a higher number of hidden neurons has a direct effect in terms of loss function optimization: it seems to us that the MLP has a sweet spot where a value neither too low nor too high of hidden neurons is the most suitable to achieve the best configuration of weights. We see that the lowest loss value is attributed to the MLP with 6 hidden neurons and 0.5 learning rate. It appears that 3 and 11 hidden neurons (with a learning rate of 0.5) do not achieve a decision boundary as optimal as 6 hidden neurons; this allows us to see that each data set has a different complexity and that by increasing the number of hidden neurons indiscriminately the decision boundary will not necessarily improve.

In conclusion, we could see that the correct setting of these two parameters is very important to obtain a proper trained model and that the values are difficult to find because: a) the learning rate cannot be too low as the weights are updated too slowly, nor too high because the global minimum could be lost and, b) the number of hidden neurons cannot be too high because the decision boundary could be too complex nor too low because it could be too simple.