P1: Basic MLP Implementation, PyTorch

Using Optimizers and AutoGrad

Group Members

Luca Franceschi, 253885
Júlia Othats-Dalès, 254435
Alejandro González Álvarez, 252658

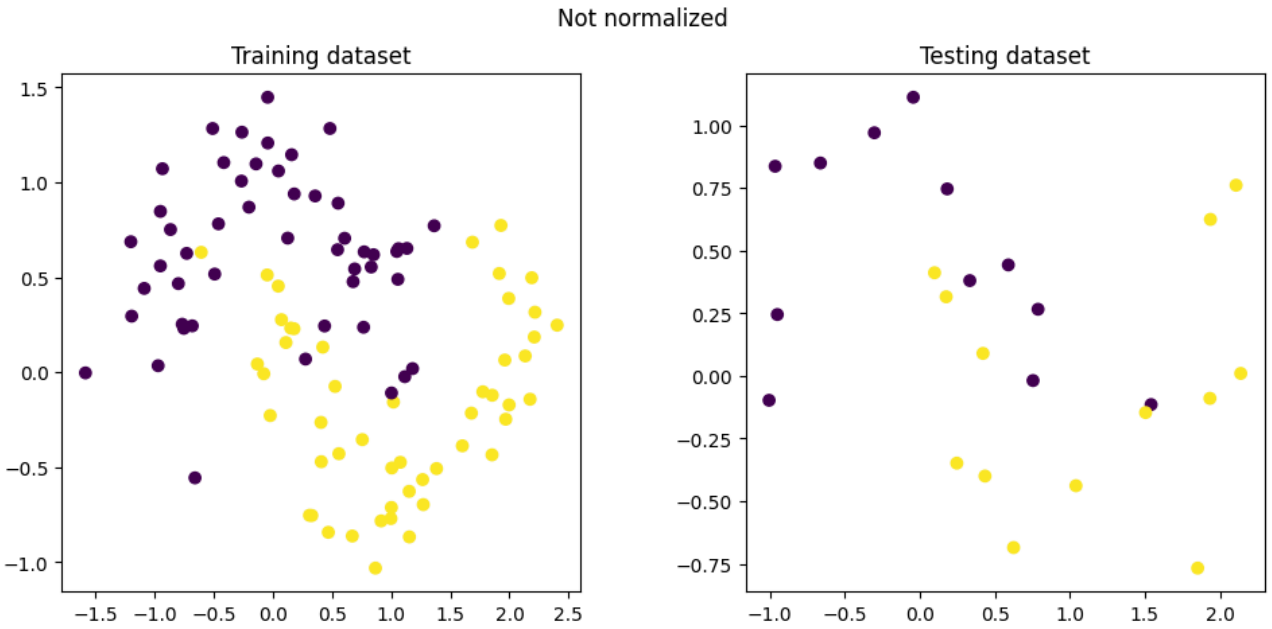
In this report we aim to provide an overview of the results obtained from our exploration of different concepts, ranging from multi-layer perceptrons, gradient descent optimization and automatic differentiation. For each exercise we will discuss the results and provide an explanation of our approach and solution, along with any experiments made.

Exercise 1

The goal of this exercise is to train a Multi-Layer Perceptron (MLP) using the Numpy library to then solve a classification problem.

We began by loading the provided datasets **train.csv** and **test.csv**, using the Pandas library.

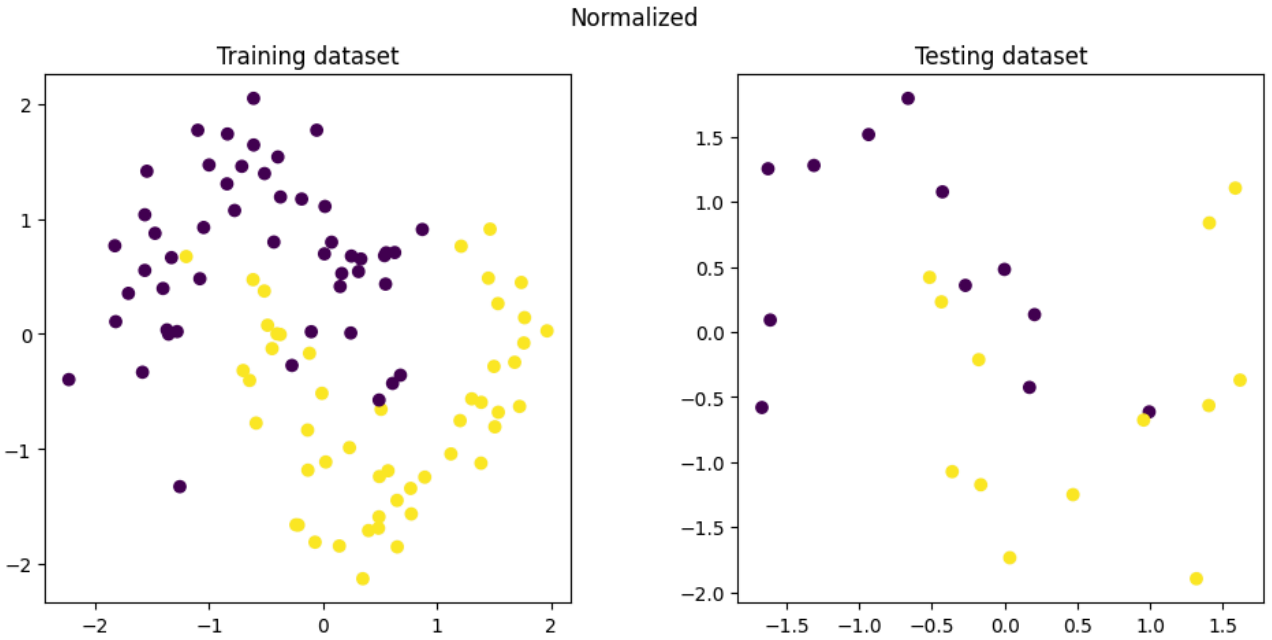
Then, in order to visualize and normalize the datasets we generated some scatter plots of the original training and testing datasets. Each plot represents the dataset with points distributed across the X and Y dimensions, color-coded based on their labels. The visualization helps to understand the distribution of the points and their separability. We obtain the following plots:



Next, we wanted to normalize the datasets' dimensions, as it helps in improving the convergence of optimization algorithms. We first extracted the X and Y features of both the training and testing datasets and stacked the arrays horizontally to create a combined dataset. After that we applied Z-score normalization to the new datasets so that the mean of all the values is 0 and the standard deviation is 1.

We used the following formula: $\frac{x-\mu}{\sigma}$.

We can then re-plot the normalized datasets to see how they changed. As we can see on the resulting plots below, the data is now centered.



In the third step we implement the MLP using Numpy, for which we used the MLP class provided in the **P1-Examples** file, modifying it to fit our needs, as it was being used for a regression task instead of the desired classification one. In this case, we will use the binary cross-entropy loss function, instead of the L2 loss, as it best fits our problem. Therefore, the loss function used is the following:

$$J = \frac{1}{m}\sum_{i=1}^{m}(-y_i * log(\hat{y}_i) + (1 - y_i) * log(1 - \hat{y}_i))$$

To define this function we used the np.mean function, given that it computes the same thing as the summation of the loss evaluated at each data point divided by the number of data points.

Asides from changing the loss function, we only modified the backward propagation function, where we simplified the partial derivative $\frac{\partial L}{\partial z^{[3]}}$, also known as *delta3* in our code. The following steps show the calculation process.

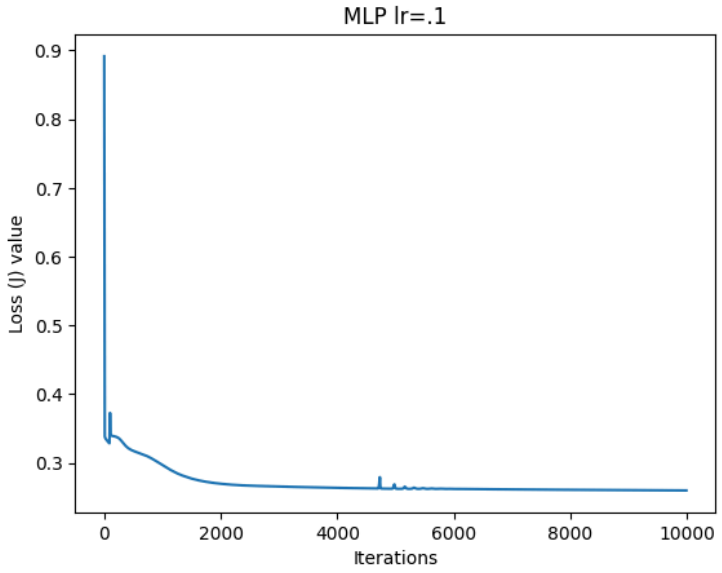
$$\frac{\partial L}{\partial z^{[3]}} = \frac{\partial L}{\partial a^{[3]}} * \frac{\partial a^{[3]}}{\partial z^{[3]}} = \left(\frac{-y}{\hat{y}} + \frac{(1-y)}{(1-\hat{y})}\right) * \frac{e^{-z}}{(1-e^{-z})^2} = \left(\frac{-y}{\hat{y}} + \frac{(1-y)}{(1-\hat{y})}\right) * \hat{y}(1-\hat{y})$$

$$= \frac{\hat{y}(1-\hat{y})(-y)}{\hat{y}} + \frac{\hat{y}(1-\hat{y})(1-y)}{(1-\hat{y})} = -y + y\hat{y} + \hat{y} - y\hat{y} = \hat{y} - y$$

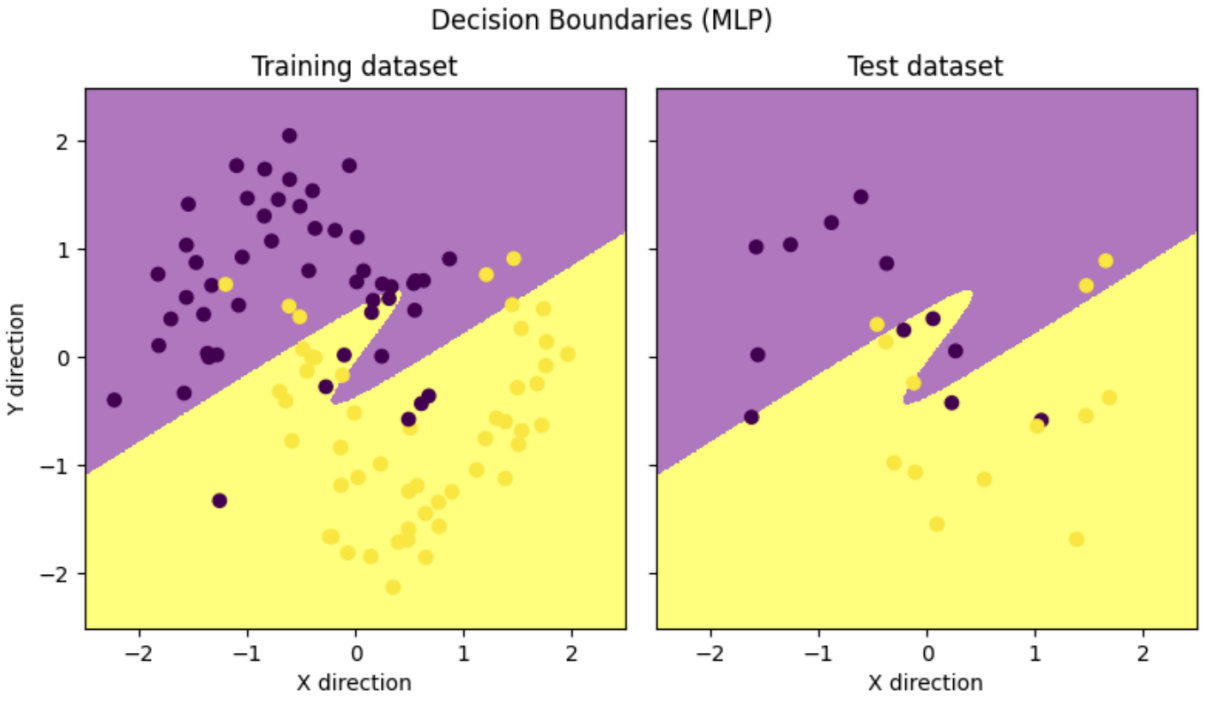
We made no significant changes when implementing the training function. Finally, in order to predict the classification of our testing data, along with its accuracy, we created a predict() function which essentially applies the forward() function of our neural network to the given data. To compute the accuracy, we applied the XOR function between the predicted labels and the actual labels, which gave us the proportion of incorrect predictions out of all predictions performed, then from that we obtained the proportion of correct ones by doing 1 - (incorrect/m) where m is the number of data points. We expressed the result as a percentage.

After those changes, all necessary functions were implemented to be able to train the MLP and visualize the decision boundary in 2D of the classification.

For the training, when analyzing the loss function graph over iterations, we can see a quick decrease in the loss, which indicates that it quickly converged and that the model was learning effectively.



To visualize the decision boundary we decided to generate a mesh grid of points that cover the same ranges as our original data. This grid covers the entire 2D space defined by the X and Y dimensions. Then we were able to —as we did with our original datasets— stack the features horizontally to then predict their classification, using the trained multi-layer perceptron (MLP). If we then plotted this mesh (plt.pcolormesh) we could clearly see the decision boundary. We then plotted both the normalized training and testing datasets on top, to better visualize the errors made by the model, and more precisely, which data points were misclassified.



With all the functions specified before we performed a predict() on the test data to obtain the predicted classification labels. We found that the accuracy obtained by our trained multi-layer perceptron was 72%. We consider this level of accuracy to be a reasonable level of performance. However, there is always room

for further optimization and improvement, for instance by modifying some hyper-parameters of the model, such as the learning rate, the epochs, number of neurons, etc.

Exercise 2

In this second exercise, we explore optimization techniques for training the Multi-Layer Perceptron. We will focus specifically on Stochastic Gradient Descent (SGD) with momentum, which is considered a more sophisticated optimization mechanism. Following the example, we will modify the provided OptimSGD class to create OptimMom. Then we will train our previously defined MLP using this new optimizer, comparing different values and their impact on training efficiency.

First of all, we got familiar with the momentum theory, which is used to help SGD navigate ravines and local minimas, helping accelerate the process and gaining faster convergence and reduced oscillation. It does this by adding a fraction gamma, the momentum, of the update vector of the past time step to the current update vector:

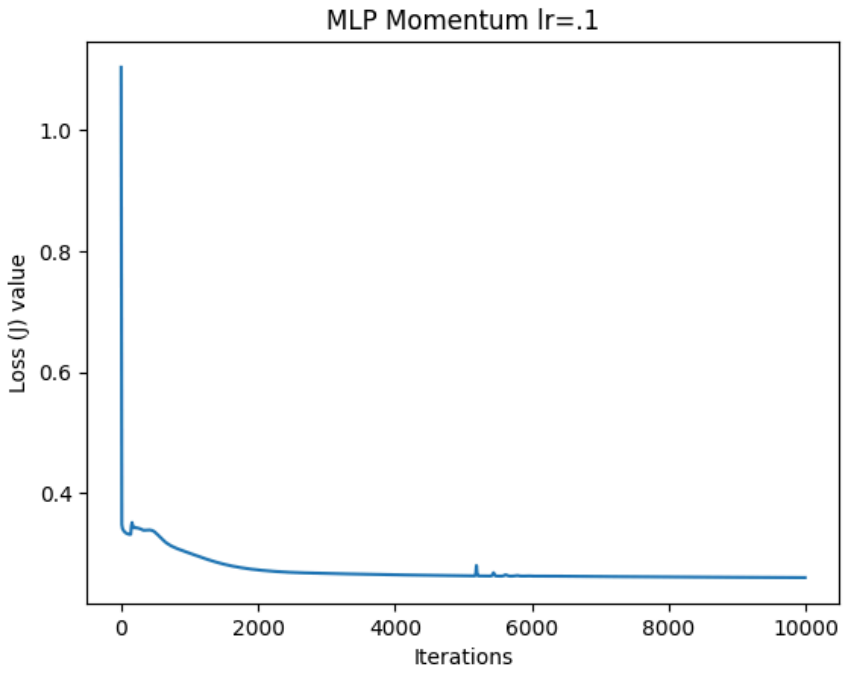
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

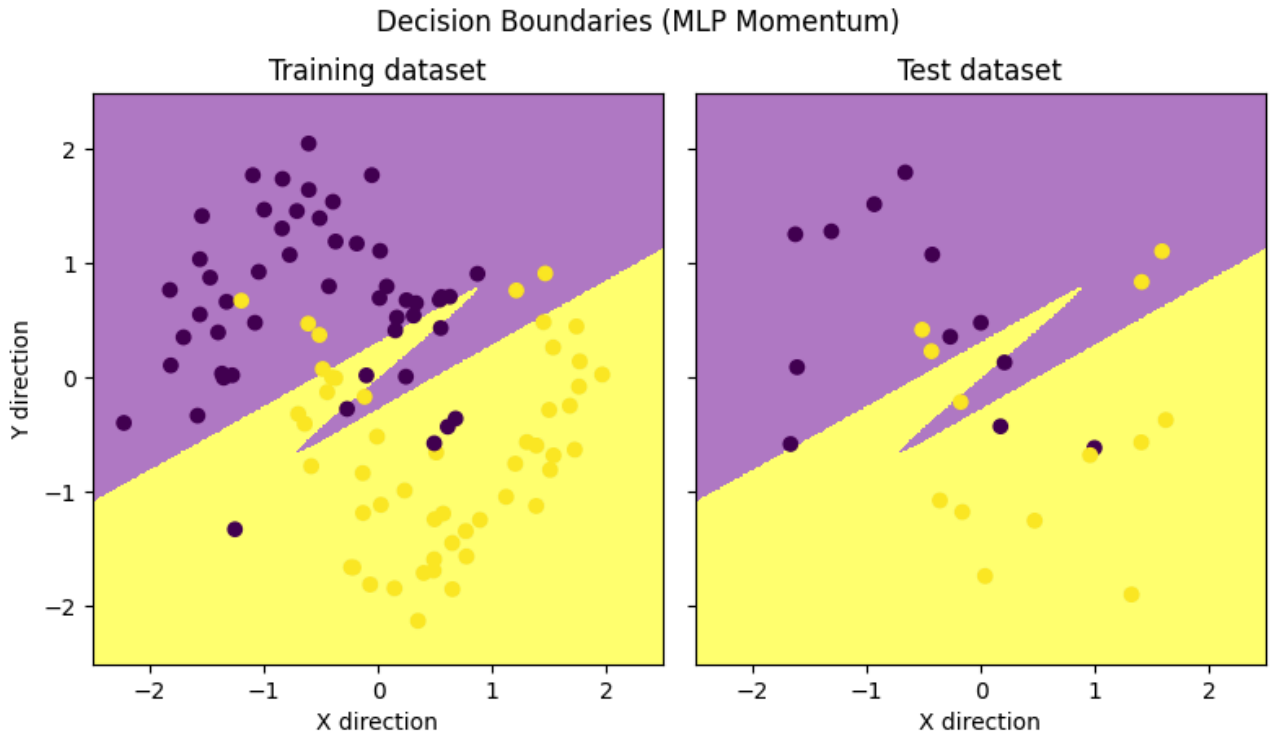
To incorporate momentum into our algorithm, we made adjustments to the existing class by introducing the new gamma parameter. In initialization, the class stores the parameter and learning rate parameters. In the step method, which is responsible for updating the model parameters based on the formulae above. For every weight and its corresponding gradient in the network, we computed the velocity term and then subtracted this velocity term from the current weight, effectively updating it.

After the class was defined, we modified the function that trained the MLP implemented in the first exercise. For that we could reuse the previous code, only needing to change the gradient descent update by the Momentum update.

Finally, we trained the MLP and plotted the resulting loss function over the iterations.



In this plot, just as we saw in the first exercise, we see that the loss converges quite quickly. The decision boundary can also be visualized again, to see if there are any major differences between one optimization technique and the other.



If we take a look at the plots above, we can see that the curve for the momentum based algorithm is less drastic as the previous one. This could potentially mean that we have less overfitting in this case. However, if we check again for the accuracy in the testing data, it stayed pretty much invariant.

To finish off this exercise, we wanted to see how different values of parameters would affect the loss function, so we defined some hyper-parameters called hyp_epochs hyp_lr and hyp_gamma to be able to test the different values.

The hyperparameter hyp_epochs controls the epochs, and as more and more epochs are used to train the model, the better it will adjust to the training data and will achieve a lower loss value. However, we have more risk of overfitting when increasing this value.

The hyperparameter hyp_lr controls the learning rate. The higher this value, the more impact each epoch will have on the parameters, as the update will be stronger. If we use a really high value, we take the risk of not converging to a minima due to the loss going from one side of the minima to the other, as the steps it takes are too big. However, if we choose a high learning rate for which this does not happen, we will converge faster than when using a low learning rate.

Lastly, the hyperparameter hyp_gamma, controls how much we are taking into account the previous “history”. Higher values of hyp_gamma will put more weight on the history, while lower values of hyp_gamma will make the Momentum mechanism resemble a traditional gradient descent approach.

Exercise 3

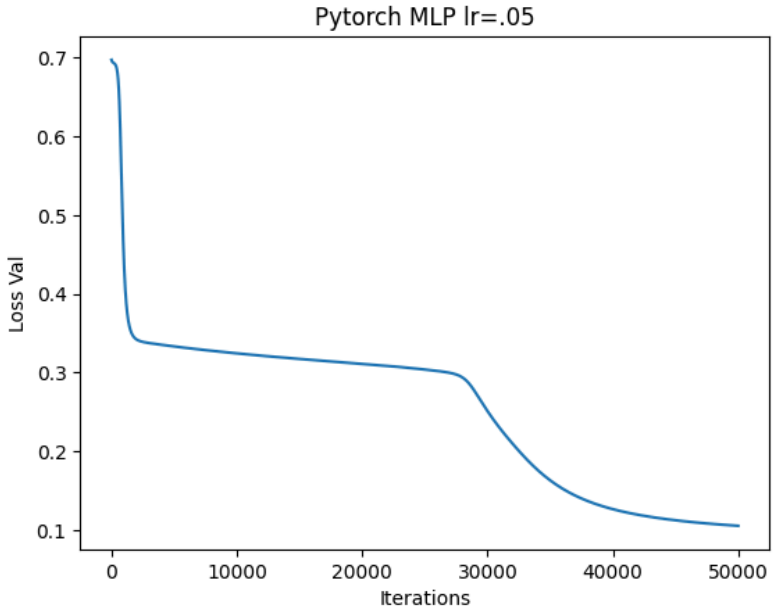
In this last exercise, the goal is to train a Multi-Layer Perceptron (MLP), as in exercise 1, but in this case using the PyTorch library. As we are using the same datasets as in previous exercises, we do not need to normalize again. We created a class called *MLP_PyTorch* that basically performs the same operations as the *MLP* class from exercise 1 but adapted to the PyTorch library.

The most remarkable changes are:

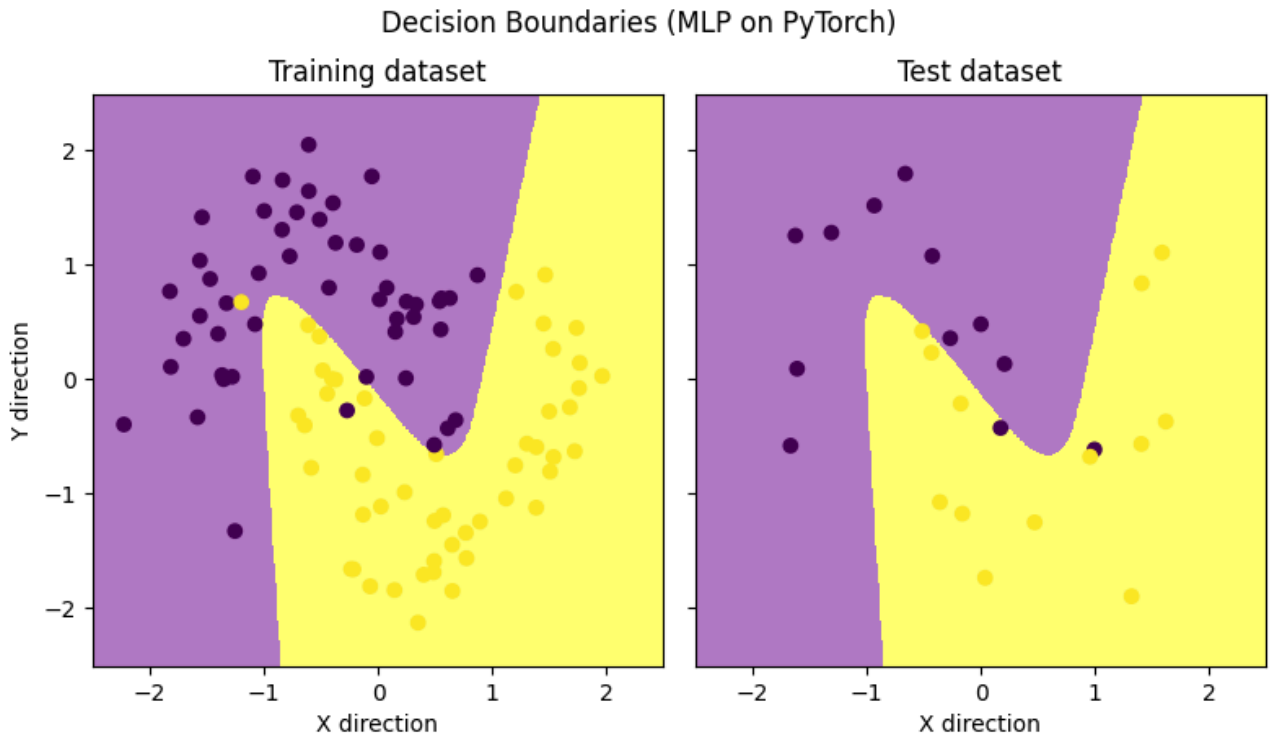
- Use of the already existing functions of PyTorch *Sigmoid()* and *BCELoss()*, that implement the sigmoid and the binary-cross entropy loss, respectively.
- No need to define a backwards function for the class, as we can use the already existing *backward()* function from the PyTorch library, that automatically computes all partial derivatives and gradients.
- Instead of updating the parameters ourselves manually, we called the *step()* function that updates them automatically.
- For the *predict()* function, we can call the PyTorch *eval()* function that performs the prediction.

A last remark would be that we are not using NumPy arrays anymore, instead we are using PyTorch's Tensors, so we needed to properly format the inputs when training and predicting the data.

When training the model with the normalized training data, this was the loss evolution over the iterations:



As we can see, it was progressing correctly at first, then it decreased much slower for a while, and then decreased significantly until reaching the lowest possible loss. What's interesting to notice here is that the model struggled a bit (around 30k iterations) to surpass the loss value of around 0.35, that is where the other two models (MLP and OptimMom) converged to. However, it managed to improve that loss significantly. We can clearly see that when looking at the 2D decision boundary below:



The accuracy obtained for this model was better than the 2 previous ones, consistently achieving around 85-90% in most runs.