Notebook - Hyperparameter Optimization

October 31, 2019

0.1 *Homework Assignment 3B - Hyperparameter Optimization*

0.1.1 Group 58:

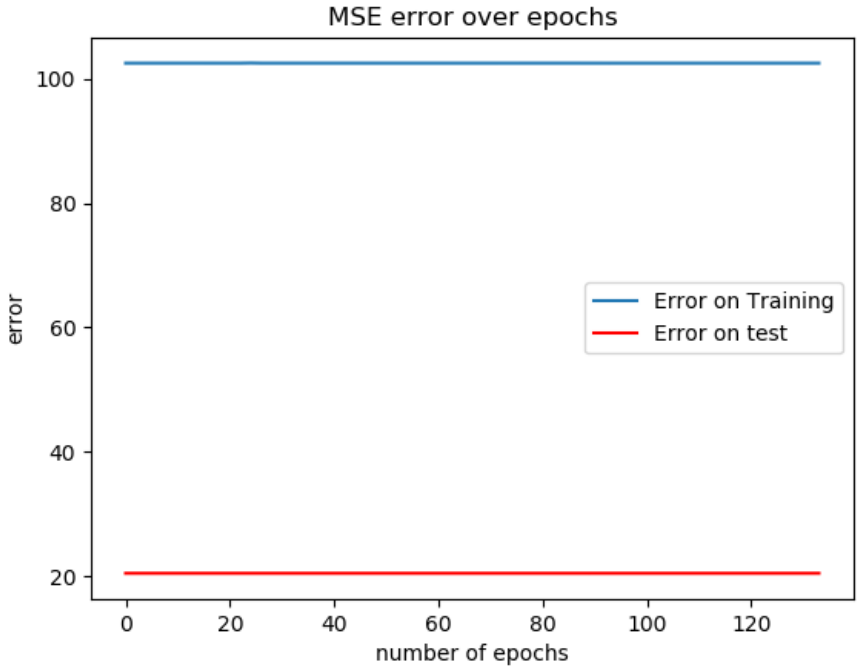
Andrea Favia, Paolo Berizzi, Tristan Tomilin, Aletta Tordai, Eliza Starr

0.1.2 *Parameter initializations*

As best performing model we chose the model in which the initial learning rate is 0.3, leading to a model able to learn fast and not overshoot the minimun. The number of epochs used is 200 but, since we use the early stopping technique, we reach the best result beforehand. The size of the batch was set to 16 and as combination of activation function we chose [relu, relu, sigmoid]. With the model depicted we were able to obtain an accuracy of 97.560% on the testing set as described in HW3A.

As required in the exercise we set the values of both the initial weights and biases to zero and train the model. It must be said that we already decided to set biases to zero. Hence, by initializing the weights to zero we can effectually see how weigths initialization affects the learning of the model.

```
n_epochs = 200
b_size = 16
l_rate = 0.3
nn_architecture = {
        'layers':[(10,2),(10,10),(1,10)],
        'activations':[relu,relu,sigmoid]
        }
mlp = MLP(nn_architecture, zeroWeights=True, zeroBiases=True)
mlp.train(df_train, df_test, epochs=n_epochs, batch_size=16, lr=0.3)
```

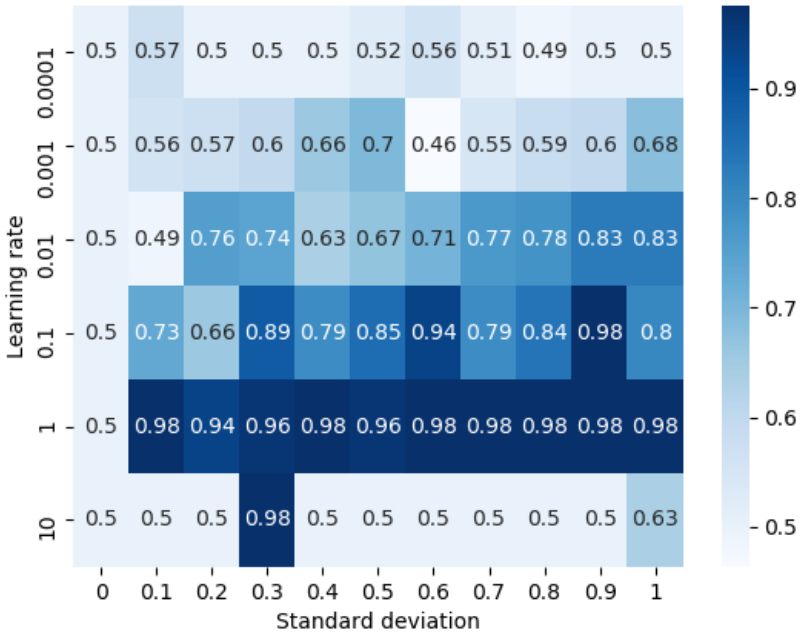


The result of the training on the model give an unpleasant, but expected result: the model does not learn during the training phase leading to the same label as predicted value. This is simply explained by the fact that the target of backpropagation is to minimize the error between the real values and the predicted ones. This happen when a gradient descent define the direction to follow in order to improve the prediction, but in this case we are creating a local mininmum that does not permit any other change. Mathematically, to compute the gradient we multiply deltas with weights and the result will always be zero. Furthermore, if all weigths are the same they will be updated by the same amount, resulting in no improvement.

Finally, we need to say something about the performance of this model. In our test set there is an equal number of observations per class. Therefore, since our model always predicts 0 or 1, the accuracy is 50%, a result that might satisfy a distracted observer. If we had a different dataset or we had a multiclass classification problem we might have ended up with an arbitrarily bad result.

0.1.3 *Learning rate vs parameter initialization*

The heatmap we obtained offers different insights for the learning process and the parameters initialization.



accuracy_heatmap

The first thing that we can notice is that the first column on the left, the one related with weights and biases sampled from a normal distribution with **standard deviation of 0** has all values equal to 0.5. Sampling from a normal distribution of mean and standard deviation both equal to zero gives always 0 as result, leading to the same results discussed in the previous section.

The second pattern that we notice is that the first row at the top has all values around the 0.5. This row is associated to a **learning rate of 0.0001**, too small to help the model to learn anything that can affect the classification.

Proceeding in the observation of the heatmap, we found out that also the last row at the bottom, the one related with a **learning rate of 10**, does not perform well apart from two cases. The fact of having a learning rate so big implies that every time we update the value of the weights we try to get closer to the optimal point. If the update is too big we overshoot the minimum point and get a bad result. This is because the gradient is only a good approximation of the loss function locally. Having a big learning rate makes the system taking huge steps, and most likely it will enter a region of the parameter space which is completely different from the previous point.

We can now shift our focus on the cases that give a higher accuracy. What stands out from the heatmap is the row corresponding to the **learning rate equal to 1**. For all of this cases, the accuracy is higher than 0.9 and for most of them is 0.98. This make us confident in saying that a learning rate of 1 is the best because no matter how the weights are distributed initially, it always resultsi in high accuracy. It, we have to point out that we use adaptive learning rate and early stopping technique, so we do not really know if in the end of the process the learning rate has been reduced. After all, we applied the same technique to all the models so it still has a significance for the model.

The rest of the heatmap, from the second row to the fourth, mainly shows the importance of having a learning rate capable of significantly improve the model by taking consecutive steps in the right direction.

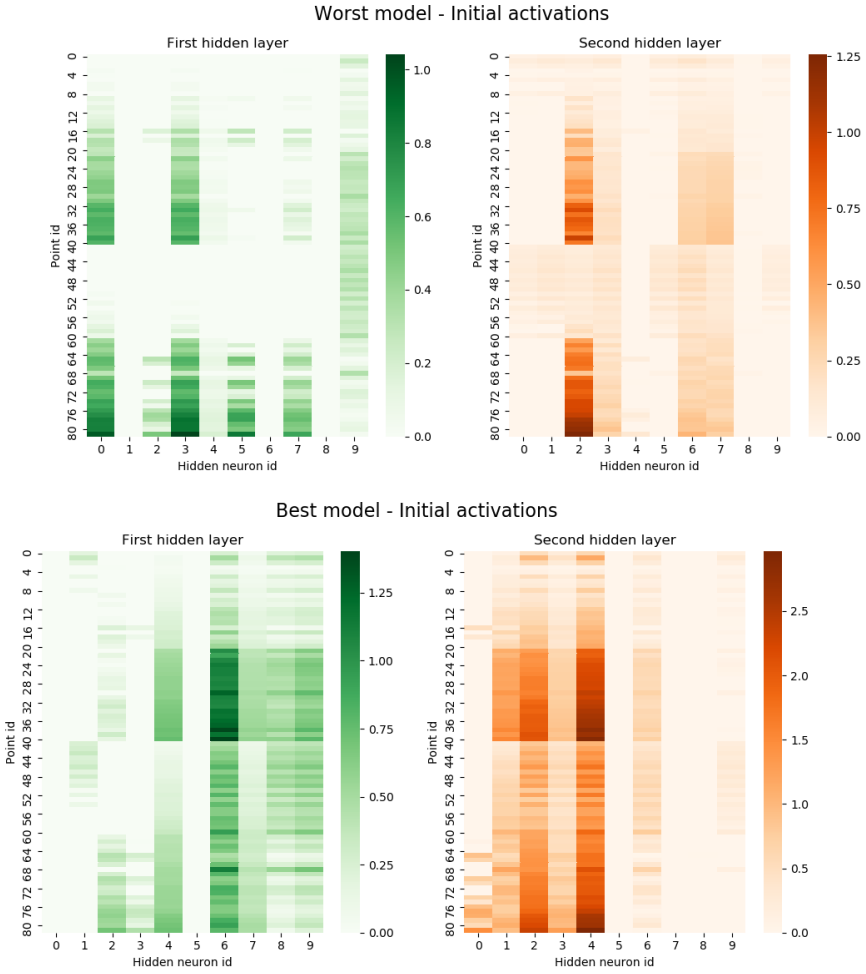
```
n_epochs = 200
b_size = 16
mean = 0
nn_architecture = {
        'layers':[(10,2),(10,10),(1,10)],
        'activations':[relu,relu,sigmoid]
        }
lrate_array = [0.0001, 0.001, 0.01, 0.1, 1, 10]
stdDev_array = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]

dataFrame = np.zeros((6, 11))
for i in range (len(lrate_array)):
    for j in range (len(stdDev_array)):
        mlp = MLP(nn_architecture, mean, stdDev_array[j])
        mlp.train(df_train,df_test, n_epochs, b_size, lrate_array[i])
        predictions,y_test = mlp.predict(df_test)
        dataFrame[i][j] = accuracy_score(predictions,y_test)
```

0.1.4 *Activations*

Computed the heatmap for the accuracy as required in the previous section it is possible to define the worst and best performing models as request by the assignment. Both have the same architecture, number of epochs and mean for the normal distribution. They only differ for the learning rate and standard deviation of the normal distribution used to initialize the weigths.

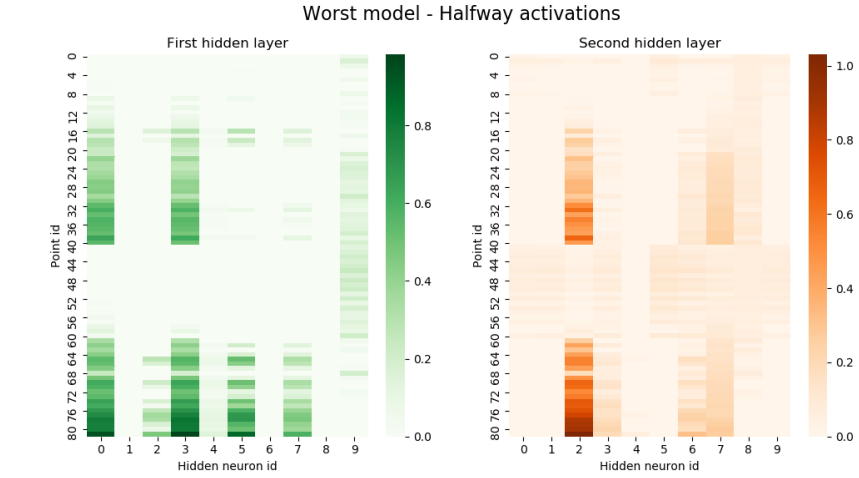
```
n_epochs = 200
mean = 0
bSize = 16
worst_l_rate = 0.001
best_l_rate = 0.9
nn_architecture = {
        'layers':[(10,2),(10,10),(1,10)],
        'activations':[relu,relu,sigmoid]
        }
worst_mlp = MLP(nn_architecture, mean, 0.6)
best_mlp = MLP(nn_architecture, mean, 0.9)
```

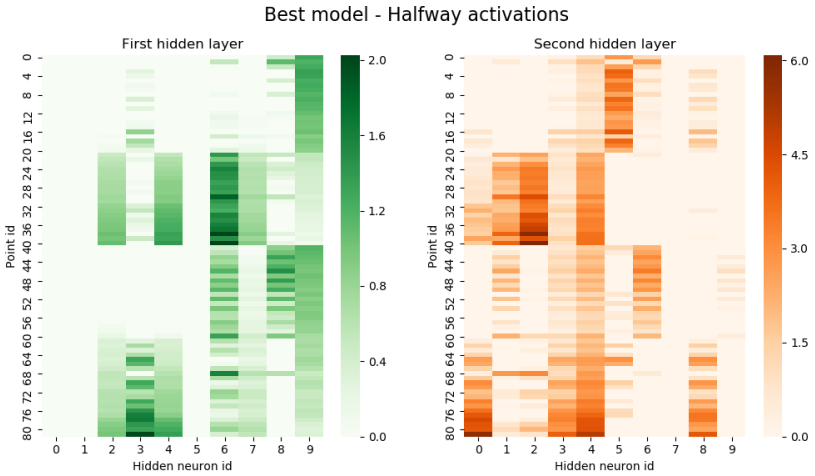
A normal distribution with higher standard deviation indicates that the values are spread out over a wider range. This also happens in our case, so the weights of the model with a higher standard deviation (the best model) have higher absolute values. Accordingly to this, the values generated by the activation functions are higher for the model with a higher variability and the heatmap clearly represent this. Moreover, it is worth notice that the values of the worst model vary between 0 and 1.25, whereas the value of the best model overreach 2.5.



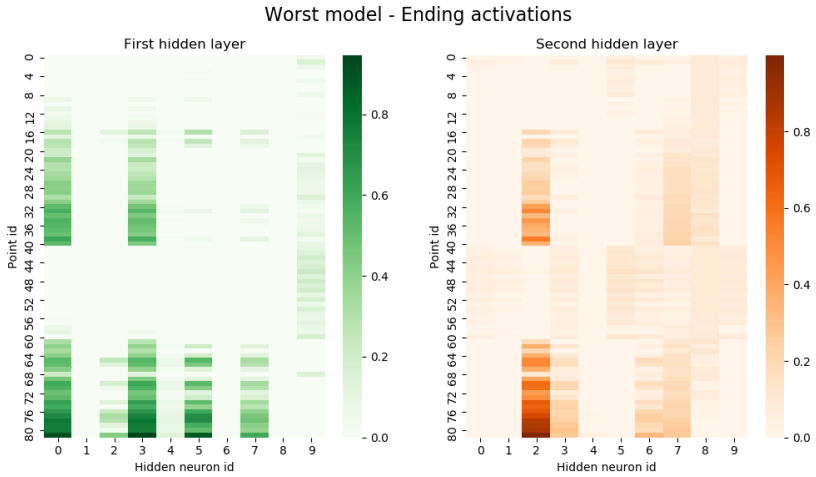
After half the training we notice a great change in the heatmap representing the best model, but only a slight change in the ones used for the worst. This difference is due to two main factors combined together: the tiny learning rate and the low variability of the initial weights that characterize the worst model.

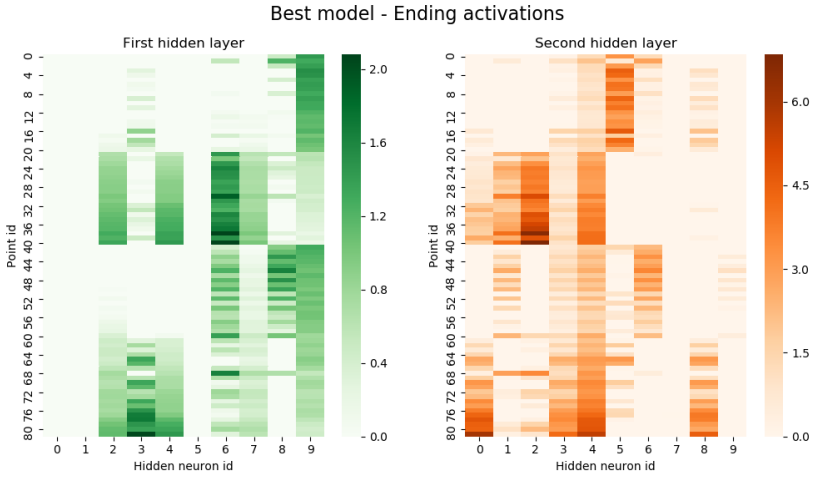
When a model starts with weights that are especially small or, on the other case, especially high the model has to make more effort to changes these weights and learns more slowly or, in the worst-case scenario, not learn at all. In our case, due to the small value of the learning rate, the model cannot affect the value of weights and biases enough and learns too slow. As a result, the best model is able to improve itself to better represent the training set, while the other ones change without any remarkable improvements. The heatmap showing the values of the activation functions clearly present these differences.





The third pair of heatmap represents the second half of the trainig. The worst model still changes a bit, but this time also the best model do not shows a great change as in the first half of the training. This is beacause the gradient descent process take bigger steps at the beginning, that is when the error of the predicted values compared to the real values is greater, and smaller steps as it get near the optimal solution.



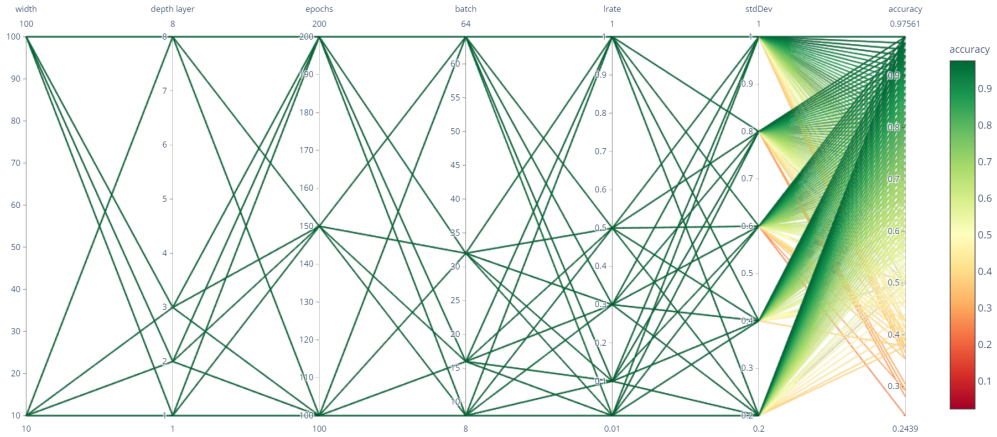


0.1.5 *Hyperparameters Optimization*

To study the effect of different combinations of parameters we use a Parallel Coordinates Plot. As hyperparameter for the model we consider the number of neurons per each layer *(width)*, the number of hidden layers used *(depth)*, the number of epochs to use to train the model, the size of the batch, the learning rate and the standard deviation for the weights initialization (where the mean for the normal distribution is always 0) It is possible to see the different values used for the hyperparameters tuning below

```
widths_array = [10, 100]
layers_array = [0, 1, 2, 7]
epochs_array = [100, 150, 200]
bsize_array = [8, 16, 32, 64]
lrate_array = [0.01, 0.1, 0.3, 0.5, 1]
stdDev_array = [0.2, 0.4, 0.6, 0.8, 1]
```

For the sake of simplicity we report the image of the plot, but it is possible to inspect it more thoroughly **here**



parallelCoordinatesPlot

The first thing we notice is that the most of the parameters combinations lead to satisfying accuracies and only few combinations result in bad classification.

When inspected, the plot shows that the three worst models are made up of **100 neurons for hidden layer** and **8 hidden layers**. This is clearly explained in the literature: neural networks that are too deep or too wide tend to overfit the training model and do not have a good result when tested on the validation set. Also, especially when using sigmoid functions for the hidden layers, we might experience the vanishing gradient problem.

Analysing the plot highlighting the **number of epochs**, it is possible to detect a difference between the model with a maximum of 100 epochs and the model that can have up to 150 or 200 epochs. In both cases there are models that perform good, but in the former case we see more result with accuracy below 0.5. Overall, the latter seems to perform better. This is due to the fact that we use adaptive learning technique and after 130 epochs the learning rate is reduced by one magnitude to get closer to the minimum. We also use early stopping technique so it is possible that cases labeled with 200 epochs actually stop learning before and therefore give similar result to the ones with a maximum of 150 epochs.

Considering the **size of the batch** used in backpropagation: the best result is clearly given by the model having batches of 16 elements. In this models just few cases perform below 0.5 of accuracy and only two models give results under 0.4. In all the other three cases, there are models that perform under 0.3 of accuracy and this clearly shows the importance of the batch size, that do not have to be too small (8), nor too big (32 or 64). Batch size also affects the speed of the traning and the memory space.

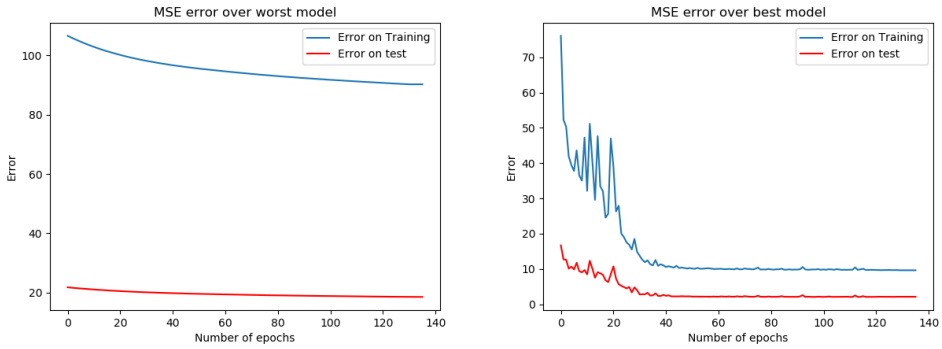
Regarding the value of the **learning rate**, best solutions are given by using learning rates equal to 0.3 or 1. For this models just few cases give values of the accuracy under 0.5. Most of them predicts correctly with 80% of accuracy. It is worth noting that for learning rates equal to 0.01, the lowest rate we used, the model learning is not significantly influenced by this parameter. The gradient descent is taking small steps towards the minimum, therefore only if it uses certain combinations of parameters it is possible to get a good accuracy. It is possible to check this behaviour by selecting the learning rate in the interactive plot to see that the accuracy values are in a wide range.

Lastly we study the effects of the **standard deviation** for the normal distribution used to generate the initial weights. For all the different values of this parameter it seems possible to obtain a good accuracy value. On the other hand, standard deviations equal to 0.6 or 0.8 lead to models that result in accuracy below 0.5 or even below 0.3.

Let us consider the worst and the best model identified:

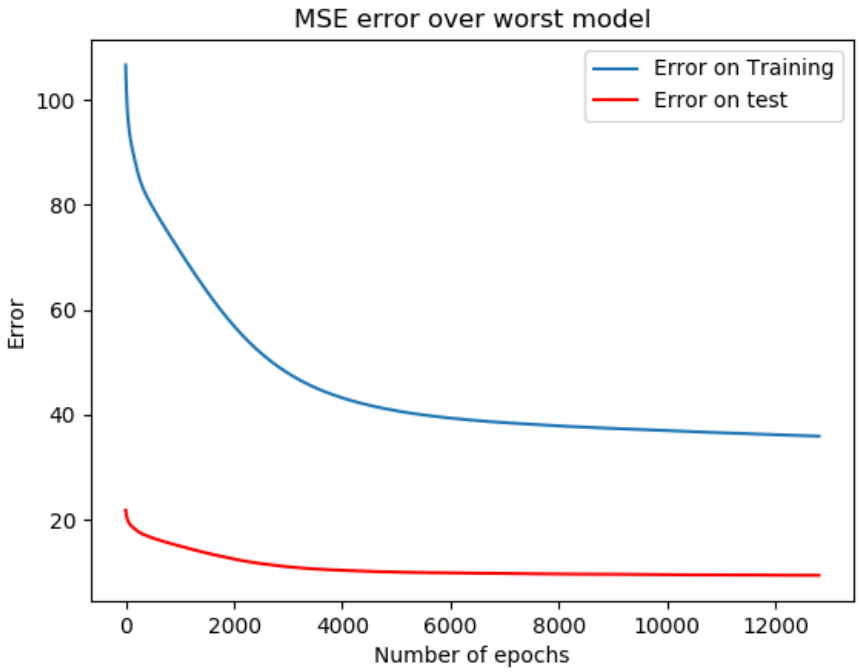
```
nn_architecture = {
        'layers':[(10,2),(10,10),(1,10)],
        'activations':[relu,relu,sigmoid]
        }

worstMlp = MLP(nn_architecture, mean = 0, std = 0.6)
worstMlp.train(df_train, df_test, n_epochs = 200, batch_size = 16, lr = 0.001)

bestMlp = MLP(nn_architecture, mean = 0, std = 0.9)
bestMlp.train(df_train, df_test, n_epochs = 200, batch_size = 16, lr = 1)
```

We can plot the cost fuction over the epochs for both the models during the training.



The difference between the two model is undeniable: the worst model [shown in the left] starts to learn in the first half of the graph, but then it does not improve significantly. The best model instead, presented in the right graph, reachs faster small cost error values resulting in a faster learning. Ideally, except for overfitting scenarios, having a good accuracy (i.e. small error on the cost function) on the traning set means that our model has properly learned, resulting in a good ability to predict the value on the validation set.

It is worth saying that even if the model on the left has a bad performance, it has a slightly slope that suggests some sort of learning. This can be proved if we increase the number of epochs to 12800: in this case the model slowly improves his performance and in the end we get an accurracy of 0.79.



worstModel_12800