Fabian Weiland

MLP for Regression

Introduction:

In this experiment I will implement a Multi-layer Perceptron for Regression. The data set for this experiment is stored inside a .csv file. We possess 1000 data points, with column x representing the independent variable and column y representing the dependent variable. Our objective is to derive a function that correlates x to y, utilizing these 1000 training examples.

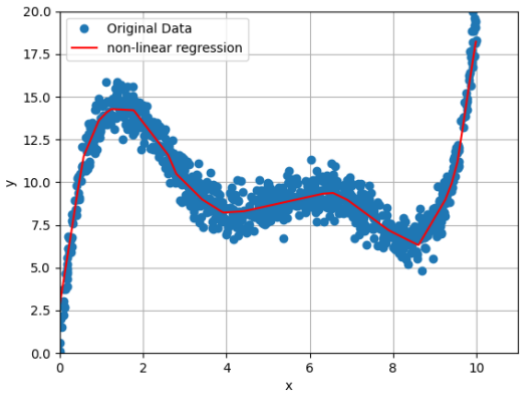
We will be using a Deep Learning Model with two hidden layers and one output layer (in total 3 layers). As we are trying to get a non-linear fit to the data, we will also use a non-linear activation function after each hidden layer.

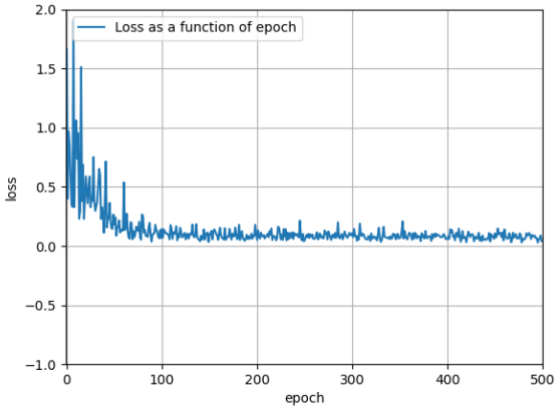
Description of Work:

First, I am normalizing my data, to avoid the vanishing gradient problem and to make the computation less expensive. I do so by doing z-score standardization. Then I am preparing our Dataset and creating the Data Loader. The MLP class contains two hidden and one output layer. Each hidden layer has a non-linear activation function at their output.

To train the model, I let it run a number of epochs. After every batch size the parameters of the neural network get updated. The optimizer we use here can be the Stochastic Gradient Descent, Adam, … .

After training our model, we are ready to evaluate it. I do so by creating a set of x values. I feat my model with this data set and get the predicted y values. As we can see in the image below, the model had learned and outputs a regression to the training data set.



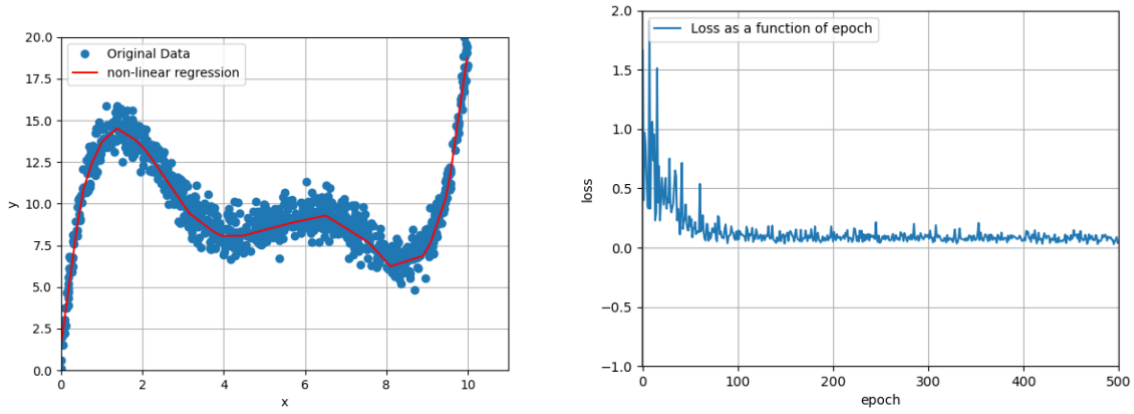


Hyper Parameter Search:

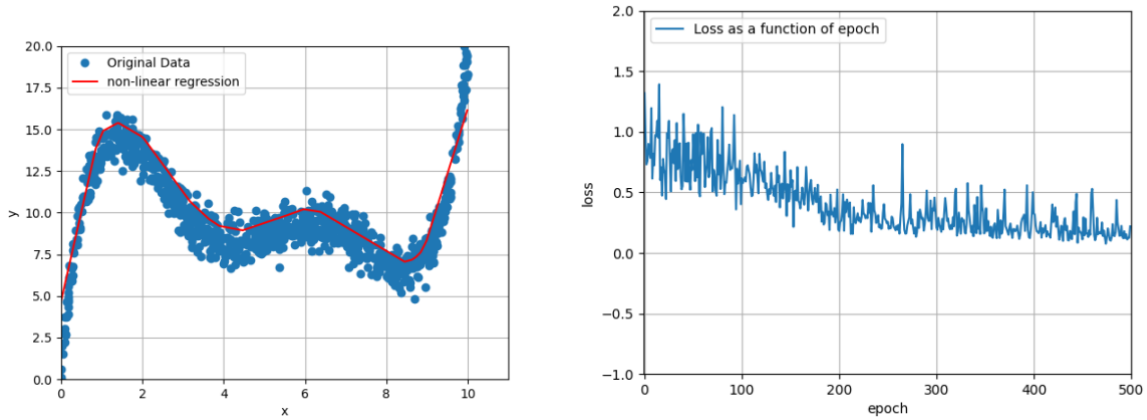
In this section I will explore the different results of the model by differing its hyper parameter. Hyper Parameter are the parameters, which the developer can change manually.

First, we will explore the difference of changing the batch_size.

Model with **batch_size = 20** & epochs = 500:



Model with **batch_size = 100** & epochs = 500:

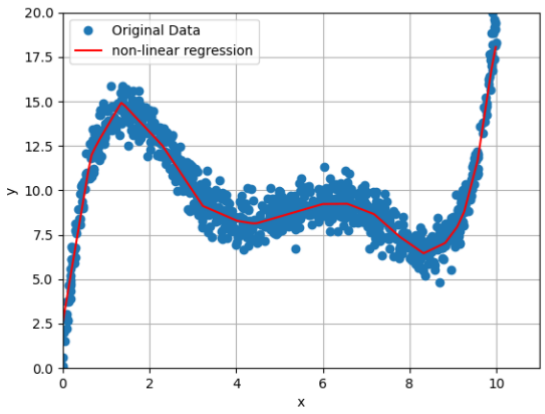


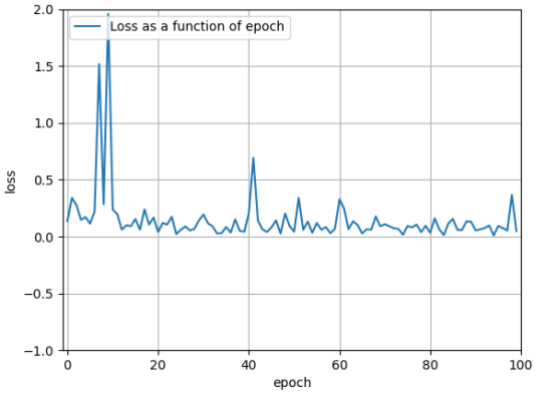
We can clearly see the difference on the regression curve. The smaller batch_size has a better regression curve. The reason for that is, that the model can update its parameters and learn already after 20 datapoints instead of after 100 datapoints. On the Loss as a function of epochs, we can see that for a smaller batch_size, the model learns faster

and therefore the loss gets faster towards zero, whereas with the bigger batch size, it needs more epochs to get closer to zero.

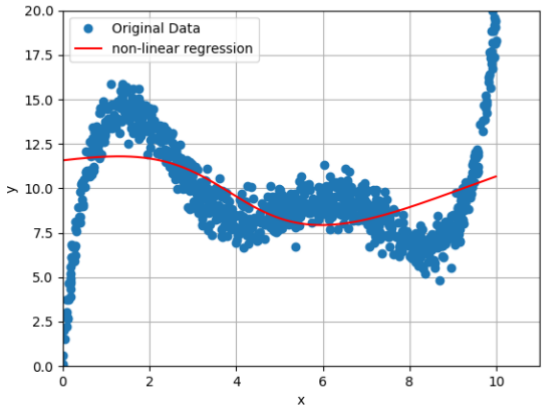
Now we will use different non-activation functions.

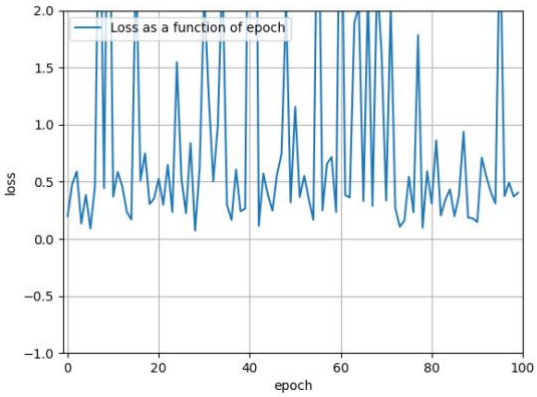
Model with **ReLu** activation function (batch_size = 5, epochs = 100):





Model with **Sigmoid** activation function (batch_size = 5, epochs = 100):





We can clearly see the difference on the regression curve. The Sigmoid activation function does not give us a good approximation for small batch_size and epochs. But the curve in general is smoother then the ReLu. The ReLu activation function is faster by approaching the correct regression curve, but it is not as smooth. The Loss function for the sigmoid oscillates, whereas the Loss of the ReLu gets smaller for more epochs.

Result:

We can see that with a minimum of one hidden layer + non-linear activation function our model can approach a non-linear function well. Changing the hyper-parameters can make a huge difference on the performance of the model as well. For smaller batch_size and bigger epochs in general the model gives us a better regression curve, but the running time takes longer. Interesting is, that the ReLu activation function approximates a good amount faster than the sigmoid activation function, but it is less smooth. The learning rate also plays a key role in the performance of the model. For to small values and too big values the model does not perform as well. The best values are somewhere in between.