Study of classification and regression problems using FFNN

Maria Emine Nylund

github.com/marianylund/fysstkprojects/tree/master/Project2

November 13, 2020

Abstract

For regression simple OLS for polynomial degree 5, with 1000 data points and 0.1 noise strength, MSE is 0.012. The best results were achieved with a neural network with one hidden layer with 5 neurons, He weight initialisation, leaky ReLU with the slope of 0.1, resulting in MSE of 0.0104. For classification logistic regression managed to reach 0.9405 accuracy, but neural network has achieved 0.9705 accuracy with shape of hidden layers [100, 20]. Xavier weight initialisation and ReLU hidden layer activations.

I. Introduction

Arguably, backpropogation is the most beautiful algorithm in our modern times. Since the late 20th century, this simple solution of applying chainrule to solve optimisation problems was found to be useful in countless areas of research. With the exponential growth of hardware's computational power and rapid development of software, we might be getting closer to those sci-fi dreams of technological singularity. But in the meantime, neural networks has proven to be a powerful prediction and classification tool.

In this article, my aim is to study regression and classification problems using newly developed feed-forward neural network (FFNN) on Franke's function and MNIST data set. I will evaluate its performance compared sklearn's methods. I will reuse data normalisation and mean-squared error algorithms from my previously implemented work. [M.E. Nylund, 2020]. First I will start by presenting the theory behind logistic regression, FFNN and stochastic gradient descent (SGD). Then the rest of the sections are divided into regression and classification parts for a clearer comparison of the problems. Further sections will be filled with discussion of the methods implemented and analysis of the results til I will conclude this article in the last section.

II. Theory

Linear Regression

Main goal of the linear regression is to learn the coefficients of a functional fit in order to be able to predict the response of a continuous variable on some unseen data [Morten Hjorth-Jensen, 2020].

$$Y = \beta_0 + \beta_1 x + \varepsilon \tag{1}$$

Logistic Regression

Logistic regression in contrast to linear regression has as its main goal to learn coefficients that let us predict which classes unseen data belongs to. It returns the probability of a data point belonging to a category. Thus it has to be between 0 and 1, but as we can see in eq.1, $\beta_0 + \beta_1 x$ is not necessarily in the given range. Logistic regression means assuming that *p(x)* is related to x by the logit funtion [Devore and Berk, 2018]. This function is also often called Sigmoid function:

$$p(x) = \frac{e^{\beta_0+\beta_1 x}}{1+e^{\beta_0+\beta_1 x}} \tag{2}$$

In order to support the new probabilistic function, a new cost function is used for logistic regression, which is called cross entropy. The second line is the shortened version:

$$\begin{aligned} C(\hat{\beta}) &= -\sum_{i=1}^{n}(y_i(\beta_0+\beta_1 x_i) - log(1+e^{\beta_0+\beta_1 x_i})) \\ C(\beta) &= -\sum_{i=1}^{n}(c_i(x_i,\beta)) \end{aligned} \tag{3}$$

The important moment with this function is that it is convex, which means any local minimizer is a global minimizer as well [Morten Hjorth-Jensen, 2019].

To be able to use logistic regression for for multiclass classification, one can use the Softmax function, which is a slight modification of the Cross-entropy eq. 3.

$$p(C=k|\mathbf{x}) = \frac{e^{\beta_k 0+\beta_k 1 x_1}}{1+\sum_{l=1}^{K-1} e^{\beta_l 0+\beta_l 1 x_1}}. \tag{4}$$

Neural Networks (NN)

Standard artificial neural networks, which usually are called just neural networks consist out of artificial neurons that are highly inspired by how our own brains work. A network consists from its building blocks, perceptrons. They take in some inputs $\{x_0, x_1, ..., x_i\}$, paired with weights $\{w_0, w_1, ..., w_i\}$, which determine how important that input is and each perceptron returns a single binary output, 0 or 1. Feed Forward Neural Networks (FFNN) is the simplest type of artificial neural network as the information only moves forward [Schmidhuber, 2014].

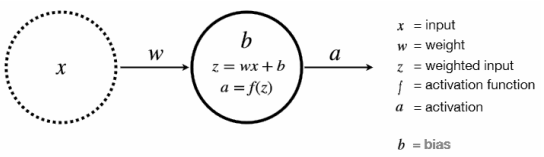

Figure 1: *Visual representation of a network with single input, single layer and single neuron*

A non-linearity allows us to learn arbitrarily complex transformations between the input and the output. There are several activation functions that can be used in the process. One of the most common ones is Sigmoid activation function, the artificial neuron using this function is called sigmoid neuron [Nielsen, 2019].

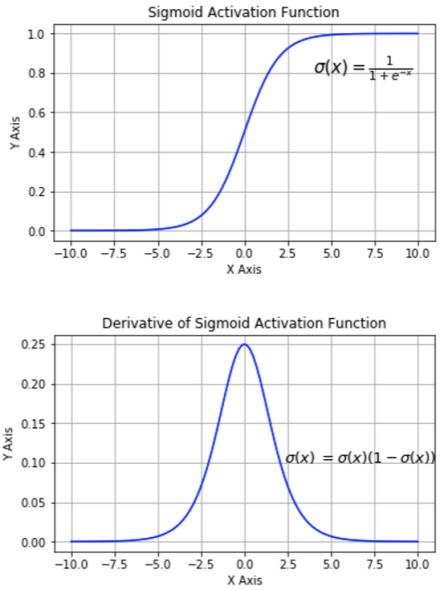

Figure 2: *Visual Representation of Sigmoid function and its derivative*

It has its drawbacks in creating vanishing gradients, because its outputs are close to 0/1 so its derivative being close to 0 which leads to weights not updating when backpropagating. Another activation function that is resistant to the vanishing gradient problem in the positive region is ReLU (Rectified Linear Unit). But if input is negative in the forward pass, the neuron remains inactive during the backward pass, because the derivative is 0.

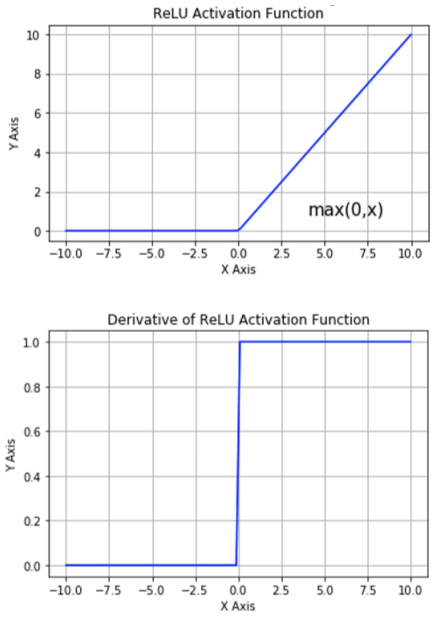

Figure 3: *Visual Representation of ReLU function and its derivative*

To find the weights and biases that will minimise the cost function the most, one can use backpropagation algorithm. Backpropagation computes the gradient of the cost function with respect to the weights of the network. To do so first we need to find derivatives in respect to the weights and to the biases:

$$\nabla_\beta C(\beta) = \sum_{i=1}^{n} \nabla_\beta c_i(x_i, \beta) \tag{5}$$

$$\frac{\partial C}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l)}} \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \circ \frac{\partial C}{\partial \mathbf{a}^{(l)}} \tag{6}$$

Stochastic Gradient Descent (SGD)

The main goal of the gradient descent is to minimize a cost function by tweaking parameters iteratively. SGD does it by randomly picking a minibatch containing the data points and learning on it. I leads to cost function fluctuating but decreasing on average. Here an important parameter is the learning rate as if it is too big it might never converge to the minimum or if it is too small it might take too long. To approach this learning rate problem, one can use the *learning schedule* to slowly reduce it for each iteration. [Géron, 2019]

$$\nabla_\beta C(\beta) = \sum_{i \in B_k}^{n} \nabla_\beta c_i(x_i, \beta) \tag{7}$$

Gradient descent step:

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(x_i, \beta) \tag{8}$$

Performance Measures

There are several methods that can be used to evaluate performance of a classifier. Here are will present the theory behind the ones I will be using in this research:

Mean Squared Error (MSE)

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2 \tag{9}$$

The MSE is the *mean* $(\frac{1}{n}\sum_{i=1}^{n})$ of the *squares of the errors* $(Y_i - \hat{Y}_i)^2$

Confusion Matrix

The confusion matrix can quickly give a description of the performance of a classifier in a matrix format. Each row represents an *actual* class and each column represents a *predicted* class. This results in the first row first being true negative, which means it has been correctly labeled false and second false positive meaning the correct answer was false, but it was labeled true. The second row contains similar pattern just for the positive answers. True positive stands for correctly labeled true and false negative meaning it should have been labeled true but was false.

True Negative (TN)	False Positive (FP)
True Positive (TP)	False Negative (FN)

Table 1: *Confusion Matrix*

Using those classes one can calculate *accuracy* of the model, how often is the classifier correct:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{10}$$

III. Methods

First I will explain the set up of the small nnreg library and later I will present how this library was used to generate results. As there were 6 different tasks from a)-e), you can find a separate python file for each one of them. But I will explain a general method to test regression and classification

nnreg

After some experimenting writing separate code for linear, logistic regressions and neural networks, it became clear that the same code base can be used for all the different cases if the configuration is set up correctly. Therefore my code base is the same for all the cases, but I will explain in detail the different set up needed to use those methods. But first let me explain the four main parts of the code: model, trainer, dataloader and config. This way to divide the code is inspired by the github repository

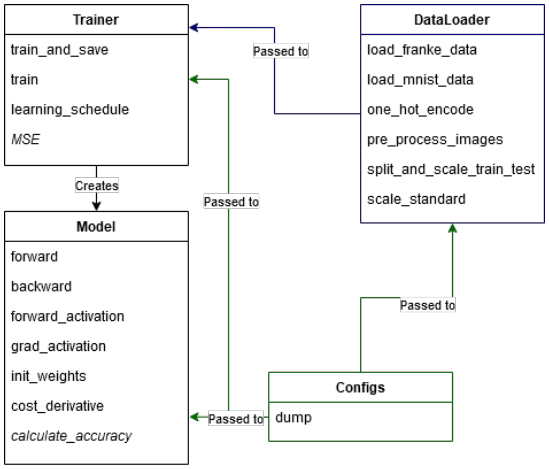

Figure 4: *Diagram of how the package works together. Only the most important functions in the classes are included in the model.*

Configuration

Neural network and regression requires a lot of different parameters set up and changed between trainings, so I needed an easy way to control and save it. The Config class is taken from the github repository: [Karan Desai, 2019]. It was changed to work with the pathlib library which makes it much easier to load and save files. The parameters themselves has been added to fit the purpose of the training. The biggest strength of the class is that it can load an existing config file and overwrite or add to the default values. It can also take a list over values to override. The values themselves are divided into 3 subconfiguration "folders": optim for optimisations values, model and data. It also runs validation tests on the configuration so that there are no contradictory settings. The config file is used by the other three parts to set up their parameters which I am about to explain.

Dataloader

This class is responsible for loading the correct dataset based on the configurations given, normalising them and splitting in to train, validation and test sets. For MNIST data it uses mnist.py file to load and save data onto your computer. Loads and configures the data, the MNIST dataset function are taken from the github repository [H. Hukkelås, 2020]. Function for one hot encoding was written by me, it creates a new empty list with the shape of the number of classes. Then for each entry it sets the values to one on the index which the correct classification is. For the Franke's data, the functions are reused from the previous project.

Model

This part can be found in model.py and is responsible for weight initialisation, neurons and gradient descent. First it saves all the configurations given in its own variables, then it initialises the weights and other parameters. The most notable functions are forward and backward. In forward function it propagates the data through all the layers using the given activation functions. It actively uses the "forward_activation" function where it chooses the correct activation based on which layer it is. Implemented activations include: identity, sigmoid, tanh, ReLU, softmax and leaky ReLU. I

followed [N. Kumar, 2019] article when implementing those functions and weight initialisation methods: random, he, xavier and zeros.

For the backward function, as the function name suggests it propagates through the levels backwards, for each layer it takes the derivative of the activation function with the output error and computes the delta cost. Which then in turn is propagated further through derivated activations. In the end of each layer it updates the gradient of the given layer by setting it to the derivative of cost function. If regularization is applied, then it is added as well.

Trainer

Trainer uses mini batch stochastic gradient descent, it is also responsible for creating the model and saving the progress of training. For each epoch it updates the learning rate if learning schedule should be used and shuffles the traning data if it is enabled. Then for the number batches per epoch, it calculates the start and end index depending on the step and batch size. After that it computes the gradient by calling forward and backward function in the model with the given batch. To update the weights it calculates the learning step by multiplying the updated gradients from the model with the learning rate and subtracts it from the weights. After that there a lot of steps and configurations in order to find and save the best model, save the checkpoint and log the process. It also checks for the exploding weights (it is infinity or nan) and early stopping, in that case it stops the training before the last epoch.

Other files

The RegLib library has been included from the previous project. Another file "analysis_fun.py" can be found in nnreg library, it mostly includes helper functions used for running parametric searches, analysing results and creating plots.

For each sub task given in the project, there exists a file that includes code needed to run one training with given configurations and testing against sklearn functions. To run the file I suggest writing "python -W ignore ’.\e)LogisticRegression.py’".

IV. Analysis

Regression

SGD

For the regression I used 1000 datapoints and 0.1 noise. Models are compared using MSE. First of all I was curious of the effects of batch size and learning rate on polynomials of degree 2, 5, 10 and 15.

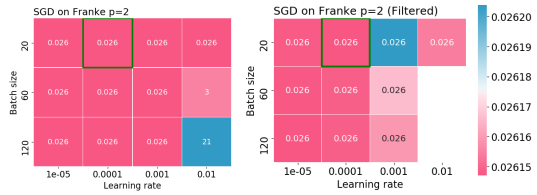

Figure 5: *Heatmap of the runs for polynomial 2, on the right side the heatmap is filtered by getting rid of all values over 1.*

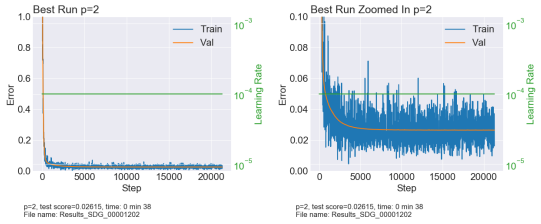

Figure 6: *For the graphs of the best run, the one on the right is zoomed in to 0.0 - 0.1 on y-axis*

For the rest of the runs I will only show the filtered and zoomed in versions for the polynomials:

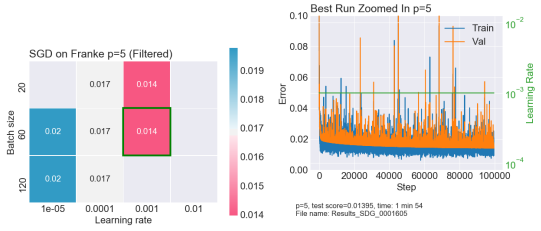

Figure 7

Figure 8

Figure 9

This satisfied my curiosity and gave an indication that the more complex the polynomial degree the bigger batch size could be used for more beneficial runs for the finite number of epochs. But the learning rate of 1e-3 performed best for all polynomial degrees as it is more closely tied to the nature of the dataset itself. In some cases I had to filter out the results over 1.0 MSE, those cases has introduced me to the exploding gradient problem which made the network unstable since the weigths first became equal to infinity and then to nan. This happened mostly when the batch and/or learning rate was too big and happened more often to higher degree polynomials, but it also showed that this model **is prone to over-saturation**.

For the next runs, I decided to concentrate on polynomial degree 5. Since batch size of 60 and learning rate of 1e-3 performed the best, I tested learning decay and ridge regularisation with various alpha values. A simple OLS for the same parameters result in MSE of 0.012. This will be used as a baseline to compare to. The best run with mini-batch SGD resulted in MSE 0.014, with run-time of 1 min 54 sec.

Neural Network

Continuing with the same dataset I started with a simple neural network with just one hidden layer with 5 neurons. Sklearn running with the same architecture resulted in MSE: 0.0127. First I compared the different weight initialisations and activation layers for that hidden layer.

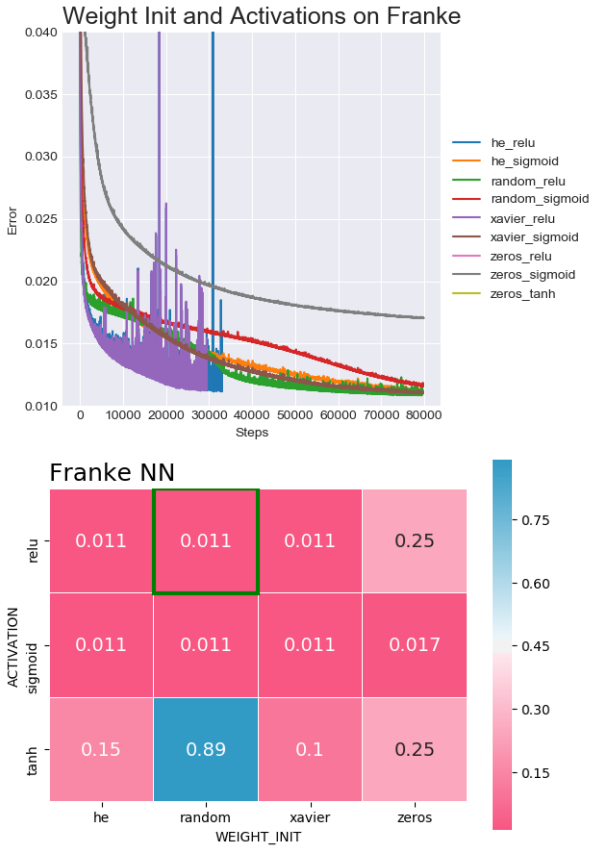

Figure 10: *Effects of weight initialisation and activation functions in the first hidden layer with 5 neurons. Some runs with zeros weight init is not visible on the graph, because it did a lot worse than the rest. Graph is zoomed in.*

The weight initialisation with zeros performed much worse than others no matter the activation function as it was expected. When all weight are set to zero, then all the layers are essentially computing the same thing in the beginning. It also makes the learning much slower since it is starting with the vanishing gradient problem. Meaning that the changes are multiplied by zero or very small number make smaller steps in the correct direction to reach the minimum.

Random weight initialisation gave much better accuracy since every neuron is no longer performing the same task, it paired quite nicely with ReLU activation function.

He and Xavier weight initialisation methods are quite similar to each other since they take into consideration the shape of the previous layers. This resulted in much quicker convergence, although there is a lot of variance in the error after the initial fall. So it means it would benefit from some regularisation. Those methods pair nicely with ReLU, which makes sure not to over-saturate the gradient.

Then I tested the leaky ReLU separately with different slope values as well.

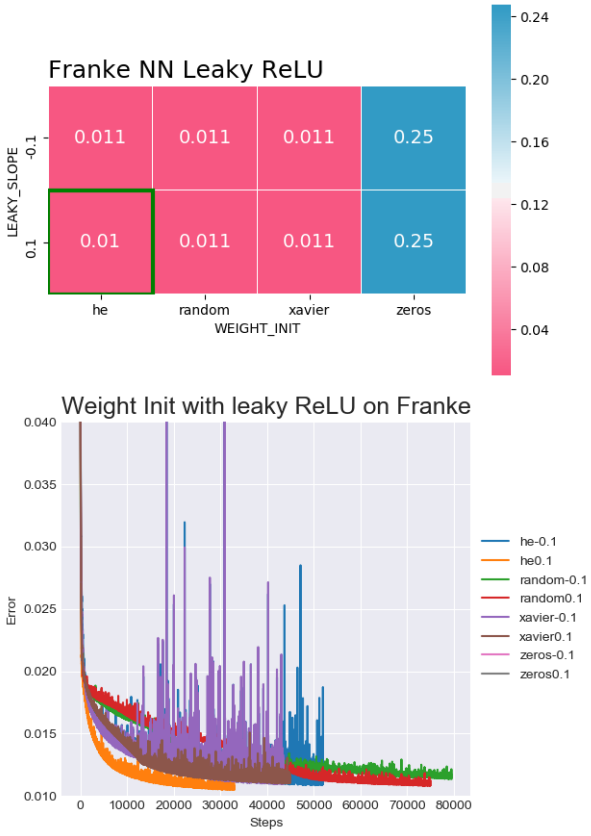

Figure 11

The best run was with He weight initialisation, leaky ReLU with the slope of 0.1, with MSE of 0.0104. Simple ReLU came quite close as well with MSE of 0.0107 and random weight initialisation.

It makes sense that leaky ReLU performed better, because when ReLU takes in zero its gradient becomes zero as well and does not converge to a good minimum. But leaky ReLU it will still have this alpha value avoiding those zero dead neurons.

Classification

Neural Network

To compare my data I am running MLPClassifier with the same hidden layers setup and enabling early stopping. For hidden layers: [100, 20], sklearn's accuracy is 0.9480. For all runs accuracy was used as evaluation function, softmax as activation function for the last layer and entropy loss as cost function.

Early stopping is turned on for all the runs, if the model does not get better for the last 2000 steps, it stops, based on the validation score.

The first test was to compare the effects of batch size is on learning rate and simple learning scheduler. This was run with maximum of 20 epochs, sigmoid as activation function for hidden layers.

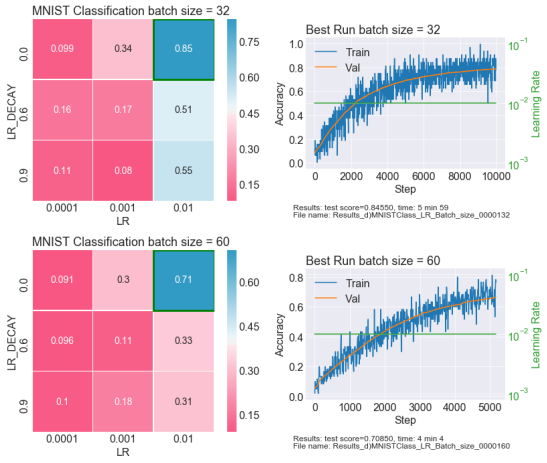

Figure 12: *The results of effects of learning rate and learning decay with various batch sizes*

As the best one from those results is the one with lr 1e-2, batch size 32 and no learning decay, I have continues the tests based on that.

Then I checked different weight initialisations with different activation functions for the hidden layers. This time I have chosen not to test weight initialisation with zeros as it gave a poor result for regression.

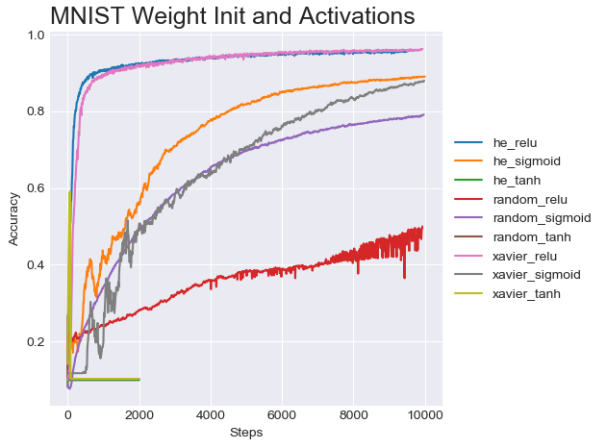

Figure 13: *The results of effects of learning rate and learning decay with various batch sizes*

The best result of 0.9705 was given by xavier weight initialisation and ReLU hidden layer activations.

Logistic Regression

For testing I used sklearn's LogisticRegression class with parameters set: solver to "sag", not fitting of intercept and penalty to "l2". This resulted in sklearn's test accuracy of 0.9350.

All tests are run with batch size of 32, accuracy as evaluation function and softmax as activation. First I wanted to see the difference learning rate, weight initialisation and regularisation has on the accuracy. Here still I do not test zero weight initialisation as it proved to be much worse than the alternatives for logistic regression.

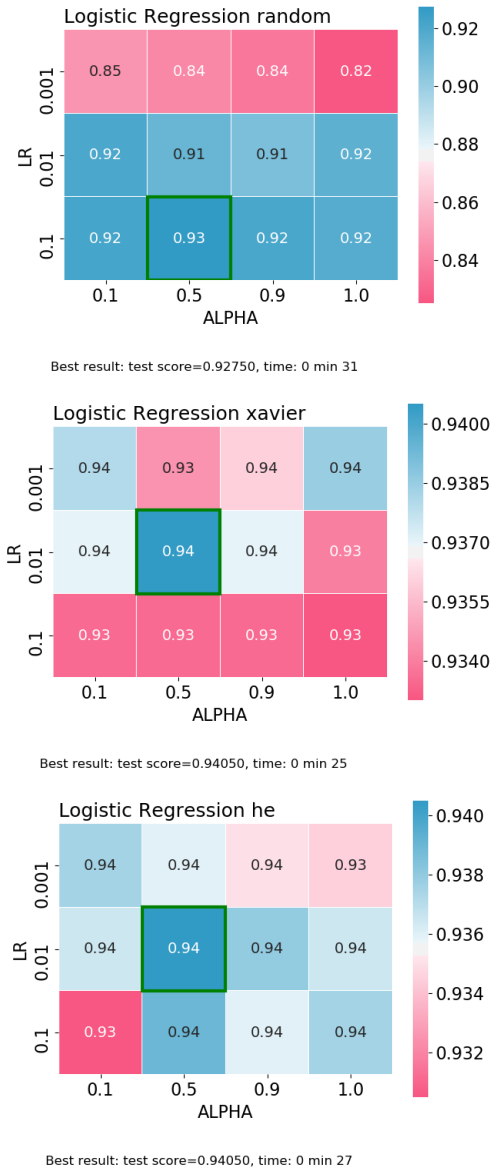

Figure 14: *Results are divided by the weight initialisation methods: random, he and xavier. With learning rate and l2 regularisation parameter alpha on the axes of the heatmap.*

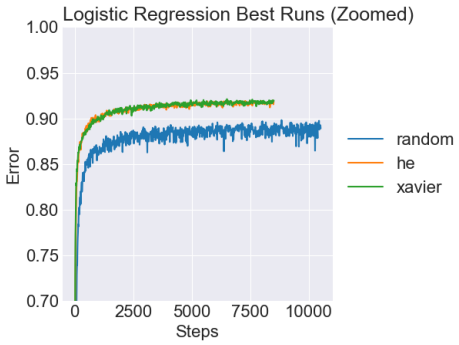

Figure 15

Best run of them all with he weight init, learning rate of 0.01 and alpha of 0.5 takes only 26 seconds with 0.9405 accuracy.

It appears as if neural networks is a better solution for the classification of 10 classes if accuracy is your only concern. Logistic regression is able to achieve impressive results though through little time, but it is still 3% shy of the best Neural Network's run.

Regularisation clearly helps LogisticRegression, but with he init it is little to no affected by the learning rate, it only shows how fast it stops converging

Linear regression is not fit for classification as it predicts continuous values when logistic regresssion predicts the probability of a given output. We can use sigmoid or softmax functions as activations for the last output. First we one-hot-encode our data, which means creating an array of the length of classes. For MNIST dataset we get an array with 10 zeros, as there are 10 classes and change the correct label to 1. So if this image is of class 3, we change the zero in the array place third to 1. To get the answer from our network we just take the index of the maximum number. The difference between sigmoid and softmax in that case is that softmax will normalise the answers, so that the sum of all probabilities will be 1.0, whilst sigmoid will just output the probability without normalising them.

V. Conclusion

In the start of the project, all the different methods such as SGD, logistic regression and neural network were implemented separately. But after some time it is clearly visible that they are based on each other and therefore can be run using the same code base, but with different configurations. Logistic regression and simple SGD for linear regression are the same as the neural network with no hidden layers. For linear regression the activation function is the identity function instead of sigmoid or softmax. This realisation helps to see that simple OLS might be quicker and more useful for simpler cases with less data and noise. But in the case of a lot of classes or more noise one need add more hidden layers to be able to track more of the important features. This is reflected in the results for the Franke's data the neural network did not achieve much better results just improving MSE by 0.0016, but for the classification neural network has managed to improve accuracy by 0.03.

For the feedback about the project itself, I found extremly fascinating to see the parallels between the different methods. It was also satisfying and motivating seeing the models slowly getting smarter by implementing the new methods. Since I did not manage to finish extensive research in my previous work as to what parameters are best fit to Franke's data, I suffered from that loss this project as well since I did not have good data to compare it with. Also for the Franke's data I should have chosen a higher degree polynomial or added more noise (from 0.1 to 0.3), since it is difficult to improve something that has already such a low MSE.

If the time allowed it I would love to dive in to exactly which numbers the classification trainer struggles with. I would love to run more tests on just binary datasets with only two numbers. When I started reading more about neural networks I found out that it is possible to propagate the image through the network and see how it is being affected, thus seeing what weights it is prioritising. This will definetly be something I will look closer into in the next project. As I have noticed the hardest part of this project was visualising what is happening in the neural networks. Since I planned my work to first generate the data, run training with different parameters and then analyse it, it resulted in of a lot useless data which made it difficult to come to definite answers. If I had more time, I would go more in depth on the models that succeeded. There were also a few methods that have been implemented in the code, such as momentum in SGD, but I did not have time to experiment properly with that. But that only makes me more driven to start on the next project.

References

[Géron, 2019] Aurélien Géron. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition. *O'Reilly Media*.

[A. Bronshtein, 2017] Adi Bronshtein. (2017). Train/Test Split and Cross Validation in Python. *https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6*.

[N. Kumar, 2019] Niranjan Kumar. (2019). Implementing and Analyzing different Activation Functions and Weight Initialization Methods Using Python. *https://towardsdatascience.com/implementing-different-activation-functions-and-weight-initialization-methods-using-python-c78643b9f20f*.

[H. Hukkelås, 2020] Håkon Hukkelås. (2020). Starter code for Computer Vision assignments *https://github.com/hukkelas/TDT4265-StarterCode/tree/master/assignment1*.

[Karan Desai, 2019] Karan Desai. (2019). Up Down Captioner Baseline for nocaps *https://github.com/nocaps-org/updown-baseline/blob/266081042bc2f0c3e6676ee0b5e204036cb6a74a/updown/config.py/*.

[M.E. Nylund, 2020] Maria Emine Nylund. (2020). FYS-STK4155: Data Analysis and Machine Learning *https://github.com/marianylund/fysstkprojects/blob/master/Project1/Report/FYS$_S$TK4155$_M$aria$_N$ylund$_P$roject$_1$.pdf/*.

[Morten Hjorth-Jensen, 2019] Morten Hjorth-Jensen. (2019). Data Analysis and Machine Learning: Logistic Regression. *https://compphysics.github.io/MachineLearning/doc/pub/LogReg/ipynb/LogReg.ipynb*.

[Morten Hjorth-Jensen, 2020] Morten Hjorth-Jensen. (2020). Data Analysis and Machine Learning: Linear Regression and more Advanced Regression Analysis. *https://compphysics.github.io/MachineLearning/doc/pub/Regression/ipynb/Regression.ipynb*.

[Devore and Berk, 2018] Jay L. Devore and Kenneth N. Berk. (2018). Modern Mathematical Statistics with Applications. 2nd Edition. *Springer*.

[Nielsen, 2019] Michael Nielsen. (2019). Neural Networks and Deep Learning. *http://neuralnetworksanddeeplearning.com/*.

[Schmidhuber, 2014] Juergen Schmidhuber. (Oct 2014). Deep Learning in Neural Networks: An Overview. *Neural Networks, Vol 61, pp 85-117, Jan 2015*. arXiv:1404.7828v4 [cs.NE].