Applications of Gradient Descent in Logistic Regression and Feedforward Neural Networks

Carl Fredrik Nordbø Knutsen, Halvor Tyseng, Benedict Leander Skålevik Parton

September 2022

In this report we explore some applications of the stochastic gradient descent algorithm, including estimating optimal parameters for linear regression problems, logistic regression problems, and for finding feedforward neural network parameters. We explore the performance of gradient descent using different parameters and learning rate scaling schemes on the Franke function. Overall we find that employing smaller mini-batch sizes leads to faster, albeit noisier, convergence. We find that RMSProp generally tends to perform quite well. Using a feedforward neural network to approximate the Franke function, we find that leaky ReLU performs well as a hidden layer activation function. Furthermore, we find that using two hidden layers with 10 nodes each outperformed the other tested architectures. And finally, we successfully approximate the Franke function using this network architecture. Next, we employ logistic regression and neural networks on classification problems. We observe that feedforward neural networks can approximate non-linear classification boundaries that cannot be approximated by logistic regression. We also analyze the Wisconsin breast cancer dataset (Wolberg, 1995). Using logistic regression, we achieve an accuracy score of of 0.930. Meanwhile, we find that a network with 2 hidden layers of 10 nodes using a learning rate $\eta = 0.005$ and l^2 regularization $\lambda = 10^{-3}$, using AdaGrad scaling and leaky ReLU as hidden activation function, gave an estimated accuracy score of 0.979.

1 Introduction

In this report we will tackle three major topics. First, we will analyze gradient descent as a method for minimizing cost functions. Here, we will use linear regression as an example for analyzing the performance of different variations of the algorithm. Second, we will introduce feedforward neural networks (FFNN) and analyze their performance when approximating a range of different functions, using different configurations for layers and activation functions. Finally, we will use a FFNN for classification tasks and compare its performance with logistic regression. For the gradient descent analysis, we will generate data using the Franke function with some added noise. For the classification tasks, we will first study an artificial example, namely the function $g(x, y)$ that is 1 if $x > 0.5$ and $y > 0.5$ and 0 otherwise. Finally, we will tackle real world data, in particular the Wisconsin breast cancer data set (Wolberg, 1995).

In section 2, we will explain the methods employed in this report. First, we introduce the concept of gradient descent and stochastic gradient descent. We move on to explain different variations of these algorithms, including AdaGrad, RMSProp, Adam, and the addition of momentum. We then

move on to FFNNs, where we first describe the forward propagation algorithm and discuss choice of activation functions. We then cover the backpropagation algorithm. Next, we introduce logistic regression, and characterize it by its cost function. Then, we introduce the data sets we use in our analysis.

In section 3, we perform our analysis on the three different data sets. We proceed to discuss these results more in detail in section 4. We proceed to evaluate the different versions of the gradient descent algorithm, along with evaluating the performance of different activation functions, number of layers, and number of nodes per layer for the FFNN. And finally, in section 5 we conclude and summarize our main findings.

2 Methods

2.1 Gradient Descent

In this project, we are concerned with minimizing cost functions. The gradient descent algorithm is an iterative scheme that can be used to find a minimum of a function. In particular, if we have a cost function C with parameters w, one can update the t'th estimate for the minimum w_t by

$$w_{t+1} = w_t - \eta \nabla C(w_t). \tag{1}$$

Here $\nabla C(w_t)$ denotes the gradient of the cost function and η is the learning rate, a hyper-parameter. In essence, the algorithm is finding the direction in which the cost grows most quickly, and taking a small step in the opposite direction in the parameter space.

The learning rate, which has to be chosen in advance, plays a crucial role in determining whether the algorithm will converge towards a minimum, and the speed at which it does so. A high learning rate can lead to divergence, meanwhile a low learning rate can lead to a slower convergence. The value of a good learning rate is often determined empirically.

An important fact to note is that the gradient descent algorithm cannot guarantee convergence towards a global minimum (even with sufficiently small learning rate and enough iterations). This is due to the fact that the gradient is a local property; it behaves identically in any local minimum as compared to a global minimum. However, we have a special case for convex functions. Since any local minimum of a convex function is a global minimum as well, these functions are well-suited for gradient descent. However, in a lot of machine learning applications one must deal with the problem of getting stuck in local minima.

2.1.1 Stochastic Gradient Descent

The gradient of the cost function be expressed as the sum of the individual costs for each training sample, for cost functions such as the mean squared error (MSE). That is, we can write

$$\nabla C(w_t) = \sum_{i=1}^{n} \nabla C_i(w_t). \tag{2}$$

Then computing the gradient entails computing the gradient for each of the data points, which can become quite computationally expensive for large data sets. Stochastic gradient descent mitigates

this concern by only computing the gradient for one data point, and updating. For $i = 1, 2, \ldots, n$ you update the weights with

$$w_{t+1} = w_t - \eta \nabla C_i(w_t). \tag{3}$$

Updating the parameters multiple times per epoch means that the cost will approach a local minimum in fewer epochs.

A middle ground between regular gradient descent and stochastic gradient descent as described above, is stochastic gradient descent with mini-batches. This algorithm entail setting a batch size B. Then you shuffle the data and split it into disjoint sets of size B, called mini-batches. In one epoch, you update the weights once for each mini-batch, using the mini-batch's cost function gradient. That is, for the j'th mini-batch you update the parameters with

$$w_{t+1} = \sum_{i=1+B\cdot(j-1)}^{B\cdot j} \nabla C(w_t). \tag{4}$$

2.1.2 Momentum Methods in Gradient Descent

When you have a noisy gradient, momentum can be a useful tool for improving convergence. Especially with stochastic gradient descent, the gradient of a single sample or a small mini-batch is not necessarily the best possible estimate one can get for the optimal direction to take a step in. Therefore, you introduce the momentum v, that you update with

$$v_t = \gamma v_{t-1} + \eta \nabla(w_t), \tag{5}$$

for regular gradient descent. For stochastic gradient descent with mini-batches, the gradient is simply replaced with the gradient of the cost of the mini-batch. Next, the parameters are updates with

$$w_{t+1} = w_t - v_t. \tag{6}$$

Analogously to physical systems, you keep a (discounted) momentum in the direction that the previous gradients have had.

2.1.3 Adaptive Gradient Descent - AdaGrad

If the learning rate is set too high, the weights might never settle to an acceptable value as each iteration causes it to make too great leaps, whilst setting the learning rate too low will cause the weights to take far too long to converge at an optimal value. AdaGrad is an optimization method which aims to tackle this problem by adaptively scaling the learning rate for each weight; making it smaller as the weight approaches its optimal value. First we define $g_t = \nabla C(w_t)$. Then AdaGrad is defined by

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_\tau \tag{7}$$

where

$$G_\tau = \sum_{\tau=1}^{t} g_\tau \odot g_\tau. \tag{8}$$

Here $\odot$ denotes the elementwise product.

2.1.4 Root Mean Squared Propagation - RMSprop

Another optimization method which adaptively scales the learning rate is RMSprop. When scaling the learning rate using RMSProp, we keep track of the quantity s_t given by

$$s_t = \beta s_{t-1} + (1-\beta) g_t \odot g_t, \tag{9}$$

for some constant β. The parameters are then updated with

$$w_{t+1} = w_t \frac{\eta}{\sqrt{s_t + \epsilon}} \odot g_t. \tag{10}$$

2.1.5 Adaptive Moment Estimation - Adam

Adam is yet another method for scaling the learning rate. Adam keeps track of the quantities m_t and s_t, where

$$m_t = \frac{\beta_1 m_{t-1} + (1-\beta_1) g_t}{1-\beta_1^t} \tag{11}$$

and

$$s_t = \frac{\beta_2 s_{t-1} + (1-\beta_2) g_t \odot g_t}{1-\beta_2^t}, \tag{12}$$

for constants β_1 and β_2. Then the parameters are updated with

$$w_{t+1} = w_t - \eta \frac{m_t}{\sqrt{s_t} + \epsilon}. \tag{13}$$

2.2 Logistic Regression

Logistic regression is a useful tool for classification. It is the result of minimizing the cross-entropy cost function

$$C(p) = -\sum_i \Big(y_i \log(p_i) + (1-y_i) \log(1-p_i)) \Big), \tag{14}$$

when the predicted probabilities p_i are given by

$$p_i = \sigma(X\beta) = \frac{1}{1+\exp(-X\beta)}, \tag{15}$$

for a design matrix X and parameters β. It is worth noting that the logistic regression cost function is convex, which makes gradient descent a reliable option for finding a global minimum.

2.3 Feedforward Neural Networks

Feedforward neural networks (FFNN) are in essence a type of non-linear function, that is computed in a layered fashion. The output of an FFNN is computed by the forward propagation algorithm.

2.3.1 The Forward Propagation Algorithm

A k-layer network consists of three types of layers: the input layer, the hidden layers, and the output layer. The input layer L_0 simply takes the input to the function, and outputs these values to the next layer. The hidden layer L_i, for $0 < i < k$, takes input the input vector a_{i-1} from the $i-1$'st layer. It then computes $z_i = W_i a_{i-1} + b_i$ for a weight matrix W_i and bias vector b_i. We then apply the activation function f_i elementwise to the vector z_i to compute the layer outputs a_i. The output of the output layer k is computed in the same way, but we note that the choice of activation function f_k will often differ from the previously used activation functions.

2.3.2 Choice of Activation Function

The output of an FFNN depends on the weight matrices W_i, the biases b_i, and notably also on the choice of activation functions f_i. In this project, we will restrict ourselves to studying neural networks where all the hidden layers use the same activation function. We will examine three commonly used hidden layer activation functions. Firstly, we will Study the sigmoid function given by

$$\sigma(x) = \frac{e^x}{1+e^x}. \tag{16}$$

We will also utilize the Rectified Linear Unit (ReLU) function, here denoted by

$$R(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{17}$$

We will also try a variation of the ReLU, namely the leaky ReLU:

$$R_{leaky}(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise.} \end{cases} \tag{18}$$

For the output layer, we will study two different activation functions. The sigmoid function σ is a reasonable choice for classification tasks since it outputs values between 0 and 1. When we want our network to be able to output arbitrary values in $\mathbb{R}$, we will simply use a linear activation function.

2.3.3 The Backpropagation Algorithm

To find a reasonable set of parameters W_i and b_i, we make the assumption that a local minimum in the cost as a function of the parameters, will give a good set of parameters. In particular, we employ gradient descent to find such a local minimum. For approximating the Franke Function, we have deemed the mean squared error a suitable cost function. That is, for our model parameters θ we minimize

$$C(\theta) = \frac{1}{n}(y - \tilde{y}(\theta))^2 + \lambda\|\theta\|_2^2, \tag{19}$$

where $\lambda\|\theta\|_2^2$ is an l^2 regularization term. For classification tasks, the cross-entropy cost function

$$C(\theta) = -\sum_i \Big(y_i \log(p_i(\theta)) + (1-y_i)\log(1-p_i(\theta)) \Big) + \lambda\|\theta\|_2^2, \tag{20}$$

is more suitable. Again, $\lambda\|\theta\|_2^2$ is an l^2 regularization term. In any case, we have reduced our minimization problem to computing the gradient of the cost function. The computation of this gradient for a FFNN uses the back propagation algorithm, which boils down to repeatedly applying the chain rule. The full implementation can be found in our code, see appendix A.

2.4 Data Sets

In this report, we analyze three data sets.

2.4.1 The Franke Function

The Franke Function is given by

$$\begin{aligned} F(x,y) = &\frac{3}{4}\exp(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}) + \frac{3}{4}\exp(-\frac{(9x+1)^2}{49} - \frac{(9y+1)^2}{10}) \\ &+ \frac{1}{2}\exp(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}) - \frac{1}{5}\exp(-(9x-4)^2 - (9y-7)^2). \end{aligned} \tag{21}$$

We generate points (x_i, y_i) from a uniform distribution on $[0,1] \times [0,1]$. We then compute the dependent variable $z_i = F(x_i, y_i) + \epsilon_i$, where $\epsilon_i \overset{\text{iid}}{\sim} \mathrm{N}(0, \sigma^2)$.

2.4.2 An Example Function for Classification

We define the function

$$g(x,y) = \begin{cases} 1 & \text{if } x > 0.5 \text{ and } y > 0.5 \\ 0 & \text{otherwise.} \end{cases} \tag{22}$$

We use this function in our analysis of classification using logistic regression and FFNNs. Here, $g(x,y)$ can be considered a probability density function for observing class 1 at a given point (x,y). We remark that the discontinuity of this function is a notable property.

2.4.3 Wisconsin Breast Cancer Data

The main goal of this report is to study the Wisconsin Breast Cancer data set (Wolberg, 1995). The data set contains features such as tumor radius, texture, perimeter, and so on. We will train on this data to predict whether a certain tumor is malignant or benign.

3 Results

The source code is linked in appendix A.

3.1 Gradient Descent Analysis on Franke Function

We will approximate optimal linear regression estimates for approximating the Franke function using gradient descent. We have done so to compare the performance of different gradient descent and stochastic gradient descent configurations.

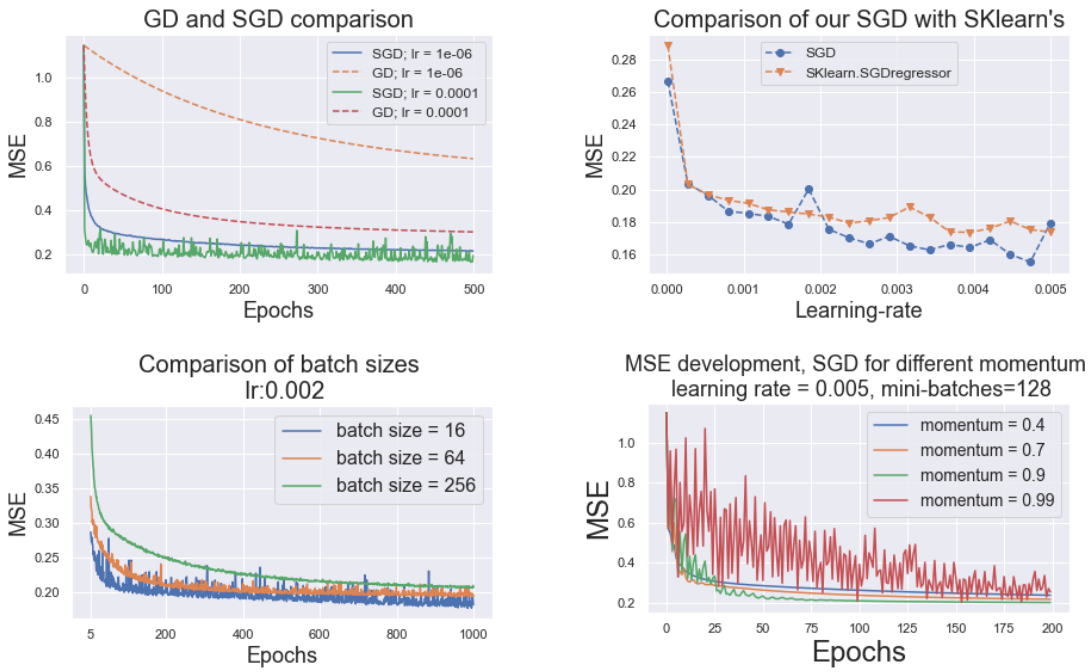

Figure 1: These plots show the basic workings of the parameters one can choose in gradient descent.

The plots in figure1 show our main findings from the initial analysis of gradient descent. This analysis was done on similar data, scaled like in project one, with polynomial degree 5 and 1000 data-points. The analytical solution yielded a MSE of 0.1265, this will then benchmark for our model's. Here we do not use methods to scale the learning rate, and it is therefor constant. For GD compared to SGD (figure 1 top right.) we see a faster descent towards the minimum, but a much more volatile, and an increasing learning rate η seems to only increase this feature. Our model seem to preform identical, or either slightly better than SKlearn's model. At increasing learning rates we see a greater different in the two model's performance. Mini batches lets us get the best from both world's, since we get a faster descent than GD with less noise.

When we add momentum one can see that that larger momentum makes the descent more noisy, but as with SGD lets us faster reach a lower score value. One can see the noisiness of the descent with 0.9 in momentum die out around 50 epochs, but with 0.99 in momentum the noisiness is significant until at least 200 epochs.

Then we analysed the scaling methods, examining how the MSE cost of the gradient descent approximation of the OLS estimate $\hat{\beta}$ evolved as a function of epochs. We tested three different learning rates for no learning rate scaling, Adam, RMSProp, and AdaGrad as shown in figure 2. In particular, we have plotted epochs 5 through 100. We see that a higher learning rate generally leads to faster convergence, except for in the case of RMSProp where it leads to increased instability. We see the best performance for Adam with learning rate $\eta = 0.04$, however we note that this is a

quite limited number of epochs.

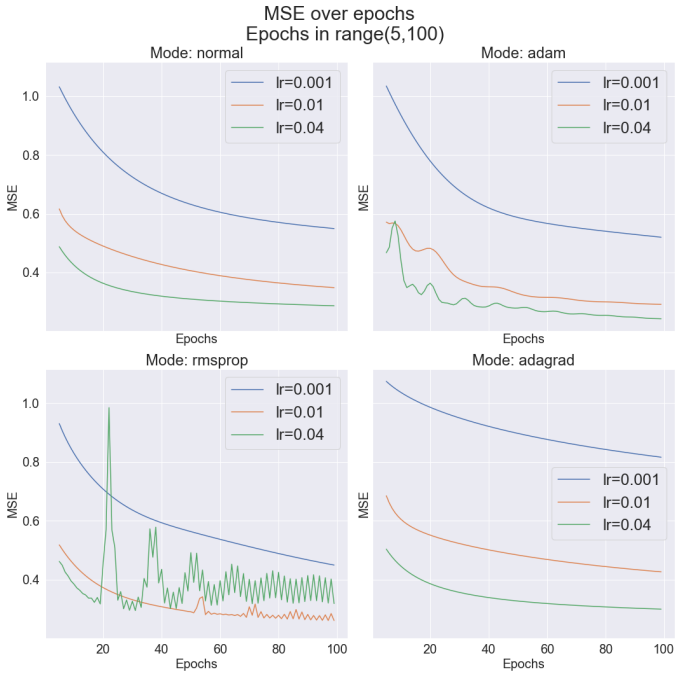

Figure 2: Plain gradient descent, for each mode. None mini-batches and no momentum. First five epochs is cut away to se the difference more clearly.

We then perform a grid search on learning rate and momentum for 2000 epochs, as shown in figure 3. For no learning rate scaling, we see that we generally get better performance for 0.01 as learning rate and the highest examined momentum, ie. 0.9. The computed MSE was NaN for the highest learning rates, we believe this means that the gradient descent diverged. For Adam, we observe that the combination of high momentum and high learning rate appears to lead to poor performance or even divergence. We got the best MSE for low momentum $(0.1 - 0.3)$ and learning rate 0.01. For the lower learning rates (eg. 10^{-5}), we note that Adam appears to perform better with higher momentum. For RMSProp, we see that a learning rate of 0.01 and momentum 0.9 leads to optimal results. And for AdaGrad, we note that higher learning rates are needed. It performed the best for the maximum momentum of 0.9 and the maximum learning rate 1.0. We see the best overall performance with RMSProp, though the differences are quite small.

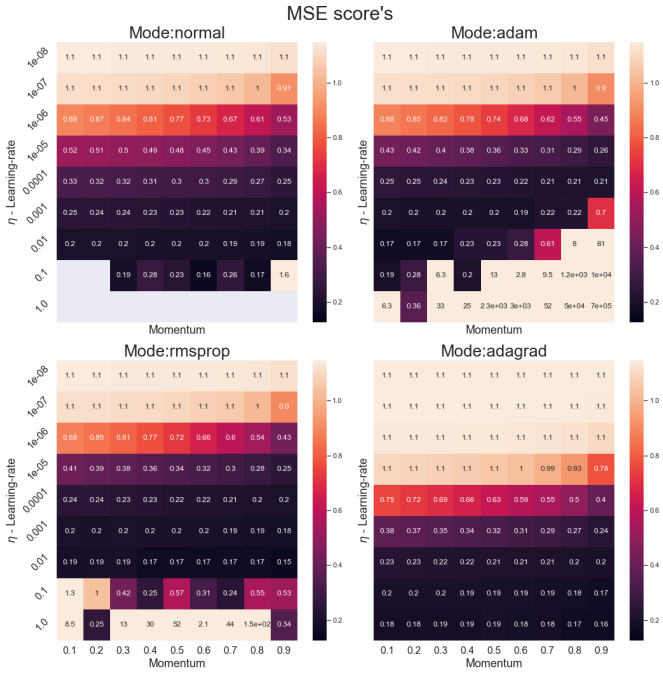

Figure 3: Grid search for momentum and learning rate for gradient descent over 2000 epochs, with lambda 0

Figure 4 shows the MSE as a function of learning rate and regularization. We observe that for this model, regularization does not appear to be of particularly good use. We again see that the best learning rate depends on the particular learning rate scaling in use.

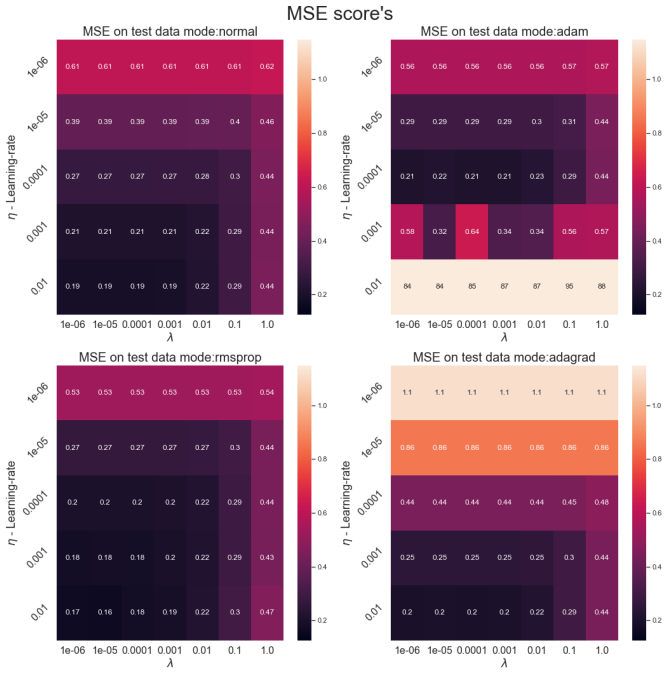

Figure 4: Grid search for lambda and learning rate, with momentum=0.9

3.2 Neural Networks on Franke Function

When analyzing the Franke Function using neural networks, we first chose to investigate the impact of different activation functions in the hidden layers on the MSE of the validation dataset. Using constant learning rate of 0.0005, a network structure of two hidden layers with 10 nodes in each, and training each model across 1000 epochs, we got our results as seen in figure 5. We see that ReLu and Leaky ReLu do approximately equally well, and so choose Leaky ReLu between them in our further analysis.

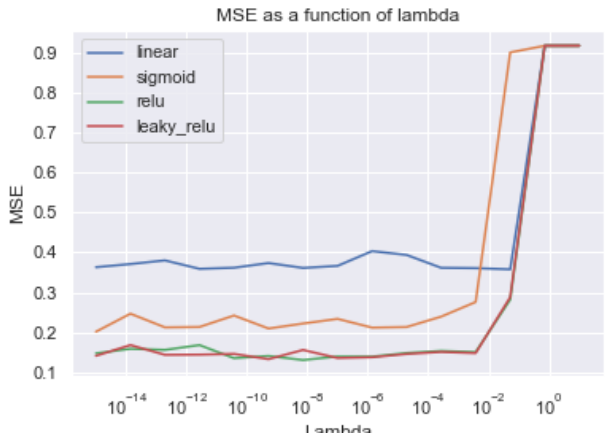

Figure 5: MSE as a function of regularization parameter λ for different activation functions. ReLu and Leaky ReLu seems to perform best across the regularization range.

Our next investigation involved observing the impact of different optimization methods on the MSE of the validation dataset. Setting all other parameters as previously, and now using Leaky ReLu as the activation function in the hidden layers, we got figure 6. We observe that RMSProp achieves the best MSE and stays stably below the other optimization methods, and thus choose this method in our further analysis.

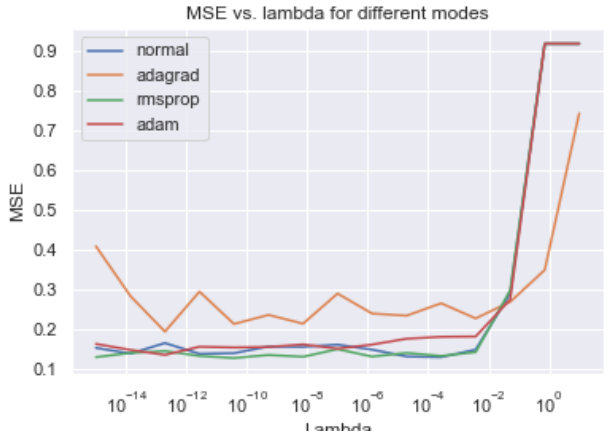

Figure 6: MSE as a function of regularization parameter λ for different modes. RMSProp seems to perform best across the regularization range.

We then continued with our previous parameters and produced a heatmap of the MSE for different

regularization parameters λ and learning rates η. This resulted in figure 7. We observe that lower learning rates seem to perform considerably worse, seemingly due to a failure to converge in time, as we only iterate over 1000 epochs. Thus, we choose the learning rate 10^{-2} in our further analysis, as this achieved the optimal MSE on the validation dataset in our gridsearch.

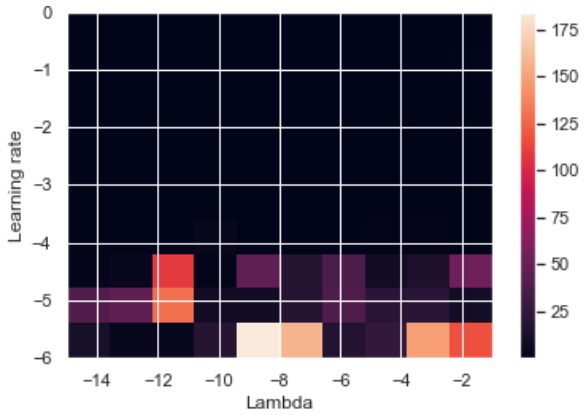

Figure 7: MSE as a function of regularization parameter λ and learning rate η. Lower learning rates seem to perform worse. Numbers on both axes are transformed with the logarithm base 10.

The neural network structure may also have a significant effect on how well the model performs. Thus, we explored how well four different neural network structures would fare with our dataset. We used four representative structures; two networks with 5 hidden layers, with respectively 4 and 10 number of nodes in all these layers, and two networks with 2 hidden layers, also with respectively 4 and 10 number of nodes in all such layers. For each of these, we found the MSE on the validation set for a range of regularization parameters, and got a plot as shown in figure 8.

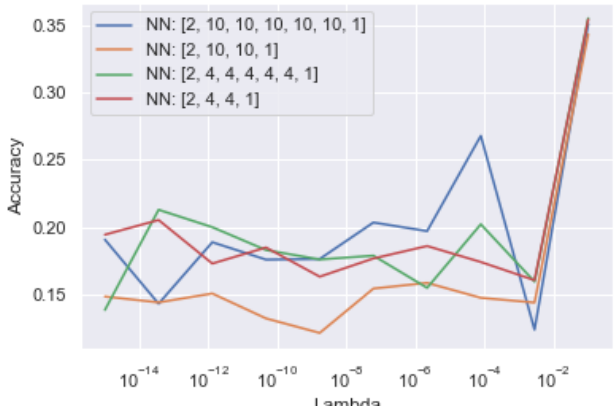

Figure 8: MSE as a function of regularization parameter λ for different neural network structures. y-axis is erroneously labelled with accuracy, this should be MSE.

We find that the network consisting of 2 hidden layers with 10 nodes in each performs the best. Finally, we used these the previously determined learning rate, along with the optimal found regularization $\lambda = 1.67 \cdot 10^{-9}$, and trained on the training and validation data. We achieve an MSE of 0.1226 on the test data set.

In figure 9, we see the optimal neural network prediction of $x, y \in [0, 1]$ trained on the Franke Function dataset, compared with the actual Franke Function data.

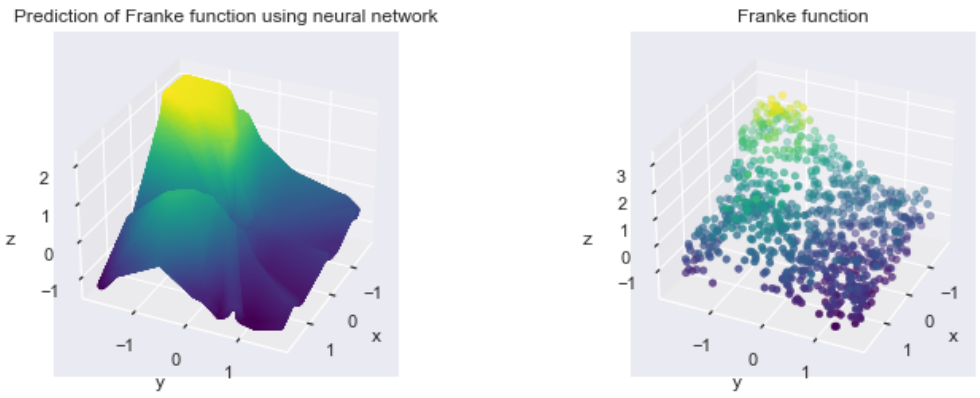

Figure 9: These plots show the neural network prediction of the Franke Function, and Franke function samples, after scaling.

3.3 Classification Analysis

3.3.1 Analyzing a Non-Noisy Example Function

We first performed logistic regression on the training data, using our implementation of stochastic gradient descent. We used a batch-size of 10, a learning rate of 0.01, and a momentum of 0.9. Figure 10 shows the cross-entropy cost as a a function of epochs.

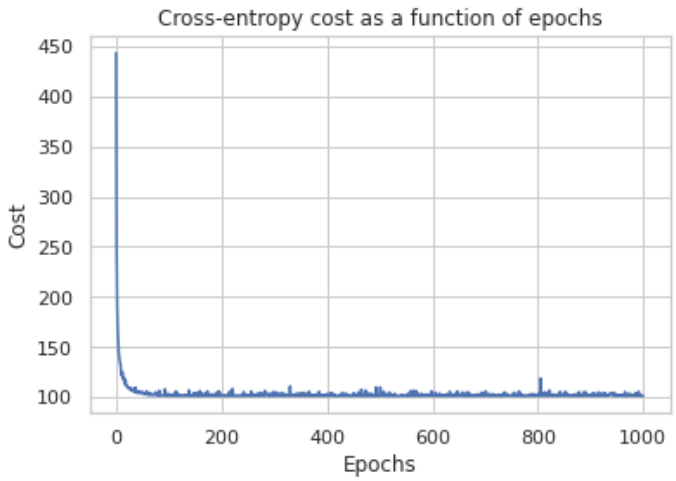

Figure 10: This figure shows how the cross-entropy evolves as a function of epochs. The cost converges quickly, with some spikes

The resulting predictions over $[0, 1] \times [0, 1]$ are shown in figure 11. We observe that the decision boundary is a line by our choice of a logistic regression model.

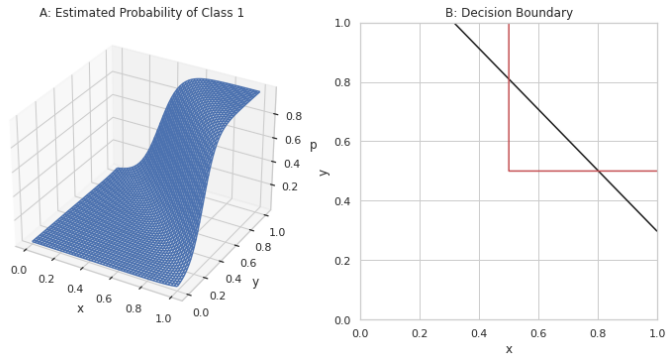

Figure 11: A: Here we see the logistic regression model's predicted probabilities p of class 1, given x and y. B: The black line shows the decision boundary for predicted class 1. The red line encloses the area where the function is actually valued 1.

Next, we considered the pairs of learning rates and regularization parameters (η, λ) for $\eta \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, \}$ and $\lambda \in \{0, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10\}$. For each pair (η, λ), we ran gradient descent using the specified parameters and computed the accuracy score on the validation set. We averaged over 10 runs, giving us figure 12. We see that as long as the learning rate and regularization is sufficiently small ($\eta < 0.1$ and $\lambda < 0.01$), we get prediction scores between 0.91 and 0.92. We decide on using learning rate $\eta = 0.001$ and regularization $\lambda = 0$ for our model. Training the model on all the training data and averaging over 100 runs, we get an average accuracy score of 0.876. Meanwhile, scikit-learn's logistic regression model yielded a slightly higher accuracy of 0.89.

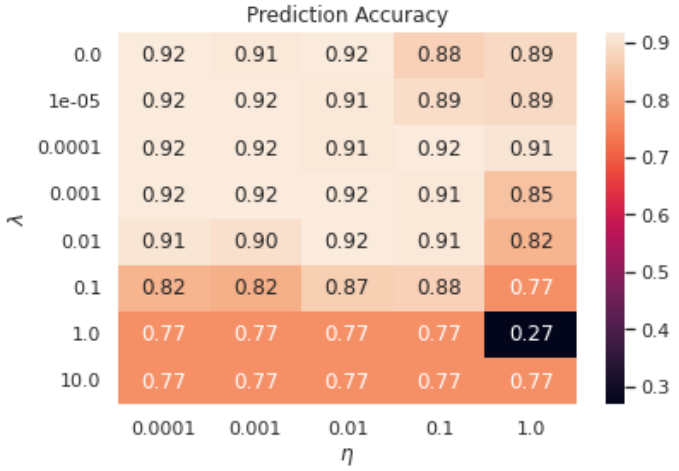

Figure 12: Accuracy score for each combination of η and λ, averaged over 10 runs.

We proceeded to repeat the analysis using an FFNN model with 3 hidden layers consisting of 4 nodes each. We used ReLU as activation function for the hidden layers, and the sigmoid as the output activation. We trained 100 such networks using a learning rate of $\eta = 0.01$, minimizing the cross-entropy cost. A histogram over the resulting accuracy scores is shown in figure 13. The mean accuracy score was 0.953, which is quite a bit better than the score from logistic regression.

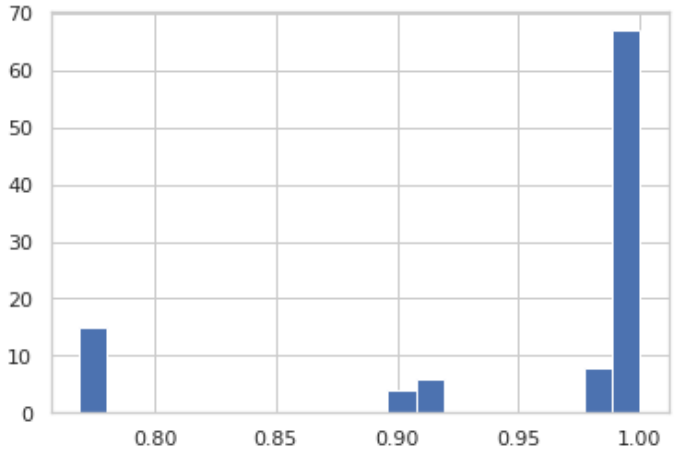

Figure 13: Accuracy score of 100 neural networks trained with $\eta = 0.01$ and $\lambda = 0$. 75 out of 100 networks had an accuracy score greater than 0.98.

We observe that the neural network quite easily gets stuck in local minima in the cost function. The predicted probability of class 1 for two network trained using the exact same parameters is shown in figure 14. We see that the first neural network almost perfectly recreates the the function, meanwhile the second one resembles the probability distribution from the logistic regression case.

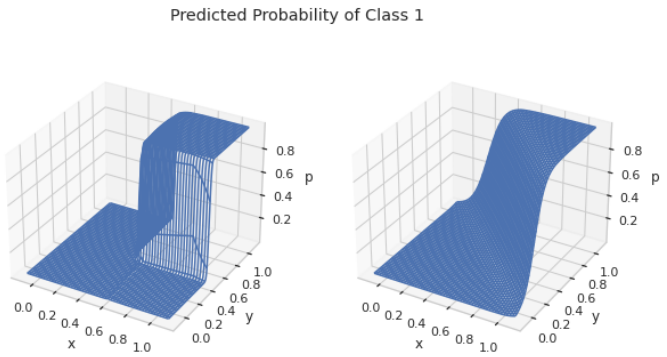

Figure 14: Predicted probability p of class 1 as a function of x and y for two neural networks trained with $\eta = 0.01$. The first network almost perfectly recreated the target function. The second network appears to be trapped in a local minimum.

We keep the architecture and activation function choice fixed, and perform a grid search for η and λ for no learning rate adjustment, for AdaGrad, for RMSProp, and finally for Adam. The results are shown in figure 15. We note that we see a slight tendency towards RMSProp and Adam performing better. We pick the best set of parameters, ie. $\eta = 0.0001$ and $\lambda = 0.001$ using Adam, and estimate the performance on the test set. Averaging over 100 runs, we get an estimated expected accuracy score of 0.975. We repeat the analysis using the default configuration of scikit-learn's MLPClassifier for 3 hidden layers of 4 nodes, still using 500 max iterations. Averaging over 100 runs, we get an estimated accuracy score of 0.908. We note that we could get a higher accuracy score if we tuned the parameters of the model. Overall, we see that the FFNN greatly outperforms logistic regression at its best, but that it also can perform quite poorly if it happens to get stuck in a local minimum.

3.3.2 Analyzing the Wisconsin Breast Cancer Data

We first performed an analysis on how the cost function of a preliminary neural network developed as a function of epochs, the result of which can be seen in figure 16. Though the precise development of the cost function varied slightly with the random initialization of the weights, we observed that it had sufficiently converged by 500 epochs in all instances, and thus chose to perform the rest of the analysis using this epoch value. We also chose a batch size of 16 throughout the analysis.

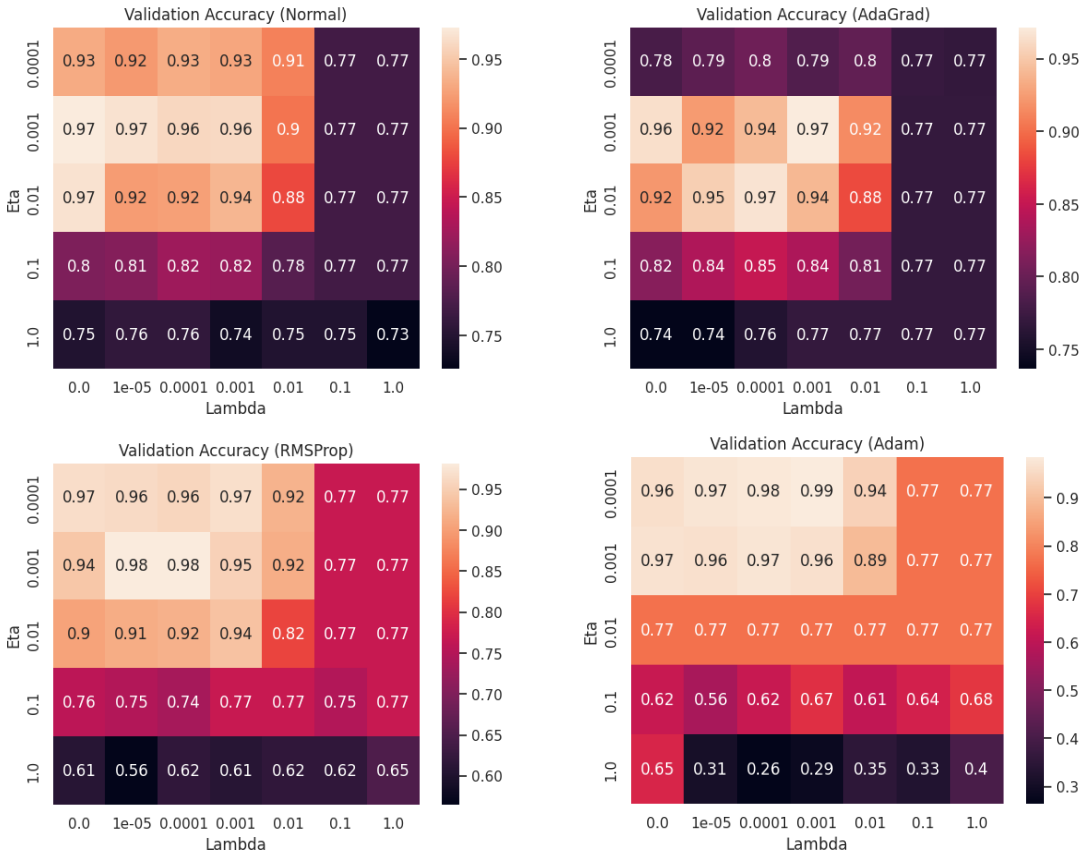

Figure 15: These plots show the estimated accuracy score from a grid search on learning rate η and regularization parameter λ, using constant learning rate, AdaGrad, RMSProp, And Adam.

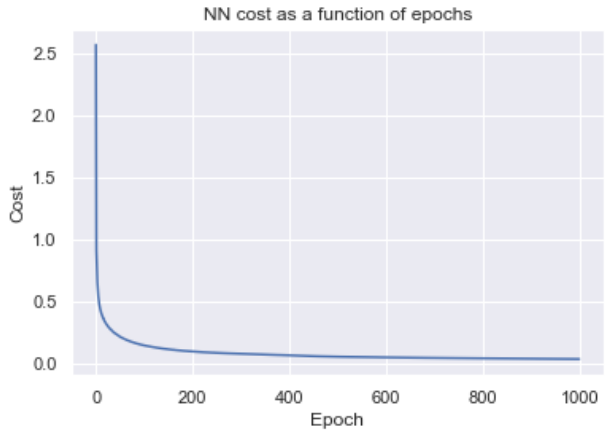

Figure 16: The development of cost of the neural network over epochs. The cost seems to sufficiently converge by 500 epochs.

We moved on to analyze how the accuracy on the validation dataset varied between the different activation functions in the hidden layers. For a range of different regularization parameters, we plotted the accuracy (taken as an average over 10 different weight initializations) for each activation function. The resulting plot can be seen in figure 17. From this figure, we observed that Leaky ReLu consistently outperformed the other activation functions, and so this is the activation function which we use for the rest of the analysis. We also did the same analysis, using Leaky ReLu, varying between optimization methods rather than activation functions. From the resulting figure 18, we observed that AdaGrad appears to outperform the other modes, and decided to use this mode in our further analysis.

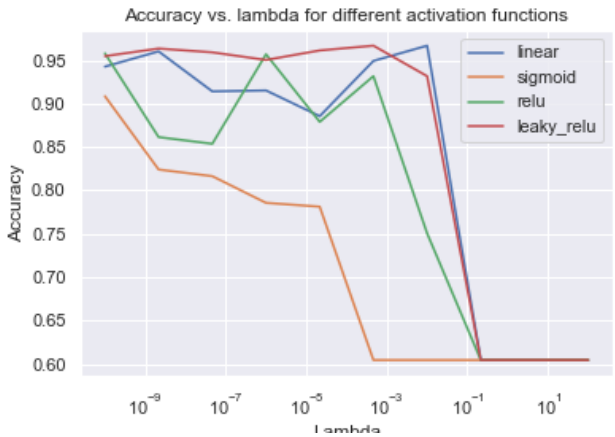

Figure 17: The average accuracy over regularization parameter λ for different activation functions. Leaky ReLu seems to be quite stable compared to the other activation functions.

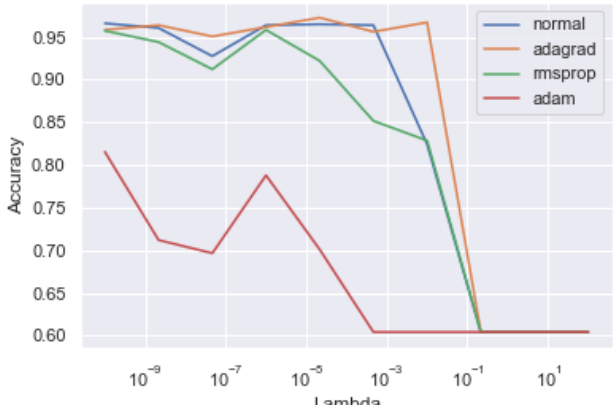

Figure 18: The average accuracy over regularization parameter λ for different modes. AdaGrad seems to be quite stable compared to the other modes.

We then proceeded to analyze how the accuracy varied with the learning rate η and regularization parameter λ for the parameters previously set. We iterated over 8 learning rates and 10 regularization parameters, logistically distributed (and, to limit computational time, had to reduce the number of epochs to 400, and averaged over 8 instead of 10 initializations), and found the accuracy on the validation set for each of these. This resulted in the heat map from figure 19, which gave us optimal accuracy for $\eta \approx 0.005$ and $\lambda \approx 0.0008$.

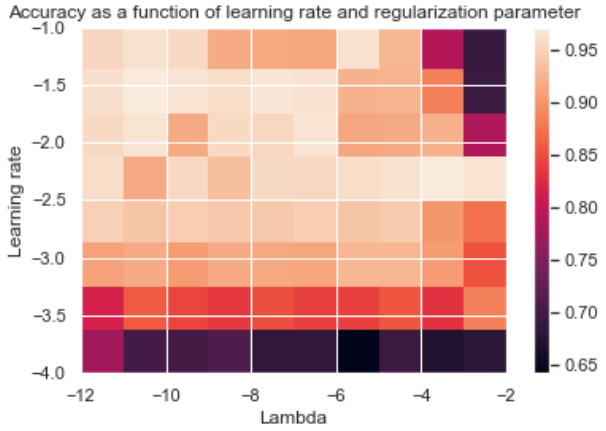

Figure 19: Heatmap of average accuracy by learning rate η and regularization parameter λ. High λ and low η appears to take a toll on the accuracy. Numbers on both axis is 10 exponential, e.g $-12 = 10^{-12}$.

Table 1: Optimal regularization parameter and accuracy on validation set for different neural network structures.

Structure	λ	**Accuracy**
[30, 10, 10, 10, 10, 10, 1]	10^{-2}	0.9670
[30, 10, 10, 1]	10^{-3}	0.9736
[30, 4, 4, 4, 4, 4, 1]	10^{-7}	0.9692
[30, 4, 4, 1]	10^{-3}	0.9703

Table 2: Confusion matrix for the optimal model on the test dataset.

	Pred. pos.	**Pred. neg**
Pos.	0.966	0.034
Neg.	0.015	0.985

Lastly, carrying over the optimal learning rate from figure 19 as well as all other previously determined parameters, we found the optimal regularization parameter with corresponding accuracy on the validation dataset for four different neural network structures. Aiming to explore how the accuracy varies between number of hidden layers and number of nodes in these layers, we chose four representative networks; two networks with 5 hidden layers, with respectively 4 and 10 number of nodes in all these layers, and two networks with 2 hidden layers, also with respectively 4 and 10 number of nodes in all such layers. From figure 20 we see that most structures, apart from the one with many layers and many nodes in each layer, stayed relatively consistent in their accuracy for different regularizations, before dipping when the regularization parameter got too high. The optimal accuracy was achieved with the neural network with few layers and many nodes, with a regularization parameter of $\lambda = 10^{-3}$. This neural network achieved an average accuracy on the validation dataset of 0.9736. The optimal regularization parameters for the other neural networks can be found in table 1.

When using all the optimal parameters we have gathered throughout this analysis, training on the train- and validation-dataset, and using this on the test-dataset (whilst averaging over 50 different random weight initializations), we find an average accuracy of 0.9789. We also produced a confusion matrix for this optimal model, as seen in table 2. We also trained an instance from the neural network class from the ScikitLearn-library on the training- and validation dataset and found the accuracy, averaged over 100 initializations, on the test dataset, which came out to be 0.9761.

Table 3: Optimal regularization parameter, learning rate and accuracy on validation set for different modes with logistic regression.

Mode	λ	η	**Accuracy**
Normal	10^{-5}	0.278	0.9670
AdaGrad	$3.6 \cdot 10^{-5}$	1.000	0.9670
RMSprop	10^{-5}	0.0215	0.9670
Adam	10^{-5}	0.006	0.9670

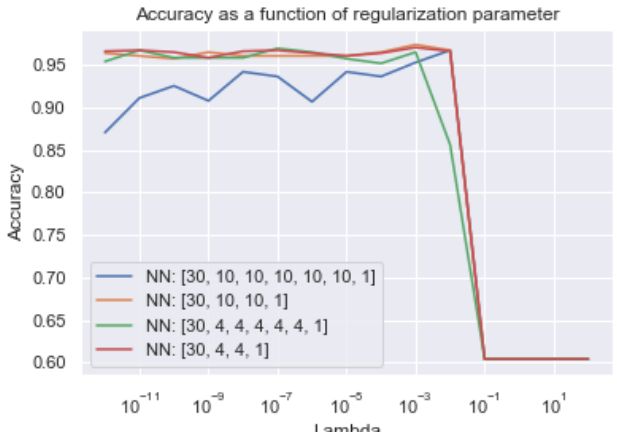

Figure 20: Average accuracy over regularization parameter λ for 4 different neural network structures. All structures, apart from the one with many layers and many nodes per layer, appear stable for different λ.

For comparison with neural networks, we also used logistic regression on the Wisconsin breast cancer dataset. We transfer most of the same parameters we used in our analysis with the neural networks; choosing a batch size of 16 and momentum of 0.9, and only investigated the accuracy with different learning rates and regularization parameters for different modes. We also did not need to average over many initializations, as the weights were always set to an initial value of 0.

For each learning rate scaling mode, we iterated over 10 values of regularization parameters, and 10 learning rates, both logarithmically selected in a range between 10^{-5} and 1. This resulted in the 4 heatmaps as seen in figure 21. The optimal regularization parameter and learning rate for each mode is shown in table 3. We found that all the optimal models for each mode achieved the same accuracy for the validation dataset.

When using the optimal model (which we choose to be with constant learning rate, as all modes resulted in the same accuracy on the validation dataset) on the test dataset, we got an accuracy of 0.9298. The confusion matrix for this model is shown in table 4. In this case, we also tried using the logstic regression class from the ScikitLearn-library to compare. We trained it on the training-

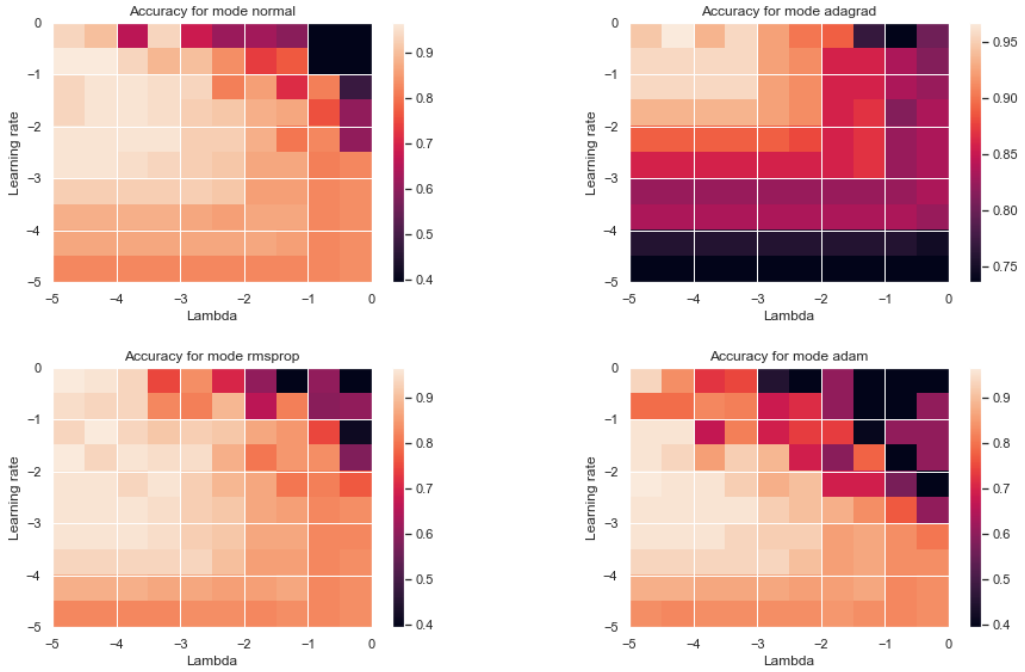

Figure 21: These plots show the estimated accuracy score from a grid search on learning rate η and regularization parameter λ, using constant learning rate, AdaGrad, RMSProp, And Adam. Numbers on all axis is in 10 exponential, e.g. $-5 = 10^{-5}$.

and validation dataset, and found the accuracy on the test dataset, which came out to be 0.9737.

4 Discussion and Evaluation

4.1 Gradient Descent Evaluation

We note that it is hard to capture the all the interactions between the different variables, including batch size, learning rate, regularization, momentum, and learning rate scaling method. We will, however, make some general recommendations and evaluations. First, we note that mini-batches can be a useful tool when handling large and complex datasets. By allowing for multiple gradients to be calculated per epoch, it can vastly increase the speed at which the cost approaches its

Table 4: Confusion matrix for the optimal model with logistic regression on the test dataset.

	Pred. pos.	**Pred. neg**
Pos.	0.905	0.095
Neg.	0.056	0.944

minimum. This does come at the cost of the cost as a function of epochs becoming slightly more noisy, but this does seem like a often-reasonable trade-off. Lower mini-batch sizes do appear to lead to higher degrees of noisiness. Therefore it seems reasonable to pick a batch-size that ensures convergence in a set number of epochs, but not much lower.

A higher learning rate can also increase the convergence speed, but too high of a learning rate can lead to divergence. Both too high and too low of a learning rate can therefore lead to not reaching a minimum in the cost. The learning rate can be problem-specific, but we have generally seen that learning rates in the range $[10^{-3}, 10^{-2}]$ often have seemed to work. Though performance of course depends on your chosen cost function, and how your data is scaled.

Regularization can play an important role in restricting model complexity, but we see that it is not always required or beneficial. Especially when the model complexity is low to begin with, we see that regularization leads to limited benefits. We see this eg. for our low-complexity logistic regression. We also see a tendency towards regularization having limited benefits for the lower-complexity neural networks. On the flip side, we did observe that the highest-complexity neural network did appear to benefit from greater regularization. As expected, we see that we need to take greater care in applying regularization when dealing with over-parameterized models.

Momentum generally appears to have a positive effect on convergence. Though we have observed that increased momentum ($\sim$ 0.8 to 0.9 and above) appears to cause divergence in the linear regression training, when scaling the learning rate using Adam. We cannot explain this phenomenon entirely, though it seems that increased momentum could require a lower learning rate. We note that we did not the same pattern for the classification tasks, where Adam performed quite well (and in some cases the best) even with a momentum of 0.9. We do not currently have a good explanation for this difference, and we would be interested in seeing if this finding can be replicated.

We observe that Adam, RMSProp, and Adam all can be beneficial modifications to the learning rate scaling. We see that both Adam and RMSProp tend to lead to faster convergence, with the cost fluctuating more with respect to epochs trained. Especially RMSProp can lead to quick convergence, while the cost is bouncing around quite a bit after it has closed in on the minimum. AdaGrad appears to require a higher learning rate parameter η, as it slows down the gradient descent quite a bit. It appears, however, to lead to a quite stable descent to the minimum. We also note that no scaling also results in quite similar results. All in all, we see that RMSProp, AdaGrad, and Adam all outperform each other certain experiments. Although, especially in the case of neural networks, our expected accuracy score estimates appear to be somewhat noisy. Therefore, we make a weak recommendation of using Adam, since it generally has appeared to lead to good convergence.

4.2 Approximating the Franke Function Using Neural Networks

The most important take-aways from our analysis of the Franke function are as follows: leaky ReLU and ReLU appear to perform quite well as activation functions, which could generalize to other tasks. And the result that the two-layered FFNN with 10 nodes per layer performed best, could hint towards low-layered high node networks being appropriate for such tasks. However, we had to limit our scope in this report to four networks. Therefore, there is highly likely some other set of nodes and layers that is optimal for this task. We do not know it yet, however

Furthermore, we note the fact that the network can approximate a relatively complex function such as the Franke function. To create similarly complex approximations using linear regression, one would have to add more features (eg. polynomial features).

4.3 Classification

In the case of the example function given by equation 22, the neural network appears to vastly outperform the logistic regression. This is a reasonable result, since the decision boundary for the logistic regression case by necessity must be a line. This holds since we get level curves in the predicted probabilities for

$$\frac{1}{1+\exp(-(w_0+xw_1+yw_2))} = C. \tag{23}$$

This implies that we get level curves that are lines, given by

$$w_0 + xw_1 + yw_2 = D, \tag{24}$$

for some constant D. Given these features, logistic regression cannot perfectly approximate the decision boundary. More features would be needed to counteract this.

The neural network, on the other hand, can establish more complex decision boundaries. The fact that the network can almost perfectly approximate function 22 (and seems to generalize quite well to outside the training domain) showcases the flexibility of FFNNs. On the other hand, the fact that the gradient descent relatively often lead to non-global minima, is notable a concern. Whereas the loss function is convex for logistic regression, that is not the case for the FFNN loss function. This can make the results vary quite widely, even when using the same set of hyperparameters and solver. Here we suggest researching methods for increasing the likelihood of finding the global minimum, as useful further work.

In analyzing the Wisconsin breast cancer dataset using FFNN, the problem of local minima for neural networks was something we handled by taking the average over a number of weight initializations. This, however, caused a significant bump in computational time. Coupled with the sheer number of parameters we wished to explore, we often relied on doing a simple analysis of a certain parameter, and assumed this would generalize to our further investigations. If equipped with more computational power and/or time, a more thorough analysis of the accuracy and its dependence on specific parameters could be a possible path of further work.

Our neural network seemed to perform approximately equally well as using the neural network class from the ScikitLearn-library, which indicates that we implemented our neural networks correctly. Our neural networks performed better than the logistic regression, which is to be expected as discussed earlier; FFNNs have greater flexibility in approximating functions than what logstic regression can perform. We do note however, that the low accuracy score using logistic regression did not match the scores we got during preliminary testing. The low score also did not align with the cost we found using scikit-learn's logistic regression implementation. Randomness in stochastic gradient descent, along with the particular train-test split could possibly explain the discrepancy. However we are not sure if this is sufficient to explain the poor performance. In any case, the neural network outperformed even our preliminary results and scikit-learn's logistic regression. So

it appears that a properly-tuned neural network provides better predictions on this particular data set.

4.4 Neural Network Architecture and Activation Functions

Throughout this report, we have repeatedly seen leaky ReLU perform on par or better than the other hidden layer activation functions. We also note that the function is computationally inexpensive to compute, which serves as another benefit. We therefore suggest leaky ReLU as a good alternative for FFNN hidden activation functions. However, other options not explored in this report could also be worth investigating

Regarding network architecture, we have explored a quite limited range of architectures. In both our tests a network consisting of two hidden layers with 10 nodes each performed quite well. We therefore suggest this architecture for similar problems to the ones explored in this report. We do note, however, that we were quite limited in the range of possible architectures we could test. It could therefore be interesting to explore FFNN performance as a function of layer count, node count per layer, and exploring networks with a different number of nodes in each layer. We suggest this as useful future work.

5 Conclusion

In this report, we have explored the effects of variables such as batch size, learning rate, regularization, momentum, and learning rate scaling on the convergence of gradient descent. Overall, we find that smaller batch-sizes lead to noisier, albeit quicker convergence towards a minimum. We see that higher learning rates lead to quicker convergence, though too high of a learning rate can lead to divergence. We also see that lower learning rates lead to better approximations of the minimum, provided enough epochs. We see that regularization is less important in low complexity models, such as linear or logistic regression with two features. We also see that the lower complexity neural networks do not benefit as much from regularization as the higher complexity neural networks. High momentum generally appears to lead to improved convergence, with some exceptions. We also see that each of AdaGrad, RMSProp, and Adam outperform each other in certain experiments, though the differences are quite small. We make no general recommendation, as the best performer appears to depend on the particular data set.

Using neural networks to approximate the Franke function, we found that the leaky ReLU activation function appears to perform quite well, along with the regular ReLU function. We later make the same observation for the classification problem. Therefore we suggest leaky ReLU as a good option for hidden layer activation function. Further in our analysis of the Franke function, we find that a neural network consisting of 2 hidden layers with 10 nodes outperforms the other architectures tested. We get a reasonably good approximation of the Franke function, showcasing the flexibility of feedforward neural networks as a tool for approximating functions.

For classification problems, we first observed that logistic regression is limited in the types of decision boundaries it can establish. In particular, logistic regression can only establish hyperplane decision boundaries in the parameter space. Meanwhile, feedforward neural networks can create arbitrary decision boundaries, making them more suited for certain types of classification tasks. In

our analysis of the Wisconsin breast cancer data, we got an accuracy score of 0.9298 using logistic regression with regularization $\lambda = 10^{-5}$ and constant learning rate $\eta = 0.278$. Using neural networks, we got an average accuracy score of 0.9789 using leaky ReLU as hidden activation with a relatively small network consisting of 2 layers of 10 nodes. All in all, the feedforward neural network performs slightly better on this task.

References

1 Wolberg, W. H., Street, W. N. & Mangasarian, O. L. (1995). Breast Cancer Wisconsin (Diagnostic) Data Set. `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29`. Accessed: 04.11.2022.

A Code

The code used to compute estimates and generate plots can be found at:
`https://github.com/carlfre/FYS-STK3155-project-2`