FYS-STK4155 - Project 2

Helene Wold & Tiril A. Gjerstad

Autumn 2021

Abstract

In this paper, we are analyzing three different and how they perform using different data sets. At first, we are considering the Franke Function, comparing the Stochastic Gradient Descent with a Neural Network. Next, a classifying problem with a breast cancer data set is applied to the Neural Network and Logistic Regression, comparing the performances of these models. For our Franke data, we found the Neural Network gave the best results, with an $MSE = 0.186$. The breast cancer data was best classified using the Logistic Regression method, with an accuracy of 0.98. This was slightly better than the Neural Network.

Contents

1 Introduction

Neural Networks and Logistic Regression are two useful techniques often used in Machine Learning for classification of data. Both algorithms are based on the Stochastic Gradient Descent (SGD), which is an iterative method used to optimize a cost function.
Throughout this project, we are first implementing the SGD, before implementing a more complex Feed Forward Neural Network (FFNN) and a Back Propagation Neural Network. These methods are applied using our Franke data set, and calculates the performance of the methods before comparing them.
We are analyzing how different choices of activation functions affects the Neural Network. The opportunity of varying different activation functions and cost functions makes it easier to adapt our network to classification problems as well, i.e. a breast cancer data set which will be applied in this paper. We are then comparing our Neural Network's accuracy with the Logistic Regression for the breast cancer classification case. Logistic Regression is based on the idea of SGD as well as the Neural Network methods, meaning we can reuse many methods.
All algorithms are dependent on several parameters, which will be analysed and compared. Throughout this analysis, the best choices of the parameters λ, learning rate η, minibatch size, amount of epochs will be investigated, as well as the preferred activation functions and cost function. This will also be affected by the amount of hidden layers and nodes at each layer.
The Neural Network is build using layers with corresponding weights. Through training our network, the weights is updated and adapted. Both our Neural Network and the Logistic Regression is adapted efficiently during training using minibatches and epochs, such as in SGD. Our goal is to find the best combinations of parameters for our methods, before analysing and comparing them to each others.

2 Theory

2.1 Regression models

2.1.1 Linear regression

Linear regression models is used to predict an outcome $\tilde{y}$ of y by using a linear model.

The following equation is used in the calculation of the linear regression

$$\tilde{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{1}$$

Two common linear regression models are the Ordinary Least Squares method and the Ridge Regression method. [4]

2.1.2 Logistic regression

Logistic regression is a linear regression model with the binary outputs 1 and 0 (true and false) to classify whether or not an input variable belongs to a chosen category. This decision is made from calculating the probability of the input being in one category, and then returning either 1 or 0 for this input. The logistic regression method avoids exploding or vanishing gradients that may come with deep neural networks, which is beneficial to avoid. The logistic regression does, just as the linear regression method, compute the coefficients β. However, the logistic regression method computes these coefficients by using the activation function *Sigmoid function*, which will be introduced later.

2.2 Gradient descent

In general, the gradient of any function describes how much the function changes with the respective variable. When a function consists of several variables, the gradient is calculated as a partial derivative. The gradient descent is a way of optimizing our algorithm, mainly used to minimize our cost function.

2.2.1 Steepest gradient descent

A function $F(x)$ has the steepest descent in the direction of the negative gradient $-\nabla F(x)$, also shown as

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k) \tag{2}$$

Here, γ_k is the learning rate, larger than 0. As long as $F(x_{x+1}) \leq F(x_k)$, we are moving towards a minimum, shown through smaller function values. The sequence is ideally converging towards a global minimum in F, but there is no guarantee not to get stuck in a local minimum in F. [10]

The gradient descent is set by several parameters, such as the learning rate η, the hyperparameter γ, and the parameter λ if we want the Ridge version of gradient descent. The choices of these parameters affects how well the gradient descent works, but they are however rather sensitive towards other parameters such as batch sizes and amount of epochs, which will be introduced later. I.e., too big of a learning rate may lead to big steps, which all together steps over

the minimum we want to locate, while too small of a learning rate may make the learning go too slow, resulting in no minimum found.

In other words, the gradient descent is sensitive towards the initial conditions and parameters. It is as well expensive on bigger data sets, and a dynamic learning rate may give better results than a fixed learning rate. A dynamic learning rate may adapt more to the landscape of the data set, taking bigger steps when the landscape is rather non variant, and smaller when the values are steeply changing. A random initialization with bigger steps at first may also be a way of adapting a dynamic learning rate.

While the gradient descent is simple to understand, and has an easy implementation, it does have some limitations. As mentioned, it might find and get stuck in a local minimum of our function, it is sensitive towards our parameters, and is expensive on larger data sets. It does as well treat all directions in the parameter space uniformly, meaning the learning rate is the same in all directions. This can lead to slow training of our model. It is more practical to take large steps on flat surfaces, and smaller steps when the surface is steep.

2.2.2 Stochastic Gradient Descent

The Stochastic Gradient Descent (SGD) is a method avoiding some of the imperfections of the normal Gradient Descent (GD). We want to minimize our cost function, which can be written as a sum over n data points:

$$C(\beta) = \sum_{i=1}^{n} \nabla c_i(\mathbf{x}_i, \beta) \tag{3}$$

where the gradients can be computed as a sum over i-gradients.

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta) \tag{4}$$

only taking the gradient of a subset of the data, called minibatches. We therefore randomly select minibatches from the data set in each iteration, before computing the gradient for this batch only. This is a less expensive operation than the GD, because the GD method calculates the gradient of the entire data set for every epoch.

To let our SGD be more dynamic than our GD when coming upon a local minimum, we may add some momentum to our computation of the parameter θ, creating what's called a momentum-based SGD. This will in turn act as a memory of direction when moving in parameter space.

$$\begin{aligned} \mathbf{v_t} &= \gamma \mathbf{v_{t-1}} + \eta_t \nabla_\theta E(\theta_\mathbf{t}) \\ \theta_\mathbf{t+1} &= \theta_\mathbf{t} - \mathbf{v_t} \end{aligned} \tag{5}$$

Here we let the momentum parameter γ be $0 \leq \gamma \leq 1$. This increases the speed of the GD algorithm when coming upon flat areas in the data, and decreases when the areas are steeper.

2.3 Minibatches and epochs

It is rather normal to calculate the gradient decent with minibatches and epochs, as this is a more efficient way to train our model. It also reduces the chance of overfit, as it does not overfit towards random noise.

When training our SGD model for different data sets, we split the data into so-called minibatches, and calculates the SGD for a random minibatch for an amount of epochs. The minibatches are random training batches with a chosen data size. Epochs is the amount of complete runs through m randomly picked minibatches. For each epoch, the training data set is split into m minibatches, before it is drawn m random minibatches to calculate and update the SGD with.

The relationship between the size of the minibatches and epochs are rather important. When using smaller batches, it is a need for a higher amount of epochs. This is to be able to run through the entire data set. If the batch and epoch size is too small, the algorithm might not be able to randomly run through every minibatch enough times, and therefore not be able to fit properly to our data. When increasing batch size, the amount of epochs can decrease.

When picking the proper amount of epochs, the batch size has to be accounted for. It is also necessary to keep the calculating time in mind when choosing epoch size.

2.4 Neural network

First and foremost, an artificial neural network is put together by several (or few) layers built up by nodes, also called neurons. One node takes at least one input value, calculates an activation of the node, and its weight and bias. The activation of the node is represented through

$$a = \sigma(z), z = w_1 x_1 + w_2 x_2 + ... + w_n x_n + b \tag{6}$$

The activation may be done using different types of activation functions, which will be presented shortly. The output of the Neural Network is the predicted values of the data. Neural Networks learns to perform tasks through the process of taking examples, without any specific rule in general. [6]

An artificial node is modelled through

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{7}$$

where y of a node is the value of its activation function. The activation function takes a sum of signals as input. [6]

The Neural Network takes in an arbitrary amount of layers, consisting of different amounts of nodes. The input and output layer does not have to have an equal amount of nodes, and the weights helps recreate the connection between layers.

There exists different types of Artificial Neural Networks, whereas we are implementing the Feed Forward Neural Network (FFNN) and Back Propagation Neural Network.

2.4.1 Feed Forward Neural Network

Throughout the Feed Forward Neural Network (FFNN), the information goes as expected from the name: *forward* throughout the layers of the network. In Figure 1, the nodes are represented using circles, and the direction and connections, and therefore weights, using arrows.

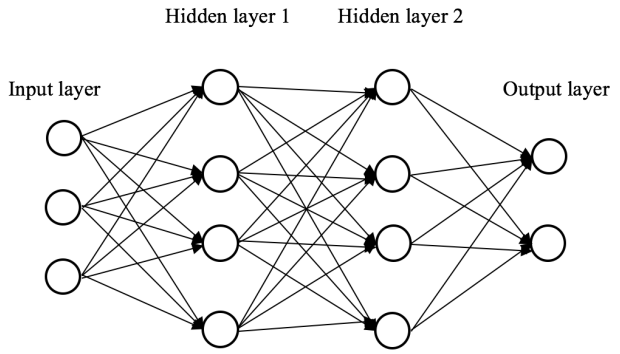

Figure 1: Illustration of a feed forward neural network with two hidden layers. Image found here: [1].

The general idea of a FFNN is that the input of our layer is the output from the previous layer.

2.4.2 Back propagation process

The back propagation algorithm is used to update the weights and biases of the neural network, and to minimize our chosen cost function. The algorithm works by first computing the weights and biases using one forward pass of the network, before updating them through a backwards pass of the network. The algorithm [7] looks something like this:

Step 1: The Input. The input layer X is sent through the feed forward algorithm.

Step 2: Output error. The output error is computed using the equation

$$\delta_j^L = f'(z_j^L)\frac{\delta C}{\delta(a_j^L)} \tag{8}$$

Here, our $f'(z_j^L)$ is the output from our derived activation function when calculated with the predicted z. The other term, $\frac{\delta C}{\delta(a_j^L)}$, is the derivative of the cost function.

Step 3: Performing the backward propagation. For $l = L-1, L-2, ..., 2$, the error is backpropagated with the equation

$$\delta_j^L = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{9}$$

Step 4: Update the weights and biases. Using gradient descent for $l = L-1, L-2, ..., 2$, the weights and biases are updated with

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta\delta_j^l a_k^{l-1} \tag{10}$$

$$b_j^l \leftarrow= w_j^l - \eta\delta_j^l \tag{11}$$

η is the learning parameter introduced in Section 2.2.1.

2.5 Activation functions

2.5.1 Identity

The identity matrix is an activation function used for linear models, which returns an output identical to the input of the function. This is to keep the input and output of the network proportional. It is defined as

$$\sigma(z) = z \tag{12}$$

with the derivative

$$\sigma'(z) = 1 \tag{13}$$

This does however not work with back propagation, and collapses all layers into one. [3] When applying the Identity activation as an output activation function, we keep our data proportional to the input of the network.

2.5.2 Sigmoid

The Sigmoid Activation function is defined as

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{14}$$

with the derivative [3]

$$\sigma'(z) = z(1-z) \tag{15}$$

The Sigmoidal Unit helps by saturating to a high value at very positive z-values, and a rather small value at negative z-values. Sigmoid is sensitive when z is close to 0. [5]

The Sigmoid activation function returns values between 0 and 1, which makes it useful when predicting probabilities in binary classification.

2.5.3 Rectified Linear Unit - ReLU

The ReLU activation function is a variation of the Sigmoid activation function, given by

$$\sigma(z) = max(0, z) \tag{16}$$

with the derivative [3]

$$\sigma'(z) = \begin{cases} 1 & z > 0, \\ 0 & z < 0 \\ undefined & z = 0. \end{cases} \tag{17}$$

The output of the ReLU is then in the range $0 \leq \sigma(z) \leq \infty$, avoiding saturation except with negative weights. This makes the gradient vanish, and stops the nodes from learning. The ReLU may in general be used when hidden layers are implemented, as it is rather quick to calculate.

2.5.4 Leaky ReLU

By multiplying in a variable α to the input z, we can now find a leaky ReLU defined as

$$\sigma(z) = max(\alpha z, z), \alpha \in \langle 0, 1 \rangle \tag{18}$$

with the derivative [3]

$$\sigma'(z) = \begin{cases} \alpha & z < 0, \\ 1 & z \geq 0 \end{cases} \tag{19}$$

We will use $\alpha = 0.01$.

This equation prevents the ReLU from saturating when the input is 0, making the nodes learn separated from the values of the weights.

2.6 Initializing the weights

To initialize the weights, it is easiest to use a random initialization with a mean zero and standard deviation of 1. This is however not an optimal solution, and may easily lead to the explosion or vanishing of the weights.

The explosion of weights means the weights grows too big, resulting in an unstable network as the gradients grow very large. These large gradients will give large updates of the weights, giving the unstable network. [2]

When weights vanishes, they shrink to the point that the neural network stops learning, and effectively dies. This is because the weights and bias will not be updated efficiently enough. A way to avoid this is to use other activation functions than the Sigmoid Activation, such as the ReLU (Section 2.5.3) or Leaky ReLU (Section 2.5.4).

Without our activation function, our system will be linear, as the multiplication and addition of weights and biases are linear operations. With our activation functions, the neural network may process more complex data, meaning non-linear data, as the activation functions are non-linear.

The most used activation functions used today is the Sigmoid, ReLU, leaky ReLU and tanh (Section 2.5). In this project, the first 3 activation functions are to be used in the different situations. Sigmoid and ReLU is the most sensitive to random initialization of the weights, meaning we should initialize them in a bit different way to avoid unnecessary problems.

2.6.1 The Xavier initialization

The Xavier initialization is defined as

$$weights \cdot \sqrt{\frac{1}{n}} \tag{20}$$

Where weights are the weights first randomly initialized, and n is the amount of input nodes for the layer. This initialization decreases the variance between the weight gradients from the first to the last layers of the network when using hyperbolic activation functions. This is a good choice for the Sigmoid activation function. [9]

2.6.2 The HE initialization

When combined with the ReLU and leaky ReLU activation, the HE initialization is the most efficient to reduce the variance of the gradients to about 1. The HE initialization is defined as

$$weights \cdot \sqrt{\frac{2}{n}} \tag{21}$$

Where weights are the weights first randomly initialized, and n is the amount of input nodes for the layer. [9]

2.7 Cost functions

A cost function is used to measure how well the neural network is learning, by measuring the error between the predicted and expected values from the network. The Mean Squared error is used for linear regression methods in this project, and for the logistic regression case, cross-entropy is applied.

2.7.1 Mean Squared Error

When using the Franke data throughout this project, the Mean Squared Error (MSE) is our main cost function. The MSE calculates how well our network are able to learn by first training our network, then testing it with some spared values.

The mean squared error is defined as

$$MSE\left(\hat{y},\hat{\tilde{y}}\right) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 \tag{22}$$

where $\hat{y}$ is the predicted data and y is the real data. The smaller the value, the better the fit.

The smallest MSE of our training data might however not be the best fit for the testing data. This can be explained by overfitting, meaning that the model has adapted to the random noise in the training data, which does not fit as well on the test data. Since the data is split into training and testing parts, the data points in the testing isn't necessarily similar to the training. [4]

For the back propagation algorithm, we are interested in the derivative of the MSE, which is:

$$MSE'\left(\hat{y},\hat{\tilde{y}}\right) = \frac{2}{n}\left(y_i - \tilde{y}_i\right) \tag{23}$$

2.7.2 Cross-entropy

The Cross-Entropy loss function is well suited for classification cases, as it returns a probability value of either 0 and 1, also denoted as False and True. As opposed to the MSE which is calculated over a set of data points, the cross-entropy looks at probabilities. [8] The Cross-Entropy is defined to be

$$C = -\frac{1}{n}\sum_{x}\left[yln(a) + (1-y)ln(1-a)\right] \tag{24}$$

This can also be denoted as

$$C(\hat{w}) = -log(P(D|\hat{w})) \tag{25}$$

D is here our data set, and w is out weights that is to be optimized.

2.8 Accuracy

When calculating the performance of our neural network in the classification cases, we are using the accuracy function. This function calculates the accuracy after the classification has been done by the neural network. It is defined as [7]

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(\tilde{y}_i = y_i)}{n} \tag{26}$$

3 Methods

The methods we use are all to be found in our ***GitHub repository***, found at this link: `https://github.uio.no/helenewo/Project2`

3.1 Briefly explained

The two data sets used is Franke data and the breast cancer data set. We have discussed the Franke data in Project 1. [4]

We now use the results from Project 1, and the Ordinary Least Squares method (OLS) to compare how well other machine learning techniques does on the same data set, i.e. the Franke Function. We know through OLS the mathematical best score possible, and we want to test how close we can get the same result with our own implementations of Stochastic Gradient Descent (SGD), neural network and logistic regression.

We have implemented a neural network as described in Section 2.4. In addition to testing the neural network on the Franke Function, we also use the neural network to classify breast cancer data. The breast cancer data set contains 30 measured features on the patients, in addition to the diagnosis of breast tissues. The diagnosis is denoted as M for malignant, and B for benign.

For the breast cancer data set, we do not have any OLS score as a reference. We do however have a goal of getting the accuracy score as close to 1.0 as we can, meaning all samples were classified correctly.

To test our network, we compare our results with the results we get using keras. For SGD and logistic regression we use sklearn to compare results.

3.2 Prepping the data

For the OLS case, we use the same methods as in Project 1 to create the data, the design matrix, and to do the scaling. However, we do not create the design matrix for the neural network similarly as for the linear regression case. Here we simply want to use the x and y values as input to the network, meaning we send in the data itself to the network. The data is split into test and training data, and scaled, such as in Project 1. [4]

When comparing different methods, we want to avoid to initialize and split the data completely random, as one random split may be more convenient and give better results than an other random split. To preserve the same random data between the methods, numpy's random seed is used. This method is used to pick out random data, but as long as the same seed is called, the same data is being randomly picked.

3.3 Stochastic Gradient Descent

In Project 1, we applied OLS and Ridge Regression to the Franke data. We will for this project use a gradient descent method, namely Stochastic Gradient Descent (SGD, Section 2.2.2) to replicate the results from Project 1 and compare.

We will begin with the gradient descent method, which is essential in neural networks to find the optimal weights and biases. The Stochastic Gradient Descent is implemented with momentum, and the SGD algorithm is run through using epochs and mini batches as mentioned in Section 2.3.

Through analysis, we study the dependency between the parameters to find the most optimal match of parameters. The parameters in mind is the momentum γ, the amount of epochs, the size of the minibatches, the learning rate η, and λ for the Ridge Regression. To measure the performance, we will try to minimize the Mean Squared Error.

3.4 Neural network

Measuring performance. To measure the performance of our neural network, we will use the input of the Franke data, as we have studied a lot, a regression problem, and also the breast cancer data, which is a classification problem. We will keep using the MSE to measure the performance of the neural network applied with the Franke data. However, we will use accuracy score for classification problems.

Cost function. MSE can be used as the cost function in our network. The accuracy score is however not a cost function. When we are looking at a classification case, the cross entropy cost function is preferred, as it returns probabilities of each case, see Section 2.7.2.

Activation function. When using our neural network with hidden layers, the activation functions Sigmoid, ReLU and leaky ReLU are applied on the input layer. For the output layer, we have to choose our activation function with the input data in mind. For our regression case, we simply use the identity function to return the correct output values. In our classification case, where we will predict a class, we use the Sigmoid activation function, and predicts the probability for the patient to be in a specific class. For our SGD case, we have several parameters which will affect the performance of the neural network.

In addition to the choice of cost and activation functions, we will look at the choice of the learning rate η, number of hidden layers, number of nodes in each hidden layer, the size of minibatches and number of epochs.

3.5 Logistic Regression

We have implemented the logistic regression as a method in the Neural Network class. The regression method applies a backwards propagation through no hidden layers. As for the neural network with the breast cancer data, we will use accuracy to measure performance, with Sigmoid as the activation function and cross entropy as cost function. Since there are no hidden layers, we only apply the output activation function.

Again, we will test several values for the input parameters, such as the learning rate γ, number of epochs and the size of the minibatches.

4 Results & Discussion

The figures we use and discuss are all to be found in our ***GitHub repository***, found at this link: `https://github.uio.no/helenewo/Project2`

Not every figure is shown in this report, we only include the figures we find most important.

4.1 Stochastic Gradient Descent

For our Stochastic Gradient Descent Method (SGD), we are implementing momentum SGD, a more dynamic SGD method. Our data is for now the Franke function with 400 data points, using a polynomial degree of $p = 5$.

This part has demanded many types of tests, as there are many parameters and combinations of parameters that may have a great effect on the performance of our model. We have chosen a more systematic testing method, as opposed to a random grid search. The random grid search is more used when the analysis is depending on many parameters.

The advantage of systematic testing, is the control we have of what is tested against each others. This method also looks at the bigger picture, and helps us avoid choosing a non ideal combination of parameters. In some cases, a certain combination of parameters may give the best result. If this result is not a part of the general trend of the model, the good result may be just a coincidence. Through systematic testing, we can choose to avoid this.

The disadvantage is however how the testing is a bit limited, making it hard to take all parameter combinations into account at the same time.

By looking at the amount of epochs vs. the size of the minibatches, we found through testing that a small batch size with 8 samples per batch gave rather good results with 1250 or 1500 epochs. Our choice of batch size and amount of epochs then temporarily settles on batch size 8 and 1500 epochs. This is to reduce the computational cost. These values has been decided using OLS, i.e. $\lambda = 0$, $\eta = 0.01$ and $\gamma = 0.6$. These values has been decided through many tests back and forth between parameter combinations.

4.1.1 η versus minibatches

Figure 2 shows that η is very sensitive with the size of the minibatches, as too small mini batches will give an overflow.

If the combination of η and our other parameters is unfortunate, it may easily give explosively high MSE values or overflow. We therefore have to be careful with our choice of η, as our algorithm is rather sensitive towards this parameter. A bigger batch size is as seen in Figure 2 more robust, but it is a small trade-off with the MSE.

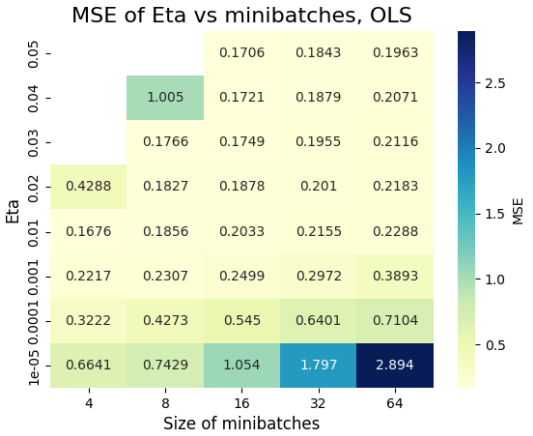

Figure 2: How η depend on the size of the minibatches.

We noticed that the lowest MSE values is for batch sizes of $[4, 8, 16]$, with $\eta = [0.01, 0.02, 0.03]$. To be able to use the colormap in the most efficient way, we excluded other values which returned too high MSE values.

In our further analysis we have chosen to use batch size $b = 8$, and $\eta = 0.01$, as this fits with our choice of batch size from our batch size vs. epoch discussion.

4.1.2 Epochs

In most cases the results gets better as we increase the number of epochs, until a minimum is reached. This is however as long the learning rate η isn't too big, which leads to unstable results. A too small η will need more iterations to stabilize, which is unfortunate. With an appropriate η, the MSE or accuracy becomes better until stabilization. Our epoch has to fit this η so that we reach our minimum, and manage to stabilize here. A too high epoch number is not bad, but the results will after a while saturate while the algorithm is still running, meaning it is a waste of computation.

We found trough our analysis the algorithm improves in very small steps after about 1500 epochs, and we will use this in our analysis to get the best results while our algorithm is rather efficient.

4.1.3 η versus λ

Figure 3 shows the relationship between η and λ when using Ridge Regression. It is noticeable that smaller λ values are preferred, which shows that the OLS implementation may be better than a Ridge regression, since $\lambda = 0$ returns the OLS. This is a result of the data set not being complex enough for Ridge Regression when using a complexity of $p = 5$ and only 400 data points. This gives no improvement when comparing the Ridge Regression to our OLS results.

Another reason for the lack of improvement from

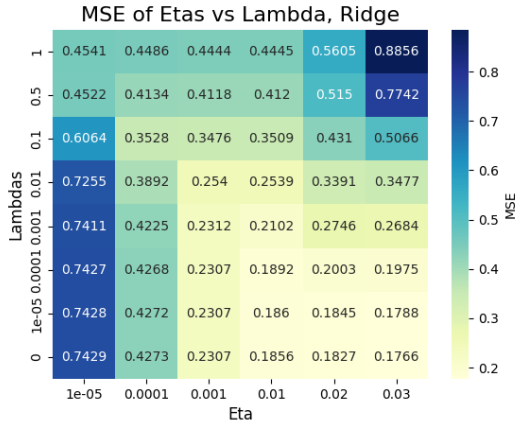

Figure 3: How η and λ depend on each other

the OLS to Ridge, is the use of minibatches. This method prevents overfitting, which is an important quality of the λ parameter in Ridge Regression. The minibatches adds a small portion of random noise, leading to a more general model.

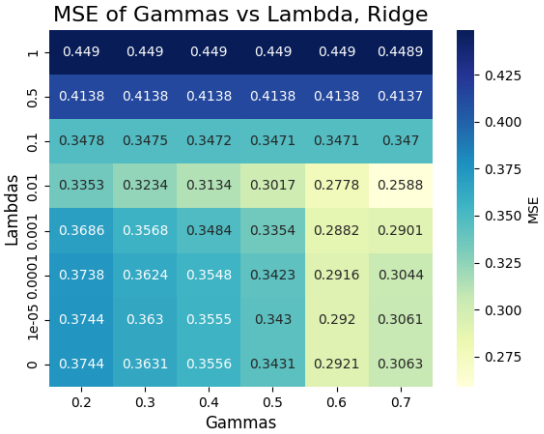

Figure 4: Dependency of γ and λ using a dynamic η in Ridge regression.

It should be beneficial to use a dynamic learning rate η, as the implementation should return a bigger value when we are moving along a flat surface, and a smaller value when coming upon a steeper area. By using dynamic η, it is possible to combine the bigger and smaller values to avoid loosing information and skipping minimums. As seen in Figure 4, our implementation of the dynamic learning rate is not the best, as it returns a significantly worse MSE than for a fixed η. We will therefore use our fixed $\eta = 0.01$.

Our dynamic learning rate depends on the inputs t_0 and t_1, which set the starting value of η. The results may improve by choosing practical starting value, which an excluded analysis shows. A starting value of $t_0 = t_1 = 20$ is our chosen values for Figure 4.

4.1.4 η versus γ at different λ

Figure 5: γ vs. η using four different λ. Top left: $\lambda = 1e-5$, Top right: $\lambda = 1e-4$, Bottom left: $\lambda = 1e-3$, Bottom right: $\lambda = 1e-2$. Bigger plot is found here

When looking at Figure 5, it is noticeable that a smaller λ is preferred, as the plot with $\lambda = 1e-5$ gives the smallest values throughout the heatmap. It is however shown through Figure 5 that $\eta = 0.01$ and $\gamma = 0.6$ is one of the better combinations of parameters.

4.1.5 In general

Through analysis, we have now found 1500 epochs with a minibatch size of $b = 8$, combined with a small or non-existing λ, $\gamma = 0.6$ and $\eta = 0.01$, returns the best Mean Squared Error of $MSE = 0.186$.

When using a dynamic learning rate we updated λ, as a smaller value gave overflow. The MSE value increased to $MSE = 0.278$. As mentioned, the implementation of dynamic learning rate is not successful, and we wish to use a set learning rate of $\eta = 0.01$ from now on.

If the amount of epochs are increased when using the dynamic learning rate, we get a better result. By doubling the epochs, the MSE value decreases slightly to $MSE = 0.272$. This shows us that the dynamic learning rate hasn't necessarily stabilized the model in a global or local minimum, but the method does however move rather slowly. We will therefore not investigate further to find the proper amount of epochs, as this is a time-costly operation.

For a set learning rate of $\eta = 0.01$, increasing the amount of epochs does not improve our MSE value, as it give the value $MSE = 0.192$.

4.2 Neural network on Franke data

Now, we wish to look at how the different activation functions presented in Section 2.5 behave with different parameters when using our Franke function as the

input to our Neural Network. Through analysis we will attempt finding the optimal parameters for our network, so it adapts the best to our Franke data.

4.2.1 Varying hidden layers

To begin with, we studied the three activation functions for three different combinations of hidden layers and nodes. All runs are done with $\lambda = 0$, using 300 epochs and a batch size of 32. The choice of batch size was decided by how the graphs stabilizes, combined with how small MSE we get at the end. Through the analysis, we will study the three activation functions, and how they behave when the neural network becomes deeper.

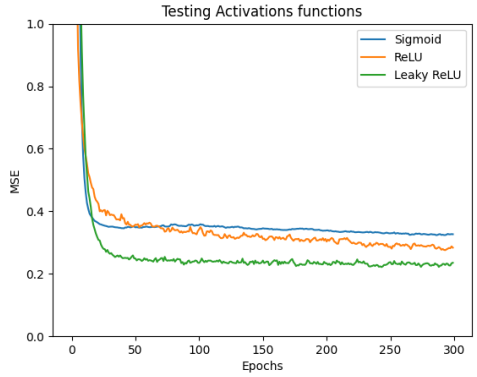

Figure 6: Test MSE of all activation functions using one hidden layer with 50 nodes, with a set $\eta = 1e-3$.

By using one hidden layer such as in Figure 6, there is no big differences between the trend of the three activation functions. All methods decrease fast before stabilizing. It is however a slight difference in MSE value, whereas the leaky ReLU stabilizing at an MSE of $MSE_{leakyReLU} \approx 0.22$. The other MSE values stabilizes to $MSE_{Sigmoid} \approx 0.34$ and $MSE_{ReLU} \approx 0.29$.

Even though the differences between the activation functions are rather small for one hidden layer using 50 nodes, leaky ReLU is the fastest learner. The leaky ReLU saturates at a small MSE value quite fast compared to the Sigmoid and ReLU, as well as being more stable than ReLU. This will be studied through some more thorough testing of different parameter combinations.

Figure 7 shows us a slightly deeper Neural Network, where the ReLU and leaky ReLU trend is rather similar as in the situation in Figure 6. The Sigmoid activation function has now a slower trend, meaning the network learns slower, and returns a higher MSE value.

Now using three hidden layers in our Neural Network, it is clear from Figure 8 that the Sigmoid activation function learns much slower from the beginning,

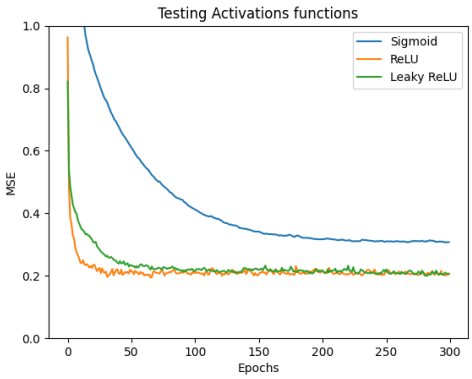

Figure 7: Test MSE using two hidden layers with 20 nodes in each layer, with a set $\eta = 1e-3$.

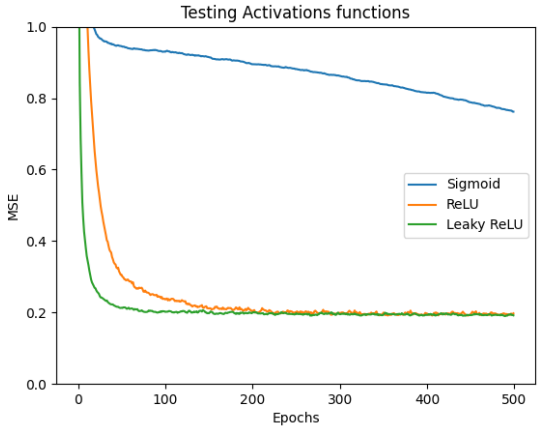

Figure 8: Test MSE using three hidden layers with $[10, 10, 10]$ nodes for the layers, with a set $\eta = 1e-3$.

and the Sigmoid is not preferred here. Both ReLU and leaky ReLU performs well, leaky ReLU being a faster learner. This is as expected, since the leaky ReLU is a more stable activation functions, as introduced in Section 2.5 and Section 2.6. The leaky ReLU is as expected the best for deeper neural networks, as it keeps the network from saturating such as with the ReLU (see Section 2.5.4).

4.2.2 Parameter combinations

Let us now take a look at the different parameter combinations for each activation function. We begin with **epochs versus batch size**, using one hidden layer with 50 nodes, and the Identity activation function as the output activation function, see Section 2.5.1.

The Sigmoid parameters in Figure 9 shows us how the MSE values varies in the range $[0.228, 0.5345]$ for our chosen combinations of epochs and batch sizes. The MSE value is lower when the batch size is smaller, and decreases when the epochs increase.

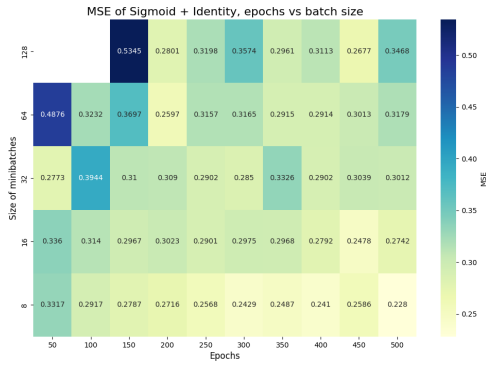

Figure 9: Epochs versus batches with the neural network using the Sigmoid activation function in hidden layers, and Identity activation function as output activation.

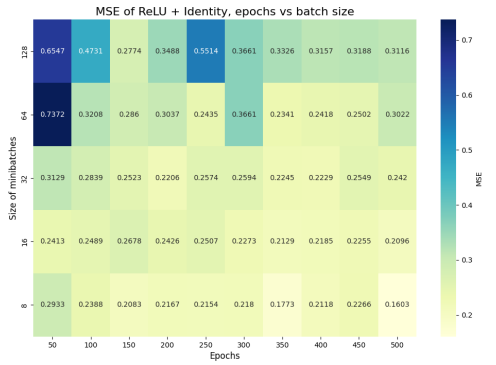

Figure 10: Epochs versus batches with the neural network using the ReLU activation in hidden layers, and Identity activation function as output activation.

When using the ReLU activation in stead of the Sigmoid activation for the same situation, we get the MSE values shown in Figure 11. For a smaller batch size of 8, the MSE shrinks. The difference between the MSE's are however a bit larger for each epoch size for the size 8 than it is for a batch size of 16.

At last we have the leaky ReLU activation function, which returns the MSE values in Figure 11. When ignoring the MSE values in the situations where the batch size is bigger than the amount of epochs, as well as for the unexpectedly higher MSE values, it is noticeable to see how the MSE of the leaky ReLU is more varying than the MSE for the ReLU and Sigmoid.

Common between the situations in Figure 9 through Figure 11, is that the smaller batch sizes is preferred, combined with 250−400 epochs, returns the best MSE values. The leaky ReLU does however have some uneven MSE values for the smaller batch sizes, as a result of a slightly unstable network as shown in Figure 6.

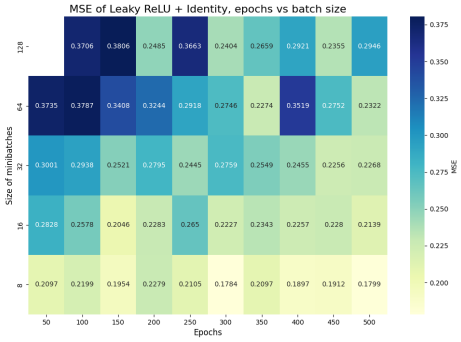

Figure 11: Epochs versus batches with the neural network using the leaky ReLU activation in hidden layers activation, and Identity activation function as output activation.

In our GitHub repository folder figs_NN_Franke, it is also three plots showing the effect of different combinations of hidden layers with different nodes, and the batch size. These three figures also show how a smaller batch size of 8 or 16 gives the best MSE values.

To avoid the possibility of an unnecessarily unstable network, we choose to use a batch size of 16 with 300 epochs, with 50 nodes in one hidden layer for our next parameter combination analysis. The next parameter combination we are looking into is the combinations of η **and** λ.

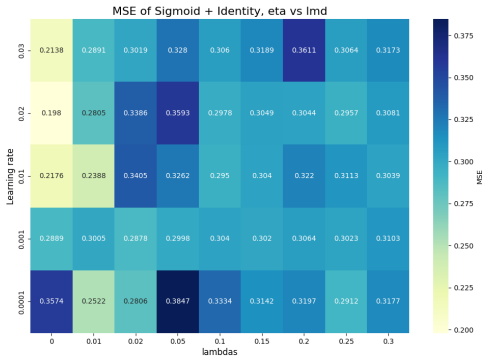

Figure 12: η versus λ with the neural network using the Sigmoid activation function in the hidden layer, and Identity activation function as output activation.

When applying the Sigmoid activation function as our activation for the hidden layer, we get the MSE results shown in Figure 12. When $\lambda = 0$, we get the definite smallest MSE values. The only exception is when the smaller learning rates, η, is applied. As shown in Figure 6, the Sigmoid learns slower than our other applied activation functions. When using Sigmoid activation function with a small learning rate,

it is necessary to increase the amount of epochs if we want better results.

In general, a small or non-existing λ combined with a higher η in the range $[0.01, 0.03]$ is preferred for the best network learning when using Sigmoid.

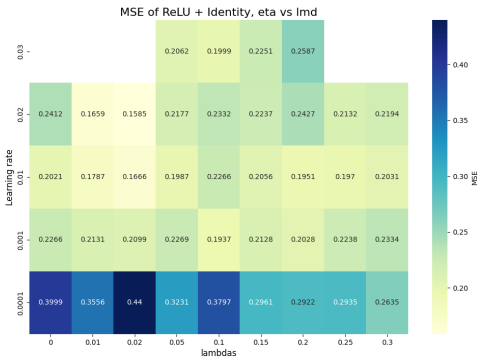

Figure 13: η versus λ with the neural network using the ReLU activation function in the hidden layer, and Identity activation function as output activation.

The MSE's of the ReLU as input activation function is generally smaller than for the Sigmoid activation function, as seen in Figure 13. The ReLU activation is however a more unstable activation function for the highest learning rate $\eta = 0.03$ when considering the MSE. This is a result of the big learning rate when using ReLU may make the network miss minimums by taking large steps on steep surfaces, increasing the MSE for this case.

A good combination is however a learning rate of $\eta = 0.01$ or $\eta = 0.02$, and $\lambda = 0.01$ and $\lambda = 0.02$. This means that the ReLU prefer a higher λ, as opposed to Sigmoid.

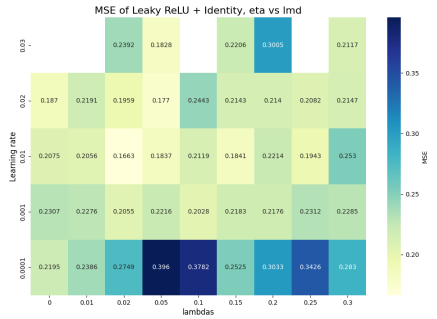

Figure 14: η versus λ with the neural network using the leaky ReLU activation function in the hidden layer, and Identity activation function as output activation.

Similar as for ReLU in Figure 13, leaky ReLU prefer combinations of values close to $\eta = 0.01$ and $\lambda = 0.02$. This specific situation gives an MSE of $MSE = 0.166$. Leaky ReLU is also, just as the ordinary ReLU, sensitive towards a high learning rate, i.e. $\eta = 0.03$.

4.2.3 In general

Figure 12 through Figure 14 shows us how the Sigmoid activation function is more robust with a higher learning rate η. This is however not the best learning rate for the Sigmoid. The MSE values for the Sigmoid is also not the best in general when we are using the Franke Function. The ReLU and leaky ReLU activation functions seems to benefit more from the use of a higher λ.

The preferred case of parameters is then 300 epochs with a batch size 16, using the ReLU activation function with $\eta = \lambda = 0.02$, as this gives $MSE = 0.1585$ in a rather stable environment.

4.3 Neural network on breast cancer data

With our Neural Network finished, we can begin applying the breast cancer data set. This data set consists of 30 features, which will give a diagnosis as mentioned in Section 3.1.

We apply the same tests as for the Franke Function, however with a new cost function. When we used the Franke Function as input data, we wanted to find how big the mean squared error was after training our neural network. Now, we are looking at a classification problem, meaning we want to use Cross-Entropy (Section 2.7.2) as our cost function, and measure the performance of our network with accuracy (Section 2.8). This is because we are now taking a classification problem into account, where we need to be able to calculate probability.

Our output activation function is now the Sigmoid activation function, as this returns a value between 0 and 1, perfect for binary classification such as the breast cancer classification.

4.3.1 Parameter combinations

We have again tested different combinations of nodes and amount of hidden layers, results are located in our GitHub figs_bc folder. From these analyses, we found no great advantage with using a higher amount of hidden layers, and choose therefore not to complicate our model excessively to avoid loosing functionality. We will then use one hidden layer with 50 nodes. η and λ is chosen trough some initial testing, set to $\eta = 0.01$, $\lambda = 0$.

Let us begin by looking at **the batch size versus epochs**.

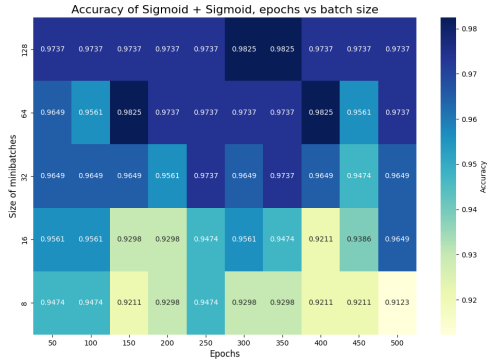

Figure 15: The batch size versus epochs with the neural network using the Sigmoid activation function in the hidden layer.

As opposed to MSE, we want values close to 1 when measuring accuracy. Figure 15 shows how a bigger batch size is the best choice relatively independent of the amount of epochs. By choosing combinations with batch size 64 or 128, and epochs between 50 and 300, we get the best results when looking at the Sigmoid function as activation function. These combinations gives a accuracy between 0.974 and 0.983.

The preference of higher batch sizes when testing for the breast cancer data set, as opposed from the Franke data, is possibly a result of the breast cancer data set being bigger, with more features. Our algorithm learns quicker, which is noticeable since our network gives high accuracy already with 50 epochs.

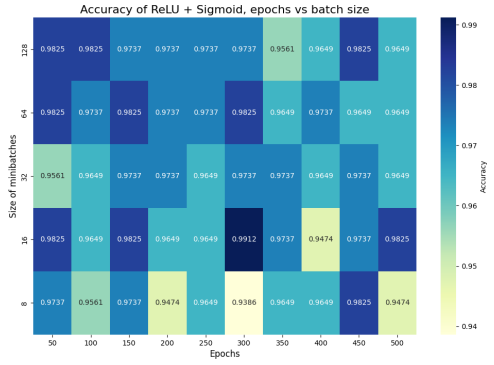

Figure 16: The batch size versus epochs with the neural network using the ReLU activation function in the hidden layer.

Moving on to the ReLU input activation function shown in Figure 16, we see a higher variation in accuracy for the different combinations. Now, our best accuracy is for batch size 16 with 300 epochs, with an accuracy of 0.991. We do assume this to be a coincidentally good value, as all accuracies surrounding this combination of epochs and batch sizes is not as good.

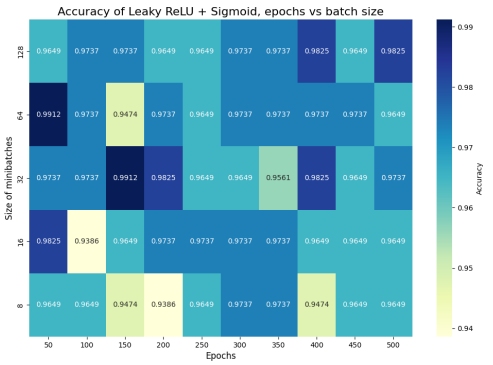

Figure 17: The batch size versus epochs with the neural network using the leaky ReLU activation function in the hidden layer.

When applying the leaky ReLU, we see through Figure 17 that the variance of the accuracy has decreased slightly. The leaky ReLU activated neural network learn fast, just like the other activations, as the accuracy is higher when the epochs are smaller. A higher batch size is however preferred, and a batch size 64 combined with 50 epochs gives an accuracy of 0.991.

Figure 15 through Figure 17 shows us how the network is not as dependent of a high epoch amount, as the network learns faster on the breast cancer data compared with the Franke data. A bigger batch size does as well give better results. The breast cancer data contains as mentioned more data than the Franke data, meaning a smaller batch size will not represent the data set as well for the new data. As an example, a batch size of 16 is a higher percentage of a data set of 400 data points, than a data set of 30×569. All activation functions will in general give high accuracy, but our preference after analyzing the epoch vs batch size is leaky ReLU as this gives less variance in accuracy and learns rather fast.

Let us however move on, and take a look at η **versus** λ when using 150 epochs with a batch size of 32. Even though 150 epochs may be a bit excessive, we choose to stay in a safe zone, avoiding the possibility of not letting our network work long enough. We keep the output activation function as Sigmoid.

As seen in Figure 18, λ does not affect the accuracy of our model as much. The learning rate η is the most influential towards the accuracy, even though this is just for the smallest learning rate tested. $\eta = 0.1$ gives the highest accuracy throughout all λ values tested, with a rather arbitrary λ.

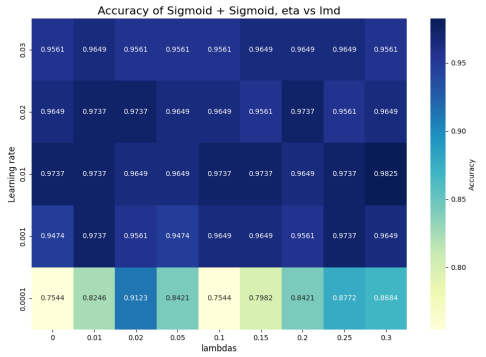

Figure 18: η versus λ with the neural network using the Sigmoid activation function in the hidden layer.

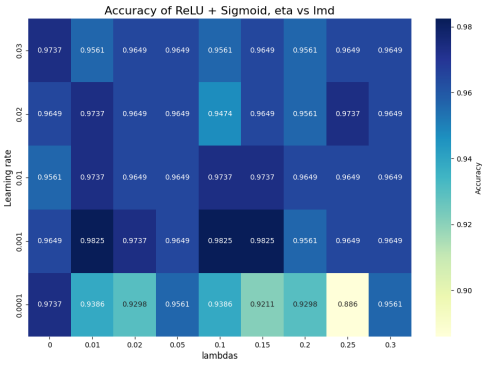

Figure 19: η versus λ with the neural network using the ReLU activation function in the hidden layer.

When looking at the ReLU input activation in Figure 19, we see again that neither of the parameters are crucial when finding the best accuracy. We want to avoid the smallest learning rate, but with $\eta = 0.001$ as a new favourite learning rate.

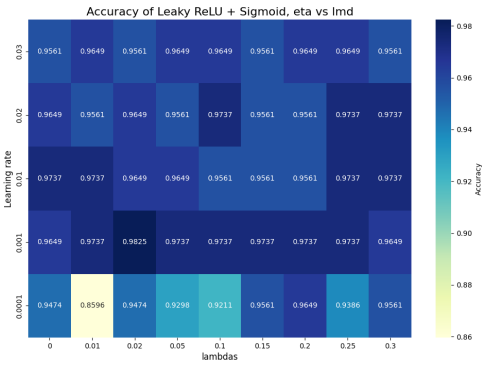

Figure 20: η versus λ with the neural network using the leaky ReLU activation function as input activation.

The leaky ReLU input activation in Figure 20 returns rather invariant accuracy throughout the parameter analysis, such as the ReLU in Figure 16. Again, avoiding the smallest learning rate is preferred, and keeping the favourite learning rate at $\eta = 0.001$.

4.3.2 In general

Throughout the testing of η versus λ, we notice how the parameter values does not greatly affect the accuracy of our neural network regardless of input activation function, but it is preferred to stay away from $\eta = 0.0001$. The accuracy also decreases slightly when going from $\eta = 0.02$ to $\eta = 0.03$, however not very noticeable. A learning rate of $\eta = 0.01$ or $\eta = 0.001$ is the best choice in general when using Sigmoid, and ReLU and leaky ReLU. The λ parameter does not seem to affect the accuracy in a notable way.

Our network is therefore rather accurate when using one hidden layer with 50 nodes, and 150 epochs with a batch size of 32, independent of the tested learning rates and λ values. The number of epochs can most likely be decreased.

4.4 Logistic regression applied on breast cancer data

Moving on to the logistic regression classification problem using the breast cancer data. When applying logistic regression, we are as mentioned in Section 3.5 applying similar methods as the neural network method. Because of this, many of the tests that has been ran on the neural network will be reused with the logistic regression.

We have excluded testing hidden layers and node values, as logistic regression does not use hidden layers. Since the hidden layers is not a part of the logistic regression, we are not using activation functions for our hidden layers. For our output activation function we use our Sigmoid activation function.

4.4.1 Parameter combinations

Let us begin with **epochs versus batch size**.

Figure 21 shows how the accuracy depends on our batch size and amount of epochs. Just as in Section 4.3.1, a bigger batch size is a better choice because of the amount of data. For the batch sizes 32, 64 and 128, the results are rather well, varying between 0.964 and 0.991. The logistic regression method has a good accuracy with 50 epochs, meaning our method learns fast, and even faster with a bigger batch size.

To keep a high accuracy when testing further, we are using 100 epochs and a batch size 64. This is to make sure our model finishes learning if i.e. our learning parameter has a great effect on the learning, making a bigger epoch necessary. We also avoid using the biggest batch size tested, as as we found through

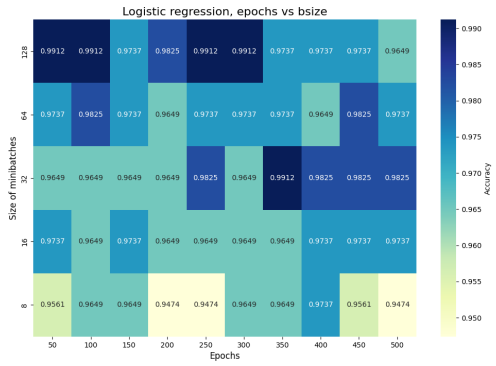

Figure 21: Accuracy of our logistic regression method when using $eta = 0.01$ and $\lambda = 0$, varying our epoch size and batch size.

our further tests that the batch size 128 gave a rather unstable accuracy. We therefore move on to η **versus** λ using the decided epoch and batch sizes.

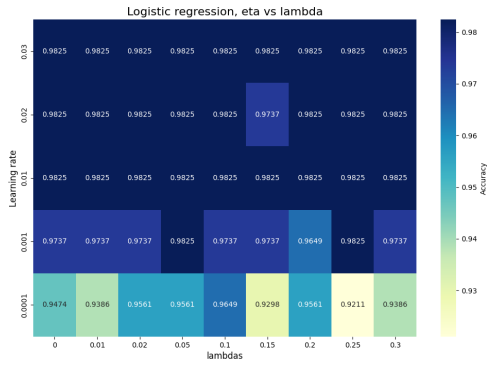

Figure 22: Accuracy of our logistic regression method when using 100 epochs and a batch size 64, varying our η and λ.

Through Figure 22 we can see how the logistic model quickly reaches a minimum of the cost function, giving us the high accuracy. The accuracy is rather stable for all $\eta's$ and $\lambda's$, and just as earlier will a small learning rate of $\eta = 0.0001$ give the lowest accuracy. Our accuracy elsewhere is for the most part 0.9825, which is a good performance. Again, λ does not contribute greatly to our accuracy.

The accuracy shown in Figure 22 shows us that our logistic regression method is performing really well on the breast cancer data set. This may however be a result of our data set not being too complicated.

4.4.2 In general

Just as when we used our neural network with the breast cancer data set, we found the parameter λ to be quite uninteresting. This is why we use $\lambda = 0$, to avoid unnecessarily complicating our model. As mentioned, the learning parameter η did not have the greatest effect on our accuracy either, as long as the smallest η tested is avoided. This gave quite good accuracy when using 100 epochs and a batch size 64.

4.5 Method comparisons

Throughout our project, we have so far analyzed how well each method performs on the Franke data and the breast cancer data set. How does the methods compare to each others for the two data sets? Let us take a look at the general results from our analysis.

4.5.1 The Franke Data

By applying the Stochastic Gradient Descent method when considering our Franke data, we found the best fit to be with the parameters $\lambda = 0$ or $\lambda = 1e-5$, $\gamma = 0.6$ and $\eta = 0.01$, using 1500 epochs and a batch size of 8. This returned a Mean Squared Error of $MSE = 0.186$ as presented in Section 4.1.5

When moving on to the Neural Network, we found the preferred parameters to be $\eta = \lambda = 0.02$ using the ReLU activation function on our hidden layers, with 300 epochs and a batch size of 16. By using 50 nodes on one hidden layer, we find the MSE value $MSE = 0.1585$.

By using the Neural Network with the optimal parameters, the model receives a better fit than using the SGD algorithm.

We will summarize our results up in a table, including the results from using keras and sklearn. We observe a very high MSE value when using Sigmoid with sklearn. This is unexpected.

	Our own	keras/sklearn
Sigmoid	0.20	0.93
ReLU	0.16	0.22
SGD	0.18	0.27

4.5.2 The breast cancer data set

When classifying our breast cancer data set, we are using the Neural Network with three different activation functions for our hidden layers, as well as the Logistic Regression method. By using 100 epochs and the batch sizes $b_{NN} = 32$ and $b_{Logistic} = 64$, with our best parameters chosen for the different environments, we get the accuracy scores presented in Figure 23.

We now see how all methods perform well with high accuracy, and all models stabilize rather quickly. It is however chosen to use 100 epochs, as this underlines how stable our models are. The best accuracy comes from our Logistic Regression, followed by the

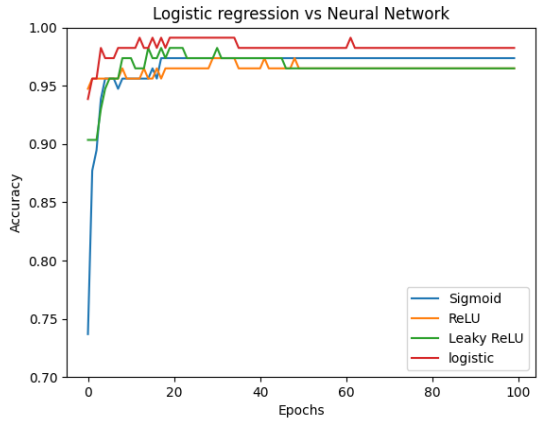

Figure 23: The accuracy of our different methods applied on the breast cancer data set.

neural network using Sigmoid as the activation function, with the ReLU and Leaky ReLU at last.

We will summarize our results up in a table, including the results from using keras and sklearn.

	Our own	keras/sklearn
Sigmoid	0.97	0.96
ReLU	0.96	0.98
Logreg	0.98	0.98

4.5.3 In general

For the Franke data, it is rather clear that the Neural Network is a better model to use when predicting the data.

When looking at our breast cancer data, the Logistic Regression method seems to be the best method. This method is a simpler method, and gives better result. There is no need to use a more complex method such as the Neural Network, when the logistic regression method performs at least as good.

However, the Neural Network applications may be a better choice to use if we want more flexibility. The logistic regression model is used for classification problems, which is why the Franke data was never applied to this method. The Neural Network gives us the opportunity to choose whether or not we want to find accuracy or mean squared error, depending on the type of data set we want to use.

The breast cancer data we have used in this project can be described using linear models, which fits well with the logistic regression model. If our data set was more complex, it would probably be a better idea to use a neural network with suitable activation functions for the complexity of the data.

5 Conclusion

5.1 Stochastic Gradient Descent values

By using SGD, we found the best parameter combinations to be $\eta = 0.01$, $\lambda = 1e-5$, $\gamma = 0.6$, epochs= 1500 and batch size= 8. The learning rate is static, since the dynamic result was proven to be quite wasteful. The best MSE then became $MSE = 0.186$

5.2 Neural Network, Franke Function

Next, the Neural Network performed the best with the parameters $\eta = \lambda = 0.02$, 300 epochs, and a batch size of 16 using the ReLU activation function on our one hidden layer with 50 nodes. This gave the best MSE of $MSE = 0.1585$.

As mentioned in Section 4.5.1, the best model for our Franke data is then the Neural Network with optimal parameters.

5.3 Neural Network, Breast Cancer Data Set

Moving on to the breast cancer data set, we found our best performance with the network with one hidden layer using 50 nodes. It was used a batch size of 32 and 150 epochs, but since the learning went rather fast, we could have reduced this. The choice of activation function did not effect the accuracy much, giving a accuracy between 0.9561 and 0.992.

5.4 Logistic Regression, Breast Cancer Data Set

Lastly, we have the logistic regression case. This method proved to give the most stable accuracy when classifying our breast cancer data.

We found the parameter λ to be quite uninteresting, and we apply $\lambda = 0$. This also goes for the learning parameter η, as long as the smallest η we tested is avoided.The accuracy is rather stable at 0.9825 when using 100 epochs and a batch size 64.

References

[1] https://www.oreilly.com/library/view/r-deep-learning/9781788478403/assets/0c814605-ee2b-49a3-9b3e-4bd88b718dc2.png. Accessed: 20-11-2021.

[2] Jason Browlee. *A Gentle Introduction to Exploding Gradients in Neural Networks*. https://machinelearningmastery.com/exploding-gradients-in-neural-networks/. Accessed: 14-11-2021. 2019.

[3] Snehal Gharat. *What, Why and Which?? Activation Functions*. https://medium.com/@snaily16/what-why-and-which-activation-functions-b2bf748c0441. Accessed: 17-11-2021. 2019.

[4] Tiril A. Gjerstad and Helene Wold. *Project 1*. 2021.

[5] Tushar Gupta. *Deep Learning: Feedforward Neural Network*. https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7. Accessed: 12-11-2021. 2017.

[6] Morten Hjorth-Jensen. *Week 40: From Stochastic Gradient Descent to Neural networks*. https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html. Accessed: 18-10-2021. 2021.

[7] Morten Hjorth-Jensen. *Week 41: Constructing a Neural Network code, Tensor flow and start Convolutional Neural Networks*. https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41.html. Accessed: 07-11-2021. 2021.

[8] Michael Nielson. *The cross-entropy cost function*. https://eng.libretexts.org/Bookshelves/Computer_Science/Applied_Programming/Book%3A_Neural_Networks_and_Deep_Learning_(Nielsen)/03%3A_Improving_the_way_neural_networks_learn/3.01%3A_The_cross-entropy_cost_function. Accessed: 13-11-2021. 2020.

[9] Christian Versloot. *He/Xavier initialization & activation functions: choose wisely*. https://www.machinecurve.com/index.php/2019/09/16/he-xavier-initialization-activation-functions-choose-wisely. Accessed: 17-11-2021. 2019.

[10] Pradyumna Yadav. *The journey of Gradient Descent — From Local to Global*. https://medium.com/analytics-vidhya/journey-of-gradient-descent-from-local-to-global-c851eba3d367. Accessed: 18-11-2021. 2021.