Classification and Regression, from linear and logistic regression to neural networks

Kamilla Julie Ida Sulebakk, Marcus Berget and Sebastian Amundsen

October 2021

ABSTRACT: In this project we have studied both classification and regression problems by developing a feed-forward neural network and a logistic regression algorithm. For the regression part of the project we studied terrain data produced by utilizing Franke's function, and the data set studied for the classification problem were the Wisconsin breast cancer data. We began the analysis by implementing the stochastic gradient descent method (SGD). This algorithm was important in order to setup both the neural network and the logistic regression method. We found that by using the momentum based version of this method we obtained the best looking fit. We implemented the momentum SGD in conjunction with both the OLS and Ridge algorithms. An observations done were that the momentum SGD was faster than the ordinary SGD, even though one performs more computations in the momentum method. We had that the time for Ridge with the ordinary SGD was at 8.91 s, while the momentum SGD used 8.45 s. Further we implemented a neural network, experimenting with three types of activation functions in the hidden layers; Sigmoid, Tanh (hyperbolic tangent), and ReLU. When studying the terrain data consisting of 100×100 data-points we found the optimal batch size, M, to be around 500. We found that when using Sigmoid and Tanh the optimal learning rate $\eta =$ 5e-4 and optimal ridge regularization parameter $\lambda =$ 1e-4. When using ReLU the corresponding values was found to be $\eta =$ 5e-4 and $\lambda =$ 1e-4. Moreover we found that using Tanh gave the best convergence-time, and that this activation function is the safest option in terms of avoiding vanishing or exploding gradients. We also found that using the recommended type of weight initialization only matter when using ReLU. Moving on to the Wisconsin Breast Cancer data, we found, by utilizing our neural network, that Tanh among the activation functions tested, gave the highest accuracy for the fit to the test data at 98.5% for two hidden layers with 20 hidden nodes each and with $M = 50$ and $\lambda = 0.001$. This result was provided when making use of the learning rate function, in which helps the model converge towards a global minimum point. In comparison the model's highest provided accuracy for the test data were 97.4% when using Sigmoid or ReLU as activation functions, with $\lambda = 0.01$. Lastly we trained our model to fit the binary cancer data using logistic regression. Even though we got an accuracy at 98.2%, in which is among the highest reached accuracies for the test data in this project, we did not see that the model provided a converging accuracy score. Thus we concluded that the neural network worked the best for the classification case.

1 Introduction

Data is more or less the lifeblood in all types of science, business and in our society in general. Among hundreds of tasks, data can help identify diseases in healthcare and medicine sciences, analyze risk and regulation in financial services and predict the terrain in the engineering industry. However, handling large and complicated data sets is no easy task, at least not without the right right tools. Machine Learning uses advanced algorithms that parse data, learns from it, and uses those learnings to discover meaningful patterns of interest. The aim of this project is to investigate various aspects of different machine learning techniques. These include polynomial regression, neural network, and logistic regression.

To study the performance of the different techniques we will use two data sets, one for regression analysis and one for classification analysis. The first data set is one replicating a terrain using the analytical function, Franke's Function. The second is the so-called Winsconsin Breast Cancer Data set of images representing various features of tumors used for determining cancer We base the code for polynomial regression of the code used when studying Regression models and resampling methods [4]. But now we minimize the gradient of the cost function using gradient descent with momentum. The structure of this code is also used for logistic regression. We will build our own feed forward neural network, with the aim of developing a better understanding of a neural network structure and algorithm.

2 Theory

Logistic Regression

In linear regression our main interest is to learn the coefficients of a polynomial to predict the response of a continuous variable y_i on unseen data based on its independent variable $\boldsymbol{x}_i$. In contrast, logistic regression commonly deals with two possible outcomes, normally denoted as a binary outcome.

We start by considering the case where the dependent variables y_i are discrete and only takes values from $k = 0, ..., K-1$. What we want to predict is the output from the design matrix $\boldsymbol{X} \in \mathbb{R}^{nxp}$ where n denotes the number of samples and p is the number of features. Our main goal is to identify the classes to which new unseen samples belong [5].

Logistic regression is the canonical example of a soft classifier. Soft classifiers explicitly estimate the class conditional probabilities and then perform classification based on estimated probabilities [6]. The probability that a data point x_i belongs to a category $y_i = \{0, 1\}$ is given by the so called logistic Sigmoid that goes as follows:

$$f(s) = \frac{1}{1+e^{-s}}, \tag{1}$$

and is supposed to represent the likelihood for a given event. Before we proceed we introduce the notation $s_i = \boldsymbol{x}_i^T \boldsymbol{w} + b_0 \equiv \boldsymbol{x}_i^T \mathbf{w}$, where $\boldsymbol{w}$ denotes the weights, b_0 is the bias, $\mathbf{x}_i = (1, \mathbf{x}_i)$ and $\mathbf{w}_i = (b_0, \boldsymbol{w}_i)$.

The cost function for the logistic regression algorithm is simply the negative log-likelihood [5] and can written as:

$$C(\boldsymbol{w}) = \sum_{i=1}^{n} -y_i \log f(\mathbf{x}_i^T \mathbf{w}) - (1-y_i) \log[1 - f(\mathbf{x}_i^T \mathbf{w})]$$

This equation is commonly referred to as the cross entropy.

Minimizing the cost function with respect to the weights w we get the following result:

$$\nabla C(\mathbf{w}) = \sum_{i=1}^{n} [f(\mathbf{x}_i^T \mathbf{w}) - y_i] \mathbf{x}_i \tag{2}$$

To measure the performance of the classification problem one commonly use the so-called accuracy score. The accuracy tells the number of correctly guessed targets t_i divided by the total number of targets, and goes as follows:

$$\text{Accuracy} = \frac{\Sigma_{i=1}^{n} I(t_i = y_i)}{n}, \tag{3}$$

where y_i is the predicted values and n is the number of targets. I is the indicator function, in which when having a binary classification problem is 1 if $t_i = y_i$ and 0 otherwise.

Gradient Descent

When training an algorithm one wants to minimize the cost function to reach its global minimum. Instead of using matrix inversion to reach this minimum we now want to use gradient descent. This is useful due to the fact that it sometimes is hard to perform matrix inversion on data sets.

Gradient Descent is an optimization algorithm for finding a local minimum of a derivable function. The underlying idea of the algorithm is that a function $F(\mathbf{x})$ decreases fastest if we take repeated steps in the opposite direction of the negative gradient $-\nabla F(\mathbf{x})$ of the function at $\mathbf{x}$. We have that:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \nabla F(\mathbf{w}_k) \tag{4}$$

which leads to $F(\mathbf{w}_{k+1}) \leq F(\mathbf{w}_k)$, when $\eta_k > 0$. This means that for sufficiently small step lengths η_0 we are always moving towards a minimum. In the Gradient Descent method we do an initial guess on weights $\mathbf{w}_0$ and compute new approximations according to equation 4.

Gradient descent then starts at a point and takes steps in the steepest downside direction until it reaches the point where the cost function is as small as possible.

It is important to note that GD is very sensitive to the chosen initial condition. In machine learning we often deal with non-convex high dimensional cost functions with many local minima. This means, that unless we have a very good initial guess, we risk getting stuck in a local minimum when the gradient converges. Moreover, the algorithm is sensitive to the step length η_0, also referred to as the learning rate. If we take too large steps, we risk stepping over the global minimum point, resulting in unpredictable behavior. The step length also needs to be large enough so we don't get "stuck" in a local minimum point. Another fact is that a small learning rate will require many iterations before we reach a minimum point, which increases CPU time.

One common problem with the GD method is that it has a tendency to overshoot the exact minimum. It has been shown that by slowly decreasing the learning rate we will obtain convergence behaviour similar to the batch gradient descent, which can help it converge towards a local or global minimum point [7]. We define a learning rate function which adjust as a function of a variable t, which is proportional to the number of epochs. The learning rate function is given by:

$$\eta(t) = \frac{t_0}{t + t_1} \tag{5}$$

Where t_0 and t_1 are constants. This equation will decrease as we run through our batches, since we are dividing by the t term. In Scikit-Learn's "SGDRegressor" we have that $t_1 = \frac{1}{0.01 * t_0}$, which we use in our own model when we compare with the method provided in Scikit-Learn.

Stochastic Gradient Descent with momentum

In almost all cases we can write our cost function as a sum over n data points:

$$C(\beta) = \Sigma_{i=1}^{n} c_i(\mathbf{x}_i, \beta). \tag{6}$$

Which means that the total gradient can be computed as a sum over i -gradients:

$$\nabla_\beta C(\beta) = \Sigma_i^n \nabla_\beta c_i(\mathbf{x}_i, \beta) \tag{7}$$

The SGD method allows us to take the gradient on randomly selected subsets of data at every step rather than the full data set. These subsets are commonly referred to as mini-batches. In other words, Stochastic Gradient Descent is just like regular Gradient Descent, except it only looks at one mini-batch for each step. The mini-batches are denoted by B_k where k runs from 1 and up to the number of batches, n/M. An iteration over the number of mini-batches is called an epoch.

Introducing randomness by only taking the gradient on a subset of the data, is beneficial as it lowers the chance of getting stuck in a local minimum point. Moreover, splitting the data points into batches reduces the time spent calculating the derivatives of the cost function, since we sum over k batches and not all n data points. Implementing momentum to the Stochastic Gradient Descent, helps accelerate gradient vectors in the right directions, which leads to faster converging. In many ways this improves the algorithms sensitivity to the learning rate. The momentum serves as a memory of the direction we are moving in parameter space and can be written as follows:

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} - \eta_t \nabla F(\boldsymbol{\beta}_t) \tag{8}$$

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t - \mathbf{v}_t \tag{9}$$

where the parameter γ represents the momentum and must be between 0 and 1. The momentum term γ is usually set to 0.9 or a similar value [7]. We used $\gamma = 0.9$. We also set the initial "velocity" to $v = 0$.

The momentum SGD algorithm

Initialize all the parameters.
Create design matrix X
Call Franke's Function with x and y. This is our z.
Guess on some β values.
for *epoch* $\in \{N_{epochs}\}$ **do**
  Shuffle the data
  for *index* $\in \{m_{minibatches}\}$ **do**
    Calculate gradients
    Find current learning rate
    $v = \gamma * v + \eta * gradients$
    $\beta = \beta - v$

For the ordinary SGD method we have one less term in the loop over minibatches: $\beta = \beta - \eta * gradients$.

Neural Networks

Artificial Neural Networks are computational systems that can learn to perform tasks by considering examples, generally without being programmed with any task-specific rules [8].

The biological neural networks of animal brains, wherein neurons interact by sending signals in the form of mathematical functions between layers, has inspired a simple model for an artificial neuron:

$$a = f\left(\Sigma_{i=1}^{n} w_i x_i + b_i\right) = f(z)$$

where the output a of the neuron is the value of its activation function f, which as input has the sum of signals $x_i, x_{i+1}, ..., x_n$ received by n other neurons, multiplied with the weights $w_i, w_{i+1}, ..., w_n$ and added with biases.

Most artificial neural networks consists of an input layer, an output layer and layers in between, called hidden layers. The layers consists of an arbitrary number of neurons, also referred to as nodes. The connection between two nodes is associated with a weight variable w, that weights the importance of various inputs. A more convenient notation for the activation function is:

$$a_i(\boldsymbol{x}) = f_i(z^{(i)}) = f_i(\boldsymbol{w^i} \cdot \boldsymbol{x} + b^i) \tag{10}$$

where $\boldsymbol{w}^{(i)} = (w_1^{(i)}, w_2^{(i)}, ..., w_n^{(i)})$ and $b^{(i)}$ are the neuron-specific weights and biases respectively. The bias is normally needed in case of zero activation weights or inputs [8].

Feed Forward Neural Network

As the name name suggests, in Feed Forward Neural Network (FFNN) the information only moves in one direction, forward through layers. This means that in an FFNN, the inputs x_i of the activation function f are the outputs of the neurons in the preceding layer.

The Universal Approximation Theorem tells us that no matter what our data set is, there is a Neural Network that can approximately approach the result and do the job. This result holds for any number of inputs and outputs [2].

Thus the basic components of a neural network are stylized neurons consisting of a linear transformation that weights the importance of various inputs, followed by a non-linear activation function. A typical example for such an activation function is the logistic Sigmoid (1).

We assume that there are L layers in our network with $l = 1, ..., L$ indexing the layer and that different layers have different activation functions. Further we denote w_{ij}^l as the weight for the connection from the j-th neuron in layer $l-1$ to the $i-th$ neuron in layer l. The bias of this neuron is written as b_i^l. The mathematical model for a FFNN is then reads:

$$a_i^l = f^l(z_i^l) = f^l\left(\Sigma_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l\right) \tag{11}$$

where l denotes the l-th layer and N_l is the number of nodes in layer l. A FFNN that is fully-connected consisting of neurons that have non-linear activation functions, receives a weighted sum of the outputs of all neurons in the previous layer (figure **??**).

We will be studying Franke's function which produces a certain type of terrain data. You can read more about Franke's function in our previous report [4].

Back propagation algorithm

The back propagation algorithm is a clever procedure that allows us to change the weights in order to minimize the cost function. At its core, back propagation is simply the ordinary chain rule for partial differentiation, and can be summarized using four equations [3].

The fist equation is the definition of the error δ_i^L of the i-th neuron in the L-th layer:

$$\delta_i^L = \frac{\partial C}{\partial(z_i^L)}, \tag{12}$$

which can be thought of as the change to the cost function by increasing z_i^L infinitesimally. By definition, the cost function measures the error of the output compared to the target data. So if the error δ_i^L is large, that would suggest the cost function hasn't yet reached its minima. The second equation is the analogously defined error of neuron i in layer l, δ_i^l:

$$\delta_i^l = f'(z_i^l)\frac{\partial C}{\partial(a_i^l)} \tag{13}$$

where $f'(z_i^l)$ measures how fast the activation function f is changing at the given activation value. Utilizing that $\delta_i^l = \frac{\partial C}{\partial z_j^l}$ we can express the error in terms of the equations for layer $l+1$. This can be done by using the chain rule, and is the third back propagation equation (for full calculations see [9] under "Final back propagation equation"):

$$\delta_i^l = \frac{\partial C}{\partial z_i^l} = \Sigma\delta_j^{l+1}w_{ij}^{l+1}f'(z_i^l) \tag{14}$$

Finally the last equation of the four back propagation equations the derivative of the cost function in terms of the weights:

$$\frac{\partial C}{\partial w_{ij}^l} = \delta_i^l a_j^{l-1} \tag{15}$$

With these four equations in hand we can now calculate the gradient of the cost function, starting from the output layer, and calculating the error of each layer backwards. We then have a way of adjusting all the weights and biases to better fit the target data. The back propagation algorithm then goes as follows:

1. **Activation at input layer:** calculate the activations a_i^1 of all the neurons in the input layer.
2. **Feed forward:** starting with the first layer, utilize the feed-forward algorithm through 11 to compute z^l and a^l for each subsequent layer.
3. **Error at top layer:** calculate the error of the top layer using equation 12. This requires to know the expression for the derivative of both the cost function $C(\boldsymbol{W}) = C(\boldsymbol{a}^L)$ and the activation function $f(z)$.
4. **"Backpropagate" the error:** use equation 14 to propagate the error backwards and calculate δ_j^l for all layers.
5. **Calculate gradient:** use equation 13 and 15 to calculate $\frac{\partial C}{\partial z_i^l}$ and $\frac{\partial C}{\partial w_{ij}^l} = \delta_i^l a_j^{l-1}$.
6. **Update weights and biases:**
   $w_{jk}^l = w_{jk}^l - \eta\delta_j^l a_k^{l-1}$
   $b_j^l = b_j^l - \eta\delta_j^l$

Structure of our neural network

The way we decided to structure our neural network is based on how to handle the design matrix containing the input data used for producing Franke's Function terrain data. We decided to take a slightly different approach to setting up the design matrix compared to in project 1 [4]. Instead of assuming a polynomial of a given complexity as we did there, we now set up the design matrix as shown below in (16). This way the columns represent all the x and y values respectively, and each row represent each its own unique combination of the input points. The design matrix then contains all the inputs used for producing the terrain, so by the universal approximation theorem there should then exist a network which should produce Franke's Function terrain. The design matrix then takes the form:

$$X = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_1 \\ . & . \\ . & . \\ x_n & y_1 \\ x_1 & y_2 \\ x_2 & y_2 \\ . & . \\ . & . \\ x_n & y_2 \\ . & . \\ . & . \\ x_1 & y_n \\ x_2 & y_n \\ . & . \\ . & . \\ x_n & y_n \end{bmatrix} \tag{16}$$

Where n is the number of points in one direction. We then define the number of input rows as $N = n \times n$. We can visualize the feed forward method like this:

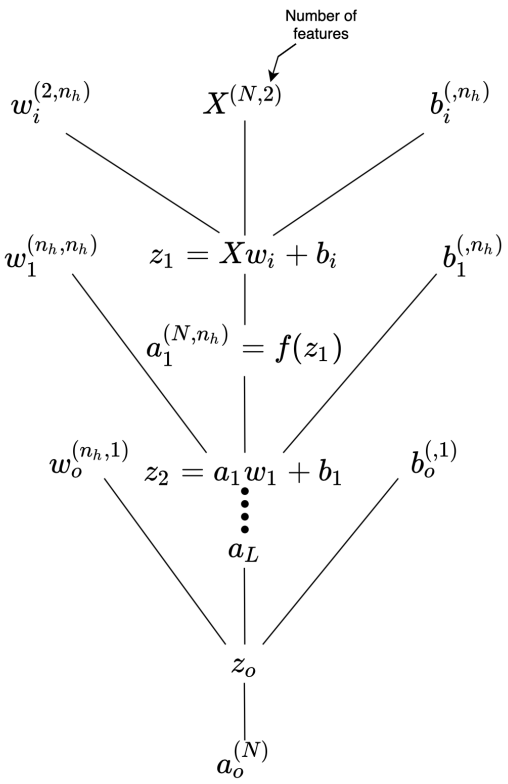

Figure 1: The feed forward method.

Where w_i is the input weights and w_o is the output weights. The numbered sub-indexes up to L denote the hidden layers. N is the input data points and n_h is the number of hidden neurons. The superscripts denote the dimensions of the matrices. Notice how we have changed the order of the input-weight matrix multiplication when calculating the z's. The reason for changing the order has to with making the dimensions add up, because when following the standard feed-forward formulas the dimensions didn't add up.

For our neural network we choose the quadratic cost function with a Ridge regularization parameter (L_2-norm), described as follows:

$$\frac{1}{2}\sum_{i=1}^{n}(a_i - t_i)^2 + \lambda||w||_2^2. \tag{17}$$

The reason for choosing the quadratic loss function, as well as the L_2-norm, is their versatility and simple derivatives. When we look at the terrain data from Franke's function it is somewhat unnecessary to us regression models and resampling methods with the Ridge regularisation parameter. The reason was discussed in our previous report [4], which was that the lack of heavy outliers.

However, when we are analyzing the breast cancer data set, we might expect more heavy outliers, which could benefit from a regularization parameter.

Activation functions

The use of activation functions is inspired by the action potential of a human brain neuron; depending on the incoming current, the neuron will either fire or not. Activation functions work in a similar way, the input to a single neuron is transformed by an activation function causing a non-linear relation between the input and output. It is in this way the network learns. There are many activation functions, but the ones we will experiment with and discuss in this project are mainly:

- The Sigmoid function: $f(z) = \frac{1}{1+e^{-z}}$
- Hyperbolic tangent: $f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- ReLU: $f(z) = \max\{0, x\}$.

Sigmoid function

A big reason for using the Sigmoid is that it's range is between $[0, 1]$. This is naturally good when predicting probabilities. Another advantage when using the Sigmoid is that it is differentiable everywhere, where its analytical expression re-uses the sigmoid: $f'(x) = f(x)(1 - f(x))$. This allows for effective computing in cases such as the back propogation algorithm which uses the derivative.

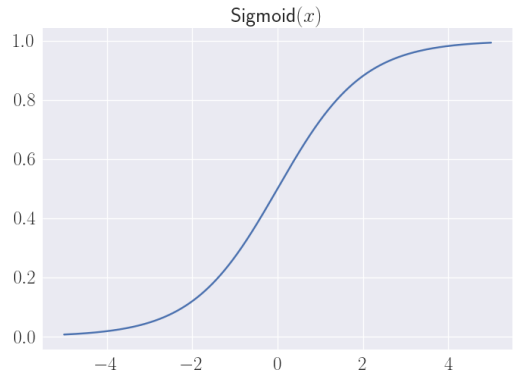

Figure 2: We see the sigmoid with its characteristic S-shape and horisontal asymptotes at 1 and 0. Note that $\text{Sigmoid}(0) = 0.5$.

Hyperbolic tangent (Tanh)

The hyperbolic tangent is similar to the Sigmoid in its shape and characteristics, but Tanh's range is instead between $[-1, 1]$ and $\text{Tanh}(0) = 0$. Similar to the Sigmoid, Tanh also reuses itself in its expression for its derivative: $f'(x) = 1 - f(x)^2$

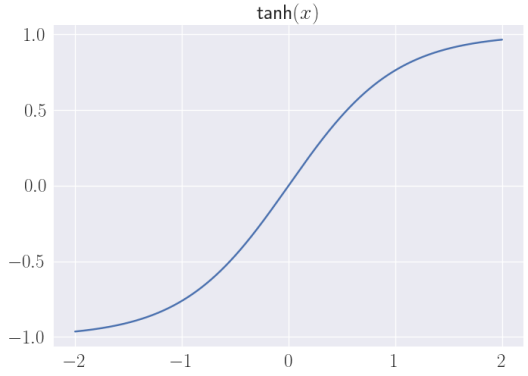

Figure 3: We see the hyperbolic tangent also with its characteristic S-shape, but with its horisontal asymptotes at -1 and 1.

ReLU

ReLU is quite simple compared to the two mentioned above, in that it returns its input value if the input is above zero, and returns zero if the input is below zero. Its simplicity is one of the main reason for its wide usage in neural networks. For input values larger than zero it's also effective in avoiding vanishing gradients which the two others mentioned above are susceptible to (This will be discussed further in the next section). Though it must be noted that also the ReLU could be sucseptible to vanishing gradients given that the derivative for negative input values are zero. One minor problem when using ReLU is that it's not differentiable at $x = 0$. This isn't necessarily a big problem as you can just define the derivative there to be zero, which is correct when x approaches zero from the left. But it is something to be wary of when using the ReLU.

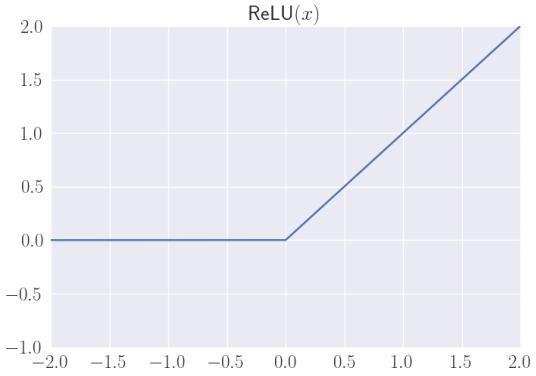

Figure 4: We see the ReLU with its non-differentiable sharp turn at $x = 0$.

Initialization of the weights and biases

The initialization of the weights and biases is seemingly arbitrary, since the network in theory should change them accordingly to fit the target data. But there are naturally some initializations that will minimize the number of epochs necessary to reach convergence, and there are some that in practice won't reach convergence at all. The optimal choices of initialization will also depend on the chosen activation function in the network. Lets use the Sigmoid as an activation function as an example. If you choose very large, or very small weights, the gradient will become very small (figure 5), causing the network to learn at an extremely slow rate. This is known as a vanishing gradient.

However, by choosing a different activation function, say the ReLU or Tanh(x), we might run into dif-

ferent problems. If we again initialize the weights to be very large, we can see that the derivative of the activation grows proportionally large (for certain areas) as the weights increase (figure 6), which in turn leads to what's normally called "exploding gradients". This problem will then only become worse as the number of layers are increased, though mainly for ReLU, since the proportionality factor from the weights carry through for each layer.

The neural network can actually also experience exploding gradients when using the Sigmoid function. But if we were to use biases equal to zero as in the example above it wouldn't happen. This is because the Sigmoid is bounded between 0 and 1, with $\text{Sig}(0) = \frac{1}{2}$ and $\text{Sig}(1) \approx 0.73$. Thus, $\text{Sigmoid}(\text{Sigmoid}(x))$ is bounded between 0.5 and 0.73. As the number of layers increase, the resulting slope will then only become flatter and flatter. However, if a bias of value $b = \frac{w}{2}$ is added to the input in each layer, the gradient will start exploding, as shown in figure 7. Though, it has to be noted that the chances of achieving such an exploding gradient when using the Sigmoid seems rather unlikely, as long as the weights and biases are not initialized in this specific way.

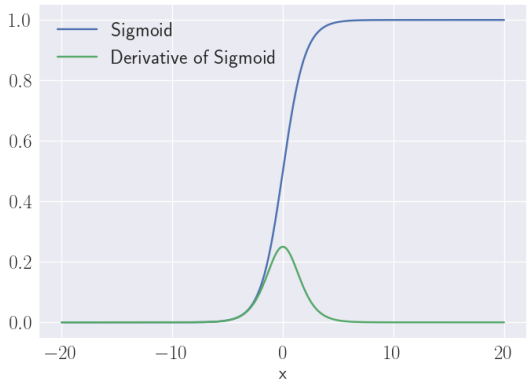

Figure 5: The Sigmoid function and its derivative. We see that for large x-values the derivative converges towards zero.

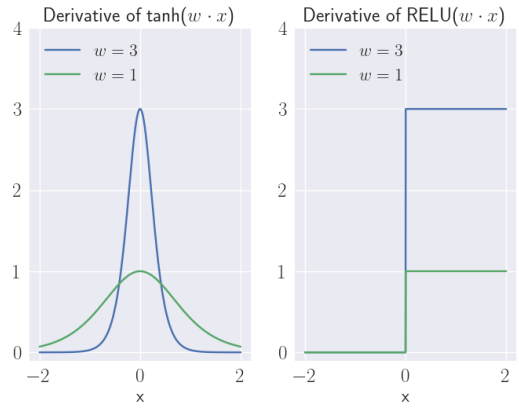

Figure 6: In the figure above we can see that the derivatives grow proportionally large as the weights increase. The difference between the two functions can be seen as the derivative of Tanh only has one spike around $x = 0$, while the derivative of the ReLU function is unbounded as $x \to \infty$

.

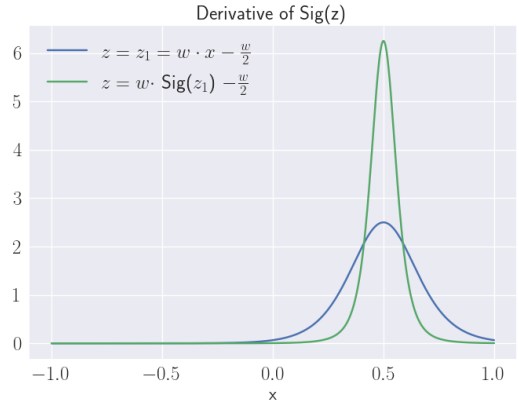

Figure 7: In the figure above we can see that when the input of the second layer is specified as $z = \text{Sig}(w \cdot x - w/2)$ the derivative starts to grow. This would continue to happen for each layer ultimately causing an exploding gradient. This is in contrast to using $b = 0$ which would cause a vanishing gradient.

The Xavier and Kaiper/He initialization methods.

From what we have discussed above, it seems that choosing reasonable initial weights and biases in combination with the right activation function is nothing other than a minefield impossible to navigate

through. Luckily for us, there are people before us which have found methods to do just this.

The first initialization method we will cover is called the Xavier weight initialization and is an approach normally used for Sigmoid as activation function. The challenge faced is to avoid shrinking the variances of the outputs through each layer. The way this is solved is to initialize the weights of each layer uniformly, and as function of the number of input nodes to that layer. The way we then initialize one layer is to draw from the uniformly distributed weights between:

$$\pm\frac{1}{\sqrt{n_l}}, \tag{18}$$

where n_l specifies the number of inputs to that layer. As Tanh is pretty similar to the Sigmoid, it's only natural that the weight initialization also is similar, which it is. For initializing the wheights when using Tanh we use a variation of the Xavier initialization called the normalized Xavier initialization. Using this the weights are initialized uniformly distributed between:

$$\pm\frac{\sqrt{6}}{\sqrt{n_l+n_{l+1}}}, \tag{19}$$

where n_{l+1} specifies the number of nodes of the outgoing layer.

The second initialization method we will cover is called the Kaiming/He initialization and is an initialization method used for the ReLU activation function. Again the challenge is to avoid shrinking variances through each layer. This method exploits that when using ReLU, the ouput of a layer has a standard deviation of approximately $\sqrt{n_l/2}$, where again n_l is the number of inputs to that layer. The idea is then to scale each layer by $\sqrt{n_l/2}$ so that each layer has an average standard deviation of 1. The weights of a given layer are then initialized from a Gaussian distribution with mean zero and standard deviation of $\sqrt{2/n_i}$.

3 Results

Comparison between SGD and momentum SGD

In this section we generate our results by using 1000 epochs, with a minibatch size of $M = 10$. We use the learning rate function (5) with $t_0 = 0.5$ and $t_1 = 1/(0.01 * t_0)$. The specific choice for the t_1 value is given by how it is defined in Scikit-Learn's SGD regressor. We perform the SGD analysis on data generated by the Franke function, with a design matrix of the size (100,21).

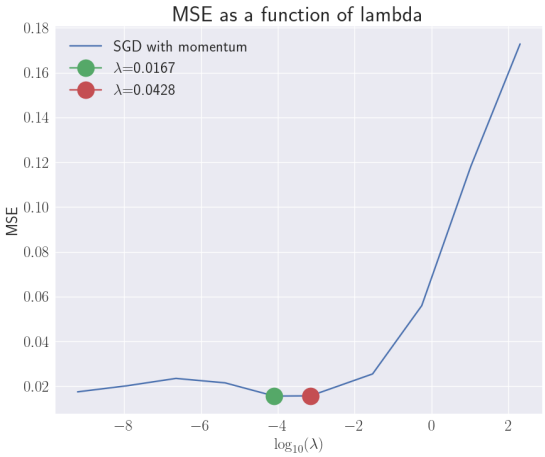

Figure 8: MSE values as a function of λ, where we have used 1000 epochs, batch size $M = 10$, $\gamma = 0.9$ and the learning rate parameters $t_0 = 0.5$ and $t_1 = \frac{1}{0.01*t_0}$.

In figure 8 we can see the MSE as a function of λ for the stochastic gradient descent with momentum. We observe that the MSE is lowest in the range from $log_{10}(\lambda) = -5$ to $log_{10}(\lambda) = -2$. The MSE is also quite low for small values of λ. The graph seems to "explode" at $log_{10}(\lambda) = -1$.

We can see the model of Franke's function using SGD with OLS in figure 9. The fit doesn't seem particularly accurate as it seems to deviate quite a lot from Franke's function data. In figure 10 we have used the SGD with momentum to model Franke's function. It seems like this gave a much better fit than when we used the ordinary Franke's function. Figure 11 presents what looks like a quite poor fit. The data does not seem to correlate with the model at all. The highest points on the model is nowhere near the actual peak in Franke's function. We observe a much better approximation with the ridge method in the momentum SGD in figure 12. This model seems to fit the data the most accurately by just looking at the figures.

From Table 1 we can see that the momentum SGD is faster than the ordinary SGD for both OLS and ridge. Moreover we can from table 2 see that the MSE from Scikit-learn's model is lower than the MSE from our own implementation by 0.046.

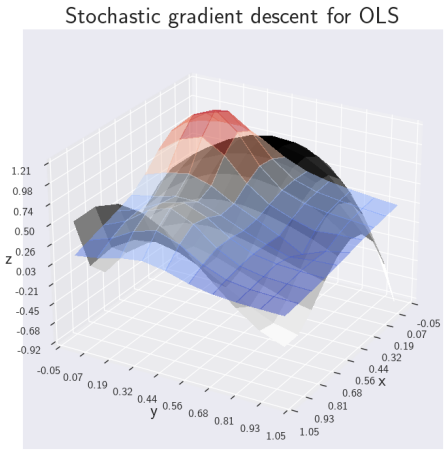

Figure 9: Model of landscape (in black) compared to the dataset from Franke's function for SGD with OLS.

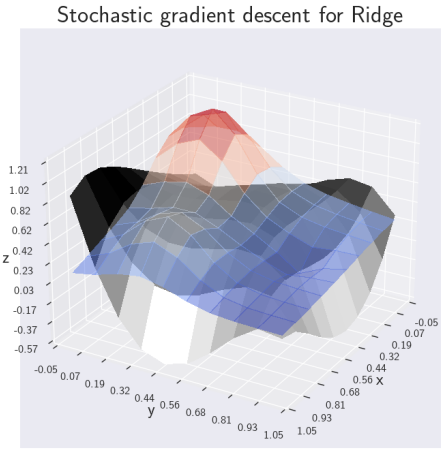

Figure 11: Model of landscape (in black) compared to the dataset from Franke's function for SGD with Ridge, where $\lambda = 4.28 \times 10^{-2}$.

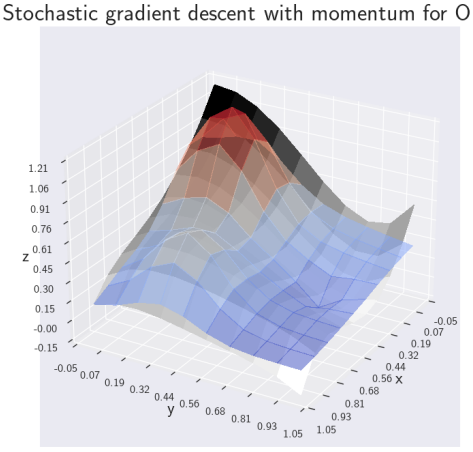

Figure 10: Model of landscape (in black) compared to the dataset from Franke's function for momentum SGD with OLS.

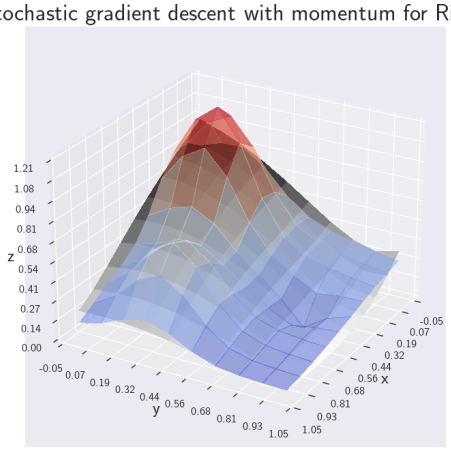

Figure 12: Model of landscape (in black) compared to the dataset from Franke's function for momentum SGD with Ridge, where $\lambda = 4.28 \times 10^{-2}$.

Table 1: Runtime for OLS and Ridge with SGD and momentum SGD.

	SGD	Momentum SGD
Time for OLS	4.76s	4.43s
Time for Ridge	8.91s	8.45s

Table 2: MSE given our own model and Scikit's model in conjunction with the OLS method.

Model for SGD	Own	Scikit
MSE	0.220	0.174

Training the neural network on terrain data using Sigmoid and varying training parameters.

The terrain data is produced by initializing the x and y-direction both in the domain $[0, 1]$ with a number of 100 points each, thus producing a 100×100 grid containing the terrain values by using Franke's Function. To emulate noise we add normal distributed values $\mathcal{N}(\mu = 0, \sigma^2 = 0.1)$ to each terrain data point. We then feed the network with the design-matrix and train using the generated terrain data. The weights are initialized using Xavier-initialization and the biases are initialized to zero.

Table 3: The table below shows the test MSE after training the network through 3000 epochs, using batch-sizes of 1000, regularization parameter $\lambda = 0.001$ and 1 hidden layer with 10 neurons. We see that $\eta = 4e-03$ results in a "Not a Number" value which was due to a vanishing gradient for the hidden bias. This problem only continued when further increasing the learning rate.

η	1e-05	1e-04	5e-04	1e-03	3e-03	4e-03
MSE	0.031	0.024	0.013	0.016	0.101	NaN

Table 4: MSE for the Sigmoid function given by the learning rate η and λ.

λ	Learning rate η	MSE
0.0001	0.0001	0.021
0.0001	0.0005	0.014
0.0001	0.00001	0.031
0.0021	0.0001	0.017
0.0021	0.0005	0.014
0.0021	0.00001	0.031
0.043	0.0001	0.026
0.043	0.0005	0.017
0.043	0.00001	0.031

In Table 4 we can see the MSE as a function of both learning rate η and the Ridge parameter λ. We can see that the lowest MSE values are given when the learning rate is equal to $\eta = 0.0005$. This is true independently of the Ridge parameters. There is only change in the MSE as we move from $\lambda = 0.0021$ to $\lambda = 0.043$, where the MSE goes up by 0.003.

Table 5: The table below shows the test MSE after training the network through 3000 epochs, setting $\lambda = 0.001$. The first five rows vary the batch size, while the remaining rows vary the layer-neuron combination.

M	Layers	Neurons	MSE	Run-time [s]
100	1	5	0.013	15.2
400	1	5	0.013	6.1
500	1	5	0.012	5.2
600	1	5	0.013	5.1
1000	1	5	0.013	4.2
500	1	5	0.012	5.4
500	1	10	0.013	7.2
500	1	20	0.013	22.0
500	2	5	0.013	7.7
500	2	10	0.013	11.2
500	2	20	0.012	30.1
500	3	5	0.013	11.6
500	3	10	0.015	16.2
500	3	20	0.016	42.1
500	6	5	0.029	22.0
500	6	10	0.095	32.2
500	6	20	0.094	80.6

From Table 5 we identify that using a batch size of 500 gives the best MSE while keeping the run-time low. From the remaining rows we can see that a combination of 1 layer and 5 hidden neurons gives both the lowest MSE score and run-time. We can also see that when using 6 layers, the MSE has risen about an eight-fold compared to the other MSE-values. By inspection we observed this was due to a vanishing gradient. In all following results we will use a batch size of 500.

Training using various activation functions

Table 6: In the table below we have trained the network until convergence using Sigmoid, Tanh, and ReLU as activation functions. The tolerance we used as a convergence-metric was MSE=0.014 (MSE of the train-data). Both when using sigmoid and Tanh we set $\lambda =$ 1e-4 and $\eta =$ 5e-4. When using ReLU we instead used $\lambda =$ 5e-4 and $\eta =$ 1e-4. The MSE in the table refers to the test-MSE. For Tanh and ReLU we checked convergence time and test-MSE for additional layers up to 6 and added the combination which yielded the best results.

$f(z_l)$	Layers	Neurons	MSE	Run-time
Sigmoid	1	10	0.016	2.0
Sigmoid	2	5	0.016	11.5
Sigmoid	3	20	0.014	30.1
Tanh	1	10	0.014	1.1
Tanh	2	10	0.014	0.8
Tanh	3	10	0.013	0.9
Tanh	4	5	0.014	0.5
ReLU	1	20	0.016	11.1
ReLU	2	20	0.014	2.4
ReLU	3	10	0.014	2.9
ReLU	5	15	0.014	0.5

Table 7: MSE and convergence-time when varying the number of layers using ReLU. The network was initialized using $\lambda =$ 5e-4, $\eta =$ 1e-4, hidden neurons=15. The tolerance we used as a convergence-metric was again MSE=0.014.

Layers	MSE	Run-time
6	0.015	1.6
7	0.016	3.8
8	0.014	3.0
9	0.016	3.8
10	0.015	2.9

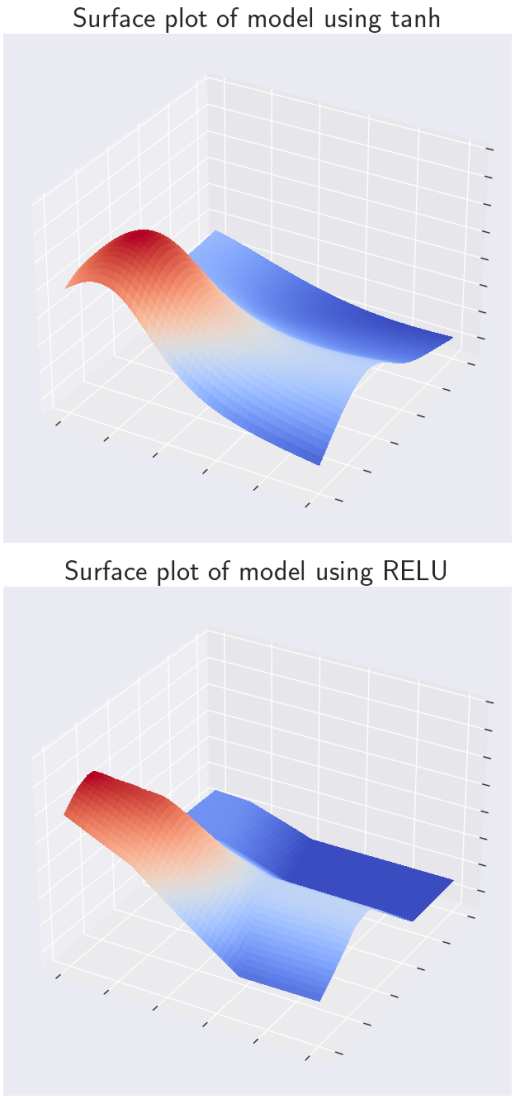

Figure 13: **Top**: Network trained using Tanh as activation function, 4 layers, 5 neurons in each layer, $\lambda =$ 1e-4, $\eta =$ 5e-4. **Bottom**: Network trained using ReLU as activation function, 5 layers, 15 neurons in each layer, $\lambda =$ 5e-4, $\eta =$ 1e-4. Both networks was trained using a batch size of 500, and stopping at a tolerance of MSE=0.013 for the training data. In both cases the total run-time for training before converging was at 0.5 s.

In table 6 we can see that increasing the layer-neuron complexity when using Sigmoid drastically increases the convergence time. This is however not the case for Tanh, where the convergence-time only varies be-

tween 0.5-1.1 s, with a slight indication of larger complexity giving the best results. For ReLU we can see a clear trend of the convergence time decreasing for increased complexity. To confirm or dis-confirm this trend we continued increasing the number of layers, as seen below in Table 7. If anything, we can from this result dis-confirm that an increased complexity necessarily gives better results when using ReLU.

In figure 13 we can see that even though both networks have converged, they produce pretty different looking terrains. We notice how the Tanh-network produce a smoothly curved terrain which strongly resembles Franke's Function terrain w/o noise. The terrain produced by the ReLU-network is on the other hand very sharp. It seems that the terrain is divided into small portions where each portion has its own gradient. We tested to see if this had something to do with the batch-size, but we saw the same tendency for both bigger and smaller batch-sizes.

Testing different initializations

Table 8: In the uppermost table below we have tested different weight initializations using the network initialization parameters shown in the bottom table.

		MSE	run-time
Sigmoid	Random	0.017	1.6
	Xavier	0.016	1.8
	Xavier normalized	0.015	1.9
	He	0.017	2.6
Tanh	Random	0.014	0.1
	Xavier	0.015	0.9
	Xavier Normalized	0.015	1.0
	He	0.015	0.6
ReLU	Random	0.027	24.0
	Xavier	0.017	19.5
	Xavier normalized	0.015	12.7
	He	0.015	2.9

	λ	η	layers	neurons
Sigmoid	1e-4	5e-4	1	5
Tanh	1e-4	5e-4	4	5
ReLU	5e-4	1e-5	5	10

In table 8 we see that the different weight initializations doesn't seem to matter too much when using Sigmoid on this data set. The same goes for using Tanh, but here we notice that initializing the weights randomly yields a convergence-time of 0.1 s! After further inspection we found that this probably was due to a lucky RNG-seed, because when changing seed the convergence-time started varying a lot. At one point the network even experienced an exploding gradient and did not converge at all. As for the ReLU, it is apparent that the best initialization here is the recommended He-initialization.

General observations

Throughout the process of writing the code and testing various initial conditions we found that the neural network is very sensitive to initial conditions. This is especially the case when using ReLU as activation function. An example of this is that when testing the layer-neuron combinations, a combination of 5 layers and 10 hidden neurons when using ReLU, suddenly resulted in a vanishing gradient.

Classification analysis using neural networks

For the classification analysis of the Wisconsin Breast Cancer data, we handled a feature matrix of 569 samples and 30 parameters. For the output layer of out neural network we sat the activation function to always be Sigmoid in order to get an output in between 0 and 1. Moreover we changed the cost function into the cross Entropy function. The Wisconsin Breast Cancer data set is a typical binary classification problem with just one single output, either 0 or 1. To measure the performance of our classification problem we used the so-called accuracy score in which is presented in the theory section.

Logistic Sigmoid Function

Table 9: The table displays the different accuracy scores for the fit with the test data set, with various numbers of hidden layers and nodes. The activation function utilized is the Sigmoid function. The results have been produced for 10000 epochs, with the batch size $M = 50$. For the learning rate function we have used $t_0 = 0.5$ and $t_1 = 100$. The parameter λ in the L2 regularization were set to 0.01 and γ used in the momentum based SGD equaled 0.8.

Accuracy	Layers	Neurons
0.974	1	10
0.974	1	20
0.974	1	30
0.965	2	10
0.974	2	20
0.974	2	30
0.974	3	10
0.974	3	20
0.974	3	30
0.965	4	10
0.965	4	20
0.974	4	30

Table 10: The table displays the different accuracy scores and costs for the fit with the test data set, for different values of the L2 regularization parameter, λ. The results have been produced with 10000 epochs, and the activation function utilized is the Sigmoid function. All other parameters are the same as in Table 9.

	$M = 10$		$M = 50$	
λ	Accuracy	Cost	Accuracy	Cost
0.20	0.632	0.658	0.632	0.654
0.15	0.632	0.658	0.632	0.593
0.10	0.632	0.623	0.974	0.303
0.05	0.974	0.216	0.965	0.118
0.01	0.965	0.115	0.974	0.129
0.00	0.965	0.134	0.974	0.160

Table 11: The table displays the different accuracy scores for the fit with the test and train data set. The results have been produced with two hidden layers, 20 hidden neurons, and a batch size of $M = 50$. The activation function utilized is the Sigmoid function. For the learning rate function we have used $t_0 = 0.5$ and $t_1 = 100$. The parameter λ in the L2 regularization were set to 0.01, and γ used in the momentum based SGD equaled 0.8.

	Test set		Train set	
Epochs	Own code	Sklearn	Own code	Sklearn
1	0.368	0.930	0.374	0.945
10	0.974	0.974	0.989	0.989
100	0.965	0.965	0.996	0.996
1000	0.965	0.965	0.996	0.996
10000	0.974	0.965	0.998	0.996
100000	0.974	0.965	1.000	0.996
1000000	0.974	0.965	1.000	0.996

Table 12: The table displays the cost for the fit with the test and train data set for different numbers of epochs. All parameters are the same as in 11.

Epochs	Test set	Train set
1	1.181	1.171
10	0.100	0.051
100	0.117	0.032
1000	0.129	0.016
10000	0.129	0.009
100000	0.125	0.007
1000000	0.115	0.008

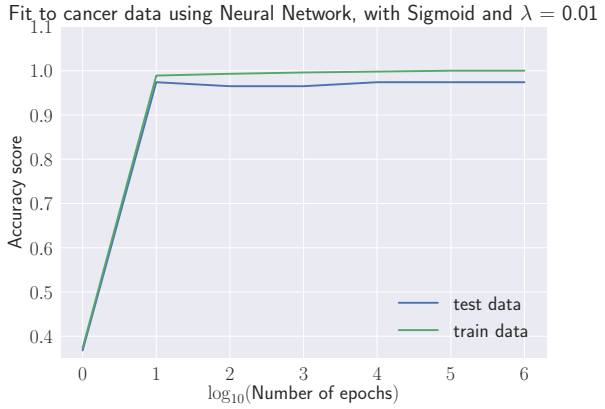

Figure 14: The figure shows the accuracy score as function of number of epochs for both the test and training breast cancer data. In the calculations we have used two hidden layers and $M = 50$. The activation function utilized is the Sigmoid function. For the learning rate function we have used $t_0 = 0.5$ and $t_1 = 100$. The parameter λ in the L2 regularization were set to 0.01, and γ used in the momentum based SGD equaled 0.8.

In table 9 we see how different numbers of hidden layers and neurons affects the models fit for the test set of the breast cancer data, when using our neural network. We see that the model not changes dramatically for different combinations of nodes and layers, when the batch size is 50. Moreover, we see that the highest accuracy reached for the different combinations is at 97.4%. With these parameters in hand table 10 presents the relation between the accuracy score, cost and the L2 regularization parameter λ, produced with two hidden layers, with 20 nodes each. For a batch size equal to 10, we see that the model provides the best accuracy when the parameter is set to 0.05. The corresponding cost is then 0.216. Looking at M = 50, we soon notice that the accuracy score is the highest when λ is set to 0.1, 0.01 and 0.00. Nevertheless, we see that the cost is smallest for λ = 0.01 at 0.129.

Figure 14 displays the accuracy score for the model as function of epochs, for test and train cancer data. In the figure we have provided the accuracy for M = 50 with λ = 0.01. It is apparent that the model reaches a high accuracy score for the test and training data after 10 epochs. We see that the accuracy score is 0.974 and 0.989 for the test and training data, respectively. From this point the accuracy for the training data increases, until it reaches 100% after 100000 epochs, and stabilizes. However, the score for the test data falls to 0.965 in between 10 and 10000 epochs, before it raises again at 100000 epochs and stabilizes at 97,4%. The precise values for the accuracy score can be seen in table 11 along with the equivalent results provided from Sklearn for both the test and training data set. From table 12 we see the cost for the fit to the cancer data.

The Rectified Linear Unit Function

Figure 15 displays two plots of the accuracy score for the model as function of epochs, for two different values of the regularization parameter λ. In the upper plot we have provided the accuracy for M = 50 with λ = 0.01. It is apparent that the model reaches a high accuracy score for the test and training data already after 1 epoch. Moreover we see that the fit to the training data, stabilizes just below a score of 1. However, the accuracy score for the test data, falls after 10000 epochs and stabilizes at 0.965. The bottom plot in the figure shows the results from the calculations when setting λ to 0.06. The highest reach accuracy for the test data is 0.982 at 1000 and 10000 epochs. The accuracy for the train data is constant at 1.0 between 10 and 10000 epochs.

The precise values for the accuracy score for λ = 0.01 can be seen in table 13 along with the equivalent results provided from Sklearn for both the test and training data set. From table 12 we can read off the corresponding costs for the fit to the test cancer data.

Table 13: The table displays the different accuracy scores for the fit with the test and train data set. The results have been produced with two hidden layers, 20 hidden neurons, and a batch size of M = 50. The activation function utilized is ReLU. For the learning rate function we have used $t_0 = 0.5$ and $t_1 = 100$. The parameter λ in the L2 regularization were set to 0.01, and γ used in the momentum based SGD equaled 0.8.

	Test set		Train set	
Epochs	Own code	Sklearn	Own code	Sklearn
1	0.974	0.974	0.976	0.954
10	0.974	0.982	0.998	0.989
100	0.974	0.974	0.998	0.996
1000	0.974	0.974	0.998	0.996
10000	0.965	0.974	0.998	0.996
100000	0.965	0.974	0.998	0.996
1000000	0.965	0.974	0.998	0.996

Table 14: The table displays the cost for the fit with the test and train data set for different numbers of epochs. All parameters are the same as in Table 13.

Epochs	Test set	Train set
1	0.135	0.067
10	0.187	0.012
100	0.252	0.003
1000	0.249	0.003
10000	0.232	0.003
100000	0.021	0.003
1000000	0.120	0.002

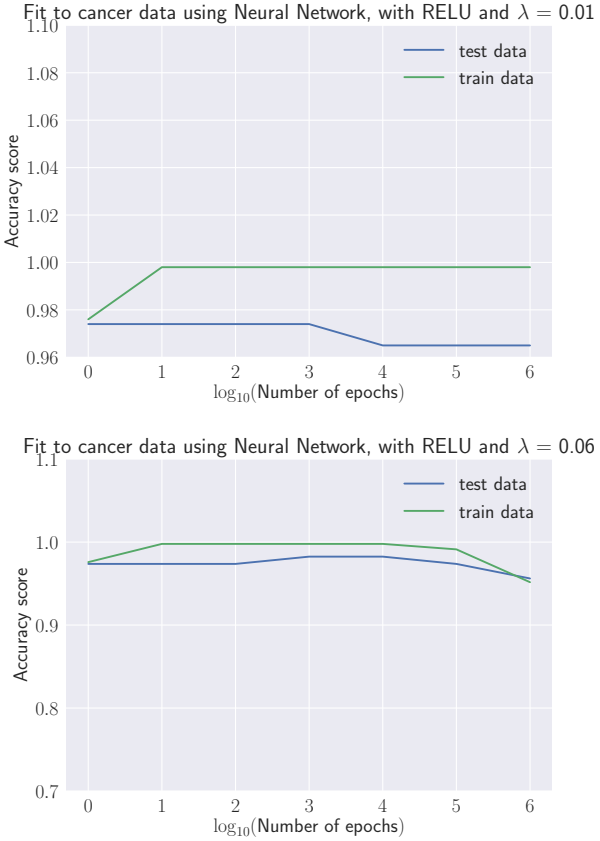

Figure 15: The figure shows two plots of the accuracy score as function of number of epochs for both the test and training breast cancer data, with two hidden layers and $M = 50$. The activation function utilized is ReLU. For the learning rate function we have used $t_0 = 0.5$ and $t_1 = 100$ in both calculations. The parameter λ in the L2 regularization were set to 0.01 and 0.06, and γ used in the momentum based SGD equaled 0.8.

Hyperbolic Tangent Function

Figure 16 displays two plots of the accuracy score for the model as function of epochs, for two different values of the regularization parameter λ. The calculations in the upper plot have been made with 50 mini batches and with λ = 0.01. For these values we see that the fit reaches a high accuracy score for the test and training data after 100 epochs. Moreover we can from table 13 see that both the accuracy for the test and train data stabilizes after 10000 epochs at 97.4% and 98.5%, respectively. The precise values for the ac-

curacy score at each tick at the x-axis in the figure can be seen in table 13 along with the equivalent results provided from Sklearn for both the test and training data set. Table 12 provides the cost for the fit to the cancer data, for λ = 0.1.

The bottom plot in figure 16 shows the results from the calculations when setting λ to 0.001. The highest reach accuracy for the test data is to be found between 1000 and 1000000 epochs at 0.982. The accuracy for the train data is constant at 1.0 between 100 and 1000000 epochs.

Table 15: The table displays the different accuracy scores for the fit with the test and train data set. The results have been produced with two hidden layers, 20 hidden neurons, and a batch size M = 50. The activation function utilized is Tanh. For the learning rate function we have used $t_0 = 0.5$ and $t_1 = 100$. The parameter λ in the L2 regularization were set to 0.01, and γ used in the momentum based SGD equaled 0.8.

	Test set		Train set	
Epochs	Own code	Sklearn	Own code	Sklearn
1	0.368	0.947	0.374	0.965
10	0.930	0.974	0.947	0.985
100	0.965	0.965	1.000	0.993
1000	0.956	0.965	1.000	0.993
10000	0.956	0.965	1.000	0.993
100000	0.965	0.965	1.000	0.993
1000000	0.965	0.965	1.000	0.993

Table 16: The table displays the cost for the fit with the test and train data set for different numbers of epochs. All parameters are the same as in 15.

Epochs	Test set	Train set
1	0.917	0.906
10	1.637	1.649
100	0.459	2e-4
1000	0.368	1e-4
10000	0.259	3e-4
100000	0.165	3e-7
1000000	0.120	0.002

Figure 16: The figure shows the two plots of the accuracy score, for two different values of λ as function of number of epochs for both the test and training breast cancer data. In the calculations we have used two hidden layers and $M = 50$. The activation function utilized is ReLU. For the learning rate function we have used $t_0 = 0.5$ and $t_1 = 100$ in both calculations. The parameter λ in the L2 regularization were set to 0.01 and 0.001, and γ used in the momentum based SGD equaled 0.8.

Logistic regression

As for the classification problem using our neural network, we used the Sigmoid activation function in order to get an output in between 0 and 1, when performing logistic regression with stochastic gradient descent with momentum. We also used the same cost function, the cross Entropy and measured the performance of our problem using the accuracy score.

Table 17 conveys the relation between the batch size, cost and the L2 regularization parameter λ. For both batch sizes, we see that the model provides alike accuracies at 96.5% for all tested values of λ. Table 18 provides the accuracy score of the fit along with the number of epochs, for $\lambda = 0.05$. We note that the accuracy score is the highest for the test set after 100000 epochs at 0.086. Table 19 provides the cost of the fit as function of epochs, for λ equal to 0.05.

Figure 17 displays the accuracy score for the model as function of epochs, for λ = 0.05. For both data sets, we clearly see that the accuracy score increases with the number of epochs. In the figure we see that the accuracy to the training data converges after 1000 epochs. However, the test data does not seem to do the same.

Table 17: The table displays the different accuracy scores and costs for the fit with the test data set, for different values of the L2 regularization parameter, λ. The results have been produced with 1000 epochs. The fit is done using logistic regression with momentum based stochastic gradient descent. For the learning rate function we have used $t_0 = 0.5$ and $t_1 = 100$. γ used in the momentum based SGD equaled 0.8.

	M = 10		M = 50	
λ	Accuracy	Cost	Accuracy	Cost
0.20	0.965	0.120	0.965	0.121
0.15	0.965	0.111	0.965	0.112
0.10	0.965	0.101	0.965	0.103
0.05	0.965	0.096	0.965	0.095
0.01	0.965	0.096	0.965	0.097
0.00	0.965	0.095	0.965	0.100

Table 18: The table displays the different accuracy scores for the fit with the test and train data set, produced with our own code and Sklearn. The fit is done using logistic regression with momentum based stochastic gradient descent. The parameter λ in the L2 regularization were set to 0.05 and the batch size M = 50. All other parameters are the same as in table 17.

	Test set		Train set	
Epochs	Own code	Sklearn	Own code	Sklearn
1	0.772	0.921	0.793	0.938
10	0.868	0.982	0.905	0.993
100	0.956	0.974	0.965	0.991
1000	0.965	0.974	0.982	0.991
10000	0.974	0.974	0.985	0.991
100000	0.982	0.974	0.985	0.991
1000000	0.974	0.974	0.985	0.991

Table 19: The table displays the cost for the fit with the test and train data set for different numbers of epochs. The fit is done using logistic regression with momentum based stochastic gradient descent. The parameter λ in the L2 regularization were set to 0.05 and the batch size M = 50. All other parameters are the same as in table 17.

Epochs	Test set	Train set
1	0.591	0.496
10	0.645	0.388
100	0.121	0.092
1000	0.095	0.078
10000	0.083	0.076
100000	0.086	0.074
1000000	0.111	0.096

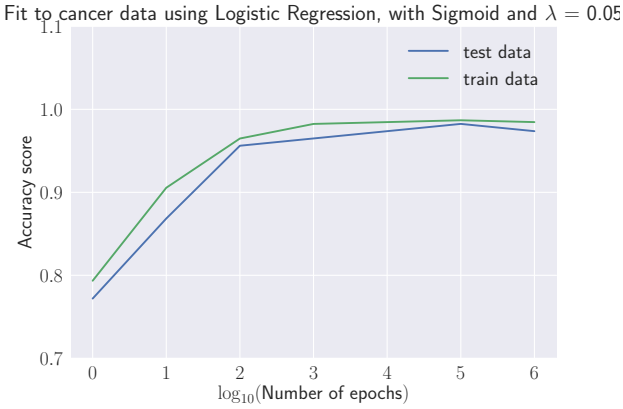

Figure 17: The figure shows the accuracy score as function of number of epochs with $M = 50$. The fit is done using logistic regression with momentum based stochastic gradient descent and the activation function utilized is the Sigmoid function. For the learning rate function we have used $t_0 = 0.5$ and $t_1 = 100$. The parameter λ in the L2 regularization were set to 0.05. γ used in the momentum based SGD equaled 0.8.

4 Discussion

Comparison between SGD and momentum SGD

We know that for the Ridge method we have an additional parameter, namely the λ value used in the cost function for Ridge [4]. In figure 8 we have the MSE values as a function of λ for the stochastic gradient descent method with momentum used in conjunction with the Ridge method. We can see that the MSE value is low for small λ values. This is to be expected as at low λ values the method will behave similarly to the OLS method, which only cares to minimize the MSE. The λ value which gives the smallest variance is plotted as the green dot in figure 8. This is not necessarily the optimal λ value to choose when fitting our ridge model.

We have previously explored which λ to choose to optimize both the bias and variance (see [4]), which was found to be $\lambda = 4.28 \times 10^{-2}$. We plotted this as the red dot in figure 8. Here we have both a low variance (MSE) and bias. This is the value we use when looking at how the Ridge model fits the data.

In figure 9 we have used stochastic gradient descent in conjunction with OLS to model Franke's function. We implemented SGD with momentum on the same regression method in figure 10, which seems to fit the data much better by just looking at the figures. This indicates that we get a better fit by adding the momentum term. The same can be said for the Ridge method in figure 11 and 12, where the latter looks far better compared to the ordinary SGD method. This could be caused by the fact that the momentum SGD accelerates the gradient vectors in the preferred direction. Since we are modelling a quite complicated function, it could be the case that the ordinary SGD does not manage to minimize the gradients enough before we have done all our iterations. This is supported by the fact that the normal SGD takes a longer time to run than our momentum SGD, which could come from the fact that our momentum method reaches the tolerance we put for gradient size before the normal SGD (norm of gradient<(tol=0.1)), while the SGD method does not reach this threshold before we have done all our iterations. We can see that the momentum SGD is faster than the normal SGD in our specific case with the tolerance we specified in table 2. We could perhaps make our normal SGD "work up to par" with the momentum SGD by increasing the number of iterations. We did not use a constant learning rate in this section, since it led to poor fits with the number of iterations we used. We therefore introduced the learning rate function given by equation 5. We also note that the momentum SGD performs more computations than the ordinary SGD method. This should also impact the run time of the two methods, which means that the time saved in minimizing the gradient is substantial for the momentum SGD, since more computations increases CPU time.

Another reason the momentum SGD fits the data better than the normal SGD could come from the fact that the ordinary SGD get stuck in a local minimum point. SGD often have trouble navigating areas where surface curves more steeply in one dimension than the other. This behaviour is often found in local optima, which we are trying to avoid. One of the properties of the momentum method is that it helps accelerate SGD in the relevant direction and dampens oscillations. In practice this may cause our momentum SGD to be able to reach the global minimum point or an acceptable halting point (local minimum point which performs nearly as well as the global one) [7].

MSE of SGD models

We also wanted to compare our own SGD method with the SGD regressor from Scikit-learn. We use the same learning rate function and number of epochs in both methods. The MSE values are given in table 2. We can see that the MSE from Scikit-learn is slightly lower than the MSE from our own model. This could imply that Scikit-learn have a better implementation of the SGD method than our own model, which means that there could be certain aspects which we could improve upon. Scikit-Learns regressor could perhaps have some additional functionalities which we are missing in our implementation. Despite of this, we think it is perfectly fine to use either one of these methods as a SGD regressor, as the difference in MSE is not that substantial.

Why do we use SGD in linear and logistic regression?

It seems that the matrix inversion algorithm which we used in our last project [4] works a lot better in modeling the landscape than both our SGD methods. Is there than any reason for choosing a gradient descent algorithm when performing linear and logistic regression?

The answer to this question depends entirely on the data set which we are studying. When we are looking at three dimensional data from Franke's function we often need to navigate trough "ravines", which the SGD method often struggle with [7]. We can see that it helps to introduce the momentum term, but it is not up to par with the matrix inversion algorithm in this specific case. There is also a few more parameters we need to keep track of in the SGD methods. Our initialization of these parameters will greatly impact how well our model fits the data. For all we know, it could be the case that we have chosen sub optimal parameters and this is the reason for our worse performance.

There is however one huge advantage point in favour of the gradient methods, namely that SGD works even when we have collinear predictors. If X is singular we cannot use the method of matrix inversion to find the solution [10], which means that the matrix inversion algorithm does not work on linear dependent columns. We will have a similar result when we have highly correlated columns. This means that the usual inverse matrix algorithm will not work when we have collinearity issues, as the path would "blow up". It is computationally difficult to invert highly correlated columns in the design matrix X and it will produce unstable regression estimates.

One important thing to note is that the SGD often saves a lot of time on calculations. In many cases it is faster to compute the solution using SGD than for example the matrix inversion algorithm. The reason being that matrix inversion is a really "expensive" calculation. In gradient descent we can also choose to only keep a piece of data in memory when computing the gradients, which lowers the requirements for the computer memory. It is also easier to allocate the calculations across multiple processors for stochastic gradient descent than for the linear algebra solution, which opens up possibilities for parallelization.

Training the neural network using Franke's function terrain data

Ridge reugularization parameter

As suggested in the results the ridge regularization parameter λ is not as important when using sigmoid and Tanh as activation functions, compared to when using ReLU. This probably has to do with the dampening nature of Sigmoid and Tanh. As discussed in the theory section, the gradients when using ReLU have a much larger probability for growing for exploding, compared to when using Tanh and sigmoid. Thus, using a stronger λ when using ReLU seems more important.

Optimal layers and neurons combinations

As mentioned in the results, when using Sigmoid as the activation function in the hidden layers, we found that of the combinations we tested for the optimal number of layers were 1, and number of neurons were 5. Initially we find it a bit strange that the lowest number of layers and neurons should give the lowest MSE. We would intuitively think that increasing the number of neurons and layers should let the network have a better capacity for learning due to the increased layer and neuron complexity, which is what apparently is the case when using ReLU as activation function. We believe that the main reason for these results is due to how the Sigmoid function is especially prone to vanishing gradients, which likely becomes a bigger problem as the number of layers increase. In

addition, since the convergence-time increases drastically it seems pointless to implement a complex network when using Sigmoid.

As for when using Tanh it is reasonable to believe that using a medium-high complexity is the way to go. Our results suggest that there is only a slight indication that the best results are obtained when using a high complexity. This trend might be due to the randomness of the initializations, and since increasing the complexity increase the probability of vanishing or exploding gradients, avoiding a too high complexity is probably a good idea. The same goes when using ReLU, but in this case it is important to use a sufficient complexity since the performance at the lowest complexity is quite poor.

Tanh-ReLU surface plot comparison

The reason for the sharp terrain when using ReLU in figure 13 is a bit mysterious. Our initial thought was that it had to do with the batch size because of the seemingly fragmented portions, but since testing with different batch sizes did not change this tendency it is most likely due to something else. We then believe it has to do with the way ReLU behaves differently for negative values compared to positive ones, without it being obvious exactly how this would cause the strange pattern.

Different weight initializations

The vanishing gradient only becomes a problem with very large or very small weights when using both Sigmoid and Tanh, so it makes sense that the various initializations (which all initializes values between 0 and 1) does not seem to affect the results of the networks. One thing to note is that Franke's Function terrain data does not have a too high variance, compared to other potential data sets. Thus the weight variances through each layer does not have an obvious reason to increase drastically, which might be the reason to why we do not experience any vanishing gradients in this case. So using the recommended initializations might be more important for different type of data-sets. ReLU on the other hand, will easier let the gradients grow large through each layer, so it makes sense that it is more sensitive to other initializations than the recommended He-initialization. It therefore makes sense that using the He-initialization is the best way to go when using ReLU.

Classification analysis using neural networks

Optimal parameters

For this part of the project we studied how the fit to the data developed for different combinations of hidden layers and nodes, number of epochs and the L2 regularization parameter λ. For the learning rate we utilized the learning schedule function given in 5.

In our calculations, we have performed the so-called trial and error method in order to find a combination of the size of the batches, number of hidden layers and neurons and the value of λ that maximized the accuracy. For logistic Sigmoid we found that the number of hidden layers and neurons, to a small extend affected the calculations. However, we continued our calculations using 2 hidden layers and 20 neurons. When setting λ, we found that this parameter predominantly affected the cost of the model. With a batch size of 50, and the regularization parameter λ = 0.01, we reached the highest accuracy and smallest cost possible for our model. However, it cannot be denied, that the searching for the different parameters not were optimal. As a matter of fact it would be surprising if our trial and error method did actually managed to find the absolute best combinations of the values of the parameters. A better option would probably have been to implement a grid search in our code. If so, this method could have substituted the learning schedule function, so that we more easily could have studied the results as function of the learning rate.

Logistic Sigmoid function and accuracy

When performing the fit of the test data set, using the Logistic Sigmoid activation function, we got the highest accuracy at 97.4%, after 10 epochs. In figure 14, it might at first sight, look like the accuracy stays constant after these 10 epochs. However, table 11 can disprove this. What we see is that the accuracy score for the test data in fact decreases to 0.965 after 100 epochs, and does not get back to the score at 0.974 before having reached 10 000 epochs. What we in fact did expect was for the accuracy to increase or stay constant as function of epochs. This seems to be the case for the training data which reaches an accuracy of 100 % after 100 000 epochs. However, comparing the results obtained for our own code with the those produced with Sklearn, we see that we get a full compliance between the fit for both the training and testing data, for 10, 100

and 10000 epochs. This of course if fulfilling. Nevertheless, this compliance ends, when increasing the number of epochs any further. This could tell us that we failed when trying to set the parameters in Sklearn equal to the ones we were using in our own code. Another explanation might be that our algorithm perhaps gives numerical errors when performing a very large number of epochs. Even so, from table 12 we see that the trend of the fit to the data is decreasing cost with increasing number of epochs. This again, could disprove numerical errors in our algorithm.

ReLU and Tanh activation functions and accuracy

When utilized the ReLU and Tanh activation functions in our calculations we used the same values for the regularization parameter, the batch size and number of hidden layers and neurons. Again, by the trial and error method, these values seemed to provide the best results for the accuracy and cost when fitting the data. However, it is to be said again, that a grid search might have given other and better values for the parameters.

Comparing the results provided when using ReLU and Tanh as activation functions in our neural network classification problem with those provided with Sigmoid function, we start by noticing that ReLU trains the model faster than Sigmoid and Tanh does. Already after 1 epoch ReLU managed to give an accuracy of 97.4% for the fit to the test data. On the other side, Tanh never reaches an accuracy higher than 96.5%. However, an odd observation, when using ReLU, is that the accuracy drops after 1000 epochs, and stabilizes on a lower accuracy score, when setting $\lambda = 0.01$. In figure 15 we see how a different value of the regularization parameter affects the fit. $\lambda = 0.06$ provides a more desired tendency when studying the accuracy as function of epochs. Nevertheless, this result is also not satisfactory, as the accuracy does not seem to stabilize.

When looking at the results provided when using Tanh as activation function, we can do some of the same observations as done when using Sigmoid. For $\lambda = 0.01$, the accuracy drops after 100 epochs, and does not get back to the same score again before having reached 100 000 epochs. Moreover, as for Sigmoid, the fit to the training data provides a constant accuracy of 100% after 100 epochs. In figure 16 we see how a different value of the regularization parameter affects the fit. $\lambda = 0.001$ provides a tendency more wished for when studying the accuracy as function of epochs. Here the highest accuracy stabilizes at 98.2% for the test data and 100% for the training data. As we see no tendency to overfitting, this result is very satisfying as it might tell us that our model is fully trained and provides a credible and stable accuracy for the test data.

To what extent the cost of the different models should be compared is not an easy question. What the cost tells us, is the predicted values deviation from the target values, but as Sigmoid in the last layer of the neural network sets the predicted value to either 0 or 1, the cost does not necessarily affect the accuracy of the model. However, what it could tell us something about is the stability of the model. A fulfilling tendency, in which we only observe when using Tanh as activation function, is that the cost decreases as function of epochs for both the test and training data, at least between 10 and 100 000 epochs. This is in alignment with our expectations, as the number of epochs can be seen in context with how well we train our model, and the cost should decrease as the model is being trained.

When comparing our own algorithms for ReLU and Tanh with the one provided from Sklearn, we only get compliance between the accuracies for some epochs, and this is only for the testing data and not the training data. Again, as for Sigmoid, this might be due to failure when we were trying to set the parameters in Sklearn equal to the ones we were using in our own code, or that our algorithm is not fully thought trough such that it gives numerical errors.

With the above discussion in hand it is hard, we see that all of our models have their shortcomings. It is therefore hard to tell which activation function that gave the ”best” model for the classification problem using neural network on the Wisconsin Breast Cancer data. However, according to the compared accuracies it might look like we got the most reliable results when using the hyperbolic tangent activation function, with $\lambda = 0.001$.

Logistic regression

Optimal parameters

When performing logistic regression with SGD, we again used the trail and error method when finding the best combinations of the L2 regularization parameter λ, batch size and number of epochs, in order

to obtain the highest possible performance. Also for this method we used the learning schedule function through out our calculations. However, it can again be discussed to which degree this technique actually gave the "right" combination, if any.

In table 18 we see how the accuracy score and cost changes with different combinations of batch size and the regularization parameter. From these results it is safe to say that neither the batch size or λ affects the accuracy. Looking at the cost we moreover see that the cost almost is the same for both batch sizes. This might be an odd result, and it would have been interesting testing for many more λs and batch sizes. However, what these results tells us is that our algorithm is quite stable regardless of values of the different parameter.

Comparison between classification using neural networks and logistic regression

As we know that the calculation when choosing a batch size of 50 takes a lower CPU time than a batch size of 10, we continued using $M = 50$ when fitting our model. Furthermore, we saw that setting λ to 0.05 gave the smallest cost provided with our model. From figure 17 we see how the accuracy develops as function of epochs. The highest accuracy is at 98.2% and is reached after 100 000 epochs. This is a higher accuracy than the one provided with our neural network, when using Sigmoid as activation function, however the performance is not necessary better. What we see it that the accuracy of the test data does not stabilize, in which is in conflict with our expectations. The cost of the fit to the test data, however, is actually smaller than the ones provided with our neural network with Sigmoid as cost function. Nevertheless, it is hard to argue for that our algorithm for Logistic regression provides an excellent model, as we do not see any sign of convergence. Another disappointing result, is that the accuracy of the fit to the training data does not reach 100%, as it did when using the neural network. Moreover, we do not see a notable consistency between the results for the accuracy provided with our own code and Sklaern. This of course, could tell us that we failed when trying to set the parameters in Sklearn equal to the ones we were using in our own code, but we do not deny that this lack of agreement might be due to systematical and numerical errors in our algorithm.

Nevertheless, if a pros of using Logistic regression were to be mentioned it would be that the computational takes a shorter time than when using Neural Network. The reason for this is that Logistic regression can be understood as s Neural Network wothout any hidden layers. It should also be said that the accuracies for the fit provided from both methods in it self is quite impressing for such a large data set, and there is no doubt that these methods plays an important role in the machine learning

To what extent the necessity of running more than a 1000 epochs for the binary problems could have been discussed. However, when running for these larger numbers of calculations we got to see when and if the different models could provide an accuracy of 100 % for the training data.

All in all, we would say that the classification model using neural network with Sigmoid as activation function provides a more trustable fit for the breast cancer data, than when utilizing logistic regression. However, if we were to have more time, it would be interesting to study multiple binary data sets, in order to see if this observation would have been repetitive.

5 Conclusion

We managed to successfully implement the stochastic gradient descent iterative method. We found that the momentum variant of the stochastic gradient descent performed better than the ordinary stochastic gradient descent. We also found that our implementation of the SGD had larger MSE than the SGD from Scikit- Learn, which could indicate that our iterative method have certain aspects which we can improve upon. These results were important in order to setup both the neural network and the logistic regression method.

When using our neural network for the terrain regression it seems that using the hyperbolic tangent as an activation function, with a medium high complexity and the hyper-parameters as specified in the results section, yields the best results. Also, it seems that the type of initialization (from the ones we tested) does not matter much when using the hyperbolic tangent. By instead using Sigmoid, it seems that the probability of the network not converging increases, due to vanishing gradient problems. Using ReLU as an activation function is not recommended due to high sensitivity to initializations and it producing

strange terrain patterns.

Moving on to the Wisconsin Breast Cancer data set with classification analysis using our neural network we observe that the different combinations of hidden layers and neurons, batch size and choice of the value of the regularization parameter, does not affect our results to a great extent. If we emphasize the score of the accuracy, it might seem like we get the best fit when using the hyperbolic tangent activation function. However, the results might have been different if we implemented a grid search in our analysis, as this would have resulted in many more trial combinations when searching for the optimal parameters.

When fitting model for the same binary breast cancer data using logistic regression with stochastic gradient descent we had some problems producing trustable results. As the accuracy score for the test data did not seem to converge, we found that even though the model provided a high accuracy score for some of the epochs tested, using the neural network when fitting the data gave more persistent and reliable results than with logistic regression.

References

[1] Hastie, T, Tibshirani, R., Friedman, J., 2016, The Elements of Statistical Learning

[2] Sahay, M.,2020, Neural Networks and the Universal Approximation Theorem.

[3] Metha, P., Wang, C, H., Day, A., Richardson, A., 2019, A high-bias, low-variance introduction to Machine Learning for physicists.

[4] Sulebakk, K., Berget, M., Amundsen, S., 2021, Regression models and resampling methods

[5] Jensen, M, H., 2021, Data Analysis and Machine Learning: Logistic Regression

[6] Liu, Y., Zhang, H, H., Wu, Yichao, 2011, Hard or Soft Classification? Large-margin Unified Machines

[7] Ruder, A., 2016, An overview of gradient descent optimization algorithms

[8] Jensen, M, H., 2021, Constructing a Neural Network code, Tensor flow and start Convolutional Neural Networks

[9] Jensen, M, H., 2021, From Stochastic Gradient Descent to Neural networks

[10] Goodfellow, I., Bengio, Y., Courville, Aaron., Deep Learning