Analysis of Classification and Regression Problems Using a Feedforward Neural Network

Achraf Atifi
Department of Geosciences, University of Oslo

https://github.com/achat97/FYS-STK4155

(Dated: November 20, 2023)

We have used plain gradient descent and stochastic gradient descent to solve a regression problem. We found that both methods could be tuned to get an equally low MSE of 0.01, but SGD with a constant learning rate of $\eta = 0.1$ was the most computationally efficient. Among the plain gradient descent, Adam was the most efficient with $\eta = 5$. A neural network was also used to solve the regression problem. We used a neural network with 1 hidden layer and 35 neurons. The sigmoid function as the activation for the hidden layer and no activation for the output layer. This was trained for 300 epochs and 80 mini-batches and achieved an error of $MSE = 0.01$ as well. However, replacing sigmoid with ReLU or Leaky ReLU would give a more efficient training.
We also used the neural network for classifying benign and malignant tumours from the Wisconsin cancer data. We used a NN with 3 layers and 20 neurons with the leaky ReLU as the hidden activation and sigmoid as the output activation to get an accuracy of 1. This was trained for 350 epochs and 80 mini-batches. Logistic regression was also used to classify, and we got an accuracy of 0.99. None of the methods mentioned above performed better with L^2-regularisation.

CONTENTS

I. INTRODUCTION

II. THEORY

In linear regression, we aim to find the optimal regression parameters that will provide us with the best fit for our target values. More specifically, we want a model

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

where $\tilde{\mathbf{y}} \in \mathbb{R}^n$ is the prediction of some target values $\mathbf{y} \in \mathbb{R}^n$. $\mathbf{X} \in \mathbb{R}^{n \times p}$ is the input data and $\boldsymbol{\beta} \in \mathbb{R}^p$ is the regression parameters. Note that the n is the number of inputs, and p is the number of features. The aim is to find the optimal parameters $\hat{\boldsymbol{\beta}}$, which minimises the error

$$\boldsymbol{\epsilon} = \mathbf{y} - \tilde{\mathbf{y}} \tag{2}$$

and there are several ways to do this. Some of the simplest ways are by implementing Ordinary Least Squares or Ridge regression. These two methods have simple analytical expressions for the optimal parameters

$$\hat{\boldsymbol{\beta}}^{OLS} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{3}$$

$$\hat{\boldsymbol{\beta}}^{Ridge} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \tag{4}$$

where λ is a tunable hyperparameter [4]. Both these analytical expressions are found by setting the derivative of their respective cost function $C(\boldsymbol{\beta})$ to zero, i.e. where we have a local minima, and solve for $\boldsymbol{\beta}$ [1][4]. To get an analytical expression, $C(\boldsymbol{\beta})/\partial\boldsymbol{\beta}$ need to be linear in the unknown $\boldsymbol{\beta}$. This is evidently the case for linear regression, hence the name. The regression function is linear in the unknown parameters.

It is not always the case that the regression function $\tilde{\mathbf{y}} = f(\mathbf{X})$ is linear in the unknown parameters. Let's assume we want to fit some input to two discrete values, 0 and 1. This is known as a binary classification problem. If we try to fit a model using linear regression, the output will be continuous, but we're just interested in two discrete values. To circumvent this, we can replace the regression function with a likelihood function $p(\mathbf{X})$ and then classify the output as either 0 or 1 in accordance with the probability. This method also needs tuning of the parameters in the likelihood function so that the prediction is as correct as possible, just like in linear regression. This is known as logistic regression, where we will return to the details later. This introduces a cost function and a derivative that is not linear in the unknown parameters. Thus, it's not possible to find an analytical expression for optimal parameters $\hat{\boldsymbol{\beta}}$, and we must resort to numerical methods [1].

A. Gradient Descent Methods

As we've seen so far, when we want to build a reliable model, there's usually a cost/loss function we want to minimise. One of the widely used techniques for minimisation is an iterative method called gradient descent, also denoted GD. This method is similar to the Newton-Raphson method, which is also an iterative method that aims to find the root of a function $f(x)$. The latter can be summarised as

$$x_{k+1} = x_k - f(x_k)/f'(x_k) \tag{5}$$

where you start with an initial guess x_0, and continue to iterate until the difference between the right and left-hand side in equation (5) is below some threshold value [1]. This can also be used for optimising our cost/loss function. If we let $\boldsymbol{g}(\boldsymbol{\beta}) = \nabla C(\boldsymbol{\beta})$, we essentially want to find the root of $\boldsymbol{g}(\boldsymbol{\beta})$. Now let $\partial \boldsymbol{g}(\boldsymbol{\beta})/\partial \boldsymbol{\beta} = \boldsymbol{H}$, which is the Hessian matrix, and let's replace the variables in equation (5) with the new corresponding variables. We then get the following optimisation method

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \boldsymbol{H}^{-1}(\boldsymbol{\beta}_k) \times \boldsymbol{g}(\boldsymbol{\beta}_k) \tag{6}$$

which is a second-order optimisation algorithm since it uses the Hessian matrix as a part of the algorithm [1][3]. This is essentially why we don't usually use the Newton-Rapshon method for optimisation. If we have a large dataset, calculating the Hessian and inverting it as well can be computationally expensive and is something we generally want to avoid [3]. Therefore, one uses gradient descent methods instead.

1. *Plain Gradient Descent*

Gradient descent methods are first-order optimisation algorithms and thus do not include the Hessian matrix in the optimisation. The simplest method is the plain gradient descent. This is essentially equal to the expression in equation (6), but now we replace the Hessian matrix with what is called a learning rate, which we're going to denote with η. The algorithm is shown for some data $\mathcal{D} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1) \dots (\boldsymbol{x}_n, \boldsymbol{y}_n)\}$, a model $\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\beta})$ and a loss function $\boldsymbol{\ell}(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\beta}), \boldsymbol{y}_i)$ which quantifies the error, in Algorithm 1 [1].

Algorithm 1: Plain Gradient Descent

Require: Learning rate η
Require: Initial parameter $\boldsymbol{\beta}$
while stopping criteria not met **do**:
  Compute gradient: $\boldsymbol{g} = \frac{1}{n}\nabla_{\boldsymbol{\beta}} \sum_{i=1}^{n} \boldsymbol{\ell}(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\beta}), \boldsymbol{y}_i)$
  Compute update: $\Delta\boldsymbol{\beta} = -\eta \boldsymbol{g}$
  Apply update: $\boldsymbol{\beta} = \boldsymbol{\beta} + \Delta\boldsymbol{\beta}$

Let's now take a step back to see what this all means and why it works. As we've mentioned several times, we want to minimise the cost function. The cost function has a local minimum where the gradient of the cost function is zero, and the local minimum is a global minimum if the cost function is convex. Now, we know that the direction of the gradient is the direction in which the cost increases the fastest. Thus, the opposite direction is the direction of the quickest decrease, which is the direction we want to go. Now, we see that we are steeping in the direction of the steepest descent, and the learning rate determines how large the steps are. This is why these methods are often dubbed as steepest descent methods [1][3].

There are some pitfalls and limitations one needs to be aware of. The learning rate needs to be tuned to provide a reliable solution. It may take an extremely long time for the algorithm to converge if it's too small. If it's too large, one may overshoot the minimum and diverge. In addition, the cost function may be rugged and complex, with several local minimums and saddle points. One may get stuck in one of these, especially if the learning rate is not tuned correctly.
Lastly, computing the gradient at each step can be computationally expensive for large datasets. Therefore, one usually uses another gradient descent method to circumvent this problem and others. This method is known as stochastic gradient descent [1].

2. *Stochastic Gradient Descent*

Stochastic gradient descent is a small but significant modification of Algorithm 1. Instead of calculating the gradient for the whole dataset, we calculate the gradient for a subset at each iteration. What we do is divide the dataset into so-called mini-batches. If we have n data points, and we want the size of each mini-batch to consist of m points, we will have n/m mini-batches. At each iteration, a mini-batch is picked at random with

equal probability, and its gradient is computed and used to step in the cost domain. This is usually done for a given number of so-called epochs, which is one cycle of n/m iterations. One can choose to stop if a stopping criterion is met, or one can finish the number of epochs that are set and store the values that trigger a criterion in the hope of a better local minimum [1]. The algorithm for SGD is shown in Algorithm 2.

Algorithm 2: Stochastic Gradient Descent

Require: Learning rate η
Require: Initial parameter $\boldsymbol{\beta}$
while stopping criteria not met **do**:
  Sample a mini-batch $\{(\boldsymbol{x}_1, \boldsymbol{y}_1) \dots (\boldsymbol{x}_m, \boldsymbol{y}_m)\}$
  Compute gradient: $\boldsymbol{g} = \frac{1}{m} \nabla_{\boldsymbol{\beta}} \sum_{i=1}^{m} \boldsymbol{\ell}(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\beta}), \boldsymbol{y}_i)$
  Compute update: $\Delta\boldsymbol{\beta} = -\eta \boldsymbol{g}$
  Apply update: $\boldsymbol{\beta} = \boldsymbol{\beta} + \Delta\boldsymbol{\beta}$

So, what makes this better than plain GD? First of all, the computational cost of the gradient is significantly lower for a small enough mini-batch since we're just calculating the gradient of a smaller subset. Secondly, the shape of the cost function will slightly change for each mini-batch since we are looking at different inputs and outputs. So, the step size in the cost domain differs for the same parameters. Thus, SGD will oscillate much more than GD on its way to the minimum. But this makes it less prone to get stuck in a not-so-desirable local minimum [1][3].

3. *Momentum*

To deal with the problem of slow convergence, one can introduce what is called momentum. The name is an analogy to the physical concept in the sense that the algorithm takes the previous step size into consideration. Like a ball rolling down a hill, its velocity at a certain point depends on its velocity at prior points. The velocity is also dependent on the surface it's rolling on, i.e. friction. Therefore, one uses a so-called friction parameter γ in the algorithm, which determines how much of the previous step size one takes into account.
We implement momentum by replacing the application of the update in Algorithm 1 and 2 by

$$\begin{aligned} \boldsymbol{\beta}_{k+1} &= \boldsymbol{\beta}_k - \eta \boldsymbol{g}(\boldsymbol{\beta}_k) + \gamma(-\eta \boldsymbol{g}(\boldsymbol{\beta}_{k-1})) \\ &= \boldsymbol{\beta}_k + \boldsymbol{v}_k \end{aligned} \tag{7}$$

where $\gamma \in [0, 1]$ is the friction parameter, as we mentioned, and we let $\boldsymbol{v}_k$ denote the update. If we set $\gamma = 0$, we return to the expression without momentum [1][3].

4. *Learning Rate Optimization*

So far, we have only considered a constant learning rate and have stressed the importance of choosing the correct learning rate in order to reach convergence within a reasonable time. This is usually just done by trial and error. However, there are moments where a constant learning rate will prove not to be ideal. As mentioned, we may deal with a non-convex landscape with several shallow and flat areas that we would like to traverse quickly. Ideally, we want to take large steps in such areas to reduce the amount of time spent in the regions that are not of interest. In addition, we want to take small steps in steep, narrow directions so that we don't overshoot a minimum of interest. A constant learning rate does not consider this and treats all directions equally. The step size is thus only modified by the local gradient and not the region's landscape. Even if we have a convex landscape, the direction of the steepest slope is not necessarily in the direction of the global minimum. This is one of the drawbacks of first-order optimisation algorithms. In second-order algorithms, information about the landscape is contained in the Hessian matrix and is thus stepping through space accordingly [1][2].

To address this issue, there are algorithms which can be used to try to mimic in some way what the Hessian matrix does by tracking the gradient and even the second moment of the gradient [1]. The methods we're going to implement are Adagrad, RMSProp and Adam.

We can start by looking at **AdaGrad**, where the name is derived from adaptive gradient. This method adapts the learning rates for each parameter of the cost function based on historical gradient information. It is implemented by replacing the update as follows:

$$\begin{aligned} \boldsymbol{r} &= \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g} \\ \Delta\boldsymbol{\beta} &= -\frac{\eta}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g} \end{aligned} \tag{8}$$

where η is a constant global learning rate, δ is a small constant usually set to 10^{-7} for numerical stability, and $\boldsymbol{r}$ is the gradient accumulation variable which is initialised to zero. This is done at each iteration. We see that the learning rates with AdaGrad become small for parameters with large derivatives and large for small derivatives. The net effect is then that we traverse gently sloped directions faster and steep directions slower [2][3]. Adagrad is great for convex functions but can offer some trouble when applied to non-convex and rugged functions. This is because one may pass several different structures on the way to a convex region, and since AdaGrad adapts the learning rates according to historical gradient information, it can lead learning rates to become very small before reaching a region of interest [3].

The other method we will use is **RMSProp**, which stands for root mean square propagation. It is a modification of AdaGrad and addresses the issue of the method in non-convex situations. It implements an exponentially weighted moving average to discard the gradient

history. Therefore, a new parameter $\rho \in (0, 1)$ is introduced, which is the decay rate. It's usual to use $\rho = 0.9$, and this is what we are going to use. The update is now calculated as follows

$$\begin{aligned} \boldsymbol{r} &= \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g} \\ \Delta\boldsymbol{\beta} &= -\frac{\eta}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g} \end{aligned} \tag{9}$$

and we see that if ρ is small, the previous gradients become insignificant rather quickly [3].

The last adaptive optimisation method we will use is **Adam**, where the name is derived from adaptive moments. This can be seen as a combination of RMSProp and momentum with some important differences. Let's first note that RMSProp approximates the uncentered second moment of the gradient by using an exponentially weighted moving average. Adam will make use of this as well and, in addition, will implement the momentum as the first moment, which will be approximated the same way. So now we need two decay rates, ρ_1 and ρ_2, which are usually set to 0.9 and 0.999 respectively, and this is what we are going to use. In addition, we let $\mathbf{s}$ be the accumulation variable for the first moment and $\mathbf{r}$ for the second, and both are initialised to zero. $\mathbf{s}$ and $\mathbf{r}$ is then updated as follows

$$\begin{aligned} \boldsymbol{s} &= \rho_1 \boldsymbol{s} + (1-\rho_1)\boldsymbol{g} \\ \boldsymbol{r} &= \rho_2 \boldsymbol{r} + (1-\rho_2)\boldsymbol{g} \odot \boldsymbol{g} \end{aligned} \tag{10}$$

but the update is not calculated quite yet, Adam includes one more step. The initialisation of $\mathbf{s}$ and $\mathbf{r}$ will bias them towards zero early in training. Therefore, Adam implements a bias correction. Let t denote the t-th iteration. The bias correction is then calculated as follows

$$\begin{aligned} \hat{\boldsymbol{s}} &= \frac{\boldsymbol{s}}{1-\rho_1^t} \\ \hat{\boldsymbol{r}} &= \frac{\boldsymbol{r}}{1-\rho_2^t} \end{aligned} \tag{11}$$

and the update can finally be computed as

$$\Delta\boldsymbol{\beta} = -\eta \frac{\hat{\boldsymbol{s}}}{\delta + \sqrt{\hat{\boldsymbol{r}}}} \tag{12}$$

where the global learning rate η is usually set to 10^{-3} [2][3].

We have now looked at three different adaptive learning rate algorithms which can perform better than a constant learning rate and require less tuning of η. All these methods can be implemented with momentum, which Adam already has included. However, some tuning of the learning rate and decay rate is needed. It is important to note that these methods will not always perform better than a constant. Sometimes, they will fail to generalise, and we can be better off without them [2].

B. Logistic Regression

We previously had a brief look at logistic regression, which started our presentation of gradient decent methods. It is closely related to linear regression in the sense that we want to find some optimal regression parameters that generalise well. But we have seen that it solves a different kind of problem, namely classification problems. The aim of solving classification problems is to predict output classes, given some input $\mathbf{x} \in \mathbb{R}^{p\times 1}$. This can be solved by finding the likelihood of each class. This is what Logistic regression does. It tries to estimate the parameters for a likelihood function that gives the best prediction [1].

The most common likelihood function too use is defined as $q : \mathbb{R} \to [0, 1]$ and is given by

$$q(x) = \frac{1}{1 + e^{-x}} \tag{13}$$

and is called the sigmoid function or logistic function [1]. Now, let's generalise the input to the case where we have an input $\mathbf{x}$ as defined above with the intercept included. Also, let the regression parameters be $\boldsymbol{\beta} \in \mathbb{R}^{p\times 1}$ as before. Now, the sigmoid function can be written as

$$q(\mathbf{x}; \boldsymbol{\beta}) = \frac{1}{1 + e^{-\mathbf{x}^T\boldsymbol{\beta}}} \tag{14}$$

for a given $\boldsymbol{\beta}$. Now, we do the same as we did for linear regression. We quantify the error with some cost function and try to find the optimal parameters that minimises the cost. If we let $q_i = q(\mathbf{x}_i; \boldsymbol{\beta})$ and let y_i denote the i-th target value, we can introduce the so-called cross-entropy, which is the cost function we're going to use. This is given by

$$C(\boldsymbol{\beta}) = -\frac{1}{n}\sum_{i=1}^{n}[y_1 \log(q_i) + (1-y_i)\log(1-q_i))] \tag{15}$$

which has no known analytical solution for the optimal parameters. The cost function is however a convex function, and gradient descent methods can therefore be used to find the global minimum [2]. After finding the optimal parameters, we can use equation (14) to make the predictions. For a binary case, this is simply

$$\tilde{y}_i = \begin{cases} 0 & q(\mathbf{x}_i; \hat{\boldsymbol{\beta}}) < 0.5 \\ 1 & q(\mathbf{x}_i; \hat{\boldsymbol{\beta}}) \geq 0.5 \end{cases} \tag{16}$$

and we can consider the likelihood function as the probability of $y_i = 1$.

C. Feedforward Neural Network

So far, we have looked at linear and logistic regression and what they can do. The alert reader may have picked

up that the decision boundary of logistic regression is just a straight line. We can show this for the binary case. This is just where

$$q(y_i = 1|\mathbf{x}_i; \hat{\boldsymbol{\beta}}) = q(y_i = 0|\mathbf{x}_i; \hat{\boldsymbol{\beta}}) = \frac{1}{1 + e^{-\mathbf{x}^T\boldsymbol{\beta}}} = \frac{1}{2}$$

and we need to solve

$$\rightarrow \mathbf{x}^T\boldsymbol{\beta} = 0 \tag{17}$$

which we know is a linear combination. If we just let

$$\rightarrow \mathbf{x}^T\boldsymbol{\beta} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 = 0$$

and solve for e.g. x_1, we get

$$x_1 = -(\frac{\beta_2}{\beta_1}x_2 + \frac{\beta_0}{\beta_1}) \tag{18}$$

which is obviously a linear function. However, what if we are dealing with a complex problem where the classification can't be divided into two groups or more by a straight line? This is where neural networks (NN) come in.

An artificial neural network is a machine learning method inspired by the biological neural network in that it tries to, at least in some sense, implement the known saying by Donald Hebb: "*Cells that fire together, wire together*" [2]. They can be used for various tasks and do not need to be preprogrammed to any task-specific rules. We can thus use the same neural network for classification problems, fitting a line to some data and much more by just giving it some input and output data. There are various types of NN, but we will look at a type called feedforward neural network (FFNN) [1].

1. *Architecture*

A feedforward neural network is a neural network where the architecture of the model is such that the data is flowing in one direction, and that is forward. We'll see exactly what this means shortly. An FFNN consists of different kinds of layers. The ones that are straightforward forward are the input layer and output layer. These two are just like what they sound like. The input layer is the layer with the input data, and the output layer is the layer with the final output. The magic the FFNN does is between these two layers [1].

These layers between the input and output layers are called hidden layers. The number of hidden layers is flexible and is determined by what kind of data you have and is something one needs to figure out by trial and error [1]. But what do all these layers contain?

They contain what one may call neurons, nodes or units. For the input layer, this is pretty simple. Each data point in a single training example is a neuron. For the hidden layers, the amount of neurons is flexible and we have to choose the amount of neurons by trial and error. The output layer has neurons equal to the amount of outputs expected, e.g. one, or two if you want, for binary classification. Now, assume we have some hidden layers. All input neurons are passed on as a weighted sum to each neuron in the next hidden layer. We denote this weight by w_{jk}, and it's the weight that adjusts the data on the way to the j-th neuron in the next layer from the k-th neuron in the previous layer. In addition to this, each neuron is equipped with a bias term, b_j, which it adds to the end of the sum. Now, let's see how all of this looks. Assume we want to pass the data in the input layer with p data points to the j-th neuron in the first hidden layer. This would look something like

$$z_j^{(1)} = \sum_{k=1}^{p} w_{jk}^{(1)} x_k + b_j^{(1)} \tag{19}$$

and when it is received by the neuron, it modifies the input further by what is called a activation function. Each layer has an activation function that the data received by each neuron is fed through. The final value for the neuron is then the output of this activation function. In practice, each hidden layer can have a unique activation function, but in this project we're just going to choose one for the hidden layers. Again, the choice of the different activation function, which we are going to cover later, is also chosen by trial an error. The output layer can also have an activation function or none, and depends on the the type of problem you are trying to solve [1].

Now, let $l = 0, \ldots, L$ denote the different layers, where $L \in \mathbb{N}$ denotes the last layer, i.e. the output layer. If we let $f(x)$ denote a activation function, the output of the j-th neuron in the l-th layer can be written as

$$a_j^{(l)} = f(z_j^{(l)}) \tag{20}$$

and now we can generalise equation (19). This can be written as

$$z_i^{(l)} = \sum_{j=1}^{N_{l-1}} w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)} \tag{21}$$

where N_l is the number of neurons in the layer l[1]. We can write this in matrix notation as well, with several input cases. Assume as before that we have some design matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$. The weight matrix is then $\boldsymbol{W}^{(l)} \in \mathbb{R}^{N_{l-1} \times N_l}$ and the bias vector is $\boldsymbol{b}^{(l)} \in \mathbb{R}^{N_l \times 1}$. We can now write the output of each layer as

$$\begin{aligned}
\boldsymbol{a}^{(0)} &= \mathbf{X} \\
\boldsymbol{a}^{(1)} &= \boldsymbol{f}(\mathbf{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}) \\
\boldsymbol{a}^{(2)} &= \boldsymbol{f}(\boldsymbol{a}^{(1)}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}) \\
&\vdots \\
\boldsymbol{a}^{(L)} &= \boldsymbol{f}(\boldsymbol{a}^{(L-1)}\boldsymbol{W}^{(L)} + \boldsymbol{b}^{(L)})
\end{aligned} \tag{22}$$

and we can compare our output with the target $\boldsymbol{t}$ using some cost function. All these steps from the input to the output layer is called a feedforward pass [1]. Now, let's look at some of the details.

2. *Activation Functions*

As we have seen, we have to choose an activation function for the hidden layers by trial and error. For the output layer, the activation function depend on the output. Let's list the activation functions we're going to use

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{23}$$

$$\text{ReLU}(x) = \max(0, x) \tag{24}$$

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases} \tag{25}$$

and we see that equation (23) is the same as (13).

We're going to use our neural network for regression and classification problems. For the regression problem, we don't use any activation function for the output layer at all. This is because we want the output as is, and not modified by some function. For the classification problem, we want to use (23) as an activation function for the output layer. As we have previously seen, it squishes all the input between 0 and 1, which we can interpret as the probability and classify accordingly. The sigmoid function is known as a soft classifier. A hard classifier for a binary case would be something like the Heaviside function, where the output is 1 if $x > 0$ and 0 otherwise. Here, we get the class from the activation directly, while for soft classifiers we have to do a bit of post-processing [1].

3. *Backpropagation*

When we have our output from the last layer, whether it has gone through an activation function or not, we want to know how well our FFNN has done. We thus pass it through a cost function to assess the error. In FFNN, the cost function is now a function of $\boldsymbol{\theta} = \{\boldsymbol{W}, \boldsymbol{b}\}$, and we can write it as

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{\ell}_i(\boldsymbol{\theta}) + \lambda \sum_{ij} w_{ij}^2 \tag{26}$$

where we have included an L^2 regularisation term for the weights, which we will use in our analysis [1].

Now, assume we have produced an output and gotten an error estimate. This is most likely a terrible prediction since we are initially just guessing on the weights and biases, as we will come back to later. Thus, we need a way to update the weights and biases to get a better prediction. This is done by the backpropagation algorithm. We will not derive it here, but see [5] for details. It is essentially just a repeated use of the chain rule, and what it does is calculate the gradient of the cost with respect to the weights and biases. We then update the weights and biases at each layer according to these gradients using a gradient descent method. When we have found the new weights and biases, we do a new feedforward pass. We rinse and repeat until we have an acceptable prediction [5].

If we let f denote some activation function, the backpropagation algorithm makes use of the following equations

$$\boldsymbol{\delta}^{(L)} = \nabla_{a^L} C \odot \boldsymbol{f}'(\boldsymbol{z}^{(L)}) \tag{27}$$

$$\boldsymbol{\delta}^{(l)} = \boldsymbol{\delta}^{(l+1)}(\boldsymbol{W}^{(l+1)})^T \odot \boldsymbol{f}'(\boldsymbol{z}^{(l)}) \tag{28}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_j^{(l)} \tag{29}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = a_k^{(l-1)}\delta_j^{(l)} \tag{30}$$

where we see that $\delta_j^{(l)}$ is the rate of change of the cost with respect to the input $z_j^{(l)}$. This is dubbed as the error. Also, it's easy to see that we first calculate the error for the last layer and use this to calculate the error for the previous layer, and so on until the first layer. This is then used to calculate the gradients (29) and (30), which is then used together with a gradient descent method to update the weights and biases [5].

4. *Weights and Biases*

So far, we have only mentioned that we use the weights and biases to modify the input going into these nodes in the hidden layers and possibly the output layer, but not the size of them. If we initialised them to zero, all neurons would have the same output, and no training would occur. So we need them to be nonzero. One of the most popular ways used before was to randomly initialise the weights from a normal distribution $\mathcal{N}(0, 1)$. However, when this initialisation was combined with the most popular activation function in the hidden layers, i.e. the sigmoid function, it sometimes resulted in what is known as vanishing gradients. This is precisely what it sounds like, and it would stop training since the

gradients would be extremely small. It was shown that in this type of initialisation and activation, the variance of the output of each layer was greater than the inputs [2].

What we're going to use instead is known as Xavier or Glorot initialisation. This method will also pick random values from a normal distribution with zero mean, but instead of a variance equal to 1, it will have

$$\mathrm{Var}[\boldsymbol{W}^{(l)}] = \frac{2}{N_{l-1}+N_l} \tag{31}$$

and thus the variance differs, depending on how many neurons there are in the previous and current layer. In this project, we are just going to use the same amount of neurons in all the hidden layers. Thus, the variance of all the layers except the first and last one will be the same [2].

The biases are not as much trouble and will be set to 0.01 such that all neurons have something they can backpropagate [1].

D. Validation

We have to assess how well our model does. For regression models, we will use the mean-square error and R^2-score, as we did in [4]. These are given by

$$MSE(\mathbf{y},\tilde{\mathbf{y}}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i-\tilde{y}_i)^2. \tag{32}$$

$$R^2(\mathbf{y},\tilde{\mathbf{y}}) = 1 - \frac{\sum_{i=0}^{n-1}(y_i-\tilde{y}_i)^2}{\sum_{i=0}^{n-1}(y_i-\bar{y})^2} \tag{33}$$

where $\bar{y}$ is the mean of the target values. These expressions are going to tell us how well our test data does. For the classification problem, we're going to use the accuracy score

$$\text{Accuracy} = \frac{\sum_{i=0}^{n-1} I(\tilde{y}_i = y_i)}{n} \tag{34}$$

where I is called the indicator function. It is 1 if $\tilde{y}_i = y_i$ and 0 otherwise [1].

III. RESULTS AND ANALYSIS

We are first going to perform OLS and Ridge regression, where the optimal parameters are found by gradient descent methods. After that, we're going to build a neural network which is going to perform both regression and classification.

A. Gradient descent

We are going to perform OLS and Ridge regression using GD and SGD on the following second-order polynomial

$$f(x) = 4x^2 + 3x + 2 + 0.1\mathcal{N}(0,1) \tag{35}$$

where the last term is a normally distributed noise. We are going to use $n = 400$ data points and construct the design matrix as a Vandermonde matrix, with the intercept included. It means it will look like

$$\mathbf{X} = \begin{pmatrix} 1 & x_0^1 & x_0^2 \\ 1 & x_1^1 & x_1^2 \\ \vdots & \vdots & \vdots \\ 1 & x_{n-1}^1 & x_{n-1}^2 \end{pmatrix}.$$

so $\mathbf{X} \in \mathbb{R}^{n\times 3}$, and the regression parameters $\boldsymbol{\beta} \in \mathbb{R}^{3\times 1}$ will be initialised randomly.
The x-values will be generated randomly from $\mathcal{N}(0,1)$, and all values in $\mathbf{X}$ will therefore be between 0 and 1. Thus, all columns will be of the same order and no scaling is really needed.

The cost function we will use is the mean square error (32), with an L^2-regularisation term for Ridge. We will find the MSE and R^2-score as a function of the different learning rates η, the hyperparameter λ as well as the friction term for momentum γ, number of epochs and size of mini-batches.

1. *Plain Gradient Descent*

We can start by looking at plain gradient descent. We have done a grid search for the learning rate and hyperparameter, which we see in figure 1. These are done for at most 2000 iterations. Less if the difference between the updated and previous regression parameter is below 10^{-5}.
We can see that it does very well when $\eta = 0.1$ and $\eta = 0.01$. We also see that OLS, i.e. $\lambda = 0$, and Ridge with hyperparameter between 10^{-9} and 10^{-3} performs equally well.

We will stick with $\lambda = 0$, i.e. OLS, for convenience in the analysis of the adaptive learning rate methods and momentum. Since we don't need fine-tuning of the global learning rate in these, we found the most time-efficient and reliable way to start with $\eta = 0.1$ and test smaller and larger values systematically to see where that leads. The result of this can be seen in table I.

We see that keeping GD with a constant learning rate and nothing else is a poor decision. By just applying

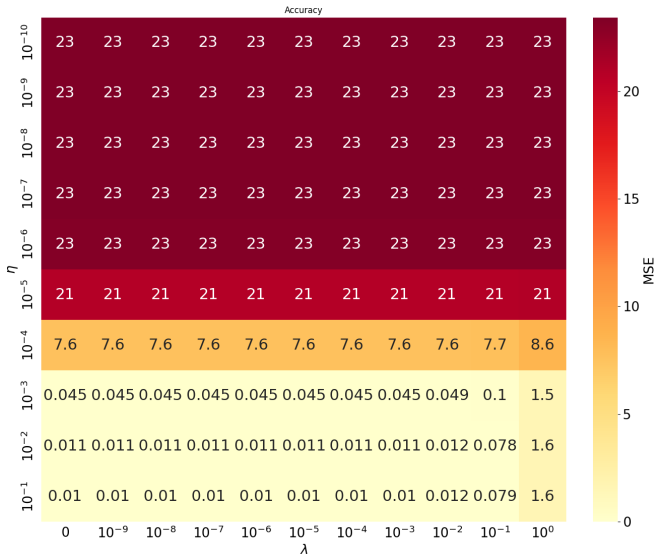

Figure 1: MSE heatmap of the parameter estimate for (34) using plain gradient descent with $n = 400$. MSE is shown for various values of the learning rate η and the hyperparameter λ.

	η	γ	iterations	MSE
Constant	0.1	-	2000	0.01
Constant	0.1	0.9	926	0.01
Adagrad	5	-	1205	0.01
Adagrad	5	0.9	326	0.01
RMSProp	0.01	-	2000	0.01
RMSProp	0.01	0.8	304	0.01
Adam	5	-	211	0.01

Table I: MSE heatmap of the parameter estimate for (34) using plain gradient descent with $n = 400$. MSE is shown for various values of the learning rate η and the hyperparameter λ.

the simple momentum term, the number of iterations is more than halved. We have even more to gain if we use one of the adaptive methods.

We notice that the global learning rate for Adagrad is very large compared to the constant case. If we recall, Adagrad divides the global learning rate by the accumulated gradients, as seen in equation (8). If we start with a small global learning rate and assume our starting point is in a more steep direction, we will be accumulating large gradients rather quickly since we are just taking small steps in that region. The learning rate would then be too small to make proper progress. Thus, a large η is necessary for good progress. When it is large, we can jump out of these steep regions and towards the global minimum rather quickly, and we avoid accumulating large gradients.
With a good global learning rate, we see that the iterations are almost halved, and with momentum, Adagrad is six times faster.

The modified Adagrad, i.e. RMSProp, has the opposite case. The global learning rate is smaller than the one in the constant case. Recall that the decay rate ρ is set to 0.9. This means that the gradient squared, which is accumulated, is first multiplied by 0.1, and then multiplied by 0.9 in the next step. Further inspection of the learning rates for two cases, 0.01 and 5, as we use for Adagrad, shows us two things. First and foremost, we know that the gradient at the first step will be the same for both cases. Thus the larger global learning rate will take a larger initial step. Then we see that after some iterations, they have approximately the same learning rate. This tells us that the case with the large global learning rate is just oscillating between steep areas, while the case with the smaller value, oscillates in the vicinity of the global minima.
Now, we see that RMSProp without momentum is not really an upgrade and does just as poorly as the constant learning rate. Adding momentum though, changes everything and makes it six times faster as well.

Lastly, we have Adam. This has momentum baked in. For this problem, Adam will converge to the same MSE for practically any global learning rate that is larger than 0.1. We found that the number of iterations continues to decrease until $\eta = 5$ and then gradually starts to increase. By inspecting the values of the updates, we see the reason for this behaviour. The update $\Delta\boldsymbol{\beta}$ is quickly reduced after the first iterations, and it will oscillate back and forth with smaller and smaller steps. For our case with $\eta = 5$, the first update will be 5 for all parameters, while the next is at most -0.3. Nonetheless, this provides us with the fastest convergence, and is almost ten times faster than GD with a constant learning rate!

2. *Stochastic Gradient Descent*

Now, let's do the same analysis for stochastic gradient descent. We see the grid search in figure 2, and we see that the behaviour is similar for the same η and λ as in GD. SGD perform slightly worse for lower learning rates. This is probably due to the oscillating nature of SGD, i.e. we are just calculating the gradient of a small subset of data and thus get a "noisy" gradient. This, in combination with a small learning rate, may lead to slower progress. Nonetheless, it performs as well as GD for a learning rate equal to 0.1 or 0.01. Ridge and OLS perform equally well, except for $\lambda \geq 0.1$. Therefore, let's do as we did with GD, and do our analysis with OLS.

So far, we have chosen the number of epochs and mini-batches arbitrarily to a certain extent. We did our grid search for 25 epochs and 40 mini-batches, i.e. 10 data points in each batch. Now, we do another grid search to find the optimal number of epochs and mini-batches. We

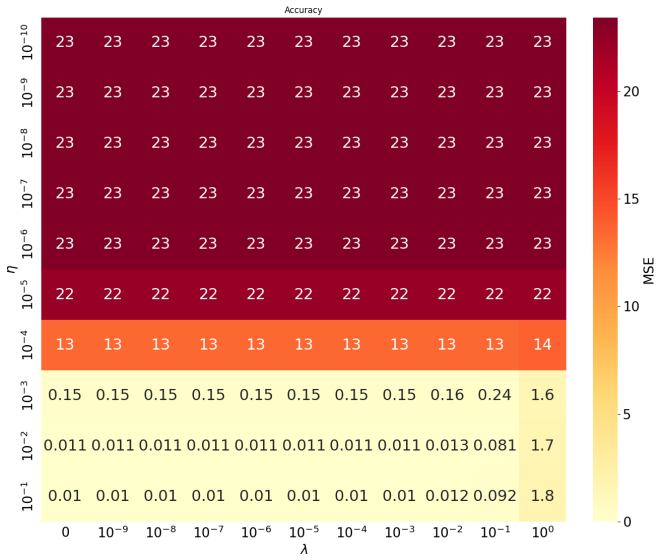

Figure 2: MSE heatmap of the parameter estimate for (34) using stochastic gradient descent with $n = 400$, 25 epochs and 40 mini-batches. MSE is shown for various values of the learning rate η and the hyperparameter λ.

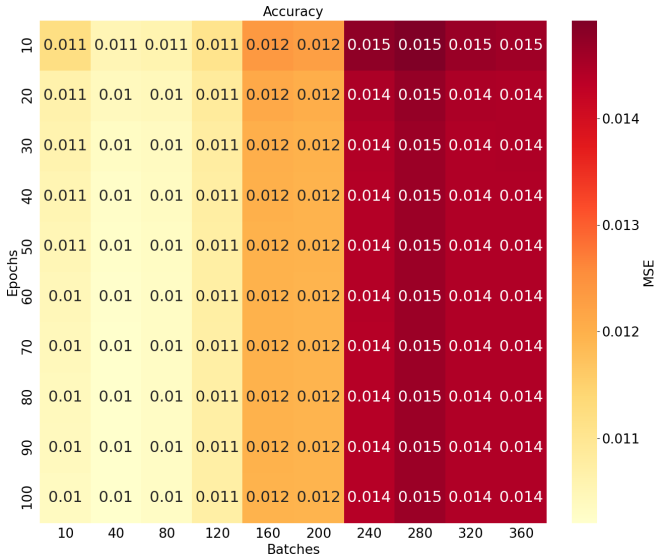

Figure 3: MSE heatmap of the parameter estimate for (34) using stochastic gradient descent with $n = 400$, $\eta = 0.1$ and $\lambda = 0$. MSE is shown for various number of epochs and mini-batches.

dod this for $\eta = 0.1$ and $\lambda = 0$. The result is shown in figure 3. We see that there are some choices that are better than others, both when it comes to the error and the amount of computational cycles, but the choice is rather insignificant. All choices give a good approximation.
But now, let's note how much faster this is than GD. We see that 20 epochs and 80 batches give us an equally good prediction as GD, but here we only do 100 iterations, and at those iterations, we only calculate the gradient for 5 inputs instead of 400! This is a massive improvement.

Now, due to the oscillating nature of SGD, it has not been easy to make the algorithm stop based on the difference between the previous and current parameters or even the gradient. We could instead choose to stop when we have reached a given MSE value, but it can be difficult to choose such a value since it depends on the size of the data at hand. This was not a problem in GD since the algorithm would not oscillate in the same way around the minima. Thus, for this problem, we have not been able to identify that the adaptive learning rates and momentum reduce the number of iterations like in GD. Nonetheless, all of these methods give the same error as in table I, but now with 20 epochs and 80 batches instead. We found by trying different global learning rates and momentum values, together with fewer epochs, would give reliable solutions as well. But as we have seen, lowering the amount of epochs in SGD with a constant learning rate also gives a reliable solution. So for this regression problem with SGD, a properly tuned learning rate and hyperparameter is sufficient.

Now, how do GD and SGD compare with standard OLS and Ridge as shown in equation (3) and (4)?

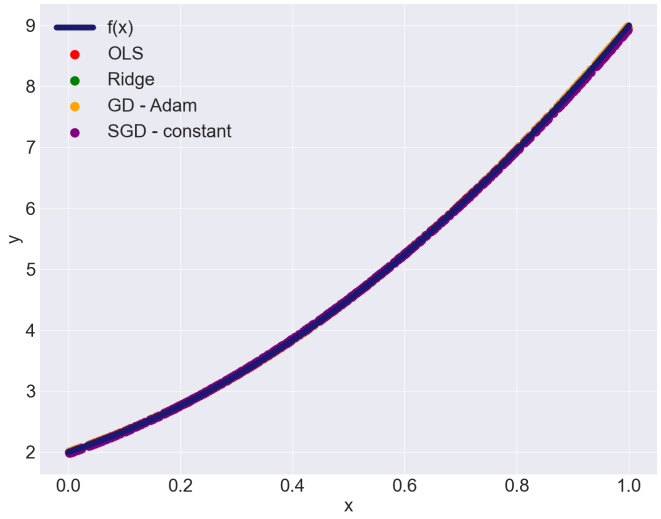

Figure 4: Plot of approximation of equation (34) using SGD, GD, OLS and Ridge as well as the true value for $n = 400$. SGD is used with a constant learning rate $\eta = 0.1$, 20 epochs and 80 batches. GD is used with Adam where $\eta = 5$. Ridge is trained with $\lambda = 10^{-5}$.

The difference is vanishingly small. They both have an MSE of 0.01 and an R^2-score of 1. From figure 4, we see that all the methods are almost indistinguishable. Thus, for such a regression problem, all methods are perfectly good choices. Of course, OLS and Ridge are preferred when we can use them, but we have seen that GD and SGD are good choices as well.

B. Neural Network

Now, we are going to move on to neural networks. We are not leaving gradient descent completely since we are going to use SGD in our neural network. We will first try to solve the same regression problem as we have looked at so far using neural networks, and thereafter move to a classification problem.

1. *Regression Problem*

So there are several things we have to have in place and eventually tune. First and foremost, we will use the same amount of data as before, i.e. $n = 400$, but we split the data into train and test data, where we keep 20% as test data. Secondly, we must choose the activation functions, the number of hidden layers $L - 1$ and the number of neurons N_l. For the activation functions, we will use the sigmoid function in the hidden layers and no activation in the output layer. If we recall, we don't want our output modified in any way when we do such regression problems, else the output would be limited by this function. After choosing the activation, we do a grid search on the number of hidden layers and neurons since this is something we need to tune. We have chosen to find the optimal weights and biases by using SGD together with Adam for 300 epochs and 80 batches. Adam is often used with a default value of for the global learning rate$\eta = 0.001$, which has worked for us [2]. Thus, there is one less thing to tune.

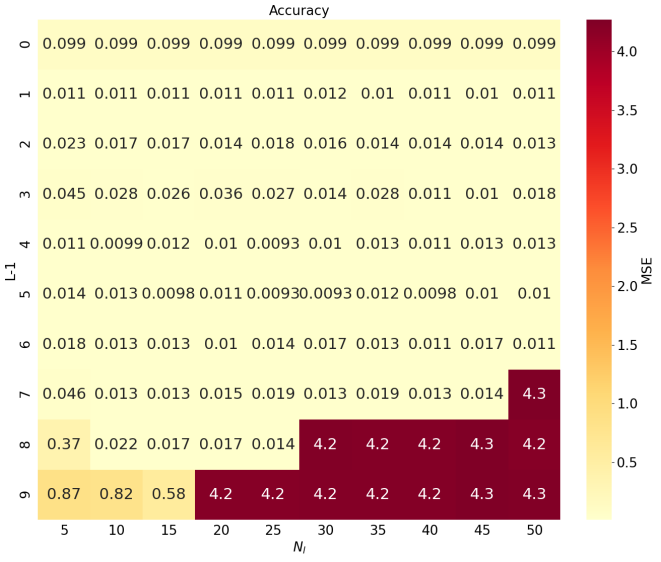

Figure 5: MSE heatmap for prediction of (34) using a neural network with test data. Shown for various number of hidden layers $L - 1$ and neurons per hidden layer N_l.

If we recall, we also have a regularisation term as shown in equation (25). We will, for this part of the analysis, set it to $\lambda = 0$.

The number of epochs was chosen to be on the safe side so that we get a reliable solution, while the number of batches was chosen according to what we found earlier. What we found earlier in the discussion of SGD is not necessarily the most optimal, but we need something to kick-start our investigation. The result of the grid search is shown in figure 5. We see that we get very good results all over the board, except for the highest amount of hidden layers and neurons. We want to choose the least amount of layers and neurons in order for our algorithm to be as efficient as possible. Thus, 1 hidden layer with 35 neurons seems to be an excellent choice and is doing as good of a job as the previous methods we have explored.

But let's see if we can get the number of iterations down. We do a grid search for various numbers of hidden layers and mini-batches for the chosen architecture. The result is shown in figure 6. We see that 300 epochs with 80 mini-batches again give us an excellent result. There was thus nothing lower.

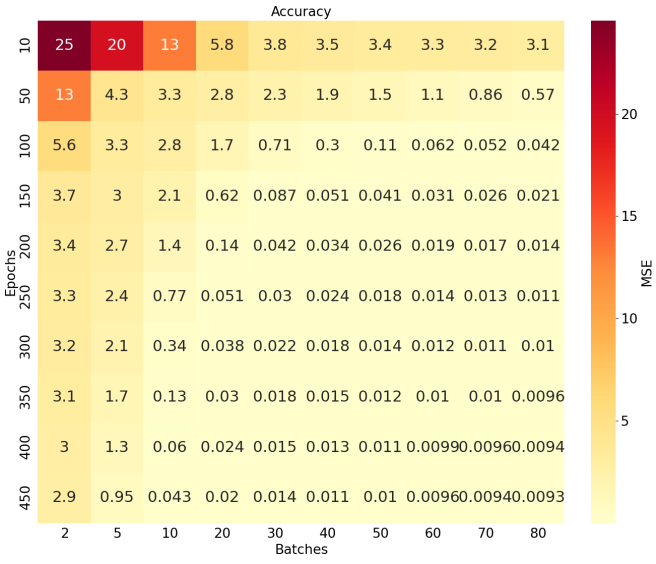

Figure 6: MSE heatmap for prediction of (34) using a neural network with 1 hidden layer and 35 neurons for the test data. Shown for various numbers of epochs and mini-batches.

Now that we're equipped with the number of layers, neurons, epochs and mini-batches, let's see if including the L^2-regularisation does as well as without. The result is shown in figure 7. We see that the error follows an s-curved shape, much like the sigmoid function. Thus, no regularisation term is optimal.

We can now compare how our neural network does,

compared to standard OLS and Ridge, as well as Scikit-Learn's own neural network for regression problems called MLPRegressor. This is shown in figure 8. We see that our neural network does a very good job, and it's almost identical to Scikit-Learn's.

At last, let's see if we could have chosen a better activation function for the hidden layers. As we discussed earlier, the sigmoid function can sometimes lead to problems, and we should explore other activations. We will look at ReLU and Leaky ReLU, which are given in (23) and (24), respectively. We test the different activation functions for different numbers of epochs and see how the error compares to the sigmoid. The result is shown in figure 9.
We see that the overall trend is the same, but both ReLU and Leaky ReLU give a better error estima earlier on and are thus a better choice since we save computational time and reduce the risk of vanishing gradients.

2. *Classification Problem*

Now, we move on to the classification problem. The data set we are going to use is the Wisconsin Breast Cancer data set. This is available through Scikit-Learn, and each data input is a tumour which has various given features. Furthermore, the data is labelled as either benign or malignant. Thus, we are going to split the data into train and test data, and we are going to try to classify the test data as either benign (0) or malignant (1) after we have trained the FFNN on the training data. We are, therefore dealing with a binary classification problem.

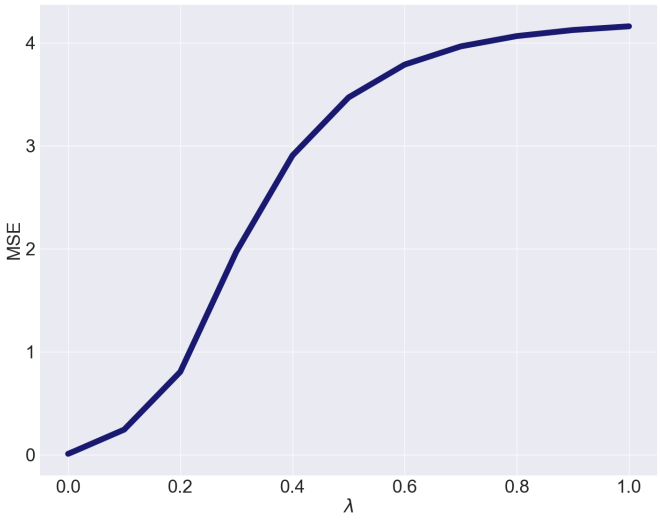

Figure 7: MSE of test data using a neural network with 1 hidden layer and 35 neurons as a function of the hyperparameter λ from equation (25).

We can start our analysis as we did for the regression case. We try to find the optimal number of hidden layers and neurons. We are still using SGD with Adam as before with the standard values. We have again chosen some arbitrary epochs and batches, namely 100 epochs and 80 batches to start off the analysis. We have also chosen for the initial analysis to avoid sigmoid and choose the leaky RelU for the hidden layers instead. The sigmoid function is used as an activation for the output layer in order for us to be able to classify. We will also include the regularisation term, but set it to zero for now and see if it can improve our result later on.

The result is shown in figure 10. We see that we get excellent results across the board. We see that the highest accuracy is 0.99, and we want as few layers as possible, but larger than zero, which we will come back to why later. We see that 3 layers with 20 neurons give us this and that is what we are going to use.

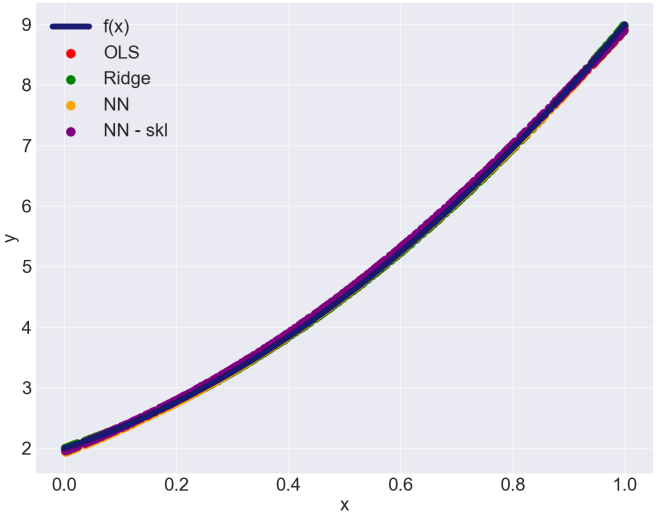

Figure 8: Plot of approximation of equation (34) using our own FFNN, Scikit-Learn's FFNN, OLS and Ridge as well as the true value for $n = 400$. The neural networks are trained have 1 hidden layer and 10 neurons. Ridge is trained with $\lambda = 10^{-5}$.

Now, let us see if can get as good of a score or even better for other combinations of epochs and mini-batches. The result is shown in figure 11. We again want as few epochs and as high a number of batches as possible. We see that for 350 epochs and 80 batches, we get a score of 1!

Let's now look at the regularisation term. So far, we have set it to $\lambda = 0$, and we already have an accuracy of 1 for the given number of epochs and batches above. But this is computationally expensive, and using 10 epochs with 100 batches for an accuracy of 0.99 is just as good in practice and much cheaper. So, let's try different values of λ for this.

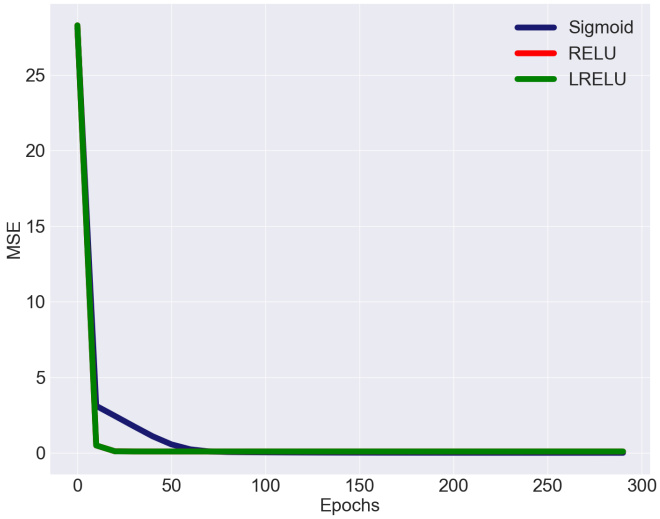

Figure 9: MSE as a function of number of epochs. Shown for three different activation functions in the hidden layer: sigmoid, ReLU and Leaky ReLU

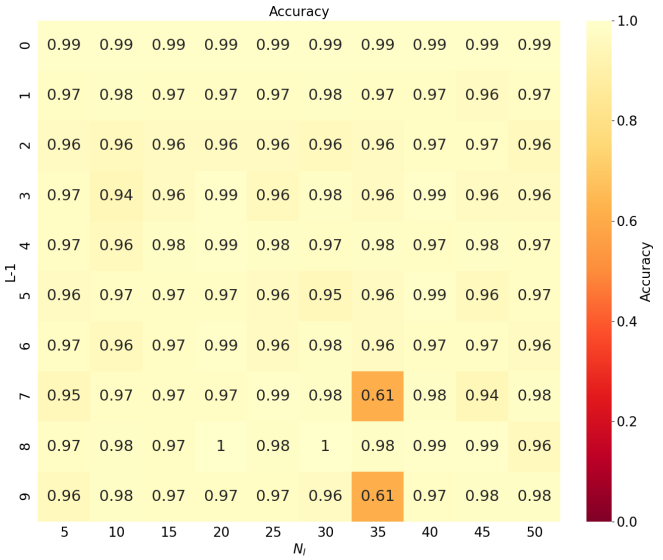

Figure 10: Accuracy heatmap for prediction of benign and malignant tumours from Wisconsin cancer data using a neural network. Shown for various numbers of hidden layers $L-1$ and neurons per hidden layer N_l.

We can see the result in figure 12, and we can see that increasing the hyperparameter actually performs worse, and the best accuracy was achieved for $\lambda = 0$ as we have used so far.

Now, we have no reasoning behind choosing the leaky ReLU as an activation function before the others except that sigmoid may lead to vanishing gradient. So let's see how the different activations compare. The result is shown in figure 13.
It seems like we made a good decision choosing leakt ReLU. We see that it has the highest accuracy for the lowest amount of epochs. sigmoid is overall the worst, but regular ReLU seems to be the winner when the epochs start to increase.

C. Logistic Regression

Lastly, let us see how logistic regression compares to our neural network on the Wisconsin cancer data. We are not going to build a new code for logistic regression though. Let's think about how logistic regression is built up. It is essentially a neural network with no hidden layers which use the sigmoid function as an activation for the output layer. That is precisely what we are going to use. If one looks at figure 10, we have a row of 0 hidden layers. The accuracy for this is 0.99 and is thus performing better than many of the different architecture choices of the neural network.

Still, the neural network can reach an accuracy of 1. Let's see if we can make the logistic regression even better if we set λ¿0. The result is shown in figure 14. We see

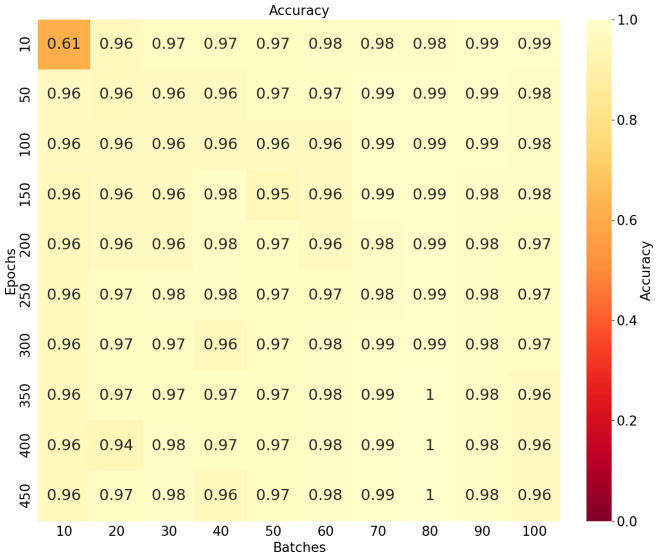

Figure 11: Accuracy heatmap for prediction of benign and malignant tumours from Wisconsin cancer data using a neural network with 3 layers and 20 neurons. Shown for various numbers of epochs and mini-batches.

that we get no improvement and we are better off with no regularisation. Also, the Scikit-Learn's own logistic regression perform as good as our own with an accuracy of 0.99.

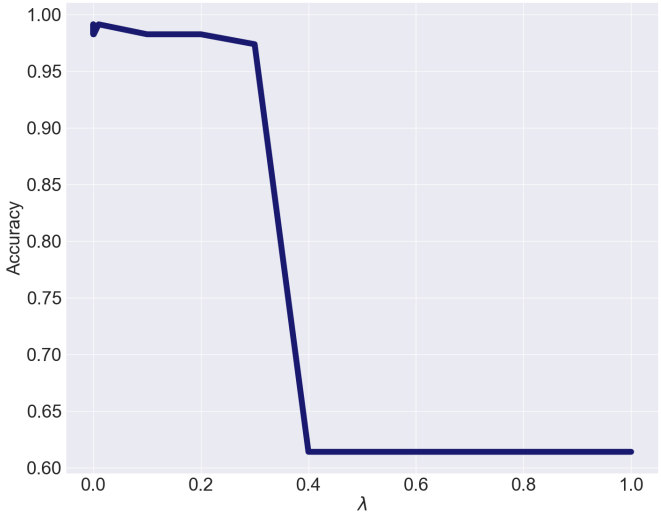

Figure 12: Accuracy of test data using a neural network with 3 hidden layer and 20 neurons as a function of the hyperparameter λ from equation (25).

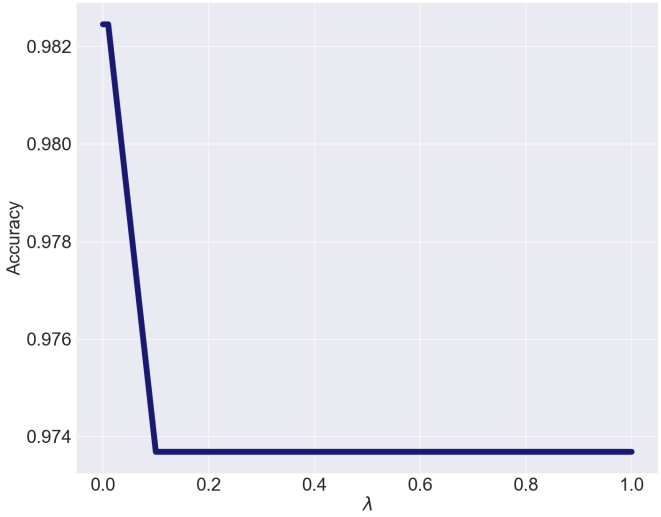

Figure 14: Accuracy of test data using logistic regression as a function of the hyperparameter λ from equation (25).

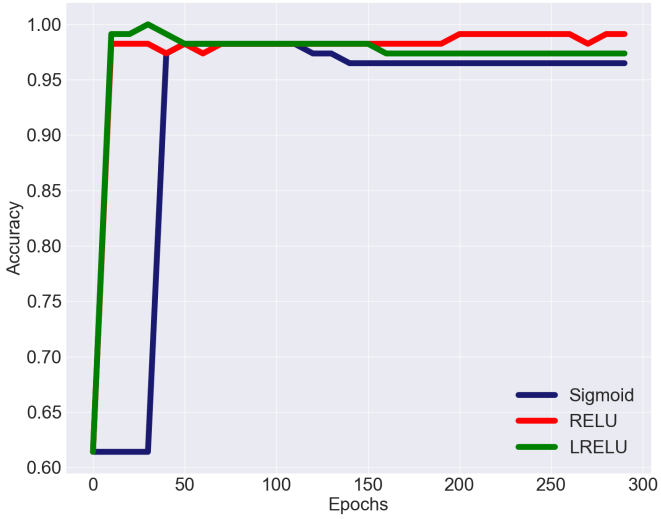

Figure 13: Accuracy of the NN on the Wisconsin cancer data as a function of number of epochs. Shown for three different activation functions in the hidden layer: sigmoid, ReLU and Leaky ReLU

IV. CONCLUSION AND SUMMARY

In this project, we have explored various topics. We have looked at plain gradient descent and stochastic gradient descent, and how we can use these methods to find the optimal parameters for a regression problem. We have also explored constant and adaptive learning rates such as Adagrad, RMSProps and Adam. We found that when using a plain gradient descent, Adam was 10 times faster than a constant learning rate but SGD was the fastest. SGD didn't make any faster progress with the adaptive learning rates, thus a constant learning rate is sufficient.

We also tried to solve the same regression problem using a feedforward neural network. This was done by trying to find the optimal number of layers, neurons, epochs, batches and values for the hyperparameter for the L^2-regularisation. In the end we found that a neural network with 1 hidden layer, 35 neurons was the best. The sigmoid function was used as an activation for the hidden layer and no activation was used for the output layer. We used SGD with Adam to find the weights and biases and found that 300 epochs and 80 mini-batches gave the best result. Lastly, we found that no regularisation gave the best result. In the end, we saw that our and Scikit-Learn's FFNN did as well as standard OLS and Ridge.

In the end, we looked at a classification problem of the Wisconsin cancer data and tried to solve it with both an FFNN and logistic regression. We found that a neural network with 3 hidden layers and 20 neurons gave us the best estimate, where leaky ReLU was used for the hidden layers and sigmoid was used for the output layers. We found 350 epochs and 80 mini-batches, as well as no regularisation to be the most optimal. The accuracy was at 1. For the logistic regression we found an accuracy of 0.99 with no regularisation.

[1] M. Hjorth-Jensen. (2021). *Applied Data Analysis and Machine Learning.* Available at: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html (Accessed: November 20, 2023).

[2] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow,* 2nd edt. Sebastopol, California: O'Reilly Media, 2019.

[3] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning.* Cambridge, MA: MIT Press, 2016. Available at: http://www.deeplearningbook.org (Accessed: November 20, 2023).

[4] A. Atifi. (2023). *Linear Regression Analysis of the Franke Function and Terrain Data.* Available at: https://github.com/achat97/FYS-STK4155/tree/main/Project1/Report (Accessed: November 20, 2023).

[5] M. A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015.