Optimising Simple Neural Networks as Regressors and Classifiers

Anna Hjertvik Aasen, Carl Martin Fevang, Håkon Kvernmoen

November 18, 2022

Abstract

At first we implemented ten GD and SGD algorithms and optimised OLS and ridge linear regression cost functions. Expanding a polynomial design matrix to degree 5 and fitting to the Franke function, we found SGD with Adam to perform the best, with a validation MSE of 0.1660 ± 0.0018, compared to the analytical OLS solution with a validation MSE of 0.1377. Ridge penalisation did not improve the GD methods, but improved the analytical solution to 0.1338 with $\lambda = 0.002$. We employed GD optimisation of NN regressors fitted to Nicaraguan terrain data. A network architecture with 5 hidden layers of 200 nodes each achieved a validation MSE of 0.078 with an L_2 penalisation hyperparameter $\lambda = 10^{-4.1}$, beating a linear regression with a polynomial design matrix of degree 11, achieving an MSE of 0.171. We performed classification on the Wisconsin Breast Cancer dataset using logistic regression and small neural networks. A stable logistic regression model was found with an accuracy of 0.983 with an L_2 penalisation term $\lambda = 10^{-3.5}$. Despite achieving an accuracy of 0.993 using a one layer neural network, we argue that logistic regression is sufficient for reliable classification.

Contents

1 Introduction

Neural Networks (NNs) have taken the world of regression by storm, and have already been applied to an incredibly rich plethora of different problems. Here we will introduce one of the simpler, yet effective NN frameworks with a number of fully connected hidden layers. Before we introduce our NN framework, we will take a dive into gradient descent (GD) methods, as these are the industry standard for optimising NNs when training them. We will apply ten different GD methods and compare the performance in optimising OLS and ridge cost functions. Building on our earlier project [Aasen et al., 2022], we will then train and apply our NNs on two-dimensional Nicaraguan terrain, and compare performance with linear regression methods.

Another area where NNs have proved effective has been in classification problems, as is common in medicine for instance. We will introduce and employ standard logistic regression and our NN on breast cancer data, a binary classification problem where we classify tumours either as malignant or benign.

We will compare our implementations with those of a standard library; `scikit-learn`.

2 Theory

When referring to the data we will be dealing without throughout this project, it is useful to define quantities clearly to limit the confusion. This was done in [Aasen et al., 2022], but we will summarise it here.

Letting $\boldsymbol{y} \in \mathbb{R}^n$ be a vector containing a series of n measurements with a corresponding to an input matrix $X \in \mathbb{R}^{n\times p}$. The rows $x_i \in \mathbb{R}^p$ of X are the p inputs, or *features*, producing the measurement y_i. The feature columns of X are referred to as $\boldsymbol{x}_a \in \mathbb{R}^n$. Together these form a dataset $\mathcal{D} = \{(y_i, x_i)\}$. Furthermore, we refer to the parameters of our models collectively by $\boldsymbol{\theta}$.

The underlying assumption of our regression is that the measurements y_i are produced from some *exact* function $f(x_i) : \mathbb{R}^p \to \mathbb{R}$, with an added stochastic noise ϵ which follows some unknown distribution. We will further assume that this noise is generated from some normal distribution $\epsilon \overset{d}{\sim} \mathcal{N}\left(0, \sigma^2\right)$ with some standard deviation (std) σ.

In fitting our models to the data, we want to produce predictions $\hat{y} = \hat{f}(x)$ that approximates new measurements $y \sim f(x)$.

2.1 Gradient Descent Methods

Optimisation of the parameters of the models we will be employing in this project is formulated as minimisation problems, with an associated cost function $C(\boldsymbol{\theta})$ to be minimised to find the optimal parameters.

2.1.1 Plain Gradient Descent

We will compare different variants of GD algorithm, starting first with the plain one. Assume the gradient of the cost function $C(\boldsymbol{\theta})$ w.r.t. the parameters $\boldsymbol{\theta}$ is well-defined on the entirety of our parameter space, and is calculable either analytically or approximated numerically.[1] The idea is that the fastest direction to move in parameter space to reduce $C(\boldsymbol{\theta})$ is to follow the direction of $-\boldsymbol{\nabla}_{\boldsymbol{\theta}} C(\boldsymbol{\theta})$. Plain gradient descent implements this idea iteratively, starting from an initial guess $\boldsymbol{\theta}_0$, finding the next point by

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_{k-1}),$$

where $k = 1, 2, \ldots, N$ for N iterations, and η is an introduced hyperparameter called the learning rate.

2.1.2 Adding Momentum

The simple GD can be fleshed out by giving the algorithm a sense of inertia, helping it continue to move in a certain direction. This combats tendency to oscillate around saddle points, for instance. It starts as earlier, but with the additional initialisation $\boldsymbol{p}_0 = 0$, finding $\boldsymbol{\theta}_k$ iteratively by

$$\boldsymbol{p}_k = \gamma \boldsymbol{p}_{k-1} + \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} C(\boldsymbol{\theta}_{k-1}), \tag{1a}$$
$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \boldsymbol{p}_k, \tag{1b}$$

where γ is a new hyperparameter characterising the memory of earlier gradients.

2.1.3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is based on functions to minimise that are on the form $C(\boldsymbol{\theta}) = \sum_{i\in A} c_i(\boldsymbol{\theta}) + c_0(\boldsymbol{\theta})$ for $A = \{1, 2, \ldots, n\}$. This is often the case, as in MSE-scores, where a cost is associated with every observation y_i separately. The idea of SGD is to avoid computing the gradient of the whole of C, but rather split it into mini-batches $C_k(\boldsymbol{\theta}) = \sum_{i\in A_k} c_i(\boldsymbol{\theta}) + c_0(\boldsymbol{\theta})$ for some subset $A_k \subseteq A$, and compute the gradient only on one of these smaller batches. This helps solve two problems: First, it makes the computational cost of calculating the

[1] Now this is not always the case, e.g. with L_1-penalisation on the parameters.

gradient smaller, as we no longer compute the full one, and second, it adds a degree of stochasticity to the movement in parameter space since we no longer will follow the 'true' gradient, which prevents getting stuck in local minima.

SGD is done in so-called epochs, where we divide the full set A into equally sized batches A_k, and do one GD iteration on every batch once. After this, A is reshuffled and new mini-batches are drawn.

2.1.4 Improving Convergence and Stability

As stochasticity is added to the algorithm, the stability might decrease. As such, it would be nice to add algorithms that can adapt the learning rate η in such a way as to increase it. The AdaGrad algorithm does this with the idea to use different learning rates along different directions in parameter space; i.e. use larger learning rates in flat directions, while keeping it small in steep directions. This is done by keeping a running sum G_k of the outer product of the gradients $\boldsymbol{g}_k$ at the points $\boldsymbol{\theta}_k$, and use this to adapt the learning rate.

$$\boldsymbol{g}_k = \boldsymbol{\nabla}_{\boldsymbol{\theta}} C_k(\boldsymbol{\theta}_{k-1}) \tag{2a}$$
$$G_k = G_{k-1} + \boldsymbol{g}_j \otimes \boldsymbol{g}_j \tag{2b}$$
$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \eta G_k^{-1/2} \boldsymbol{g}_k \tag{2c}$$

where $\boldsymbol{v} \otimes \boldsymbol{u}$ denotes the outer product of two vectors $\boldsymbol{v}, \boldsymbol{u}$, and a matrix $m = M^{-1/2}$ is understood to the matrix satisfying $m^2 = M^{-1}$. Inverting and finding the root of G_k comes at a computational cost, so we will employ the much cheaper version where we update

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \frac{\eta}{\sqrt{\text{diag}(G_k)} + \epsilon} \circ \boldsymbol{g}_k, \tag{2d}$$

where the arithmetic operations are understood to be element-by-element and $\boldsymbol{v} \circ \boldsymbol{u}$ represents the element-wise multiplication of two vectors $\boldsymbol{v}, \boldsymbol{u}$, and ϵ is a small number to avoid zero-division.

The AdaGrad algorithm serves to tune the learning rate, and can be applied to gradient descent with momentum too. This gives the update rule

$$\boldsymbol{g}_k = \boldsymbol{\nabla}_{\boldsymbol{\theta}} C_k(\boldsymbol{\theta}_{k-1}) \tag{3a}$$
$$\boldsymbol{p}_k = \gamma \boldsymbol{p}_{k-1} + \eta \boldsymbol{g} \tag{3b}$$
$$G_k = G_{k-1} + \boldsymbol{g}_j \otimes \boldsymbol{g}_j \tag{3c}$$
$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \frac{\eta}{\sqrt{\text{diag}(G_k)} + \epsilon} \circ \boldsymbol{p}_k \tag{3d}$$

Another way to adapt the learning rate to the parameter space 'terrain' is found in the Root Mean Square Propagation (RMSprop) algorithm. Here we approximate a running average of the second moment of the gradient $\boldsymbol{s} = \mathbb{E}(\boldsymbol{g} \circ \boldsymbol{g})$, then use this to scale the learning rate appropriately.

$$\boldsymbol{g}_k = \boldsymbol{\nabla}_{\boldsymbol{\theta}} C_k(\boldsymbol{\theta}_{k-1}) \tag{4a}$$
$$\boldsymbol{s}_k = \beta \boldsymbol{s}_{k-1} + (1-\beta) \boldsymbol{g}_k \circ \boldsymbol{g}_k \tag{4b}$$
$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \frac{\eta}{\sqrt{\boldsymbol{s}_k} + \epsilon} \circ \boldsymbol{g}_k \tag{4c}$$

where again ϵ is a small number to avoid zero-division, and the arithmetic operations are understood to be element-wise. Here $\beta \in [0, 1]$ is a new hyperparameter parametrising the rate at which previous gradients should inform the current average of the second moment.

Finally, an algorithm which aims to combine RMSprop with momentum is Adam. Here we also keep a running average of $\boldsymbol{m} = \mathbb{E}(\boldsymbol{g})$, i.e. the first moment of the gradient. Moreover, bias-corrected values are used in lieu of the 'bare' estimates of the first and second moments, here denoted with hats. This ensures that a bias for low values of $\boldsymbol{m}_k, \boldsymbol{s}_k$ in early iterations with high β-values is avoided.

$$\boldsymbol{g}_k = \boldsymbol{\nabla}_{\boldsymbol{\theta}} C_k(\boldsymbol{\theta}_{k-1}) \tag{5a}$$
$$\boldsymbol{m}_k = \beta_1 \boldsymbol{m}_{k-1} + (1-\beta_1) \boldsymbol{g}_k \tag{5b}$$
$$\boldsymbol{s}_k = \beta_2 \boldsymbol{s}_{k-1} + (1-\beta_2) \boldsymbol{g}_k \circ \boldsymbol{g}_k \tag{5c}$$
$$\hat{\boldsymbol{m}}_k = \frac{\boldsymbol{m}_k}{1-\beta_1^k} \tag{5d}$$
$$\hat{\boldsymbol{s}}_k = \frac{\boldsymbol{s}_k}{1-\beta_2^k} \tag{5e}$$
$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} - \frac{\eta}{\sqrt{\hat{\boldsymbol{s}}_k} + \epsilon} \circ \hat{\boldsymbol{m}}_k \tag{5f}$$

2.2 Neural Networks

Artificial neural networks (ANN) are biologically inspired machine learning algorithms which can improve performance as they are exposed to data and without being programmed with any task-specific rules. In this project fully connected Feed-Forward Neural Networks (FFNN) will be implemented.

2.2.1 Structure

First a non-mathematical, intuitive introduction. An ANN consists of an input layer and an output layer. Often the network will also have layers between the input and output, called *hidden layers*. In that case the neural network is often referred to as deep. Each layer contains *nodes* which are supposed to represent the neurons in a biological neural network, or rather their activation. In a fully connected neural network, all nodes in one layer are connected to all the nodes in the previous as well as the next layer. These connections, which represent the synapses, are referred to as *weights*, and the sizes of the weights are the synaptic strength. Each node also have an activation *bias* which says something about how easily activated that specific neuron/node is. See Fig. 1 for an illustration of a simple artificial neural network.

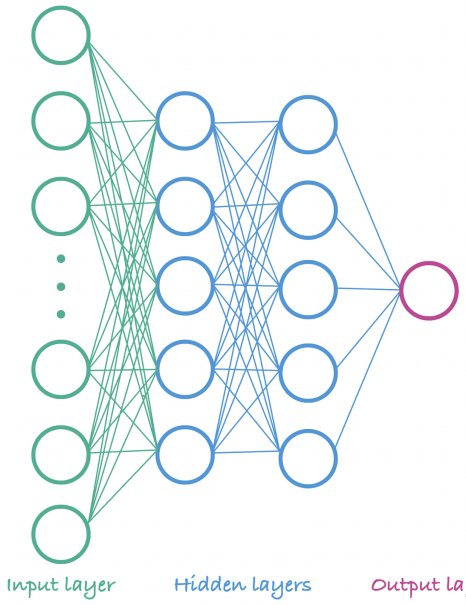


Figure 1: Illustration of a simple deep neural network with an input layer, two hidden layers, and an output layer. The circles are the nodes and the lines between are the weights.

The nodes, weights and biases are all essentially just numbers, but following an algorithm for combining these numbers creates an ANN.

2.2.2 Feed Forward

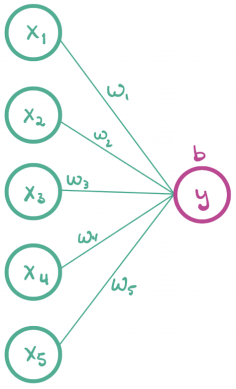


Figure 2: Illustration of a simple neural network with an input layer, and an output layer consisting of one node.

The simplest of neural networks is the *Feed Forward Neural Network(FFNN)* which, as its name implies, feeds activation from the input layer and forward through the network, eventually ending up in the output layer. Given a NN with only an input layer, and an output layer consisting of one node (see Fig. 2), the output $\hat{y}$ is then

$$\hat{y} = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = f(z) \tag{6}$$

where x_i constitutes the input, w_i are the weights corresponding to each input variable, and b the bias of the output node. We define z as the sum over all the weighted inputs with bias added; $z = (\Sigma_{i=1}^{n} w_i x_i + b)$. f is called the *activation function* and depends on the analysis being executed; e.g. regression, classification, etc. Each node in a network has a corresponding activation function which defines the possible outputs of that node. The activation function closest to replicating the behaviour of a biological neuron is the Heaviside function which yields either 0 or 1; firing or not firing. However, there are in some cases advantages of abandoning this biological model. Some other, and currently more used activation functions are: identity, sigmoid, ReLU, tanh, and leaky ReLU some of which are plotted in Fig. 3.

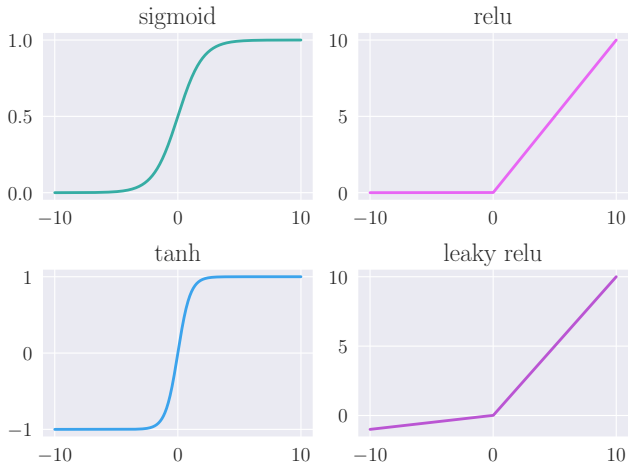


Figure 3: Plots of some commonly used activation functions.

In the output layer the activation function constitutes what analysis the network does. E.g. a form of binary classification can be done by implementing the sigmoid activation function in the output layer, and for regression problems the identity function, $f(x) = x$, is implemented. It is normal that all the nodes in a layer have the same activation function f.

If we were to have a more complex neural network than the one from Fig. 2, now also including hidden layers, $\hat{y}$ (ref Eq. 6) would be a node in the hidden layer after the input. Each node in the network would have its own activation bias and would be connected with the input, or activation from the previous layer, through independent weights. Eq. 7 shows the calculated activation of a specific node i in layer l of the network.

$$a_i^l = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b^l\right) = f^l(z_i^l) \tag{7}$$

$$z_i^l = \sum_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l \tag{8}$$

This can be written in a more compact form like in Eq. 9.

$$\boldsymbol{z}^l = \left(W^l\right)^T \boldsymbol{a}^{l-1} + \boldsymbol{b}^l \tag{9}$$

The activations of the first layer go into calculating the activations of the next layer, which again go into the next, and then the next, until the activations of the output layer, $\boldsymbol{a}^L \equiv \hat{y}$, are found.

2.2.3 Backpropagation

The essence of neural networks as machine learning is *training*. This means getting the network to adjust its parameters to better fit the data. A common algorithm used to do this is the *Backpropagation algorithm*. A feed forward calculation of the network yields an output which can be compared with the true data output through a cost function. The goal is to minimise the cost. The only changes the network can do to get a different output, is to adjust its weights and biases. Backpropagation is a method of finding the gradients of the cost function with respect to these parameters. The gradients are calculated through the chain rule and for one layer at the time, starting at the last layer and propagating backwards through the network, hence the name Backpropagation. When these are found one can apply a gradient descent method for finding the parameters yielding the lowest cost and implementing them in the network. After a feed forward pass with the new parameters, the process of backpropagation, gradient descent, and updating can be repeated. This is a way of training the network.

Mathematically, finding the gradients of the individual weights, w_{ij}, and biases, b_i, for the last layer L can be done as in Eq. 10.

$$\frac{\partial C}{\partial b_i^L} = \frac{\partial C}{\partial z_i^L}\frac{\partial z_i^L}{\partial b_i^L} = \frac{\partial C}{\partial z_i^L} \equiv \delta_i^L \tag{10a}$$

$$\frac{\partial C}{\partial w_{ij}^L} = \frac{\partial C}{\partial z_i^L}\frac{\partial z_i^L}{\partial w_{ij}^L} = \delta_i^L a_j^{L-1} \tag{10b}$$

$$\delta_i^L = \frac{\partial C}{\partial z_i^L} = \frac{\partial C}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L} = \frac{\partial C}{\partial a_i^L} f'(z_i^L) \tag{10c}$$

All of these equations are independent of the cost function implemented and enables the use of automatic differentiation through tools like e.g. `Autograd`. It is also possible to calculate all of these sizes analytically, but that yields less flexibility in the neural network. By the same approach, the gradients for the layers $l = L-1, L-2, ..., 1$ can be found:

$$\delta_i^l = \frac{\partial C}{\partial z_i^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_i^l} = \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_i^l} \tag{11a}$$

$$z_k^{l+1} = \sum_{j=1}^{N_l} w_{ki}^{l+1} a_i^l + b_k^{l+1} \tag{11b}$$

$$\delta_i^l = \sum_k \delta_k^{l+1} w_{ki}^{l+1} f'(z_i^l) \tag{11c}$$

$$\frac{\partial C}{\partial b_i^L} = \delta_i^l \quad \wedge \quad \frac{\partial C}{\partial w_{ij}^L} = \delta_i^l a_j^{l-1} \tag{11d}$$

Having found the gradients of the parameters for all the layers, gradient descent can be implemented.

2.2.4 Universal Approximation Theorem

A lot of the promise of NNs comes from the fact that they are *universal approximators*. It has been shown ([Hornik et al., 1989]) that a neural network with only a single hidden layer can approximate arbitrarily well any function $f(x) : \mathbb{R}^n \to \mathbb{R}$ for any (positive) integer n. This requires the activation function of the hidden layer fulfil certain requirements: Namely being a non-constant, bounded and monotonically increasing, continuous function [Hjorth-Jensen].

2.3 Classification

Instead of predicting targets in a continuous domain ($y_i \in \mathbb{R}$), we aimed to put the target y_i in a specific class ($y_i \in \mathcal{Q}$), where $\mathcal{Q}$ is the set of possible outcomes. This is named *classification*, where in the following we will only consider binary cases. This means we will try to classify mutually exclusive properties, that is either we have the outcome O or not the outcome O ($\mathcal{Q} = \{O, \neg O\}$). We will label these as $O = 1$ and $\neg O = 0$.

Two different methods for classification will be introduced here. Firstly we will look at *logistic regression*, maybe the most common method in the domain of classification. In addition, under some restrictions, our neural network can be used for classification problems. Both of these models have an output $p(x_i)$ on the domain $[0, 1]$, which we will interpret as the probability of the outcome $y_i = 1$. We will classify the data point x_i as $\hat{y}_i = 1$ if $p(x_i) > \tau$ and $\hat{y}_i = 0$ if $p(x_i) < \tau$, where τ is the *threshold*, a priori set to $\tau = 0.5$.

2.3.1 Logistic Regression

For comparison with our neural network we introduce maybe the most common classification method there is, *logistic regression*. One might be confused by inclusion of the word *regression* in the name of a classification method. This comes from performing a fit to the *log-odds* using a function linear in the input variables:

$$\log\left(\frac{p(x_i)}{1-p(x_i)}\right) = x_i^T \boldsymbol{\theta},$$

where we have defined $p(x_i) \equiv p(y_i = 1|x_i, \boldsymbol{\theta})$, the probability that we guess $\hat{y}_i = 1$ given a data point x_i and parameters $\boldsymbol{\theta}$. Solving the above expression for $p(x_i)$ yields the Sigmoid function from Tab. 6 evaluated in $x_i^T\boldsymbol{\theta}$.

$$p(x_i) = \sigma(x_i^T\boldsymbol{\theta}) = \frac{1}{1+e^{-x_i^T\boldsymbol{\theta}}}$$

Following the approach from [Aasen et al., 2022], we have the likelihood for a Binomial distribution:

$$P(\boldsymbol{y}|X, \boldsymbol{\theta}) = \prod_{i=1}^{n} \sigma(X_{i*}\boldsymbol{\theta})^{y_i}[1-\sigma(X_{i*}\boldsymbol{\theta})]^{1-y_i},$$

where a sum over a is implied. The cost function then follows from taking logarithm of the likelihood, presented with an overall negative sign giving a minimisation problem of the parameters $\boldsymbol{\theta}$. In the literature this is often called the *Binary Cross Entropy* (BCE) cost function [Ramos et al., 2018].

$$\begin{aligned} C_{\text{BCE}}(\boldsymbol{\theta}) &= -\frac{1}{n}\log P(\boldsymbol{y}|X, \boldsymbol{\theta}) \\ &= -\frac{1}{n}\sum_{i=1}^{n}\Big\{y_i \log[\sigma(X_{i*}\boldsymbol{\theta})] \\ &\quad + (1-y_i)\log[1-\sigma(X_{i*}\boldsymbol{\theta})]\Big\} \end{aligned} \tag{12}$$

Taking the derivative wrt. to a parameter θ_b recalling the derivative of the Sigmoid function from Tab. 6 we find:

$$\begin{aligned} \frac{\partial}{\partial\theta_b}C_{\text{BCE}} &= -\frac{1}{n}\sum_{i=1}^{n}\Big\{y_i\left[1-\sigma(X_{i*}\boldsymbol{\theta})\right]X_{ib} \\ &\quad - (1-y_i)\sigma(X_{i*}\boldsymbol{\theta})X_{ib}\Big\} \\ &= \frac{1}{n}\sum_{i=1}^{n}\left[\sigma(X_{i*}\boldsymbol{\theta}) - y_i\right]X_{ib} \end{aligned} \tag{13}$$

Finally, with $p(x_i) = \sigma(X_{i*}\boldsymbol{\theta})$, we can express the gradient of the BCE cost function in vector notation.

$$\nabla_{\boldsymbol{\theta}} C_{\text{BCE}}(\boldsymbol{\theta}) = \frac{1}{n}X^T(\boldsymbol{p}-\boldsymbol{y}) \tag{14}$$

2.3.2 Neural Network Approach

Something Something since we want to interpret output as probability we must choose Sigmoid. Hidden activation functions can be whatever.

3 Method

3.1 Datasets

We will use some of the same datasets as in Project 1 [Aasen et al., 2022], namely the Franke function data and the Nicaraguan terrain data.

3.1.1 Franke Function

The Franke function $F : \mathbb{R}^2 \to \mathbb{R}$ is a two-dimensional scalar function made up of four Gaussian functions.

$$F(x,y) = f_1(x,y) + f_2(x,y) + f_3(x,y) + f_4(x,y) + \epsilon, \tag{15}$$

where

$$f_1(x,y) = \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right), \tag{16a}$$

$$f_2(x,y) = \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right), \tag{16b}$$

$$f_3(x,y) = \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right), \tag{16c}$$

$$f_4(x,y) = -\frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right), \tag{16d}$$

and we have added a noise term $\epsilon \overset{d}{\sim} \mathcal{N}\left(0, \sigma^2\right)$ with some standard deviation (std) σ.

3.1.2 Terrain Data

We have taken terrain data from the Nicaraguan mountains, sampling height data within the square defined by longitudinal coordinates [86° 33′ 20″ W, 86° 28′ 20″ W] and latitudes [13° 26′ 40″ N, 13° 31′ 40″ N]. The data has one arcsecond resolution.

3.1.3 Wisconsin Breast Cancer Dataset

For regression problems, the Wisconsin Breast Cancer data set [Dua and Graff, 2017] was used. Tumour features are computed from digitized images, describing characteristics of the cell nuclei present in the image. In total there are $n = 569$ unique instances, where each instance has the respective tumour marked as either malignant (M) or benign (B). We aimed to classify the tumour as M or B based on 10 different attributes ranging from tumour radius to texture. Each of these 10 attributes were again split up into three different measurements, named mean, standard error and worst. The latter describes the mean of the three largest measurements. This data set serves a good real-world benchmarking set, containing not too many instances and no missing fields.

3.1.4 Data Preparation

All the data we handle, we will scale before using it for training and testing our algorithms. We will use the Standard scaling detailed in [Aasen et al., 2022]. This ensures that features are all of the same order of magnitude, making the mean zero and std 1 for all of them individually. In the regression problems, we will also scale the observation data $\boldsymbol{y}$ too, whereas this is not done in the binary classification case.

3.2 Assessing Gradient Descent

Our first analysis was a comparison of the various GD algorithms, as these make the basis for our neural network optimisation. We used the GD algorithms to optimise OLS and Ridge cost functions [Aasen et al., 2022], training them on the Franke function with 600 data points, with a train-test split of 3/4, and a noise term added with std $\sigma = 0.1$. This was done in anticipation of applying our Neural Network on terrain data, which the Franke function imitates.

3.2.1 OLS Regression

First we focused on the OLS cost function

$$C_{\text{OLS}}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(X_{i*}\boldsymbol{\theta} - y_i\right)^2, \tag{17}$$

with a polynomial expansion of degree 5 of the Franke function arguments x, y. We plot the MSE resulting from the predictions of the optimised parameters on the validation data as a function of learning rates. We looked at learning rates from close to zero to where 'exploding gradients' occurred, which was different between the different algorithms.[2]

Our preliminary analysis was a comparison of ordinary GD and stochastic GD, with and without momentum. We used 500 iterations for the ordinary GD, and 500 epochs with a batch size of 200 for the stochastic GD. For the momentum methods we used a hyperparameter value $\gamma = 0.8$. Furthermore, we investigated the effect of increasing the number of epochs or batches with plain SGD, using 2000 epochs and a batch size of 50.

Then we looked at algorithms that tune the learning rate, implementing ordinary and stochastic AdaGrad, again with and without momentum. Then we moved on to application of stochastic RMSprop and Adam, and compared how these do in comparison to AdaGrad with SGD.

To get an overview over how the algorithms converge, we finally plot the validation MSE epoch by epoch for GD with momentum, ordinary SGD, SGD with momentum and Adam SGD, up to epoch 500 with a batch size of 200. Here we used the best learning rates from our OLS analysis for each algorithm.

3.2.2 Ridge Regression

Adding an L_2-penalisation term to the OLS cost function gives us the Ridge cost function

$$C_{\text{ridge}}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(X_{i*}\boldsymbol{\theta} - y_i\right)^2 + \lambda\|\boldsymbol{\theta}\|_2^2. \tag{18}$$

[2]Exploding gradients happen when the learning rate is large enough to overshoot the minima, and then start oscillating wildly away from the minima; blowing up.

We selected the four momentum based algorithms from the OLS analysis to see how they responded to changing of learning rate and λ-parameter in predicting the validation data. Again we used 500 iterations for the GD method, and 500 epochs with batch size 200 with the SGD methods. Varying $\lambda \in [10^{-8}, 10^{1}]$, we compared first with a fixed learning rate between GD and SGD with momentum ($\gamma = 0.8$). Then we used a tunable learning rate and SGD with AdaGrad with momentum ($\gamma = 0.8$) and finally Adam ($\beta_1 = 0.9, \beta_2 = 0.99$).

3.2.3 Initialisation of Descent Problems

GD methods are quite sensitive to the initial parameters from which to descend, so made sure to initialise all the methods from the same point when comparing performance. We always initialised with randomly drawn parameter values from the standard normal distribution with mean zero and std one. Since we always deal with scaled features, we can safely initialise the parameters with a standard normal distribution without fear of the dimension of the parameters being terribly out of sorts. To further combat the sensitivity to initial conditions, we did all the descents from five different starting points, and computed the average MSE scores we got.

3.3 Creating a Neural Network and Regression Problems

We started by creating an FFNN with a flexible number of hidden layers and nodes as well as variable learning rate, multiple possible activation functions, and with an optional regularisation parameter. Subsequently, we varied these hyperparameters, then trained the various networks and tested them on the Nicaraguan terrain data from Project 1. We compared the best network's performance with the results from the linear regression methods (OLS and Ridge) employed in that project. Our dataset consisted of 600 datapoints in total, and we used a test-train-split of 3/4, meaning our training data was 450 datapoints.

3.3.1 Initialisation of the Network

For optimising the neural networks, we employed SGD with Adam and 500 epochs with a batch size of 200. This configuration resulted from our initial analysis of the GD algorithms in our library.

In the neural network we chose to implement MSE as the cost function, C, for the output. We started by setting the activation of the hidden layers to sigmoid, and since we wanted to perform regression, we selected the identity function for the final output layer. Initially we set the regularisation parameter to zero.

When setting up a neural network, the weights and biases need initial values. Choosing these values can affect how the network trains and eventually performs. From

Eq. 10 we see that setting all the parameters to 0 means the network will not backpropogate initially and all the nodes will yield the same output. In this project we have initialised the weights with a distribution of mean zero and std 1. We set the initial biases to 0.01. Since the mean of the initialised weights is 0, setting the initial biases to a small value like this ensures every z_i^L has a value to backpropogate in the first iteration.

3.3.2 Varying the Network Architecture and Learning Rate

We wanted to explore how our neural networks vary depending on their structure. We therefore varied the number of layers, number of nodes in each layer, and the learning rate while computing the measures for goodness of fit. In this case we implemented MSE. See Project 1 for elaboration on these types of measures [Aasen et al., 2022]. We chose to only evaluate somewhat symmetrical networks, meaning we used the same number of nodes in all the hidden layers.

To have a compact way of referring to the various network structures we have created, we now introduce a new notation. A network will hereby be referred to as $\mathbf{N}_{nodes}^{layers}$ with the number of hidden layers as a superscript and number of nodes in the hidden layer as a subscript. E.g. a network with 5 hidden layers and 100 nodes in each of those layers will be referred to as $\mathbf{N}_{100}^{5}$.

We looked at 1, 3 and 5 number of hidden layers, with number of nodes between: 2-200, and learning rates in the range: 0.8-0.005. We created three heatplots; one for each #layers.

To verify whether our implementation is effective, we compared our performance with that of a neural network from `scikit-learn` [Pedregosa et al., 2011] with the same hyperparameters as our networks. In that case we chose to look at just one architecture with 3 hidden layers.

3.3.3 Exploring Activation Functions in Hidden Layers

As activation functions can largely affect the performance of the network, we explored various implementations. We started by fitting networks to a 1D-data of the polynomial x^2 with noise (normal distribution with scale=0.1) and plotted the results. We used 600 datapoints and train-test-split of $^3/_4$. We did this to see the shapes of the approximations of the different activation functions.

Next we made similar heatplots as before, again training and testing on the terrain data, but now employing ReLU and leaky ReLU.

3.3.4 Implementing Regularisation

Penalising large weights can hinder them from growing arbitrarily large and therefore prevents overfitting. We added the regularisation hyperparameter to the gradients of the weights. If we had implemented a stopping criteria, we would also have included it in the cost function.

Firstly we chose the best network from the previous explorations: $\mathbf{N}_{200}^{5}$, with sigmoid and $\eta = 0.001$. We employed penalisations, λ, in the range 10^{-9} to 10^{-1} to this network and plotted the validation MSE as a function of these λs.

Secondly we used the same network, but now also varied the learning rate, η. We made a heatplot of the various ηs, λs and the R2 score these networks produced.

3.3.5 Terrain Data and Comparison with OLS

Lastly we compared the optimal network with an OLS-model like the ones made in Project 1. From the previous explorations, we had found that the network with the best scores for this data was our implementation of $\mathbf{N}_{200}^{5}$, with sigmoid activation function, $\eta = 0.01 \text{and} \lambda = 10^{-4.1}$. We trained and tested this network and plotted its prediction of the Nicaraguan terrain. We compared the MSE and plot of the network with an analytical optimalisation of the OLS cost function of a linear regression with a polynomial expansion of degree 11. We also plotted the true data for a qualitative comparison.

3.4 Classification Problems

For our case using the Wisconsin Breast data set presented in Sec. 3.1.3, we aimed to classify either M or B. These categories are mutually exclusive, thus determining M or not M is sufficient (i.e. $\mathcal{Q} = \{M, \neg M\}$). This gave us a binary problem, represented with $y_i \in \{1, 0\}$ where $y_i = 1$ implies M and $y_i = 0$ not M.

To measure the goodness of fit in for our classification problems, we used the *accuracy* score. This metric simply counts the number correctly classified cases, divided by the total number of cases:

$$A(y_i, \hat{y}_i) = \frac{1}{n} \sum_{i=1}^{n} I(\hat{y}_i = y_i), \tag{19}$$

$$I(\hat{y}_i = y) = \begin{cases} 1 & \text{if } \hat{y}_i = y \\ 0 & \text{else} \end{cases},$$

where I is the indicator function, and $\hat{y}_i \in \{1, 0\}$ is our models classification based on the probability $p(x_i)$ and the threshold τ. We note that if all cases are classified correctly, the sum will evaluate to n and if all are classified wrongly, it will evaluate to 0. Thus, the accuracy is bounded by $0 \leq A(y_i, \hat{y}_i) \leq 1$. This will be calculated using the test set from a $^3/_4$ train-test split for both the neural network and logistic regression.

3.4.1 Logistic Regression

The optimal parameters for our BCE cost function Eq. 12 can be found by the various optimisation schemes introduces in Sec. 2.1. Taking the BCE cost function, we add an L_2-penalisation to have some constraint on the parameter sizes. This gives the cost function

$$C_{\text{CLF}}(\boldsymbol{\theta}) = C_{\text{BCE}}(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_2^2. \tag{20}$$

Initially, we investigated how different optimisation methods from the GD and SGD family performed for a range of different learning rates, using the accuracy as scoring. This was done without any penalisation, that is $\lambda = 0$ in Eq. 20. GD and mGD ($\gamma = 0.8$) was tried, using 5 and 50 iterations. The SGD based methods Adam SGD ($\beta_1 = 0.9, \beta_2 = 0.99$) and AdaGrad SGD was tried with 5 epochs, while plain SGD was tried with both 5 and 50 epochs, all using 200 batches. For easier comparisons, all optimastation methods was tried with learning rates $\eta \in [0.01, 1.0]$.

Thereafter, to get a better view of how many iterations/epochs was required to achieve a good model, the validation accuracy as a function of iterations/epochs was plotted for different models with a fixed learning rate. Concretely, GD with ($\gamma = 0.8$) and without momentum was tried, in addition to SGD, Adam SGD ($\beta_1 = 0.9, \beta_2 = 0.99$) and AdaGrad SGD. The learning rate was set to $\eta = 0.05$ for all methods, with iterations/epochs in the range $[0, 200]$. SGD based methods all used a batch size of 200.

To see how penalisation influenced the model, the validation score was calculated for a range of different penalisation parameters λ and learning rates η. We used five logarithmically spaced penalisation parameters $\lambda \in [10^{-6}, 1]$ and five linearly spaced learning rates $\eta \in [0.001, 0.5]$. Lastly, we used `scikit-learn`'s implementation of logistic regression to compare with the results obtained using our implementation.

3.4.2 Neural Network Approach

To use our neural network for classification problems, we must apply some restrictions to our neural network. As discussed, since we classify two mutually exclusive outcomes, a single output neuron will be sufficient. Since we wish to interpret this output as the probability for M, the output activation function must be bounded by $f(x) \in [0, 1]$. Look at our activation function repertoire from Tab. 6, the only applicable functions (without any modifications) is the Sigmoid function. Assuming that the targets we wish to predict are drawn from a normal distribution does not resonate with what we know about classification problems. Therefore, using the OLS/Ridge cost function is not feasible. For this binary problem, a cost function derived form a Bernoulli distribution would be more feasible. The BCE cost function Eq. 20 is a better choice in this case.

Despite these restrictions, there are a plethora of components to tweak. Initially, we began investigating how different hidden activation functions performed as a function of learning rate, using eight linearly spaced points $\eta \in [0.10, 0.40]$. Specifically, we used Sigmoid, Hyperbolic tangent, ReLU and Leaky ReLU from Tab. 6. This analysis was repeated for four different network structures, $\mathbf{N}_5^1$, $\mathbf{N}_{10}^1$, $\mathbf{N}_5^2$ and $\mathbf{N}_5^3$. Optimisation was done using the overall best GD/SGD method found from our initial tests.

Taking complexity and performance into account, we chose two of these structures with two different activation functions and performed an analysis using the L_2 penalisation. By varying both λ and η as hyperparameters, we saw how different network structures behaved with constraints on weight sizes. We used five logarithmically spaced penalisation parameters $\lambda \in [10^{-8}, 10]$ and five linearly spaced learning rates $\eta \in [0.001, 0.1]$ for the hyperparameter variation. Lastly, by choosing a specific structure, a comparison with `scikit-learn` was performed. Here both learning rate and the penalisation parameter α (not corresponding to our λ) was varied.

4 Results

4.1 Gradient Descent Analysis

First we looked at our library of gradient descent methods to figure out what and how they worked. This with the goal of finding a descent recipe for optimising our neural networks down the line.

4.1.1 OLS Optimisation

All plots regarding the optimisation of the OLS cost function are found in Fig. 5. The best MSE scores and learning rates are tabulated in Tab. 1. We found that the none of the algorithms come really close to the analytical OLS solution, found by matrix inversion, with a validation MSE of 0.1176. The best performing algorithms were SGD with Adam (MSE 0.1660 ± 0.0018), SGD with momentum (MSE 0.1729 ± 0.0022) and GD with momentum (MSE 0.1775 ± 0.0026).

In Fig. 5a we see that both the non-stochastic and stochastic algorithms performed better when momentum was added, which helped avoid local minima to converge faster and prevent exploding gradients. Generally, the stochastic algorithms had tighter confidence intervals, signalling that they are not as sensitive to the initial conditions as the plain algorithms.

Exploring the increase in epochs and decrease in batch size in Fig. 5b, we see that decreasing batch size made the algorithm less stable, and more easily prone to exploding gradients. Increasing the number of epochs improved the

result, and made increased the convergence, but did not affect stability.

Adding the tuning of the learning rate with AdaGrad, we see in Fig. 5c a marked increase in stability. However, the algorithms struggled to converge, with the momentum based algorithms doing best; SGD slightly bettering the GD results. However, even AdaGrad SGD with momentum could not beat the ordinary momentum based GD and SGD algorithms.

Introducing RMSprop and Adam to tune the learning rate provided algorithms with a tunable learning rate that converged faster than AdaGrad, seen in Fig. 5d. Instead of exploding, the MSE of RMSprop gradually increased from around $\eta = 0.1$. Ultimately, it performed similarly to the AdaGrad algorithms, with an optimal MSE of 0.1863. The momentum based Adam algorithm performed the best, with an optimal MSE of 0.1660; but performing well from very low η-values up to around $\eta = 0.35$. After this point Adam was markedly less stable than the AdaGrad algorithms. Aside from converging faster, it was also notable that both RMSprop and Adam were much less sensitive to initial conditions, with generally much tighter confidence intervals.

Fig. 4 shows how GD with momentum, plain SGD, SGD with momentum and Adam SGD converged by epoch. The non-stochastic method converged smoothly, and was very stable. Adding stochasticity clearly exhibited unstable convergence, but overall trended downwards faster. Notably, SGD with momentum was quite explosive, whereas Adam managed to implement stochasticity and momentum in a much more stable way.

Method	MSE	η
Analytic	0.1377	-
Plain GD	0.1982 ± 0.0061	0.072
Momentum GD	0.1775 ± 0.0026	0.13
Plain SGD	0.1862 ± 0.0028	0.069
Momentum SGD	0.1729 ± 0.0022	0.12
AdaGrad GD	0.2135 ± 0.012	0.50
AdaGrad Momentum GD	0.1836 ± 0.0024	0.52
AdaGrad SGD	0.1906 ± 0.0038	0.50
AdaGrad Momentum SGD	0.1822 ± 0.0020	0.47
RMSprop SGD	0.1859 ± 0.0027	0.019
Adam SGD	0.1660 ± 0.0018	0.32

Table 1: Table of the best validation MSE scores by GD algorithm, together with the learning rate that produced the best result.

4.1.2 Ridge Optimisation

The results from the ridge analysis are found in Fig. 6, and a summary is tabulated in Tab. 2.

We did not see much change in the performance of the algorithms from the OLS optimisation. There was in fact a slight decrease in the optimal MSE of all the

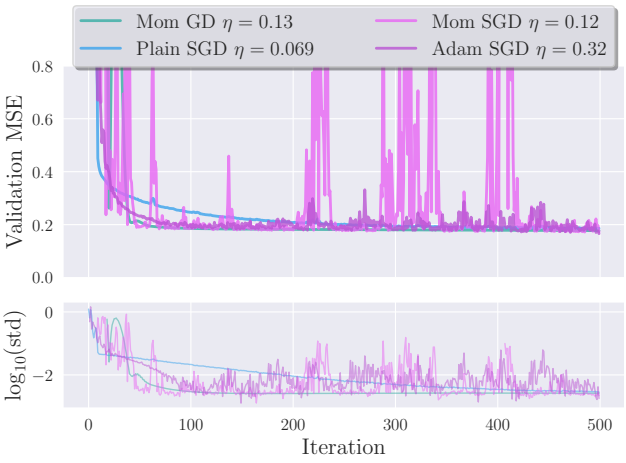


Figure 4: Plot of the mean validation MSE of four a linear regression on Franke function data as optimised with GD algorithms after every epoch up to 500. Underneath, the logarithm of the std of the mean across descent from 5 starting points is shown. The momentum methods used $\gamma = 0.8$, Adam had $\beta_1 = 0.9, \beta_2 = 0.99$ and the stochastic methods all used a batch size 200.
Best MSEs were: GD w/momentum (0.1775 in iteration 500), Plain SGD (0.1862 in iteration 495), SGD w/momentum (0.1729 in iteration 486), SGD Adam (0.1660 in iteration 500).

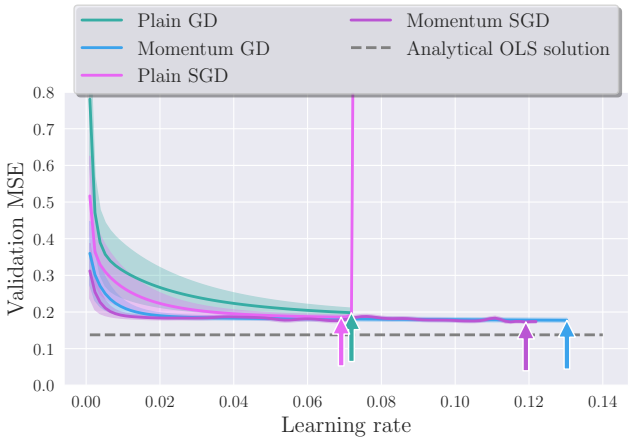
algorithms except AdaGrad with momentum, however, the std of the MSEs generally went down. There was no universal λ-value that gave the best results. It is worth noting that there was little effect on where the exploding gradients occur; in fact with larger λ-values they occur earlier for both GD and SGD with momentum.

The best performing algorithm was again SGD with Adam to tune the learning rate (MSE of 0.1673 ± 0.0020), as before with the OLS cost function.

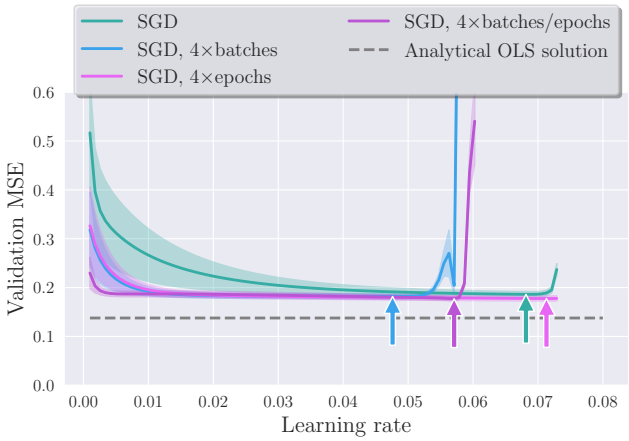
To see whether ridge actually had the potential to do a difference, we plotted the validation MSE of the analytical ridge solution as a function λ against the analytical OLS solution in Fig. 7. We see that the ridge solution does actually perform better than OLS between $\lambda \in [0.0001, 0.01]$, meaning that there is potential for improvement with the ridge penalisation.

Method	MSE	η	λ
Analytic	0.1338	-	0.002
Momentum GD	0.1777 ± 0.0025	0.13	10^{-4}
Momentum SGD	0.1747 ± 0.0020	0.12	10^{-8}
AdaGrad Momentum SGD	0.1822 ± 0.0013	0.48	10^{-3}
Adam SGD	0.1673 ± 0.0020	0.32	10^{-5}

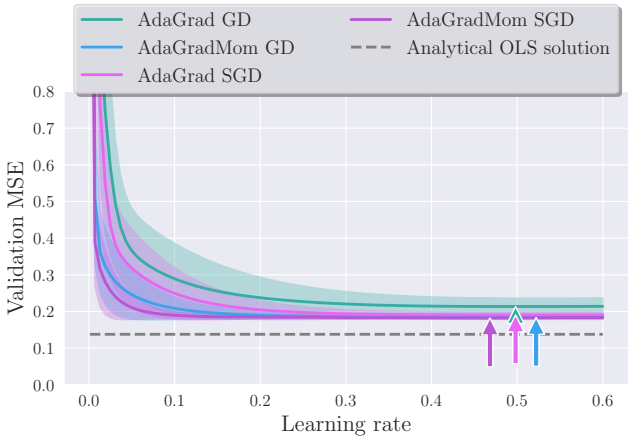
Table 2: Summary of the best validation MSEs resulting from the optimisation of the ridge cost function using various GD algorithms. The data is taken from the plots in Fig. 6.



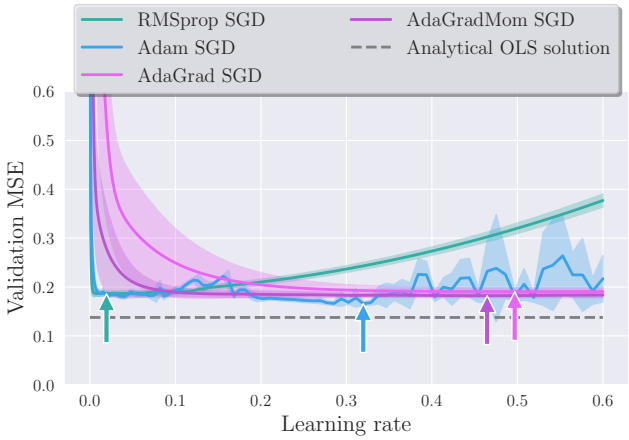
(a) The minimal MSEs by algorithm: plain GD 0.1982 ($\eta = 0.072$), GD w/momentum 0.1775 ($\eta = 0.13$), plain SGD 0.1862 ($\eta = 0.069$), SGD w/momentum 0.1729 ($\eta = 0.12$)



(b) The minimal MSEs of plain SGD by number of epochs and batch size: 500 epochs/200 batch size 0.1862 ($\eta = 0.068$), 500 epochs/50 batch size 0.1819 ($\eta = 0.0026$), 2000 epochs/64 batch size 0.1776 ($\eta = 0.071$), 2000 epochs/50 batch size 0.1773 ($\eta = 0.057$)

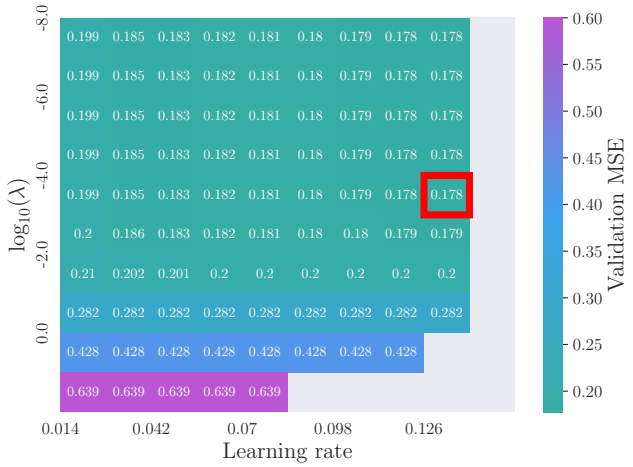


(c) The minimal MSEs by algorithm: AdaGrad GD 0.2135 ($\eta = 0.50$), AdaGrad GD w/momentum 0.1836 ($\eta = 0.52$), AdaGrad SGD 0.1906 ($\eta = 0.50$), AdaGrad SGD w/momentum 0.1822 ($\eta = 0.47$)

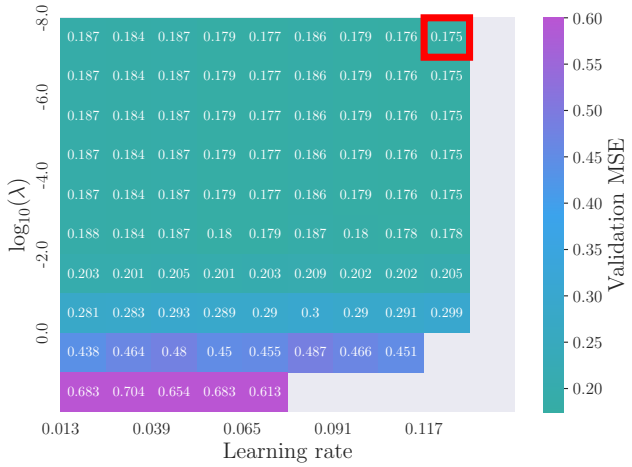


(d) The minimal MSEs by algorithm: RMSprop SGD 0.1859 ($\eta = 0.019$), Adam SGD 0.1660 ($\eta = 0.32$), AdaGrad SGD 0.1906 ($\eta = 0.50$), AdaGrad SGD w/momentum 0.1823 ($\eta = 0.46$)

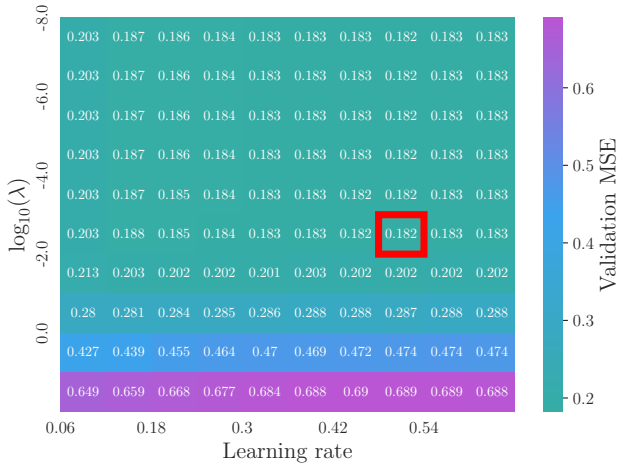
Figure 5: Plots of the validation MSE of the parameters found from optimising the OLS cost function on Franke function data with $n = 600$ data points with a train test split of $^3/_4$. For the momentum methods we used $\gamma = 0.8$, for RMSprop we used $\beta = 0.9$ and for Adam we used $\beta_1 = 0.9, \beta_2 = 0.99$. The stochastic methods used a batch size of 200 and 500 epochs, while the standard GD did 500 iterations unless specified otherwise. Overlaid are 95% confidence intervals based on optimising with five different starting points. Exploded gradients are clipped out of the plot. The analytical OLS solution achieved an MSE of 0.1377.



(a) GD with momentum. Best MSE value was 0.1777 with $\lambda = 10^{-4}, \eta = 0.13$.



(b) SGD with momentum. Best MSE value was 0.1747 with $\lambda = 10^{-8}, \eta = 0.12$.



(c) SGD AdaGrad with momentum. Best MSE value was 0.1822 with $\lambda = 10^{-3}, \eta = 0.48$.

(d) SGD Adam. Best MSE value was 0.1673 with $\lambda = 10^{-5}, \eta = 0.32$.

Figure 6: Plots of the validation MSE of the parameters found from optimising the ridge cost function on Franke function data with $n = 600$ data points with a train test split of $^3/_4$. For the momentum methods we used $\gamma = 0.8$ and for Adam we used $\beta_1 = 0.9, \beta_2 = 0.99$. The stochastic methods used a batch size of 200 and 500 epochs, while the standard GD did 500 iterations unless specified otherwise. Overlaid are 95% confidence intervals based on optimising with five different starting points. Exploded gradients are clipped out of the plot.

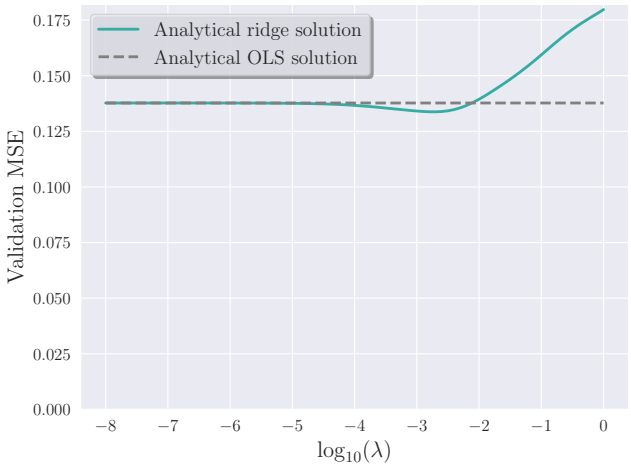


Figure 7: Plot of the validation MSE found from optimising our linear regression parameters to the ridge cost function as a function of the penalisation parameter λ. The optimal MSE is 0.1338 with $\lambda = 0.002$.

# Layers	Best network	η	MSE
1	$\mathbf{N}_{50}^{1}$	0.8	0.13
3	$\mathbf{N}_{200}^{3}$	0.01	0.094
5	$\mathbf{N}_{200}^{5}$	0.001	0.088
3, SciKit	$\mathbf{N}_{10}^{1}$	0.08	0.18

Table 3: Table of the best networks' MSE value.

4.2 Neural Network and Regression

4.2.1 Varying Network Architecture and Learning Rate

When varying the number of hidden layers (1,3,5), nodes in each layer (between 2-200), and learning rate (0.8-0.005), we found that the best MSE results, for each # layers, came from the networks in table Tab. 3.

See Fig. 8 for all results in heatmap. We see that increase in the #layers leads to more networks that are unable to converge (see high MSE-values).

4.2.2 Exploring Activation Functions

The results in Fig. 9 show that the implementation of activation functions affects the networks' performance. Fig. 9a illustrates that the shape of a network's predicted graph shares similarities with the graph of the implemented activation function; identity is linear, sigmoid and tanh are rather smooth, and ReLU and leaky ReLU are more discontinuous. The MSE values of the predictions are given in Tab. 4.

We notice that in the implementation of our network, the optimal network is still $\mathbf{N}_{200}^{5}$ with the sidmoid activation function (from above). However, the `scikit-learn` improves when changing from sigmoid to ReLU. There

Activation Function	Validation MSE
identity	0.113
sigmoid	0.013
tanh	0.014
ReLU	0.013
leaky ReLU	0.013

Table 4: Table of the MSE values from the predictions in Fig. 9a.

are now more non-converging networks and occurence of exploding gradients.

4.2.3 Exploring Regularisation

From Fig. 10a the λ yielding the lowest MSE value was $\lambda = 10^{-4.1}$ giving an MSE of 0.078. This was for the network with $\eta = 0.001$. When also varying the learning rate (see Fig. 10b), the optimal network also had $\eta = 0.001$ and now $\lambda = 10^{-4}$.

4.2.4 Terrain Data

In Fig. 11 the plots of the true and predicted Nicaraguan terrains are presented. The MSE of the network-predicted was 0.078 and of the OLS model 0.171. We saw, qualitatively and quantitatively, that the network outperformed the linear regression. By eye, it seemed also that the network replicated many of the key elements of the true terrain data.

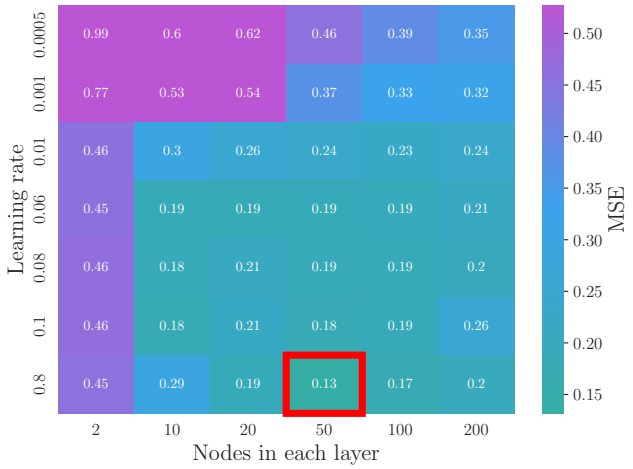
4.3 Wisconsin Breast Cancer

4.3.1 Logistic Regression

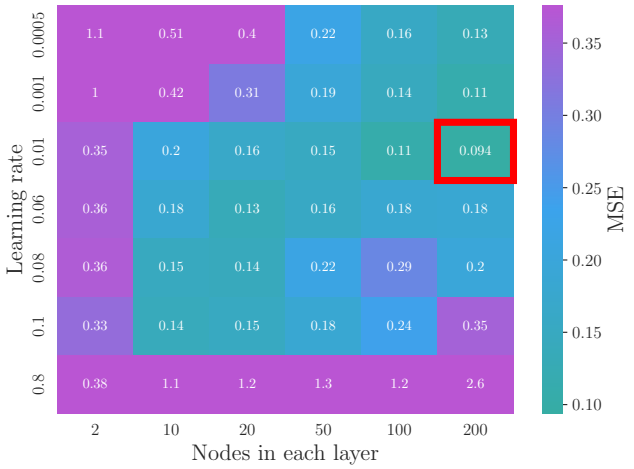
The different optimisation algorithms from our library of gradient decent methods were applied to optimise the BCE cost function without penalisation Eq. 12 using Logistic regression. This is presented in Fig. 12a for GD methods with and without momentum, while SGD based methods are presented in Fig. 12b. We found that every method, given the correct learning rate and number of iterations, managed to obtain an accuracy above 0.9.

Plain gradient decent running 5 iterations performed the worst, at best acheiving an accuracy of 0.958 at $\eta = 0.859$. However, when allowed to run for 50 iterations, accuracy scores > 0.95 were achieved across all learning rates. The addition of momentum made the optimisation converge faster, where the case for 50 iterations is especially stable. In the domain of SGD based methods, both Adam and AdaGrad SGD out-competed plain SGD for 5 epochs. However, when allowed to run for 50 epochs, plain SGD also managed to converge acheiving accuracy scores > 0.95 across all learning rates.

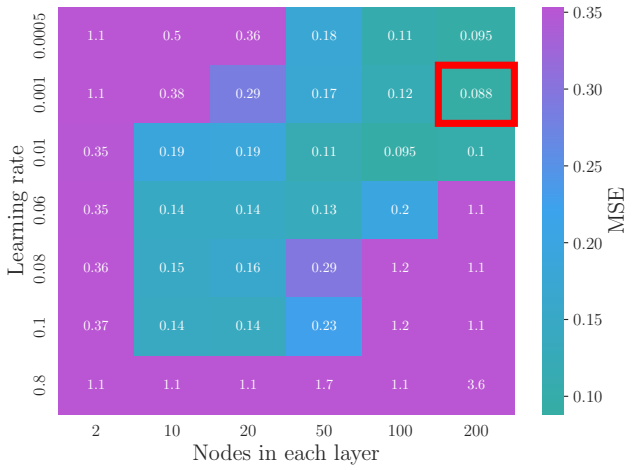
Validation accuracies calculated while optimising using GD and SGD based methods are presented in Fig. 12c. Adam SGD and mGD achieved an accuracy of 0.95 using



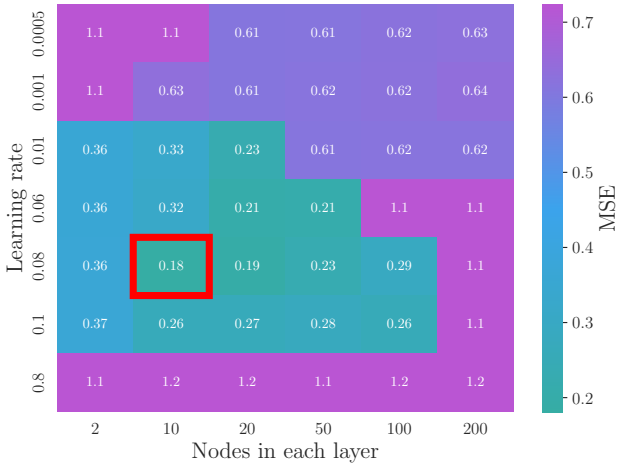
(a) 1 Layer The best result from $\mathbf{N}_{50}^{1}$ MSE = 0.13.



(b) 3 Layers The best result from $\mathbf{N}_{200}^{3}$ MSE = 0.094.

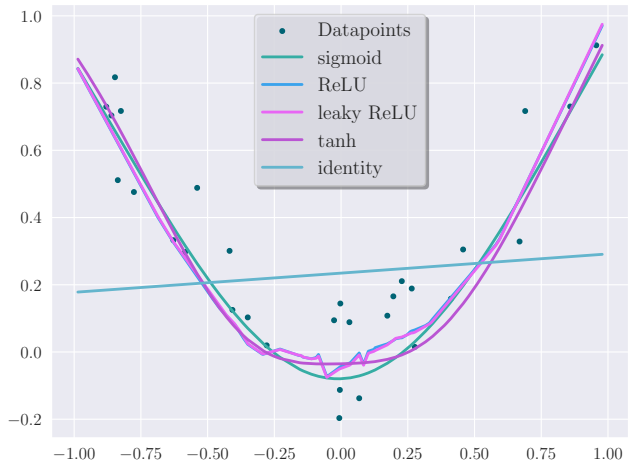


(c) 5 Layers The best result from $\mathbf{N}_{200}^{5}$ MSE = 0.088.

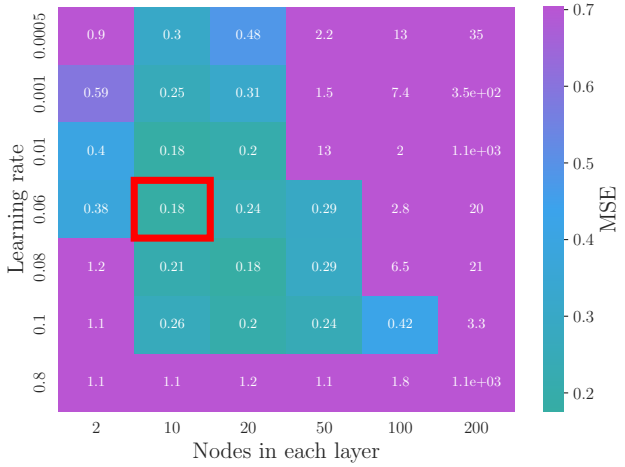


(d) 3 Layers SciKit-Learn The best result from $\mathbf{N}_{10}^{3}$ MSE = 0.18.

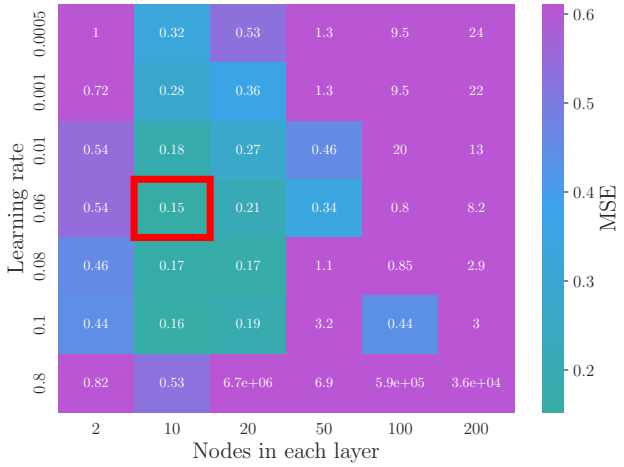
Figure 8: Heatmaps of the various networks constructed varying number nodes in each layer and learning rate for different number of layers.



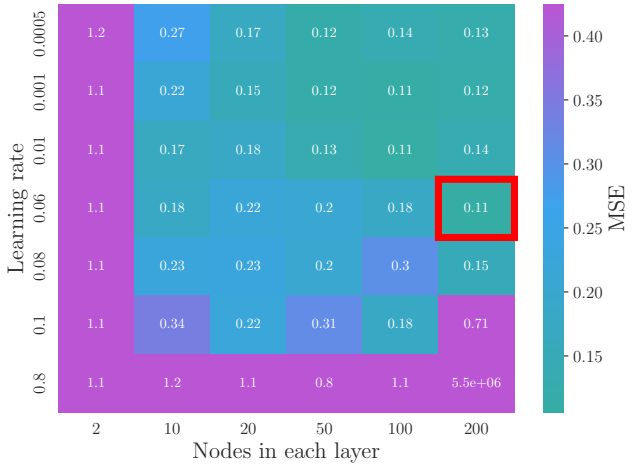
(a) Networks with various activation functions trained $x^2 + \epsilon$ ($\epsilon \overset{d}{\sim} \mathcal{N}(0, 0.1^2)$) with $x \in [-1, 1]$ plotted.



(b) 3 layers, ReLU activation function implemented. Best MSE value is 0.18.

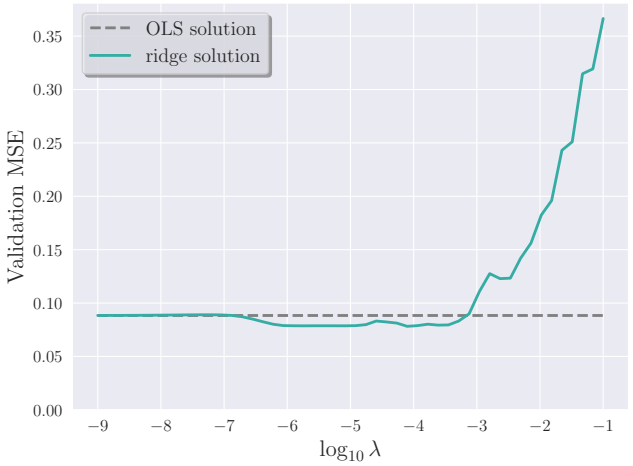


(c) 3 layers, Leaky ReLU activation function implemented. Best MSE value is 0.15.

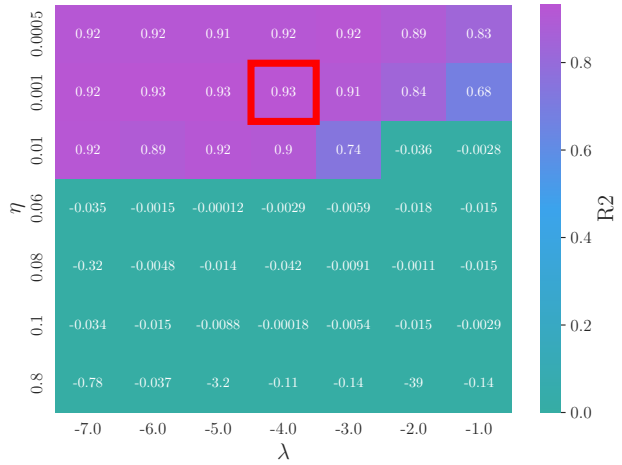


(d) 3 layers, SciKit-Learn with ReLU activation function implemented. Best MSE value is 0.11.

Figure 9: Plots from exploration of the activation functions.

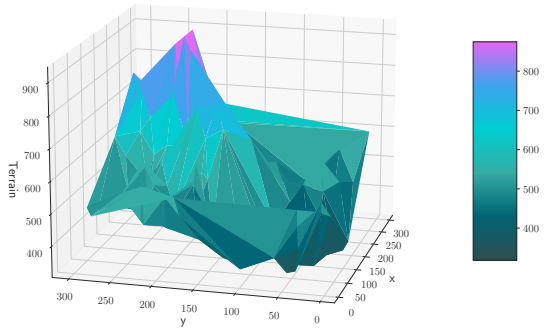


(a) Plot of the validation MSE as a function of the regularisation, λs. $\mathbf{N}_{200}^5$, with sigmoid and $\eta = 0.001$.

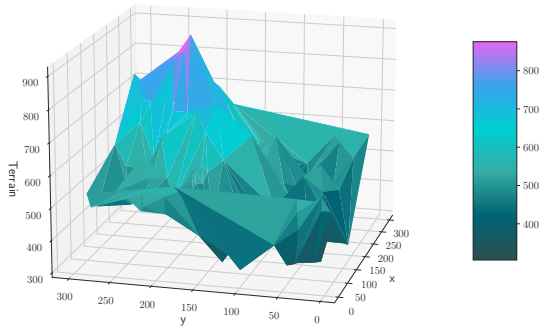


(b) Heatmap of various ηs, λs and the produced R2 score. $\mathbf{N}_{200}^5$, with sigmoid and $\eta = 0.001$.

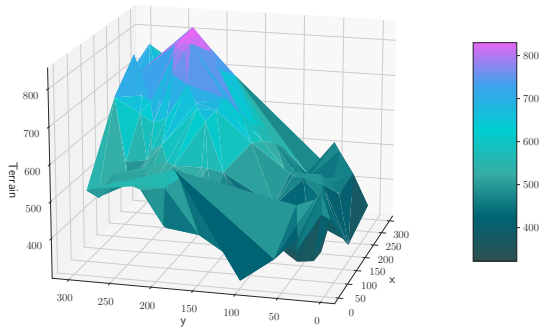
Figure 10: Results from exploring an added regularisation hyperparameter.



(a) True data from the Nicaraguan terrain.



(b) Network-predicted Nicaraguan terrain. MSE = 0.078



(c) OLS-predicted Nicaraguan terrain. $p = 11$. MSE= 0.171

Figure 11: Plots of the true and predicted Nicaraguan terrains.

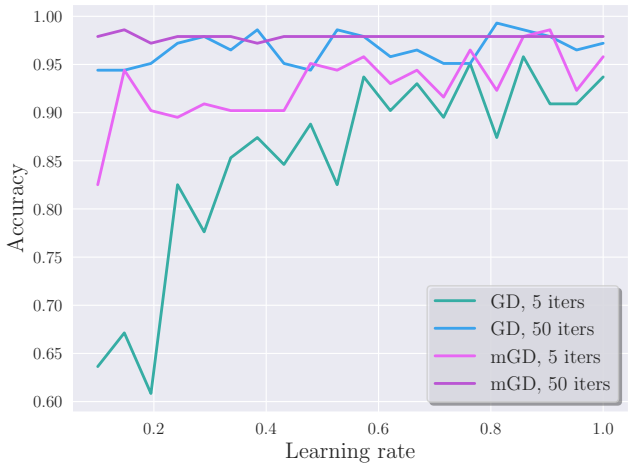
just a few iterations, while again GD and SGD performed the worst.

Using Adam SGD with $\beta_1 = 0.9, \beta_2 = 0.99$ as an optimisation algorithm, the validation accuracy varied for both penalisation λ and learning rate η is shown in Fig. 13. We found that multiple models gave the same optimal validation accuracy, for both low penalisation ($\lambda = 10^{-6}$) and higher penalisation ($\lambda = 10^{-1.5}$). The ($\lambda = 10^{-6}$) used a learning rate of $\eta = 0.13$ while the $\lambda = 10^{-1.5}$ models used higher learning rates in the range $\eta \in [0.25, 0.5]$. When using the logistic regression implementation from `scikit-learn`, we found a validation accuracy score of 0.986 without any penalisation, which lay in the vicinity of the results obtained from our own implementation.

4.3.2 Classification Neural Network

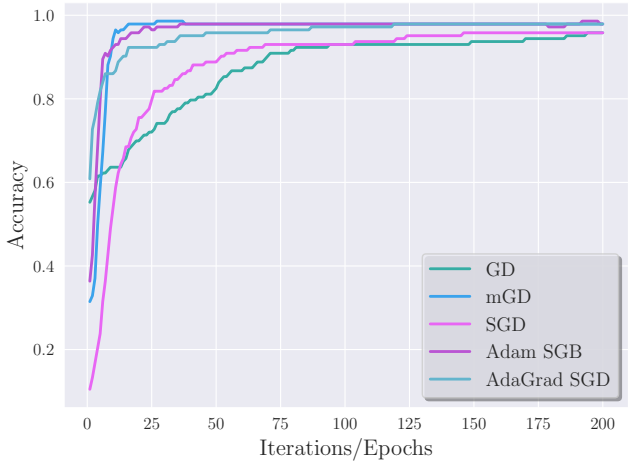
The validation accuracy of the network models $\mathbf{N}_5^1$, $\mathbf{N}_{10}^1$, $\mathbf{N}_5^2$ and $\mathbf{N}_5^3$ using different hidden activation functions is shown in Fig. 14, with the optimal accuracy scores presented in Tab. 5. For $\mathbf{N}_5^1$ Fig. 14a, the sigmoid function seemed always to perform the best, with only Leaky ReLU matching its best accuracy of 0.993 for $\eta = 0.23$. From Fig. 14b, $\mathbf{N}_{10}^1$ also performed the best using the sigmoid function, with hyperbolic tangent being the second best. For this structure, both ReLU and Leaky ReLU performed worse across all learning rates tested here. The two and three layer structures $\mathbf{N}_5^2$ and $\mathbf{N}_5^3$ seen in Fig. 14c and Fig. 14d respectively was less clear on the preferred hidden activation function. $\mathbf{N}_5^2$ seemed to be more learning rate dependent when choosing a hidden activation function, where all four functions tried here yielded an accuracy of 0.993 for different learning rates. Increasing the complexity to $\mathbf{N}_5^3$ did not yield any increase in validation accuracy, but the results from the sigmoid function was particularly stable in this case.

Due to the stable performance of the $\mathbf{N}_5^1$ structure, we chose to include this in the penalisation optimisation. To include slightly deeper structure, $\mathbf{N}_5^2$ was also chosen, since there were no significant increase in validation accuracy when moving to $\mathbf{N}_5^3$. Using the sigmoid and hyperbolic tangent hidden activation functions, the λ and η optimisation for these two structures is shown in Fig. 15. The $\mathbf{N}_5^1$ structure had its best validation accuracy when using the sigmoid function for learning rates $\eta = 0.08, 0.1$ with penalisations $\lambda = 10^{-3.5}, 10^{-5.75}$ and $10^{-8.0}$ as seen in Fig. 15a. The slightly lower accuracy of 0.986 was found using hyperbolic tangent function as seen in Fig. 15b, with optimal parameters $\lambda = 10^{-5.75}, 10^{-8.0}$ for $\eta = 0.05$ and $\lambda = 10^{-3.5}$ for $\eta = 0.08$. When moving to the structure $\mathbf{N}_5^2$, both the sigmoid and hyperbolic tangent functions yielded an accuracy of 0.993 as seen in Fig. 15c and Fig. 15d respectively. For the sigmoid, five sets of hyperparameters gave this accuracy: $\lambda = 10^{-8.0}, 10^{-5.75}$ for $\eta = 0.03, 0.05$ in addition to $\lambda = 10^{-5.75}$ for $\eta = 0.1$. The hyperbolic tangent only



(a) Showing accuracy as a function of learning rate, optimised using GD with ($\gamma = 0.8$) and without momentum for 5 and 50 iterations.

(b) Showing accuracy as a function of learning rate, optimised plain, Adam ($\beta_1 = 0.9, \beta_2 = 0.99$) and AdaGrad SGD while varying the number of epochs. For all optimisations, a batch size of 200 was used.



(c) Showing accuracy as a function of iterations/epochs. All methods used $\eta = 0.05$, with Adam having $\beta_1 = 0.9, \beta_2 = 0.99$ and momentum GD $\gamma = 0.8$.

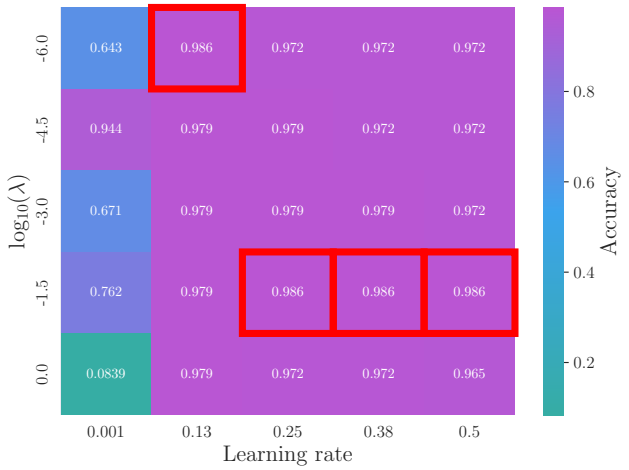


Figure 13: Validation accuracy for logistic regression as a function of learning rate and penalisation parameter λ. Largest accuracy of $A = 0.986$ found at $\lambda = 10^{-6}$ using $\eta = 0.13$ and $\lambda = 10^{-1.5}$ with $\eta = 0.25, 0.38$ and 0.50

$\mathbf{N}_5^1$	$\mathbf{N}_{10}^1$	$\mathbf{N}_5^2$	$\mathbf{N}_5^3$	f_H
0.993	0.993	0.993	0.986	sigmoid
0.986	0.986	0.993	0.993	Tanh
0.986	0.979	0.993	0.986	ReLU
0.993	0.979	0.993	0.986	Leaky ReLU

Table 5: The optimal validation accuracy scores calculated using different network structures and hidden activation functions f_H, across different learning rates. Multiple learning rates correspond to the same accuracy score.

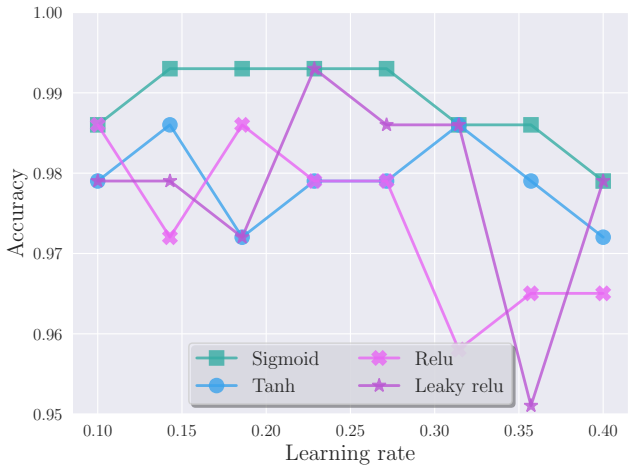
had a single set, $\lambda = 10^{-3.5}$ for $\eta = 0.03$. Lastly, using `scikit-learn`'s neural network implementation, the $\mathbf{N}_5^2$ structure using sigmoid activation function for both the hidden and output layer yielded a validation accuracy of 0.986 for multiple penalisation and learning rates, as shown in Fig. 16. Again, the obtained validation accuracy scores are in the vicinity of the results from our own +implementation.

5 Discussion

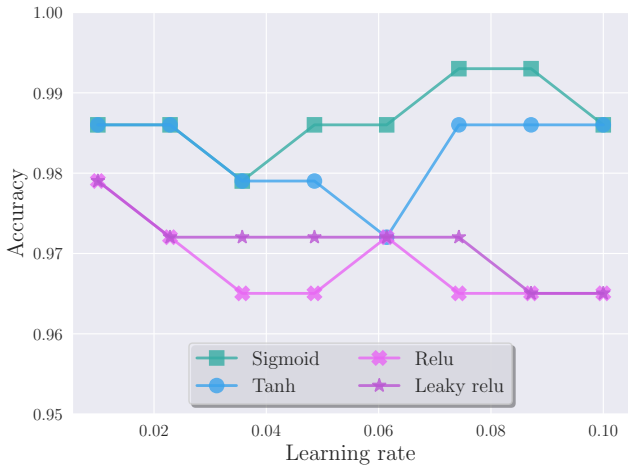
5.1 Gradient Descent Methods

The GD methods we employed were not especially impressive in optimising our linear regression parameters when compared to the analytical solution obtained from matrix inversion. However, we used a pretty limited dataset, and did not use that many features (20 in total). Matrix inversion will quickly become computationally expensive, and can also become unstable when there is correlation between the features. Some notes on the different algorithms employed follow.

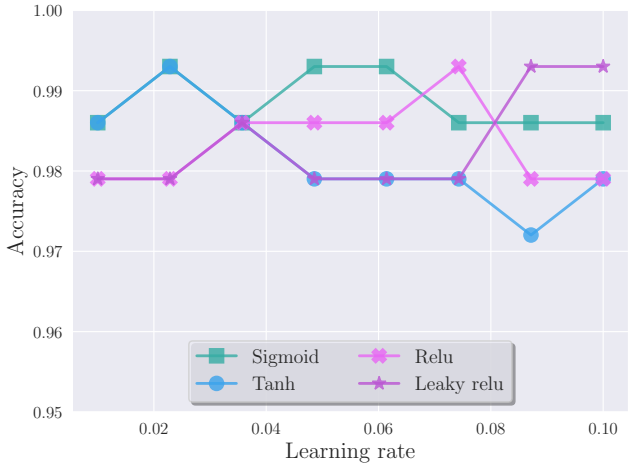
We did not analyse fully the role of epochs and batch size with the stochastic methods, focusing rather on the



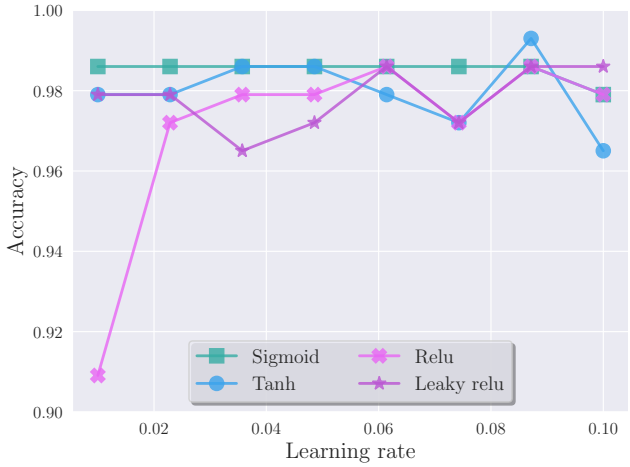
(a) Network $\mathbf{N}_5^1$, plotted for $\eta \in [0.1, 0.4]$ using 8 linearly spaced points



(b) Network $\mathbf{N}_{10}^1$, plotted for $\eta \in [0.01, 0.01]$ using 8 linearly spaced points

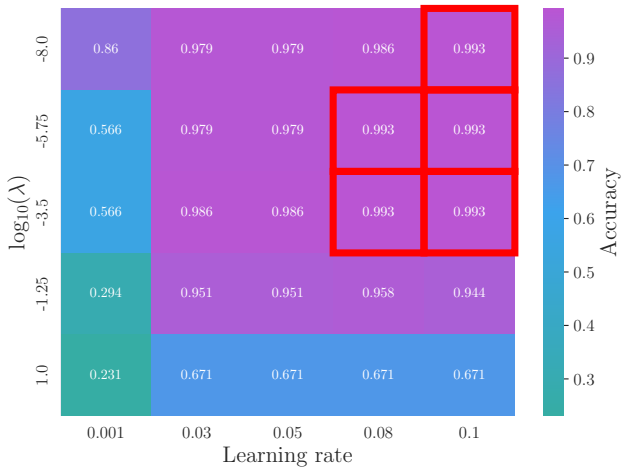


(c) Network $\mathbf{N}_5^2$, plotted for $\eta \in [0.01, 0.01]$ using 8 linearly spaced points

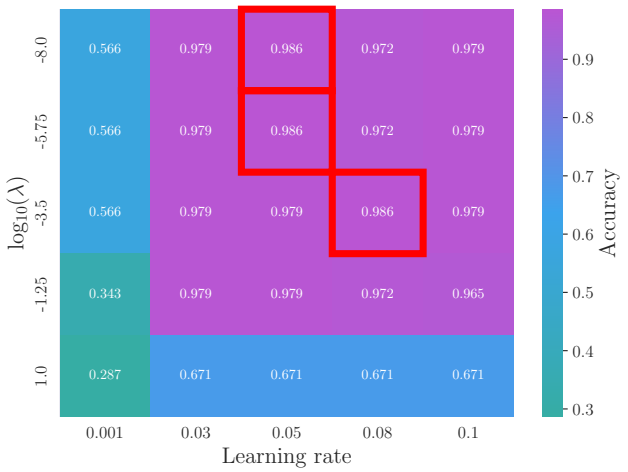


(d) Network $\mathbf{N}_5^3$, plotted for $\eta \in [0.01, 0.01]$ using 8 linearly spaced points

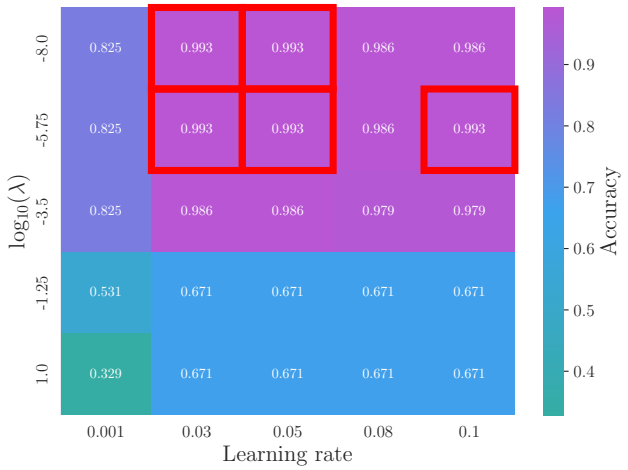
Figure 14: Plots of the accuracy score against learning rate for different activation functions for used for the hidden layer(s). Different network models have been used for the four plots. The AdaGrad algorithm with momentum $\gamma = 0.8$ has been used for optimisation, running 100 epochs with a batch size of 200.



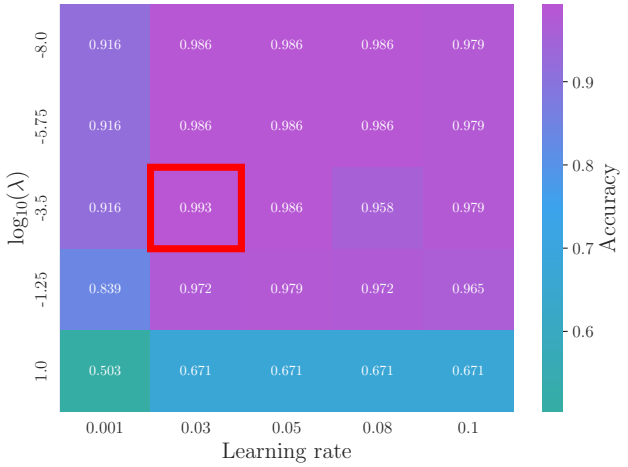
(a) Network $\mathbf{N}_5^1$ with a sigmoid activation function for the hidden layer. Multiple maxima was found with an accuracy of 0.993



(b) Network $\mathbf{N}_5^1$ with a hyperbolic tangent activation function for the hidden layer. Multiple maxima was found with an accuracy of 0.986.



(c) Network $\mathbf{N}_5^2$ with a sigmoid activation function for the hidden layers. Multiple maxima was found with an accuracy of 0.993.



(d) Network $\mathbf{N}_5^2$ with a hyperbolic activation function for the hidden layers. A single maximum was found with accuracy of 0.993, using $\eta = 0.03$ and $\lambda = 10^{-3.5}$

Figure 15: Plots of validation accuracy score varied against learning rate and penalisation term λ. Red squares mark the best accuracy scores. The first row contains the network structures $\mathbf{N}_5^1$ with sigmoid (a) and hyperbolic tangent (b) activation functions for the output layer, while the second row contains $\mathbf{N}_5^2$ with sigmoid (a) and hyperbolic tangent (b). The AdaGrad algorithm with momentum $\gamma = 0.8$ has been used for optimisation, running 100 epochs with a batch size of 200.

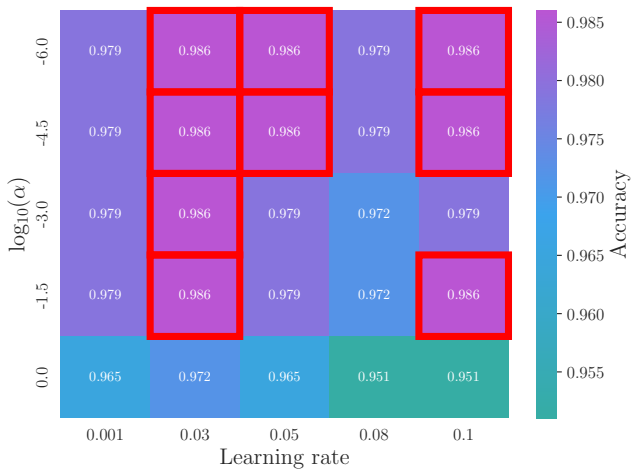


Figure 16: Network $\mathbf{N}_5^2$ validation accuracy scores using SciKit-Learn's neural network implementation, varied against learning rate η and L_2 penalisation term α. Multiple maxima was found with an accuracy of 0.986.

more overarching differences between stochastic and non-stochastic methods, together with the differences between algorithms. To obtain better results in our optimisation, looking closer at the tuning of these parameters would have been helpful.

5.1.1 Ordinary Gradient Descent

The non-stochastic GD was characterised by being sensitive to initialisation, but performed well on our simple optimisation problems when momentum was added, both on the OLS and ridge cost functions. These algorithms were faster than their stochastic counterpart, largely due to the fact that they update the parameters less often. It is clear that ordinary GD can be powerful when the parameter terrain is sufficiently non-complex, or if you have the computational resources to initialise multiple times.

5.1.2 Stochastic Gradient Descent

Adding stochasticity to the GD methods seemed to make them converge faster, at the expense of less stability. This is likely due to not being caught up in local minima, something both stochasticity and momentum can help with. Thus, it makes sense that the gap in performance between plain SGD and SGD with momentum was smaller (MSEs of 0.1862 and 0.1729 respectively) compared to the 'bare' GD (MSEs of 0.1982 and 0.1775 respectively). However, one of the significant improvements was the improvement in sensitivity to initialisation, which makes sense seeing as the stochastic algorithms can escape the pull of the local parameter space terrain more easily to venture towards greater minima.

5.1.3 AdaGrad methods

With the AdaGrad algorithms, the diagonal of G accumulates every iteration, adding g_i^2, meaning that the learning rates $\eta \propto G^{-1/2}$ will only decrease. This results in AdaGrad 'turning off' the descent, effectively implementing a cap to the distance it will travel in parameter space. This in turn makes it very safe, combating exploding gradients very effectively, as evidenced in Fig. 5c. However, it also makes it more sensitive to the initialisation of the parameters: If it starts far from a 'good' minimum in parameter space, it will struggle to reach it before turning off.

5.1.4 RMSprop and Adam

Both RMSprop and Adam married some of the stability of the tunable AdaGrad across learning rates, but managed to do so without turning off the learning rate, but rather using running averages $\boldsymbol{s}_k$ that do not monotonically increase. This meant that they were significantly less sensitive to the initialisation, as they managed to converge regardless. Again, adding momentum as with Adam increased stability and convergence.

5.1.5 Ridge Penalisation on Linear Regression

Adding a ridge penalisation on the parameters of our linear regression optimisation seemed to have little to no effect. We might attribute this to the fact that our linear model was not complex, and the data was 'nice' enough, as to not be prone to overfitting. Essentially, the degrees of freedom in our fit seemed appropriate, and did not need penalisation. It was interesting to note that it did decrease the sensitivity to the initialisation of the GD methods, however.

5.2 Neural Networks as Regressors

5.2.1 Initialisation of parameters

As the data is scaled such that the features and data all have mean zero and std one, initialising the weights with the same distribution ensures that they are all of the same order of magnitude. Thus, it is safe to assume that the weights are initialised without too much bias to away from the area of parameter space that should be of interest. We could have explored other ways of initialising the parameters of the network, like initialising with different distributions, or changing the std, and see how the network performed differently. Perhaps different architectures would do better when initialised in different ways.

5.2.2 Network Architecture

We saw that the optimal network with sigmoid activation function in the hidden layers was our implementation of $\mathbf{N}_{200}^5$ with learning rate 0.001. This was the most

complex network we tested, meaning we might get even better results with by further increasing the complexity. However, the universal approximation theorem tells us that for the appropriate network and training we can perfectly reproduce the training data, leading to overfitting. This network had 161 601 parameters (160 600 weights and 1001 biases), meaning it is enormous compared to the data we trained on, and could be prone to overfitting. The network did best with lower learning rates, so the fact that it avoided overfitting so well might well be due to the network not training enough. Furthermore, there is no actual noise added to the data, so the training and validation data may be quite similar, which can diminish the problem of overfitting. The data we sent in had two input variables, and we sent in 450 datapoints in the training process.

We saw that our and the `scikit-learn`-model had similar performances, implying our implementation is consistent with the norm.

5.2.3 Choice of Hidden Activation Function

It is interesting to note that the shape of a predicted graph in some sense inherits the shape of the activation function. The reason the network with the identity activation is linear, is that no non-linearity is ever introduced, so the output can only ever be linear in the input arguments.

5.3 Classification, Wisconsin Breast Cancer

Both Logistic regression and our Neural network managed to achieve a very good accuracy score calculated on the test set. The entire data set contained 569 unique instances, and by using our train-test split of $^3/_4$, the test set contained 143 points. Looking back at Tab. 5, we found optimal validation accuracies of 0.933 for many models, using both different network structures and hyperparameters. This corresponds to a single instance being wrongly classified, which one might consider a highly effective network. However, the Wisconsin Breast Cancer data set is known to be ideal and 'noise free' [Wolberg, 2001]. Therefore, having a repertoire of good performing models is not groundbreaking. For further analysis, the usage of lower train-test split ratios would be interesting. Model selection and hyperparameter optimisation could be performed as a function of train set size, which could indicate how little training data would be necessary to achieve good results.

Every feature from the data set has been included for all fitted models. This might be an over complication of the model, where the decisive factors could be a subset of the features present in the data set. Usage of some feature selection techniques would help to reduce the complexity of the model. In [Hasan et al., 2019], principal component analysis (PCA) was performed on the Wisconsin Breast Cancer, where they found that the number of features could be reduced to 15 covering 99.6% explained variance, by making linear combinations of the original features.

The accuracy score has consistently been used to evaluate models, but looking at the number of false positives (FP) and false negatives (FN) might also be an interesting metric to use. Distinguishing between the type of misclassification could help to narrow down the possible models, where an FN would be worse than an FP.

5.3.1 Logistic Regression

When investigating the effect of different optimisation algorithms, Adam SGD and mGD achieved a high accuracy for the least amount of iterations, as seen in Fig. 12c, both yielding a validation accuracy of 0.986 after about 25 iterations/epochs. When penalisation was added Fig. 13, no increase of validation accuracy was achieved, but two different classes of models performed equally good. For a weak penalisation, $\lambda = 10^{-6}$ the learning rate of $\eta = 0.13$ was found, resembling the model without penalisation. Using the higher penalisation of $\lambda = 10^{-1.5}$, the 0.985 validation accuracy was achieved for a bigger range of learning rates $\eta \in [0.25, 0.5]$. Despite such high learning rates, no exploding gradients or large variation in validation accuracies was found, indicating that the parameters converges to a good minimum. The L_2 regularisation also helps to set non-influential weights small, reducing the complexity of the model.

5.3.2 Neural Network

The validation accuracies from Tab. 4 yielded good results across structures and activation functions. The $\mathbf{N}_5^1$ structure using the sigmoid hidden activation function achieved the most stable 0.993 accuracy across all the structures, classifying with a single mistake for $\eta \in [0.15, 0.27]$ as seen in Fig. 14a. When doubling the number of nodes in this single layer ($\mathbf{N}_{10}^1$, Fig. 14b), performance decreased for the learning rates tried here. When increasing to $\mathbf{N}_5^2$ (Fig. 14c) and $\mathbf{N}_5^3$ (Fig. 14d), the preferred hidden activation functions was no longer as clear, without increasing performance or accuracy. Adding penalisation to $\mathbf{N}_5^1$ and $\mathbf{N}_5^2$ using the sigmoid and hyperbolic tangent seen in figure Fig. 15, lower penalisation was preferred. Using $\mathbf{N}_5^1$ and $\mathbf{N}_5^2$ the sigmoid function achieved an accuracy of 0.993 for multiple penalisation values $\lambda \in [10^{-8.0}, 10^{-3.5}]$ and $\lambda \in [10^{-8.0}, 10^{-5.75}]$ respectively. The same structures using the hyperbolic tangent function achieved fewer optimal values, where $\mathbf{N}_5^1$ had a lower accuracy score of 0.986. Using `scikit-learn`'s neural network implementation, the best validation accuracy scores achieved was 0.986 as seen in figure Fig. 16. Having multiple optimal scores in the vicinity of the results obtained using our implementation, serves

as a good benchmarking case validating that our neural network implementation is correct. The since the `scikit-learn` penalisation parameter α is not equal to our λ, direct comparisons between penalisation term is not feasible.

When comparing logistic regression with the neural network models, we found no significant benefit of using the more complex neural network with the approach applied in this work. In a real world scenario, understanding how the model classifies and how different features influences the result might be more important than a slight increase in accuracy. By directly looking at coefficient sizes, logistic regression is more easily interpreted, especially when compared to multi layered neural networks.

5.4 Model Selection

Model selection is prevalent in all parts of this project, and in many cases, the difference between models has been marginal. To get a more thorough examination of which GD methods, regressors and classifiers work best, it would have been helpful to employ resampling techniques like bootstrap or cross-validation, as in [Aasen et al., 2022], to get better estimates for the scores we have employed throughout, e.g. MSE, R2 and accuracy.

6 Concluding remarks

We have introduced and implemented a wide range of GD and SGD algorithms, and found that they perform satisfactorily when minimising OLS and ridge cost functions of either linear models, logistic models or neural networks with up to many thousands of parameters. Minimisation of linear models was done on the Franke function data with stochastic noise, using a polynomial expansion of degree 5 of the two arguments. Fitting to 600 points split into train- and test portions with a $^3/_4$ split, we found that adding momentum to the GD algorithm increased convergence rate and decreased sensitivity to initialisation, but decreased stability. Adding stochasticity did much the same, but using tunable learning rates as with the AdaGrad, RMSprop and Adam algorithms increased stability. Overall, the momentum based SGD with Adam performed the best with a validation MSE of 0.1660 ± 0.0018 using the OLS cost function with 500 epochs and a batch size of 200.

This led us to employ the Adama SGD when training our NNs both for regression of Nicaraguan terrain data similar to the Franke data, and classification of breast tumours from the Wisconsin breast cancer dataset. In the regression case, we compared our results with those found from linear regression based on methods from Aasen et al. [2022], expanding the two arguments of the terrain data polynomially to degree 11 and analytically finding the optimum parameters with an OLS cost function. Using a NN with 5 hidden layers with 200 nodes each, and training on 450 of the total 600 data points with 500 epochs of SGD with Adam and a batch size of 200, we got a validation MSE of 0.078 compared with 0.171 from the linear model. With a network of this size, we found that adding a penalisation on the size of the weights to improve the results, using an L_2 ridge penalisation parameter $\lambda = 10^{-4}$. This result was the best performed best among the various network architectures we tried out, all of which were smaller networks. In the future, it would be interesting to take a deeper dive into how the architectures are best trained by looking more closely at different numbers of epochs and batch sizes used with our SGD methods.

We tried using different activation functions for the hidden layers of our network, both for regression and classification, and found that the sigmoid activation function gave us the best results in both cases, with the ReLU functions close behind for regression and tanh for classification. Is this right???

The Wisconsin Breast Cancer dataset was found to be quite conducive to creating good models, and as such logistic regression and small NNs both produced excellent validation accuracies, even with relatively little training or hyperparameter tuning. Notably, learning rate required very little tuning. The highest validation accuracy (0.993) was achieved with a shallow NN with just one hidden layer with 5 nodes, with a penalisation on the weights of $\lambda = 10^{-3.5}$, but we argue that with more training, either logistic regression or NN could probably achieve a full accuracy of 1. In the future, it would be interesting to explore a smaller train-test split of the data, seeing whether we could predict more unseen data with less training data, or do a feature selection analysis to find which features were more important in classifying the tumours.

References

Anna Hjertvik Aasen, Carl Martin Fevang, and Håkon Kvernmoen. Bias-variance tradeoff in simple linear models, 2022. URL https://github.com/hkve/FYS-STK4155/blob/main/Project1/Project1.pdf.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

Md Hasan, Md Haque, and Mir Kabir. Breast cancer diagnosis models using pca and different neural network architectures. pages 1–4, 07 2019. doi: 10.1109/IC4ME247184.2019.9036627.

Morten Hjorth-Jensen. Neural networks. URL https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter9.html.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Daniel Ramos, Javier Franco-Pedroso, Alicia Lozano-Diez, and Joaquin Gonzalez-Rodriguez. Deconstructing cross-entropy for probabilistic binary classifiers. *Entropy*, 20(3), 2018. ISSN 1099-4300. doi: 10.3390/e20030208. URL https://www.mdpi.com/1099-4300/20/3/208.

William H . Wolberg. *Sparsity Through Automated Rejection*. University College London, 2001.

A Activation Functions

Name	**Function**	**Derivative**
Identity	$f(x) = x$	$f'(x) = 1$
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \sigma(x)(1 - \sigma(x))$
Hyperbolic tangent	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$\tanh'(x) = 1 - (\tanh(x))^2$
ReLU	$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$
Leaky ReLU	$f(x) = \begin{cases} 0.01x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$

Table 6: Table of the various activation functions implemented in this project and their derivatives.