Exploring the Transition from Regression Models to Neural Networks

Adam Jakobsen, Daniel Haas, Nils Taugbøl, Vilde Ung*
University of Oslo, Department of Physics
(Dated: January 13, 2023)

We explore implementation, tuning, and optimisation of the following methods: gradient descent (GD), stochastic gradient descent (SGD) and feed-forward neural networks (FFNN). We seek to establish which of these methods, along with the tuning of their various hyperparameters, is best suited for regression and classification tasks. Learned models using the different methods are applied to perform regression on test data and compare to an analytical solution in the case of regression. In the case of classification we use a `Scikit-learn` SGD model as a benchmark. We find that the learning rate is the most impactful hyperparameter for all our models. SGD with momentum yields the best results when performing a regression task on a one-dimensional polynomial. SGD with the Adam optimiser performed best for the classification task and both the SGD and FFNN models outperformed the `Scikit-learn` SGD model.

Keywords: Gradient Descent, Stochastic Gradient Descent, Feed-Forward Neural Network

I. INTRODUCTION

The desire to make accurate predictions from data has motivated researchers to develop statistical models inspired by biological brains. In 1943, Warren McCullen and Walter Pitts developed a simple algorithm based on information processing in biological neural networks. The model neuron, called the perceptron, receives signals from other units and modulates the output based on previously learned weights for each connection. Signals from distinct units add up with different weights in a non-linear way, allowing for machines to perform more robust learning tasks, as done in the 90s to identify hand-written digits [1].

Machine learning has since become an exciting and ever evolving field, with a wide variety of tools and concepts continually being developed. It spans simple, classical methods like linear regression models, as well as more advanced algorithms such as deep neural networks. In essence, these methods involve finding optimal parameters for a model based on data available for training. One of the most commonly applied tools for performing this optimisation is called gradient descent. The basic idea is simple: to iteratively calculate the gradient such that we continuously minimise the cost function. Nevertheless, there exists a wealth of different ways one might implement gradient descent, which leads to a wide range of possible hyperparameters and settings. This, in turn, requires close assessment and system-dependent tuning.

In this article, we develop and train models for both regression and classification tasks. We compare how the different methods perform in terms of making accurate predictions as well as ease of implementation. In Section II, we begin by presenting an overview of linear and logistic regression, gradient descent methods, and feed-forward neural networks. We also introduce the relevant adaptive methods used in gradient descent. In Section III we perform regressions on a one-dimensional polynomial function, comparing the results with a neural network model. Analogously, we apply a logistic regression model on the Wisconsin breast cancer data set [2–4] and compare its performance with a neural network model. In Section IV we go into a detailed discussion on the performance of the algorithms and possible improvements. In Section V, we conclude by summarising the advantages and disadvantages of the implemented algorithms for regression and classification, both with or without the use of an artificial neural network.

II. METHODS

A. Overview: Linear Regression

Perhaps the simplest example of a regression model is linear regression, which models the functional relationship between independent variables $\boldsymbol{x_i}$ and a dependent continuous variable y_i in a linear way. This linear relationship is determined by the parameters $\boldsymbol{\beta}$ and an added normally distributed noise term $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \sigma^2)$. In matrix form, this relation can then be written as

$$\boldsymbol{y} = \boldsymbol{\beta X} + \boldsymbol{\epsilon},$$

where the design matrix, $\boldsymbol{X}$, contains p column vectors $\boldsymbol{x_i}$ of N entries, each vector representing a feature of the data. The goal is to find the optimal parameters $\boldsymbol{\beta}$ based on the data available for training the model.

As is done in [5, Ch. 2.3.1], the process of deriving optimal $\boldsymbol{\beta}$ parameters for the linear regression model, given a squared-error cost function, can be analytically simplified to a matrix inversion.

* Repository: https://github.com/ungvilde/FYS-STK4155

Nevertheless, we note that obtaining an analytical expression for the model parameters is not always effective or even feasible. In this report, we will not focus on the analytic solution to the linear model. In the following section, we see that the logistic regression model is not analytically solvable, motivating the gradient descent algorithm that we later introduce.

B. Logistic Regression

An important part of supervised machine learning is the task of classification. In contrast to regression, a classification algorithm predicts what discrete category or class a data point belongs to, rather than assigning it a value. In this article, we will consider the logistic regression model, which models the probability that a data point belongs to one of two possible classes.

With binary classifiers, given the predicted class and target, respectively $y_i, t_i \in \{0, 1\}$, the accuracy of the model can be measured as

$$\text{Accuracy} = \frac{\sum_{i=1}^{N} I(t_i = y_i)}{N}, \tag{1}$$

where I is the indicator function given by $I(x) \in \{0, 1\}$ depending on whether x is false or true, respectively. The function responsible for this classification process is called a *classifier*, and can be either a hard or a soft classifier. A hard classifier returns the class to which the input data most likely belongs, without giving any information regarding the probability of it belonging to the other classes. On the other hand, a soft classifier returns the probability of the input belonging to each of the possible classes. The logistic regression model is a soft classifier.

For soft classifiers there are different ways of assigning the probability $p(y_i|\mathbf{x}_i)$ given an input vector $\mathbf{x}_i$ of length N. It is important, however, to maintain the constraint that the sum of the probabilities over the domain is one. The classifier assumes a functional relationship between the dependent and independent variables. With logistic regression, we apply the logistic function to model this relationship. The logistic function is given by

$$p(z) = \frac{\exp(z)}{1 + \exp(z)},$$

which, when applied to the classification of two possible categories, yields the conditions

$$p(y_i = 1|\mathbf{x}_i) = \frac{\exp(\boldsymbol{\beta}^T \mathbf{x}_i)}{1 + \exp(\boldsymbol{\beta}^T \mathbf{x}_i)}$$
$$p(y_i = 0|\mathbf{x}_i) = 1 - p(y_i = 1|\mathbf{x}_i).$$

The aim is then to find optimal parameters $\boldsymbol{\beta}$ which maximize the likelihood of obtaining the individual probabilities $p(y_i|\mathbf{x}_i, \boldsymbol{\beta})$. By focusing on the case in which $y_i \in \{0, 1\}$, the probability of the possible two outcomes are complementary, and the cost function to minimize can be obtained by the negative log-likelihood

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{N} \left\{ y_i \boldsymbol{\beta}^T \boldsymbol{x}_i - \log\left(1 + \exp\left(\boldsymbol{\beta}^T \boldsymbol{x}_i\right)\right) \right\}.$$

This equation is also known as the *cross entropy*. The challenge is then minimizing the cost function with respect to the parameters $\boldsymbol{\beta}$. However, equating the derivative with respect to the model parameters to zero does not yield analytical results. This motivates the use of numerical methods to find the minima of the cost function.

C. Gradient Descent

Our goal is to find values of $\boldsymbol{\beta}$ that gives a local minimum of the cost function $\mathcal{C}(\boldsymbol{\beta})$, with the hope of it being a global minimum. In some cases we can find analytical expressions for the optimal parameters $\boldsymbol{\beta}$ [6]. However for most cases, such as logistic regression, that will not be possible. Therefore, in this report we turn to first-order numerical methods for finding the best parameters that yield the lowest value for the cost function.

The gradient of the cost function evaluated at some set of parameters $\boldsymbol{\beta}^i$ is the direction in which it grows the fastest, i.e. the steepest tangent. Taking the negative of this direction we get the direction of steepest descent given by $-\nabla_{\boldsymbol{\beta}} C\left(\boldsymbol{\beta}^i\right)$. We can then express a new set of parameters $\boldsymbol{\beta}^{i+1}$ from the parameters $\boldsymbol{\beta}^i$ and $-\nabla_{\boldsymbol{\beta}} C\left(\boldsymbol{\beta}^i\right)$ as taking a step in the direction of steepest descent

$$\boldsymbol{\beta}^{i+1} = \boldsymbol{\beta}^i - \eta_i \nabla_{\boldsymbol{\beta}} C\left(\boldsymbol{\beta}^i\right), \tag{2}$$

where the *learning rate* η_i controls the step size of the iteration. This is the general expression and idea of *gradient descent*; to update the parameters of the function in the direction of the steepest descent, given by the gradient of said function.

Practically, $\boldsymbol{\beta}^i$ for $i = 0$ is a trial guess and through this iterative process we hope to find the set of parameters for which the gradient is zero. The learning rate can be fixed for all i or it can be adaptive and/or scheduled. The learning rate has to be tuned, as too large a learning rate will cause great fluctuations and too low a learning rate will cause the method to converge too slowly. We will look into this in a following section.

We generally want to choose a sufficiently small η_i such that

$$C\left(\boldsymbol{\beta}^{i+1}\right) < C\left(\boldsymbol{\beta}^i\right),$$

which ensures that we will be evolving the parameters $\boldsymbol{\beta}$ towards the minimum of the cost function. We also note that for large data sets, calculating the gradient of the cost function can be computationally expensive.

Ideally we want to find the global minimum of a given cost function, which is the absolute lowest value of said function. Iterating through (2) with an appropriate learning rate does not guarantee that we will find the global minimum. Gradient descent is deterministic, it stops when the gradient of the function is zero, and high dimensional cost functions may have multiple local minima or saddle points. This means that gradient descent is highly sensitive to the trial guesses $\beta^{i=0}$ as well as the learning rate. In practice we can evaluate the "goodness" of a found minimum and either be satisfied or not. That can be avoided if the cost function is convex since we will be guaranteed to have found a global minimum, since local minima are global for convex functions [7].

D. Momentum

Gradient descent (GD) moves toward a minimum at a fixed rate where each new iteration is independent of the general evolution of the parameters. To accelerate the rate at which the parameters are being updated toward the best parameters we can add a momentum term to the iterations:

$$\boldsymbol{\beta}^{i+1} = \boldsymbol{\beta}^{i} + \gamma\Delta\boldsymbol{\beta} - \eta_i \nabla_{\boldsymbol{\beta}} C\left(\boldsymbol{\beta}^{i}\right). \tag{3}$$

Here, $\Delta\boldsymbol{\beta} = \boldsymbol{\beta}^{i} - \boldsymbol{\beta}^{i-1}$ and γ is the *momentum coefficient*, with $0 \leq \gamma \leq 1$. The momentum coefficient sets a value for how much the previous updates should count for the next update, i.e. how much momentum the updates get. We see that if $\gamma = 0$ we get the normal gradient descent algorithm.

The momentum term increases the step size in the direction of consistent descent while reducing it in the directions which oscillate greatly. Which means that we will be focusing the direction of our updates toward the minimum.

E. Stochastic Gradient Descent

Stochastic gradient descent addresses the challenge that gradient descent has when using large data sets. The gradient of the cost function in gradient descent can be expressed as the sum over the gradient of cost functions at each data point $\mathbf{x}_i$ in the data set:

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

We introduce stochasticity by making an estimate of the gradient $\nabla_\beta C(\beta)$ based on a random subset of the data points, a *mini-batch*. In this report, the size of the mini-batches M will not be tuned adaptively when performing stochastic gradient descent. Equation (3) can now be expressed, using the quality in (4), as

$$\boldsymbol{\beta}^{i+1} = \boldsymbol{\beta}^{i} + \gamma\Delta\boldsymbol{\beta} - \eta_i \sum_{j\in k}^{n} \nabla_\beta c_j(\mathbf{x}_j, \beta).$$

When updating β^i using this equation, we iterate over the total number of mini-batches, and k is the collection of the indices of the data points in a mini-batch, which are chosen at random. Such an iteration over the mini-batches is an *epoch* and we will iterate over a chosen number of epochs when performing stochastic gradient descent.

Approximating the gradient of the cost function using mini-batches has the advantage of not increasing computation time per update with an increasing data set. This means that we can monitor the amount of mini-batches and epochs and find reasonably good parameters without running through the entire data set. The randomness introduced by the mini-batches also reduces the chance for the method to get stuck at a local minimum.

The random sampling of the training data from the mini-batches introduces noise to the gradient, which means that the gradient does not vanish even at minima of the cost function. In this report we choose to stop the search for optimal parameters after a chosen number of epochs. Each time stochastic gradient descent is performed on a data set, the method will propose different parameters. The set of parameters chosen will be the one yielding the lowest discovered value of the cost function.

F. Optimisation Methods

There is no guarantee that SGD will converge with a fixed learning rate. Therefore we introduce scheduling and adaptive methods to tune the learning rate with the intent of guaranteeing convergence for SGD.

1. *Scheduled Learning Rate*

One simple way of guaranteeing convergence is to make the step size diminish and converge toward a constant as the method iterates and updates the parameters. We schedule η_i, for every iteration i, by decreasing it as we iterate through the epochs, such that when nearing the final epochs the step size goes to zero or a constant. In this report we will use the scheduling

Algorithm 1 The AdaGrad algorithm for the learning rate.

$\boldsymbol{r} = 0$ ▷ Initialise gradient accumulation variable
for every epoch **do**
  for every mini-batch **do**
    [...]
    $\boldsymbol{g} \leftarrow \sum_{j \in k}^{n} \nabla_\beta c_j(\mathbf{x}_j, \beta)$ ▷ Calculate gradient of mini-batch
    $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$ ▷ Update gradient accumulation variable
    $\eta_i \leftarrow \frac{\eta_i}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
    [...]

Algorithm 2 The RMSprop algorithm for the learning rate.

$\boldsymbol{r} = 0$ ▷ Initialise gradient accumulation variable
$\rho = 0$ ▷ Initialise the decay rate
for every epoch **do**
  for every mini-batch **do**
    [...]
    $\boldsymbol{g} \leftarrow \sum_{j \in k}^{n} \nabla_\beta c_j(\mathbf{x}_j, \beta)$ ▷ Calculate gradient of mini-batch
    $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$ ▷ Update gradient accumulation variable
    $\eta_i \leftarrow \frac{\eta_i}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
    [...]

of the learning rate proposed by Goodfellow et. al [8] (page 287), which is given by

$$\eta_i = \left(1 - \frac{i}{\tau}\right)\eta_0 + \frac{i}{\tau}\frac{\eta_0}{100},$$

where τ will be last iteration over all the epochs and mini-batches. The initial learning rate will have to be tuned in the same manner as for gradient descent. In this case the learning rate will converge to a constant which is 1% of η_0 as $i \to \tau$.

This scheduling of the learning rate reduces the step size over time but does not guarantee the best parameters β. Just as stochastic gradient descent without a scheduled learning rate, the best parameters will have to be chosen in a similar manner as described above.

This learning rate scheduling will be used when applying all following optimisation methods.

2. AdaGrad

The AdaGrad optimisation method adaptively scales the learning rate for each individual parameter such that each update of the parameters is focused in the direction of smaller gradients. The scaling is inversely proportional to the square root of the sum of the squared gradients which has an added positive value $\delta \ll 1$ to avoid division by zero. To keep updating in the direction of smallest oscillations the accumulation of the gradients is kept in a gradient accumulation variable $\boldsymbol{r}$ which is updated every iteration. The algorithm for scaling the learning rate using Adagrad is shown in Algorithm 1.

AdaGrad reduces the learning rate from the first iteration and keeps the memory of all gradients. Therefore it is faced with the problem of prematurely shrinking the learning rate and hence slowing the search for the optimal parameters. We note that AdaGrad can be implemented both with and without momentum.

3. RMSprop

RMSprop is a modification to the AdaGrad where a fixed decay rate ρ is introduced to change the gradient accumulation variable into a weighted moving average. Effectively, RMSprop exponentially discards old memory of the gradient and allows for faster convergence. The algorithm for RMSprop's learning rate optimisation is shown in Algorithm 2 where we can observe that it is simply an extension of AdaGrad. The $\boldsymbol{r}$ is the second order moment of the gradient as it is calculated from the squared gradient. This optimisation method will be applied with a momentum term in this report.

4. Adam

Adam improves upon the RMSprop method by also keeping in memory the first moment of the gradient with a smaller decay rate than the second moment. There are no clear reasons why this method works well in practice, though empirical results show that it does [9]. The algorithm for the optimisation of the learning rate using Adam is shown in Algorithm 3. We denote the decay rate of the first moment as ρ_1 and the decay rate of the second moment as ρ_2, in this report we will use $\rho_1 = 0.9$ and $\rho_2 = 0.99$ as proposed by Goodfellow et. al [8, p. 301]. All other parameters are similar to RMSprop.

G. Feed-forward Neural Network

A *feed-forward neural network* (FFNN) is an artificial neural network in which the information moves forward in the network. The neurons, also known as nodes, are organised into layers that form the structure of the network. The layers can be classified into three categories:

Algorithm 3 The Adam algorithm for the learning rate.

$\boldsymbol{s} = 0$ ▷ Initialise first moment gradient accumulation variable
$\boldsymbol{r} = 0$ ▷ Initialise second moment gradient accumulation variable
ρ_1, ρ_2 ▷ Initialise the decay rate
for every epoch **do**
  for every mini-batch **do**
    [...]
    $\boldsymbol{g} \leftarrow \sum_{j\in k}^{n} \nabla_\beta c_j(\mathbf{x}_j, \beta)$ ▷ Calculate gradient of mini-batch
    $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1-\rho_1)\boldsymbol{g}$ ▷ Update first moment gradient accumulation variable
    $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1-\rho_2)\boldsymbol{g} \odot \boldsymbol{g}$ ▷ Update second moment gradient accumulation variable
    $\eta_i \leftarrow \eta_i \frac{\boldsymbol{s}}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$
    [...]

- *Input layer*: The first layer containing one node for each data feature.
- *Hidden layers*: The layers between the input and output layers, where weights and biases and activation functions are applied to the input data.
- *Output layer*: The final layer where the final output is generated

1. Forward Propagation

Forward propagation is the process of passing the input data through the network to generate the output. We start by sending the data from each input node into every node in the first hidden layer, where weights and biases are applied to the data. The outputs from the i^{th} node, denoted by z_i^l, in the layer $l \in \{1, 2, ..., L\}$, then has an activation function f applied to it. Then, it is passed to the next layer where the procedure is repeated. We then have

$$z_i^l = \sum_j w_{ij}^l a_j^{l-1} + b_i^l \tag{5}$$

and

$$a_i^l = f(z_i^l),$$

where w_{ij}^l is the weight between the j^{th} node in the $(l-1)^{th}$ layer and the i^{th} node in the l^{th} layer. Furthermore, a_j^{l-1} denotes the output from the j^{th} node in the $(l-1)^{th}$ layer, b_i^l is the bias of the i^{th} node in the l^{th} layer, and a_i^l is the activation function of the i^{th} node in the l^{th} layer. The forward propagation algorithm is outlined in Algorithm 0.

For the first hidden layer, the input data is used instead of an activation, resulting in the special case of (5), given by

$$z_i^1 = \sum_j w_{ij}^1 x_j + b_i^1,$$

where x_j denotes the j^{th} input feature.

Algorithm 4 Forward propagation

$a_i^0 = x_i$ for $i \in 1, 2, ..., n$ ▷ input layer with n nodes
for $l = 1, \ldots, L$ **do**
  for $i = 1, \ldots, n$ **do**
    $z_i^l = \sum_{j=1} w_{ij}^l a_j^{l-1} + b_i^l$
    $a_i^l = f(z_i^l)$

2. Back-propagation

In order to train the network, we need to find the optimal weights and biases for each node in the network. At the heart of the training process of a neural network is the *back-propagation* algorithm. The back-propagation algorithm is an iterative process which employs the chain rule and updates the weights and biases of the network to minimize the error between the predicted output and the actual output. Starting from the output layer and propagating backward through the network, the algorithm calculates the error of each node and uses it to update the weights and biases.

Given some cost function C, and considering the last layer of the network L, we use the chain rule and get

$$\begin{aligned} \frac{\partial C}{\partial w_{ij}^L} &= \frac{\partial C}{\partial z_i^L}\frac{\partial z_i^L}{\partial w_{ij}^L} \\ &= \frac{\partial C}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L}\frac{\partial z_i^L}{\partial w_{ij}^L}. \end{aligned} \tag{6}$$

From (5) we get

$$\frac{\partial z_i^l}{\partial w_{ij}^l} = a_j^{l-1},$$

and we can thus write (6) as

$$\frac{\partial C}{\partial w_{ij}^L} = \frac{\partial C}{\partial a_i^L} f'(z_i^L) a_j^{L-1}$$

where $f'(z_i^L)$ is the derivative of the activation function at the i^{th} node in the L^{th} layer.

For the bias we get

$$\begin{aligned}\frac{\partial C}{\partial b_i^L} &= \frac{\partial C}{\partial z_i^L}\frac{\partial z_i^L}{\partial b_i^L} \\ &= \frac{\partial C}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L}\frac{\partial z_i^L}{\partial b_i^L} \\ &= \frac{\partial C}{\partial a_i^L} f'(z_i^L)\end{aligned}$$

We now define the error of the i^{th} node in the l^{th} layer as

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

which for the output layer is given by

$$\delta_i^L = \frac{\partial C}{\partial a_i^L} f'(z_i^L). \tag{7}$$

In matrix form, we can write

$$\delta^L = \nabla_a C \odot f'(z^L),$$

where $\nabla_a C$ is the gradient of the cost function with respect to the activations, in this case the output layer, and $\odot$ is the Hadamard product.

Thus we can express the error δ^l in terms of the error of the next layer δ^{l+1} as

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot f'(z^l).$$

Finally, we update the weights and biases as

$$\begin{aligned} w_{ij}^l &\leftarrow w_{ij}^l - \eta \delta_i^l a_j^{l-1} \\ b_i^l &\leftarrow b_i^l - \eta \delta_i^l \end{aligned}$$

The back-propagation algorithm is outlined in Algorithm 5.

Algorithm 5 Back-propagation

for $i = 1, 2, \ldots, m$ **do** ▷ m outputs
  $\delta_i^L \leftarrow \frac{\partial C}{\partial a_i^L} f'(z_i^L)$
  $\frac{\partial C}{\partial b_i^L} \leftarrow \delta_i^L$
  $\frac{\partial C}{\partial w_{ij}^L} \leftarrow \delta_i^L a_j^{L-1}$
for $l = L-1, L-2, \ldots, 1$ **do**
  for $i = 1, 2, \ldots, p$ **do** ▷ p nodes
    $\delta_i^l \leftarrow (\sum_k \delta_k^{l+1} w_{ki}^{l+1}) f'(z_i^l)$
    $\frac{\partial C}{\partial b_i^l} \leftarrow \delta_i^l$
    $\frac{\partial C}{\partial w_{ik}^l} \leftarrow \delta_i^l a_k^{l-1}$
    $w_{ik}^l \leftarrow w_{ik}^l - \eta \frac{\partial C}{\partial w_{ik}^l}$
    $b_i^l \leftarrow b_i^l - \eta \frac{\partial C}{\partial b_i^l}$

3. Initialising the Weights and Biases

The standard way of initialising the weights and biases is to set the weights to random values by sampling from a normal distribution

$$w^l \sim \mathcal{N}(0, 1),$$

while biases are initialised to some small number, 0.01 in our case.

Another way of initialising the weights for a layer with m inputs and n outputs, suggested by Glorot and Bengio[10], is using *normalised initialisation*

$$w^l \sim \mathcal{U}\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right).$$

In other words the weights are sampled from a uniform distribution, with the goal of reaching a compromise between the layers having the same activation variance and the same gradient variance.

4. Activation Functions

Choosing an activation function is an important part of designing a neural network. Regardless of the architecture of the network, the outputs will be linear functions of the input. Therefore, activation functions are used to introduce non-linearity into the network. We will look at three of the most common activation functions: the *Sigmoid* function, the *Rectified Linear Unit* (ReLU) and the *leaky ReLU*.

The Sigmoid function is given by

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

and is a smooth function that is bounded between 0 and 1. It has the drawback that the gradient of the Sigmoid function is very small for large values of z. This means that the network will learn slowly.

The ReLU activation function maps negative values to 0 and positive values to themselves. It is given by

$$f(z) = \max(0, z).$$

The ReLU function has the advantage that it is fast to compute and that it does not saturate for large values of z. However, it has the drawback that when all the nodes in a layer are negative, the network will not learn anything because the gradient will be zero for all nodes in the network .

The leaky ReLU activation function is a modified version of the ReLU function. It has a linear mapping of negative values, while positive values are mapped to themselves. It is given by

$$f(z) = \max(cz, z),$$

where c is a small constant. The leaky ReLU function has the advantage that it does not saturate for large values of z and that it does not have the problem of the ReLU function where the gradient is zero for all nodes in the network. We consider the case where $c = 0.01$.

5. Choice of Cost Function

Depending on the nature of the problem, we consider two different cost functions: the mean squared error and the cross entropy.

For classification problems, we use the cross entropy, which for the final layer of a neural network is given by

$$C = -\frac{1}{m}\sum_{j=1}^{m}(y_j \log\big(a_j^L + (1-y_j)\log\big(1-a_j^L\big)\big)$$

where y_i is the target output for the i^{th} training example.

From equation (7) we get

$$\begin{aligned}
\delta_i^L &= \frac{\partial C}{\partial a_i^L} f'(z_i^L) \\
&= -\sum_j \left[\frac{y_j}{a_j^L} - \frac{1-y_i}{1-a_j^L}\right] f'(z_i^L) \\
&= -\sum_j \left[\frac{y_j}{a_j^L} - \frac{1-y_i}{1-a_j^L}\right] \frac{\partial a_i^L}{\partial z_i^L}\frac{\partial z_i^L}{\partial b_i^L}\delta_{ji} \\
&= -\left[\frac{y_i}{a_i^L} - \frac{1-y_i}{1-a_i^L}\right] a_i^L(1-a_i^L) \\
&= a_i^L - y_i
\end{aligned}$$

where we have used the fact that $f'(z_i^L) = a_i^L(1-a_i^L)$ for the sigmoid activation function.

For regression problems, we use the mean squared error cost function, which for the final layer of a neural network and a single output is given by

$$C = \frac{1}{2}\sum_j (y_j - a_j^L)^2. \tag{8}$$

Following the same procedure as for the classification case, with the mean squared error cost function and assuming a linear activation function, e.g ReLU, we get

$$\begin{aligned}
\delta_i^L &= \frac{\partial C}{\partial a_i^L} f'(z_i^L) \\
&= \sum_j (y_j - a_j^L) f'(z_i^L)\delta_{ji} \\
&= a_i^L - y_i.
\end{aligned}$$

H. Data Sets

1. Regression

In order to test the regression methods, we generated data based on a third degree polynomial, given by

$$f(x) = 0.5 + 0.5x + 5x^2 - 4x^3.$$

We use this function to generate 1000 data points, where we add a normal noise term $\epsilon \sim \mathcal{N}(0, 0.1^2)$ to each observation. The data set is then given by $\mathcal{D} = \{(x_i, f(x_i) + \epsilon_i) \,|\, i = 1, \ldots, 1000\}$. Each x_i was generated from a uniform distribution, $x_i \sim U(0, 1)$.

2. Classification

To test the classification methods, we use the Wisconsin breast cancer data set [2]. This has 569 observations of cell nuclei, where the different cell nuclei are given a diagnosis of being benign or malignant. In other words, the response variable has two possible categories. There are 212 malignant cells and 357 benign cells. Each cell nucleus has 30 real-valued features. The data set was downloaded using `Scikit-learn` from the `Python` library.

3. Scaling and Splitting the Data Sets

In order to improve the convergence of the gradient descent algorithms, we scale the data. Specifically, each column in the design matrix is centered at 0 and scaled to have standard deviation of 1. In mathematical notation, that is

$$x_{i,j}^* = \frac{x_{i,j} - \bar{x}_j}{\sigma_j},$$

where $\bar{x}_j$ is the mean of column j and σ_j is the standard deviation of column j. Here, $x_{i,j}$ is the data point for observation i of feature j, and $x_{i,j}^*$ is the scaled version of this data point. By scaling the data we also ensure that we penalise the model parameters fairly when we do Ridge regression.

Finally, we shuffle and split the data sets into test sets and training sets, where the test sets consist of 20 % of the total data sets. We use cross-validation to validate the results from training with various hyperparameters, and finally evaluate the model using the unseen test set.

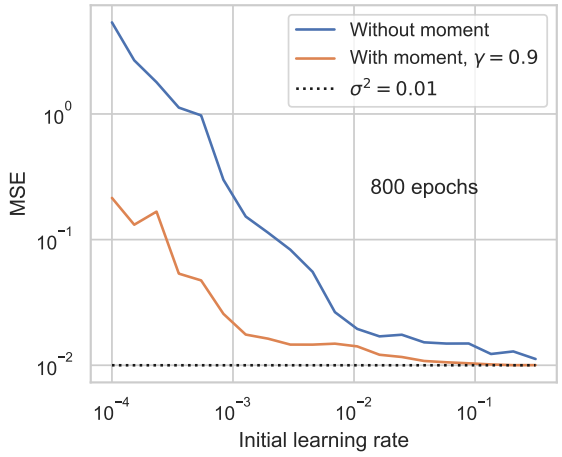

Figure 1: MSE of GD with 800 epochs, applied on a linear model, as a function of learning rates both with and without momentum. We applied a momentum coefficient $\gamma = 0.9$.

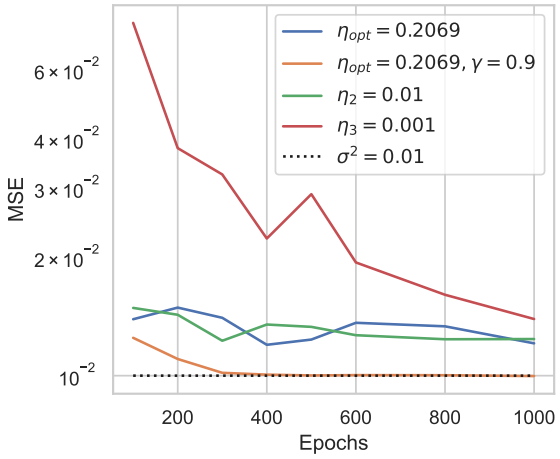

Figure 2: MSE of GD, applied on a linear model, as a function of epochs. It was solved using different learning rates without momentum except for the optimal learning rate η_{opt}, with momentum coefficient $\gamma = 0.9$. The optimal learning rate η_{opt} The dotted line indicated the minimal possible error of the model, given by the variance $\sigma^2 = 0.01$ in the data.

III. RESULTS

A. Linear Regression

We modelled the polynomial data presented in [**?**] using linear regression. The optimal parameters were found using the MSE cost function (8). All the results were validated using 5-fold cross-validation.

1. *Gradient Descent*

For gradient descent we first scanned for an appropriate learning rate, both with and without momentum, at a high fixed number of epochs. The result from this scan can be seen in Figure 1. The learning rate which yielded the lowest MSE at 800 epochs with momentum can be seen in Table III. We then applied the found learning rate with different epochs to find the most appropriate number of these. This can be seen in Figure 2 where we also applied learning rates at different orders of magnitude to see how the MSE behaved for those. Once the best number of epochs was found we scanned again for a better learning rate which in this case was the same learning rate η_{opt}.

2. *Stochastic Gradient Descent*

With stochastic gradient descent, we first performed a grid search for the optimal number of epochs and the optimal size for the mini-batches. We did this using a fixed initial learning rate of $\eta = 0.1$. We did not include momentum or any optimisations at this stage. The results from the grid search can be seen in Figure 3. We found the optimal batch size to be 5, and the optimal number of epochs to be 800.

Next, we explored the relationship between the initial learning rate and the number of epochs. In Figure 4 we have plotted the MSE as a function of the number of epochs for three different learning rates. This plot demonstrates how a larger initial learning rate is needed in order for the solution to converge within a reasonable number of iterations. We further explored how the learning rate affected convergence. In Figure 5 we have plotted the MSE as a function of learning rate. We included the results with and without momentum, and we see that using momentum improves convergence with small learning rates. Additionally, we looked at how optimisations methods like AdaGrad, RMSprop and Adam affected convergence. Figure 6 shows how the different optimisations affected the MSE as a function of epochs. The plot shows that the solution eventually converges in all cases, but that momentum converges particularly well.

Finally, we apply the above results and do a grid search for the optimal initial learning rate and regularization. The results of this grid search can be seen in Figure 7. Here, we found that the optimal initial learning rate was $\eta = 0.147$ with regularization $\lambda = 10^{-6}$. In Table I we provide an overview of the hyperparameters and their optimal setting.

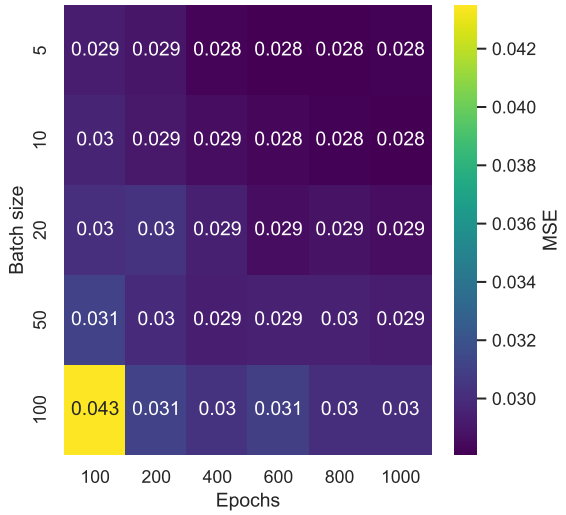

Figure 3: MSE computed using 5-fold cross-validation. Here, we applied a linear model solved with SGD. The MSEs were computed using a range of different epochs and batch sizes. We used an initial step size of $\eta = 0.1$.

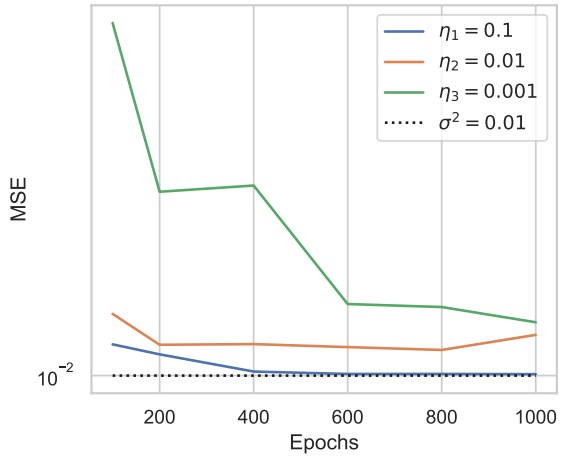

Figure 4: MSE as a function of epochs, solved using initial learning rates $\eta_1 = 0.1$, $\eta_2 = 0.01$ and $\eta_3 = 0.001$. We applied a linear model, solved using SGD. The dotted line indicated the minimal possible error of the model, given by the variance $\sigma^2 = 0.01$ in the data.

3. *Feed-forward Neural Network*

We modelled the polynomial data set using a neural network. In order to tune the model, we first did a grid search for the optimal network architecture. We started out using the sigmoid function as the activation in the hidden layers, and initialised the weights using

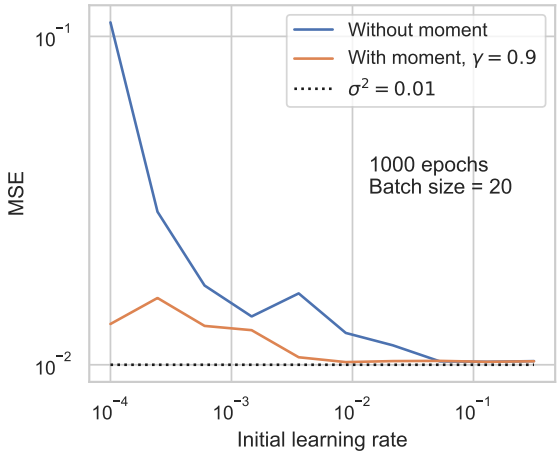

Figure 5: MSE as a function of the initial learning rate. We applied a linear model solved with SGD, both with and without momentum. When momentum was included in the algorithm, we used a momentum coefficient $\gamma = 0.9$. The dotted line indicates the minimal possible error of the model, determined by the inherent noise of the data $\sigma^2 = 0.01$.

Table I: Overview of what hyperparameters were chosen for the polynomial regression model solved with SGD.

Hyperparameter	Value
Epochs	800
Batch size	5
Initial learning rate	0.147
Optimisation	Momentum ($\gamma = 0.9$)
Regularisation	$1.0 \cdot 10^{-6}$

a standard initialisation scheme. In Figure 8 we have plotted the MSE found using a varying number of layers and neurons. We used an initial learning rate of 0.001, based on an initial manual search. Note that when we included multiple layers, all the layers were set to have the same number of neurons. Ideally we could have explored a wider range of possible architectures, but our results show that we attain a reasonably accurate fit with a single layer. The lowest MSE was found with 1 layer of 80 neurons.

To assess how the learning rate affects convergence, we looked at the MSE as a function of epochs for different initial learning rates. The results can be seen in Figure 9. We see that $\eta = 10^{-3}$ outperformed both $\eta = 10^{-2}$ and $\eta = 10^{-4}$. Convergence was further improved when adding momentum.

To tune the learning rate and regularization, we did a grid search. We applied the optimal network architec-

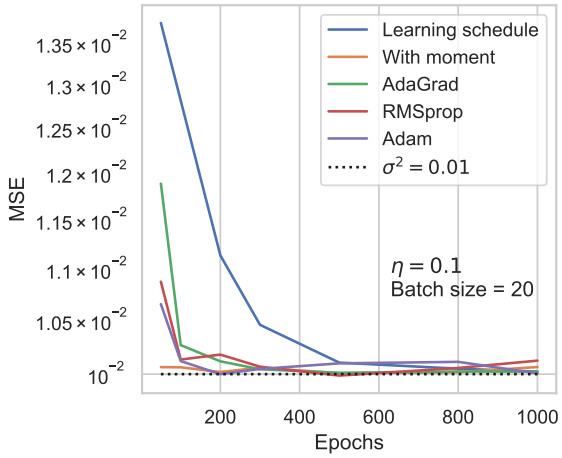

Figure 6: The MSE as a function of epochs, where different optimizations schemes were applied to the SGD algorithm. We applied the SGD algorithm to a linear model. The MSE values were computed using 5-fold cross-validation. The initial learning rate was set to $\eta = 0.1$, and we used a mini-batch size of 20. The black dotted line marks the value of the inherent noise in the data, $\sigma^2 = 0.01$.

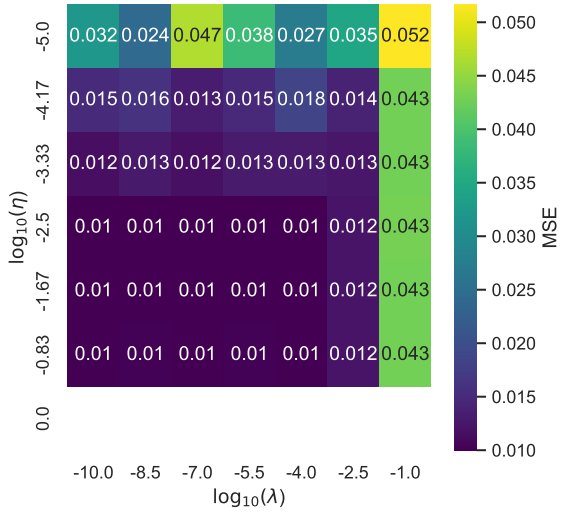

Figure 7: MSE found using different learning rates η and regularization parameters λ. We applied a linear regression model and solved for the optimal parameters using SGD. The MSE values were computed using 5-fold cross-validation. In the SGD algorithm, we used 800 epochs and a mini-batch size of 5. We applied a momentum coefficient of $\gamma = 0.9$. For $\eta = 1.0$, we got exploding gradients and a numerically unstable algorithm, which produced NaN-values in the output.

Table II: Overview of hyperparameters and their optimal value for the FFNN applied on a linear model.

Hyperparameter	Value
Num. hidden neurons	80
Num. hidden layers	1
Initial learning rate	$3.16 \cdot 10^{-3}$
Momentum coef.	0.9
Epochs	200
Batch size	20
Regularisation	$1.0 \cdot 10^{-8}$
Activation function	Leaky ReLU
Initialisation	Normalized

Table III: Overview of what hyperparameters were chosen for the polynomial regression model solved with GD.

Hyperparameter	Value
Epochs	400
Initial learning rate	0.2069
Optimisation	Momentum ($\gamma = 0.9$)

ture found previously, and we included momentum in the SGD algorithm. We did this using different activation functions in the hidden layers. In Figure 10 we have done a grid search for optimal learning rate and regularization parameter using the sigmoid function as the activation function. When performing the same analysis with the ReLU or leaky ReLU functions, we observed that our algorithm was unstable. Specifically, we got overflow warnings when the learning rate was 10^{-3} or larger, due to "exploding gradients", where the gradient becomes very large during training. To get around this problem, we used normalised initialisation for the weights. This gave us a more stable algorithm when using ReLU and leaky ReLU. In Figure 11, we have done a grid search for optimal parameters using the ReLU function and normalised initiation. In Figure 12 we have the analogous plot for leaky ReLU. In the end, we found the lowest MSE when applying leaky ReLU with normalised initialisation. The optimal learning rate was $\eta = 3.16 \cdot 10^{-3}$, with regularization $\lambda = 10^{-8}$. See Table II for a summary of the optimal hyperparameters.

After tuning the SGD, GD and neural network algorithms, we applied the optimal hyperparameters and tested the models on an unseen test set. We also tested the model using an ordinary least-squares analytic solution for comparison. The results can be seen in Table IV, which shows that the SGD algorithm gave the best results.

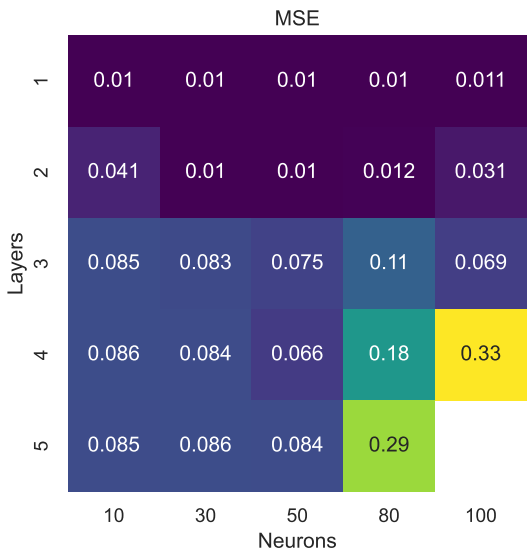

Figure 8: MSE computed using a grid search of different neural network architectures. We explored networks with at least one and up to five hidden layers, with either 10, 30, 50, 80 or 100 neurons in each hidden layer. We were unable to build a model using 5 layers of 100 neurons, leaving that part of the grid blank.

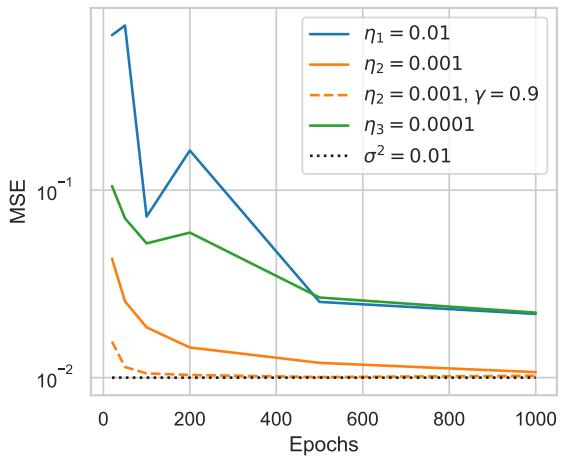

Figure 9: MSE as a function of the number of epochs, where three different initial learning rates were used. We used $\eta_1 = 0.01$, $\eta_2 = 0.001$ and $\eta_3 = 0.0001$. For η_2, we also plotted the solution when solved with momentum, with a momentum coefficient of $\gamma = 0.9$. The black dotted line indicates the minimal possible error. The network had a single hidden layer of 80 neurons.

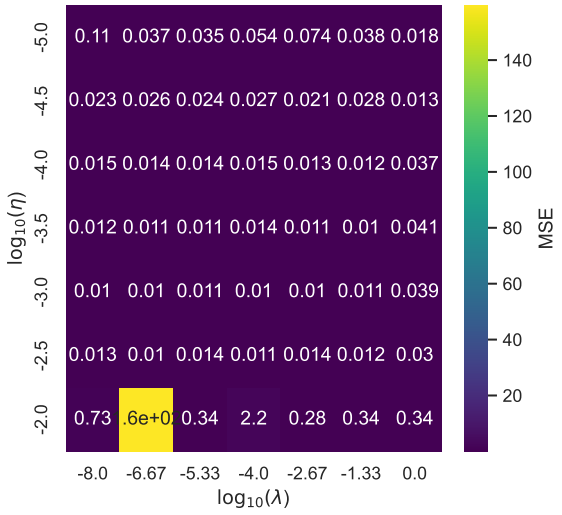

Figure 10: Grid search for optimal initial learning rate and regularization parameter. We used a neural network with a hidden layer of 80 neurons. In the SGD algorithm, we used 200 epochs with batch size 20 and momentum coefficient of 0.9. Here, we used the sigmoid activation function and standard normal initialization of the weights.

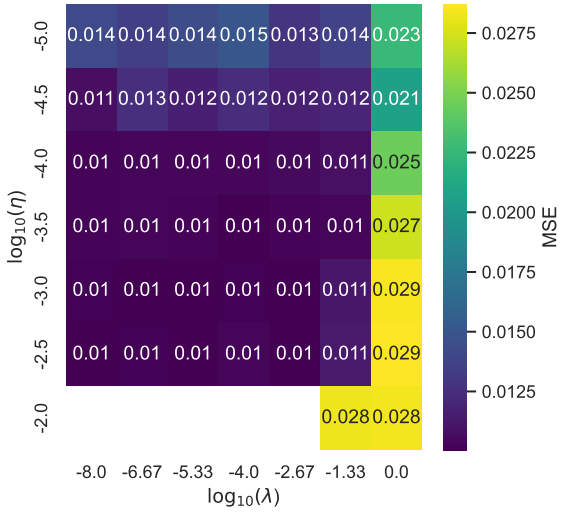

Figure 11: Grid search for optimal initial learning rate and regularization parameter, using the ReLU activation function and normalized initialisation of the weights. The results were computed using a neural network with one hidden layer of 80 neurons. We trained the network over 200 epochs with mini-batches of size 20. The MSE values were computed using 5-fold cross-validation.

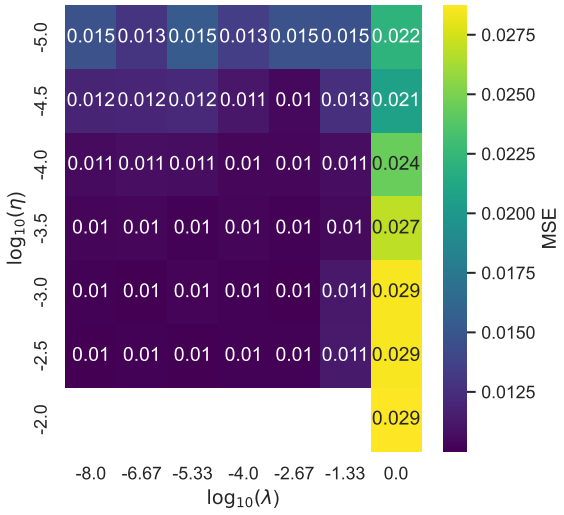

Figure 12: Grid search for optimal initial learning rate and regularization parameter, using the leaky ReLU activation function and normalized initialisation of the weights. The results were computed using a neural network with one hidden layer of 80 neurons. We trained the network over 200 epochs with mini-batches of size 20. The MSE values were computed using 5-fold cross-validation.

Table IV: The MSE found when testing the models on an unseen test data set. Optimal hyperparameters for SGD regression, FFNN and GD were applied. We have included the MSE found from an analytic solution using ordinary least-squares regression.

Model	Test MSE
Analytic	0.01014
SGD Regression	0.01018
FFNN	0.01020
GD	0.01393

B. Logistic Regression and Classification

With the objective of performing a classification study on the Wisconsin cancer data set, we implement several logistic regression fits, with the gradient descent method in its plain and stochastic variants in order to optimize the cost function. All the accuracy scores were obtained with 5-fold cross-validation.

1. *GD and SGD*

We start by analysing the accuracy score (1) as a function of the number of epochs for logistic regression

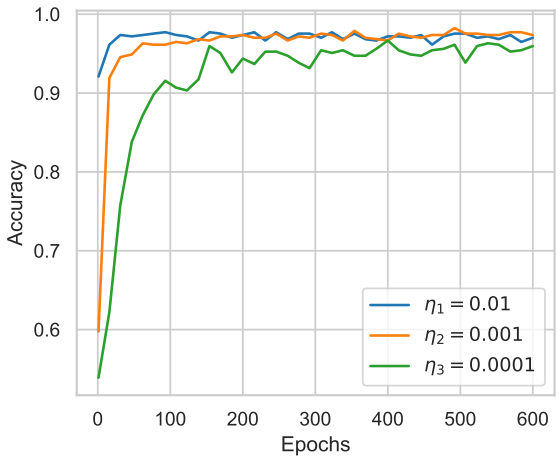

Figure 13: Accuracy as a function of the number of epochs, with three different learning rates $\eta_1 = 0.01$, $\eta_2 = 0.001$, and $\eta_3 = 0.0001$. Here, GD was used without momentum. The best accuracy is around 0.985 for η_2 and 400 epochs.

with plain gradient descent. The scores, presented in Figure 13, also show the dependence of the fit with respect to three commonly implemented learning rates.

By selecting the learning rate which presented the best convergence and accuracy for the proposed range of epochs, $\eta = 0.001$, we then proceed to find the optimal number of epochs and batch size for a stochastic gradient descent with an added momentum term of $\gamma = 0.9$ via the grid search in Figure 14. As a result of the search for those parameters, the best obtained accuracy score was 0.967 with a batch size of 20 for 50 epochs.

With the optimal batch size of 20, we proceed to compare different stochastic gradient descent optimisation methods in Figure 15, including a case without any optimisation. Figure 15 contains the moving average of the obtained scores as the non-averaged plot exhibited too much noise to properly evaluate the asymptotic behavior. The best obtained score was 0.988 with the SGD with momentum term of 0.9 and 74 iterations.

To compare how two optimization methods for SGD change under the addition of a l_2 regularisation term, λ, and find optimal learning rates, we perform the grid searches in Figure 16 and 17. The choice of momentum optimisation for the grid search was motivated by its superior accuracy score visible in Figure 15. Additionally, the choice to perform a grid search for Adam was due to its steady increase in accuracy for a larger number of epochs and its popularity.

We summarize the hyperparameters which yielded the best accuracy results in table V.

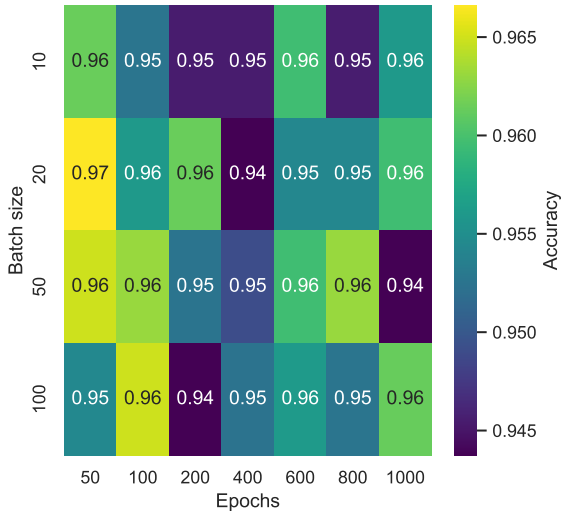

Figure 14: Grid search of accuracy score varying mini-batch size and number of epochs. Here we are using stochastic gradient descent with a momentum term of 0.9. We used an initial learning rate of $\eta = 0.001$.

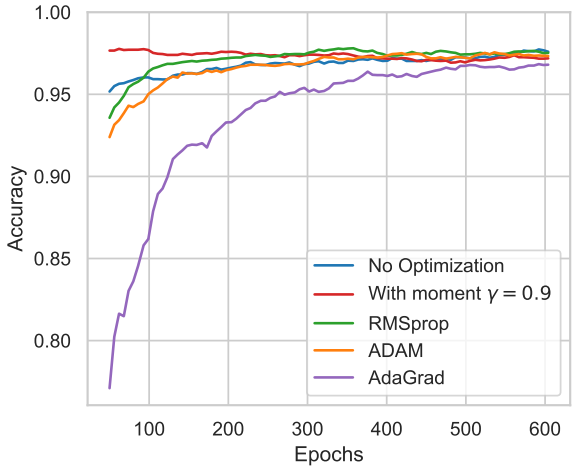

Figure 15: Moving average (10 epochs window) of accuracy as a function of the number of epochs for different stochastic gradient descent methods without regularization. All methods use an initial learning rate of $\eta = 0.001$ and mini-batches of size 20. The best accuracy was of 0.988 with a momentum term of $\gamma = 0.9$ at 74 epochs.

Table V: Overview of what hyperparameters yielded the best accuracy for the breast cancer data set with SGD logistic regression.

Hyperparameter	Value
Epochs	400
Batch size	20
Initial learning rate	0.1
Optimisation	Adam
Regularisation	0.046
ρ_1	0.9
ρ_2	0.999

Table VI: Overview of hyperparameters and their optimal value for the classification of the breast cancer data set with the FFNN.

Hyperparameter	Value
Num. hidden neurons	10
Num. hidden layers	1
Initial learning rate	0.032
Momentum coef.	0.9
Epochs	400
Batch size	20
Regularisation	0.002
Activation function	Sigmoid
Initialisation	Normalised

2. *Feed-forward Neural Network*

We then use the previously implemented feed-forward Neural Network to perform classification for the same Wisconsin cancer and compare it to the logistic regression analysis. First, in Figure 8, we explore the ideal network architecture for a sigmoid activation function as the output layer. To simplify the analysis, the grid search varies the number of layers and neurons in each layer, while maintaining every layer with the same number of units.

After finding that the best architecture consisted of a network with 10 neurons and one layer (and the output layer), we inspect the effects of adding an l_2 regularization term while also fine-tuning the initial learning rate for the network iterations in Figure 19. This search resulted in a classification score of 0.986 with an initial learning rate of $\eta = 0.032$ and regularisation term of $\lambda = 0.002$. This configuration consisted of the best score obtained for our FFNN classification, and an overview of the best found hyperparameters can be seen in Table VI.

The scores of models with optimal hyperparameters are summarised in Table VII. As a benchmark comparison, we also included the score for a `Scikit-learn` SGD logistic regression with fine-tuned initial learning rates and regularization parameters. As a final comparison between the optimal models, we show in Figure 20 the moving average of the accuracy as a function of the number of epochs for the three models with highest scores.

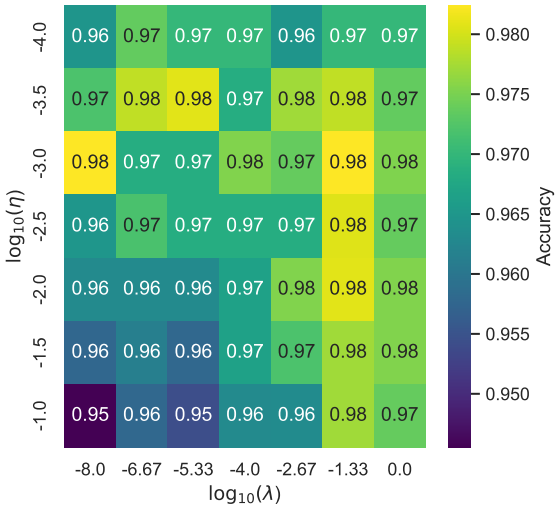

Figure 16: Grid search of accuracy score varying learning rate and regularisation term. Here we are using stochastic gradient descent with momentum term of 0.9. The maximum number of iterations was 400, and the largest accuracy achieved was around 0.982 with $\eta = 0.001$ and $\lambda = 0.046$

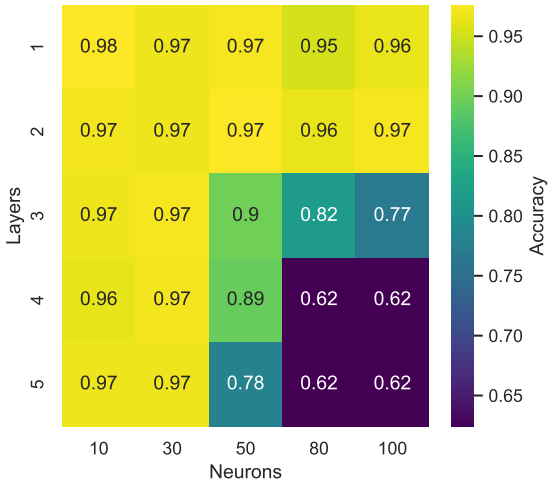

Figure 18: Grid search of the FFNN architecture accuracy score varying number of neurons and number of layers. Here we are using stochastic gradient descent with mini-batches of size 20, momentum term of 0.9, sigmoid activation layers and initial learning rate of $\eta = 0.001$. The maximum number of iterations was 400, and the largest accuracy achieved was around 0.976 with 1 layer of 10 neurons.

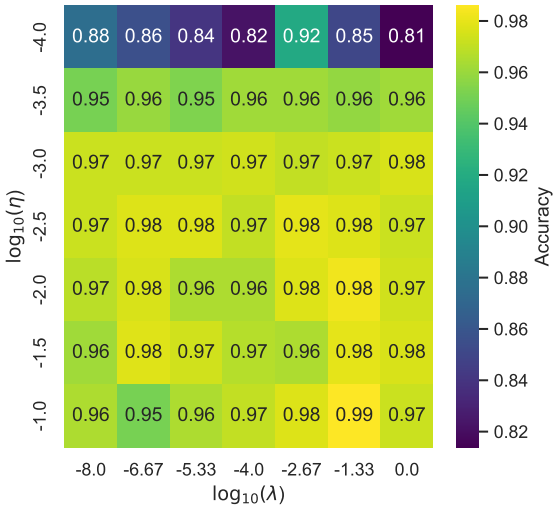

Figure 17: Grid search of accuracy score varying learning rate and regularization terms. Here we are using stochastic gradient descent with mini-batches of size 20 and Adam optimizer. The maximum number of iterations was 400 and the largest accuracy achieved was around 0.986 with $\eta = 0.1$ and $\lambda = 0.046$

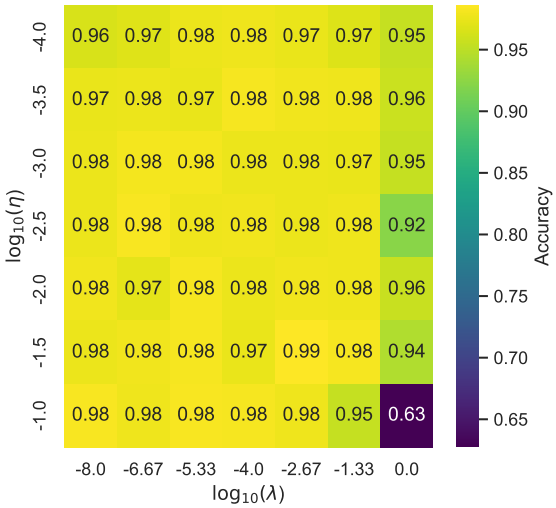

Figure 19: Grid search of accuracy score varying the FFNN learning rate and regularization terms. Here we are using stochastic gradient descent with mini-batches of size 20, momentum term of 0.9, sigmoid activation layers, and 1 layer of 10 hidden neurons. The maximum number of iterations was 400 and the largest accuracy achieved was around 0.986, obtained for $\eta = 0.032$ and $\lambda = 0.002$.

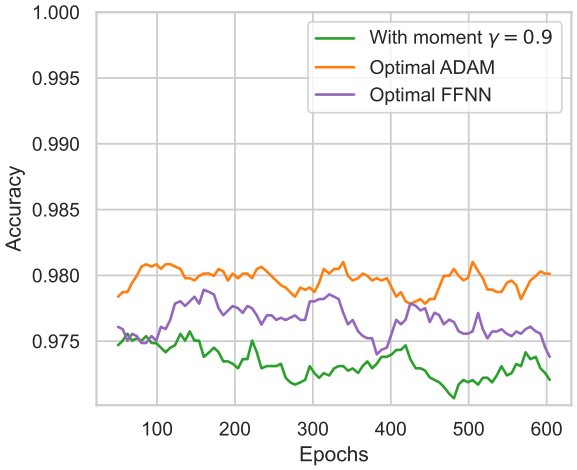

Figure 20: Comparison of optimal Adam SGD with optimal learning rate and regularization parameters obtained by grid search versus momentum SGD with a learning rate of 0.9 versus FFNN with optimal learning rate and regularization parameter found by grid search. Moving average (window = 10 epochs) The best obtained scores (not averaged) were 0.988 for both Adam and the FFNN and 0.984 for simple momentum implementation.

Table VII: Best accuracies obtained on test data. Optimal hyperparameters for SGD logistic regression, `Scikit-learn` SGD logistic regression, and FFNN implementation were applied.

Model	Test Accuracy
`Scikit-learn` SGD	0.981
SGD	0.986
FFNN	0.986

IV. DISCUSSION

A. Regression

A first observation of our results is Table IV. We see that an analytic linear regression method performs better than SGD and a FFNN. As this is a regression case on a one-dimensional third-order polynomial it can be expected that an analytic tool will outperform numerical approximation methods. However, both SGD and the FFNN perform with a satisfying MSE for our purposes. The advantage when applying SGD and FFNN is that we can apply the methods without relying on analytic solutions and so in our case of regression we can use the analytical solution as a benchmark for our models.

1. GD

We chose to find the optimal parameters of gradient descent by restricting the number of epochs instead of having a convergence criteria. This was done for easier comparison to SGD and the FFNN. From Figure 2 we see that the best learning rates without momentum converge to an MSE that is unsatisfactory compared to the analytical approach (see Table IV). This might indicate that the descent was halted by a flat area with a small gradient.

Adding momentum to the GD model greatly improved its performance, as seen in Figure 1, and allowed for an MSE within an acceptable range. These results may be improved through adaptive learning rates. The advantage of GD in our case with a simple data set is that it has the fewest hyperparameters to tune compared to the other numerical approaches. However, SGD and FFNN had a significantly lower MSE when tested which outweighs the benefits of a simpler implementation of GD.

2. SGD

We chose to approach the optimisation of the hyperparameters of stochastic gradient descent on our data set (see Section II H 1) in a pipeline fashion. Firstly we found a range of combinations of epochs and mini-batch sizes which yielded low MSE values, as seen in Figure 3. This was done without any form of optimization and a fixed learning rate, $\eta = 0.1$. The optimisation of the learning rate and regularisation term were tuned based on this initial grid-search. We can expect that a new grid-search for optimal combinations of number of epoch and mini-batch sizes for SGD, using the found optimal learning rate and regularisation term as well as momentum (see Table I), would yield different results. However, in our case we observe that the MSE of our optimal values for SGD tested on an unseen data set is satisfactory. The MSE yielded by the SGD is nearly the same as an analytic solution using ordinary least squares and 98.2% that of the inherent noise in the data.

In the case of our one-dimensional data with 1000 data points, doing the process of finding even better hyperparameters would not be beneficial. For smaller and/or more complex data sets, this process can be repeated multiple times until yielding satisfactory test results for the case in study.

A different approach could be to first find a satisfactory learning rate for a fixed mini-batch size and number of epochs. After which we could search for an optimal regularization term to finally find the best combination of mini-batch size and number of epochs. This could be

an equally valid approach and would have to be tested. We were satisfied with the results our approach yielded and so did not venture to try out different ones.

The optimal parameters presented in Table I are from the results found through Figures 3 and 4. When investigating the different optimisation methods for the learning rate and the impact of the momentum term we used different mini-batch sizes and number of epochs. This was done to better observe the behaviour of the model as a function of these optimisation methods.

From Figures 5 and 6 we observe that adding a momentum term to the SGD greatly increases the convergence rate of the model. Though momentum speeds up the convergence rate there seems still to be a higher dependence on the choice of learning rate. In Figure 5 we see that a momentum term can allow for smaller learning rates, about one order of magnitude smaller, than an optimal learning rate chosen without momentum. Momentum then helps with increasing the convergence rate though we have to be reminded again that we are testing this model on a simple case. The increase in convergence rate might be different and less evident for increasing complexity in the data set.

In this case we also observe in Figure 6 that a momentum term behaves equally good as the adaptive learning rates AdaGrad, RMSprop and Adam. All said methods converge at an MSE value close to the inherent noise in the data between epoch 200 and 300, which shows that these optimisation methods are effective compared to only using a scheduled learning rate. For further testing the applicability of these adaptive learning rate algorithms they can be applied to larger and more complex data sets, where we can expect different behaviour of the optimisation method. Considering the simplicity of the data set, implementing these adaptive learning rate algorithms increases the complexity and computation cost of the model without improving it. The momentum term is here satisfactory as it is the simplest method to implement.

The impact of the regularisation parameter λ is not obvious in Figure 7. For the higher value of $\log_{10}(\lambda) > -2.5$ we observe an increase in MSE compared to the lower values. The MSE does not improve for decreasing λ once the MSE is that of the inherent noise. The impact of the regularisation parameter is dependent on the learning rate as with a decreasing learning rate the MSE as a function of λ becomes unpredictable. In this case we have not applied momentum to the SGD model which we can expect would have overshadowed the impact of the regularisation parameter, which was helpful to find the optimal regularisation parameter.

Through these observations we see a continuum in the importance of the learning rate. We observe that the convergence rate of SGD is highly dependent on the learning rate (see Figure 4). In the case of a small learning rate the model will converge, though slowly. On the other hand, if the learning rate is too great the model might not converge at all (see Figure 7). Finding a good learning rate early in the look for optimal parameters might the most critical point to making a good model. The other hyperparameters can be tuned with regard to a range of, early found, well suited learning rates.

It is clear that momentum is an effective optimisation method for our model. This method introduces another hyperparameter that should be tuned. We followed the proposition of Goodfellow et. al. [8, 290] and used $\gamma = 0.9$. A few preliminary tests showed that it indeed was a well suited value. We chose not to adapt this value further. The momentum hyperparameter can be adapted over time which can be a point of interest for future implementation of SGD though we have chosen not to do so in this report.

When tuning the hyperparameters we could have looked at other dependencies such as MSE as a function of learning rate and batch size or number of epochs. In the case of this report we do not expect it to have any impact on our results as the hyperparameters (Table V) already yield an excellent MSE on the data set we are using. We could also have looked at the behaviour of the adaptive learning rate algorithms when used with momentum. Momentum keeps the memory of the direction in which the parameters are being updated, $\Delta\beta$, and the adaptive learning rate algorithms keep a memory of how the gradient changes. Therefore we could expect that combining both methods would further increase the convergence rate of the model. In the case of data sets with low complexity we do not expect this to have any effect as both methods already have a high convergence rate on their own.

3. FFNN

To find the optimal hyperparameters to perform the regression with a FFNN we chose to start with $\gamma = 0.9$, a mini-batch size of 20, 500 epochs and a learning rate $\eta = 10^-4$ which we used to find the best architecture, see Figure 8. In our case, increasing the complexity of the architecture, that is increasing the number of hidden layers and neurons, led to increasing MSEs. The number of neurons per hidden layer has a bigger impact on the MSE as we increase the number of hidden layers. We still have an increasing MSE with an increasing number of layers but this increase is small with a small number of neurons per layer. This is expected as fewer neurons means fewer parameters to optimise and so there is a smaller chance of overfitting.

We can also see the simplicity of the regression problem through the simplicity of the optimal architecture. The optimal architecture proposed by Figure 8 is with a single hidden layer where the MSE does not vary greatly

with number of neurons per layer.

When tuning the learning rate of the FFNN we find that the MSE as a function of epochs, see Figure 9, we observe optimal learning rate without momentum does converge toward the inherent noise in the data though slowly. We also see that the optimal order of magnitude for the learning rate is an order of magnitude lower than the one found for SGD. It is expected that different models and methods have different optimal parameters when dealing with the same problems. We can see that both a small and a large learning rate impedes the convergence more so than it did for SGD. Through Figure 9 we see that η_2 is the better option of learning rate but that both η_3 and η_1, which are both an order of magnitude different than η_2, produce an equally large MSE. This gave us an expectation to find the optimal learning rate within the order of magnitude 10^{-3} but to find a better fitting learning rate a fine-scan of MSE as a function of the learning rate is required. This can be seen in Figures 10 - 12 through which we found our best found values presented in Table II. Again we observe that adding a momentum term allows for notably faster convergence, where we have the desired result at epoch ≈ 300 when using momentum and ≈ 1000 when not.

The activation functions were sensible to the way the weights and biases were initialised. When initialising with a normal distribution of the weights and biases we had numerical overflow issues. For the sigmoid activation function our solution was to add a cut-off, at $|x| > 500$, to the input value. This may not be satisfactory for other models but in our case the results were acceptable and within the expected MSE. For the ReLU and leaky ReLU we applied normalised initialisation. This shows that they are both sensible to the variance in the gradients and weights, and need a compromise between their variances to avoid exploding gradients. The sigmoid function with cut-off produced highly similar results when using both initialisation methods.

When applying the different activation functions (sigmoid, ReLU and leaky ReLU) we observe the lowest MSE in the expected order of magnitude of the learning rate, $\log_{10}\eta \approx -3$. The use of the sigmoid activation function shows a smaller range of good learning rates, see Figure 10. The MSE increases quickly when the learning rate diverges from $\log_{10}\eta \approx -3$ compared to the ReLU and leaky ReLU, see Figures 11 and 12.

We see that the network cannot produce results for most values of λ when $\log_{10}\eta = -2$. The reason for this might again be an exploding gradient problem, and so these activation functions show a higher sensitivity to both initial weights and biases but also to too large learning rates. Both these methods perform better than the sigmoid function in our case of regression and they perform similarly to each other.

The regularisation parameter does grant a variation in the MSE of the model but the most notable impact is when the parameter is too large. As opposed to our implementation of SGD to find the optimal regularisation parameter we did include momentum when search with the neural network. This is an indicator that the FFNN is more sensible to its hyperparameters. As FFNNs use SGD for every neuron in every layer we could expect that adaptive learning rates show an increase in performance. In our case however, we were satisfied with the low MSE gotten from our optimal parameters, shown in Table II, which yield an acceptable MSE, even though OLS and SGD produced a lower MSE (see Table VII).

B. Classification

1. *GD and SGD*

The results in Table VII show that for the classification task of the Wisconsin breast cancer data set, thoroughly tuning the models can yield equally good results for SGD logistic regression and FFNN implementations. Furthermore, both methods outperform a default `scikit-learn` SGD logistic regression with optimized learning rate and regularization parameters.

The proximity between the neural network and logistic regression results is somewhat expected given the characteristics of the data set. For one, the size of only 569 instances results in a small test split of the data. Additionally, the samples can be considered balanced (212 malignant cells, and 357 benign cells), which minimizes bias in the training data and improves the probability of the models learning true patterns from the population. The slight advantage of both algorithms we developed when compared to scikit-learn baseline implementation reassures that a fine-tuning of the models, in this case, can allow for better performance. Although we are not sure why, we were able to obtain slightly better accuracy than what is reported in the literature [11] with an equal train-test split proportion of 80-20. For comparison, M. F. Ak was able to obtain an optimal score of 0.981 with logistic regression methods, while we obtained 0.986. Nonetheless, it has been shown that ensemble methods classifications with XGBoost [12] can yield an accuracy of up to 0.99.

Although the accuracy score of 0.988 was obtained in both Figure 15 and 20, it should be noted that those scores were a result of using a different number of epochs, namely 74 and 222, respectively. In order to standardise the comparison, we, therefore, considered 400 epochs when performing the grid searches, from which the best results are summarized in Table VII.

Figure 13 shows two expected behaviors of gradient methods in general. First, it shows that the accuracy

of the predictions tends to increase with the number of epochs until it reaches a plateau. Secondly, both the rate of convergence and the value of the y-axis plateau are dependent on the learning rate. Despite not being shown here, this behavior extends to the stochastic gradient descent and also to the FFNN.

Simple, weak classifiers usually output as predictions, either the most preponderant class in the data set, or make uniformly random guesses, regardless of the input data. In our data set, the class of benign cells composes 62.7% of the data. Therefore, for a small number of epochs, accuracy scores around this value are expected. Both learning rates of $\eta = 0.0001$ and $\eta = 0.001$ in Figure 13 start with scores in that range. Despite the fact that the learning rate of $\eta = 0.01$ start with considerably better performance, the learning rate of $\eta = 0.001$ is able to surpass its score with a greater number of epochs. This in turn motivates our selection of learning rate $\eta = 0.001$ for the following grid-search.

In Figure 14 we do a grid search for the optimal batch size and the number of epochs for SGD logistic regression with momentum. This grid search shows that a batch size of 20 gives good scores. Nonetheless, we notice that increasing the number of epochs for a fixed batch size did not significantly affect the accuracy, indicating early convergence.

The grid search in Figure 16 and Figure 17 show that, while fine-tuning the learning rate and regularization parameters can yield improvement in accuracy scores for SGD logistic regression with different optimizations, this effect is small. This could be explained by the fact that both grid searches are done for a number of epochs that already exhibit accuracy score convergence (see again Figure 15). Both grid searches were also performed for a batch size of 40, not yielding significantly different results.

2. FFNN

When applying the neural network on the same data set, Figure 18 shows the preference for a simpler network architecture. There is not much intuition behind this observation, beyond the small size of the data set and its noise-free profile as stated in [13]. The model does not benefit from a greater complexity when the data is of simple nature. For example, linear data without noise does not benefit from a polynomial fit of high degree. However, each observation has 30 features, making the task of assessing the complexity of the data set difficult.

Following the same procedure done with the SGD logistic regression, the optimal learning rates and regularization parameters were investigated, see Figure 19. Again, the search shows similar accuracy score results, despite a slightly better score of 0.986 for one specific configuration. While the values for the best parameters do not hold profound intuition about the data or the model, we noted that similarly good and stable scores can be obtained for grid searches with a batch size of 40 and ReLU output activation function.

As a way of comparing the three fine-tuned models (SGD with momentum term of 0.9, SGD with Adam optimizer, and the FFNN) we generate Figure 20. Here we see the moving average of the model's accuracy as a function of the number of epochs. The moving average gave a more stable trend with less noise and shows that the best model, on average, was consistently the Adam implementation. The Adam hyperparameters can be verified in Table V.

We end the comparison between the classification models by bringing attention to added momentum optimization in the SGD algorithm. Its competitiveness over other pre-tuning optimization methods was unforeseen (see Figure 15). As is clear by the large number of grid searches that we have presented, tuning the SGD optimizers and adding regularization terms can yield better results than those obtained with just the momentum term. However, the simplicity of implementing momentum compared to other methods make it an excellent choice.Another important aspect with regard to momentum is fast convergence. Figure 15 shows that even at 50 epochs, it has high accuracy. For larger data sets and models with a big number of parameters, this small number of epochs for obtaining a good accuracy can largely accelerate the iterative process of model fitting.

C. Future work

Due to the plethora of possible hyperparameters to tune in a neural network, there will always be unexplored possibilities for further improving the model. Some of the possibilities include looking at different training schedules for the learning rate, tuning the momentum coefficient and decay rates used in adaptive methods, implementing early stopping, exploring other activation functions, etc. In addition to further tuning the hyperparameters, an approach that could have been interesting to explore is randomly sampling the hyperparameters when tuning the model, rather than doing a manual search or a grid search. This has been suggested by [14] to be a more efficient approach to tuning neural networks. Randomly sampling hyperparameter space could have alleviated one of the weaknesses of our methodology, which arises from the fact that we tune one or two parameters at a time. This approach is somewhat inefficient, as it takes a long time to run through all the parameters, and we risk not finding the optimal combination of the various parameters.

Another point for future improvement is the neural

network architecture. For the grid search simplicity, all the layers in the neural network contained the same number of neurons regardless of their depth. This is not the usual setup. A more common approach is a gradual decrease in each layer's number of units with an increasing depth. It is however unclear what would be the consequences of those changes in our current models.

V. CONCLUSION

We have looked at different methods for learning from data, comparing how regression models and neural network models perform on classification and regression tasks. We have used gradient descent methods to find the optimal parameters of the models, and explored how different gradient descent implementations affected the convergence of the models.

We find that the polynomial regression model solved with SGD performs marginally better than the neural network. The test errors of these methods were close to that obtained from an analytic ordinary least-squares solution. Both SGD regression and the neural network outperformed the gradient descent approach. We found that adding momentum to the SGD algorithm significantly improved convergence, and that there was no further benefit to be gained with adaptive methods. We also found that the neural network had overflow issues when the learning rate was too large. We got a more stable algorithm by using a normalised initialisation scheme for the weights when applying ReLU and leaky ReLU activation functions.

When applying a logistic regression model and a neural network to the Wisconsin breast cancer data, we found that both methods outperform the default scikit-learn MLPClassifier, as well as the results found in literature. Again we found that adding momentum to the SGD algorithm improved convergence considerably for both models, and we find that the Adam optimiser performed the best with logistic regression.

Altogether, the results show that there is not a great advantage to applying neural networks to the data sets included in this article, as simpler regression models appears to do equally well. With neural networks, the great advantage is the flexibility of a non-linear model, with the cost of there being many hyperparameters to consider. Nevertheless, we have hopefully elucidated some of the many aspects of implementing and tuning neural network models and the relevant optimisation algorithms.

[1] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, Advances in neural information processing systems **2** (1989).

[2] "Breast Cancer Wisconsin (Diagnostic) Data set," https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29, accessed: 11-08-2022.

[3] O. L. Mangasarian, W. N. Street, and W. H. Wolberg, Operations Research **43**, 570 (1995).

[4] W. N. Street, W. H. Wolberg, and O. L. Mangasarian, in *Biomedical image processing and biomedical visualization*, Vol. 1905 (SPIE, 1993) pp. 861–870.

[5] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, 2nd ed., Springer Series in Statistics (Springer New York Inc., 2001).

[6] N. T. A. Jakobsen, D. Haas and V. Ung, "Exploring polynomial regression methods for predicting terrain data," (2022).

[7] S. Boyd and L. Vandenberghe, *Convex optimization* (Cambridge university press, 2004).

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) http://www.deeplearningbook.org.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," (2014).

[10] X. Glorot and Y. Bengio, in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, Vol. 9, edited by Y. W. Teh and M. Titterington (PMLR, Chia Laguna Resort, Sardinia, Italy, 2010) pp. 249–256.

[11] M. F. Ak, in *Healthcare*, Vol. 8 (MDPI, 2020) p. 111.

[12] S. A. Abdulkareem and Z. O. Abdulkareem, Int. J. Sci., Basic Appl. Res. **55**, 67 (2021).

[13] W. H. Wolberg, *Sparsity Through Automated Rejection* (2001) p. 656.

[14] J. Bergstra and Y. Bengio, J. Mach. Learn. Res. **13**, 281–305 (2012).