Exploration of Gradient decent and Neural network for linear and logistic regression

Isabel Sagen, Dorina Rustaj & Eir Eline Hørlyk

(Dated: November 6, 2024)

This study aims to find optimal approaches and parameters for implementing regression techniques using Gradient Descent (GD) and feed-forward neural network (FFNN) algorithms. These methods have broad applicability, highlighting their significance in supporting advanced data analysis and of modeling trends in various areas, including disease diagnosis and treatment outcomes. We have explored both linear and logistic regression, applying them to a simple polynomial dataset and the Wisconsin Breast Cancer dataset. In our exploration of ideal parameters for linear regression using stochastic gradient descent, we achieved the lowest mean square error (0.0104) with a learning rate $\eta = 10^{-2}$. For the FFNN we got optimal MSE (0.0021) when $\eta = 10^{-1}$, $\lambda = 10^{-5}$ in a single layer network of 10 neurons with leaky ReLU activation function. For logistic regression we got the best accuracy (0.988) when using learning rate $\eta = 10^{-3}$. For the FFNN the best accuracy (0.99) was achieved when $\eta = 0.0$ $\lambda = 10^{-1}$ in a single layer network of 100 neurons with sigmoid activation function. All computational work is displayed in the GitHub repository [a].

I. INTRODUCTION

In 1943 Warren McCulloch and Walter Pitts laid the groundwork for modern Artificial Intelligence (AI), creating the model *McCulloch-Pitts neuron* inspired by the workings of the neuron of the brain [7]. By the 1980s, David Rumelhart, Geoffrey Hinton, and Ronald J. Williams the model was advanced with the *Backpropagation algorithm* providing "a critical component in training and refining neural network models" [7]. The *feed-forward neural networks* (FFNN) has become essential tools in modern science, enabling scientists to tackle complex, high-dimensional problems such as particle interactions and quantum simulations. Beyond physics, FFNNs have been applied to a wide array of fields, including biology, finance, medicine, and environmental science, where they continue to drive innovation and discovery.

One of the applications of FFNN is tackling linear and logistic regression challenges. Linear regression models are designed to predict continuous outcomes, while logistic regression models categorize data into discrete labels, both of which have crucial applications in fields like healthcare and financial forecasting [9]. This project aims to investigate these regressions by developing both gradient descent (GD) and FFNN algorithms with focus on a simple second-order polynomial data and the *Wisconsin Breast Cancer Dataset* [4]. The main goal will be optimizing parameters to minimize the means square error (MSE) and improving predictive accuracy of the models.

For GD, this is achieved by introducing stochasticity through Stochastic Gradient Descent (SGD) and varying the parameters λ, the number of epochs, and mini-batch sizes. Additionally, we will vary the learning rate and examine methods for continuously updating the learning rate η.

Further we compared the performances of GD and FFNN for linear and logistic regression. By tuning the hyperparameters during a grid search for the FFNN, the lowest MSE for linear regression and best accuracy for logistic regression could be found.

The report will have the following structure: the theoretical foundations for the algorithms will be outlined in the *Methods* section II. This includes the analytical basis of the models and the FFNN. Further we showcase our results and a detailed analysis and reflection on the obtained results in the *Results and Discussion* section III. Finally we will summaries our work and discuss future improvements in the *Conclusion* section IV.

II. METHODS

A. Linear Regression: A Summary

Linear regression is a statistical method used to model the relationship between a dependent variable (x_i) and one or more independent variables (y_i) [6]. The primary goal is to find the best-fit line that minimizes the difference between observed and predicted values. Linear regression represents the simplest approach to best-fit modeling, resulting in a linear relationship determined by the parameter β and noise represented by a bell curve distribution (ϵ). This relationship can be expressed as

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

where $\mathbf{X}$ is the design matrix containing all the features of our data and y is the vector of target values we aim to predict. Essentially, we aim to determine the β parameters that best describe how each input influences the output.

[a] https://github.uio.no/isabesag/FYS-STK4155_P2/tree/main

To evaluate the quality of our fit, we use a squared-error cost function, which calculates the differences between actual data points and our predictions, squares those differences, and sums them up. A smaller total indicates a better fit [5]. Instead of testing different β values individually, which can be time-consuming, we can employ a mathematical method involving matrices. A matrix organizes data in rows and columns, and "matrix inversion" allows us to efficiently determine the optimal β values.

However, obtaining a formula for model parameters is not always effective. Complex data relationships may resist simple formulas, large datasets can complicate calculations, and highly correlated input variables can make it challenging to derive reliable estimates. Additionally, excessive noise or outliers can lead to inaccurate results. In these cases, alternative methods may be more effective for estimating parameters.

Note that in our project, we are working with both linear and logistic regression. Unlike linear regression, finding the formula for the parameters in logistic regression is more challenging because the logistic function is non-linear. Therefore, we introduce the gradient descent algorithm for both regressions, which iteratively adjusts the parameters based on model performance, making it a suitable approach when direct solutions are not feasible.

Cost functions

Cost functions provide a way to quantify how well a model's predictions fit the data. They measure the difference between the predicted and true values, giving us an object to minimize for optimal parameters [3]. There are many possible cost functions to choose from and we should choose depending on our problem. In the case with linear regression for OLS and Ridge we will use Mean Squared Error, MSE, (2). For logistic regression, we will later use the cross entropy cost function (3).

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_{\text{target}} - y_{\text{predicted}}\right)^2 \tag{2}$$

The final layer of the neural network is given by

$$C = -\frac{1}{m}\sum_{j=1}^{m}\left(y_j \log\left(a_j^L + (1-y_j)\log\left(1-a_j^L\right)\right)\right), \tag{3}$$

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

B. Gradient Descent

Gradient Descent is an optimization method used to find a local minimum of a cost function $C(\beta)$. The key idea here is that, at a given β, we evaluate the direction of the steepest tangent of $C(\beta)$, the gradient, and then take a step in the opposite direction [3]. The process is repeated until, ideally, we reach a global minimum. Each steps is then described by equation (4),

$$\beta_{i+1} = \beta_i - \eta_i \nabla C(\beta). \tag{4}$$

Here η_i is the learning rate and $\nabla C(\beta_i)$ is the gradient at step i. The learning rate controls the size of the step taken in the direction opposite to the gradient. It can remain constant for all iterations or change for each i according to the gradient. In either way, setting η_i to a sensible number is crucial for the method to work. For overly large η_i values, the model may "jump over" the local minimum. overly small η_i gives small steps which may lead to excessively slow convergence. In general we want a η_i value such that $C(\beta_{i+1}) < C(\beta_i)$ which ensures the model is approaching a minimum. The discussion on varying η_i will be elaborated upon in subsequent sections.

We start by noting that β_0 will be an initial guess. The objective is to converge to the *global* minimum of the cost function, as Gradient Descent can only reach one minimum (or saddle) point. For cost functions of higher order (greater than second order), the function may have multiple minima or saddle points. This means that the choice of β_0 can significantly influence the point to which the model converges. As a result, this can impact the final outcome, and we may only reach a local minimum.

Gradient Descent with momentum

From equation (4) we see that each step in the gradient descent is independent of the previous development of the descent. While the plain gradient descent works, it can have some shortcomings, such as slow convergence. This often happens when the gradient keeps oscillating, especially if the surface of the cost function is steep or irregular. In these cases the algorithm may take small, inconsistent steps, causing a slow and irregular converge to the minimum [3].

To address this issue we can introduce momentum to the gradient descent. Instead of relying solely on the current gradient, we use a combination of the current gradient and the previous update. This helps in smoothing out oscillations and speeds up convergence by "accelerating" in directions where the gradients are consistently pointing [8]. When momentum is introduced, the gradient descent is expressed as

$$\beta_{i+1} = \beta_i + \gamma\Delta\beta - \eta_i \nabla C(\beta), \tag{5}$$

where γ is the momentum coefficient and $\Delta\beta = \beta_i - \beta_{i-1}$ denotes the previous change of β. Parameter γ, the momentum coefficient, determines how much the previous step affects the current one. By incorporating momentum, the step size in the direction of oscillations is reduced, while in directions of consistent descent, the step size increases [3]. When implementing this computationally, we set $\gamma = 0$ to get the gradient descent without momentum. Note that for the first iteration, the momentum term is zero since as there is no prior step.

In our computations we set $\gamma = 0.9$, which means 90% of the velocity comes from the previous update, and 10% comes from the current gradient. This choice was done due to it being a widely accepted practice in momentum-based gradient descent optimization [8]. It uses enough momentum from past updates to help smooth out the changes, making the process more stable. However, it does not rely too heavily on past gradients, which can make the process slower, especially when the gradients change quickly.

C. Stochastic Gradient Descent

Gradient Descent can be inefficient with large datasets because it computes the gradient using the entire dataset, leading to longer computation times and slower convergence. To overcome this issue, we introduce a stochastic approach, resulting in Stochastic Gradient Descent (SGD). The gradient of the cost function is expressed as

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

In SGD, stochasticity is included by estimating $\nabla C(\beta)$ using only a random subset of the data, known as a minibatch. A new set of a random minibatch is chosen for each step. An iteration over a minibatch is called an epoch. We will choose number of epoch ourselves. If the dataset has n points and M as the size of the minibatches, there will be n/M minibatches. Thus, each update step in SGD can be written as

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

Here k is chosen randomly from $k \in [1, n/M]$.

SGD is not only faster than GD, but it also ensures that increasing the size of the dataset has a less effect on the computational time. Its speed helps the model learn faster, often resulting in quicker convergence.

A important benefit of the randomness in SGD is that it allows the model to explore more diverse paths towards the minimum. This increases the likelihood of finding a global minimum and decreases chance of becoming stuck in a local minimum. That said, the randomness can make the results noisy, meaning we expect more fluctuation in SGD than GD.

We need to be cautious when running too many epochs because the model may start to memorize the data instead of learning genuine patterns. This is known as overfitting, where the model captures noise and outliers rather than the underlying relationships, leading to poor performance on new, unseen data. To mitigate this risk, it is essential to monitor the model's performance on a validation set and use techniques like early stopping or regularization.

D. Optimization

The gradient descent method is highly sensitive to the choice of learning rate. Up until now we have assumed a constant learning rate. Ideally, for each point, we would tune the learning rate according to the surrounding landscape of the cost function. Thus, the model would adjust to having small learning rates where the gradient is steep to not miss the minimum, and larger learning rates for flat areas to speed up the process. Second order methods achieve this by calculating the *Hessian matrix*, which keeps track of the curvature of the cost function, adjusting the learning rate accordingly. However, this calculation can be computationally demanding. To address this issue, various efficient methods have been developed that require less computational power for adjusting the learning rate. We will now discuss some of these methods. Note that the only methods below that are prone to incorporate momentum in their algorithm are *RMSprop* and *Adam*.

1. AdaGrad

Adaptive Gradient Algorithm, also known as AdaGrad, adjusts the learning rate by scaling it inversely proportional to the square root of the total sum of the squared gradients from previous iterations [3]. This implies that a large gradient results in a smaller adjustment to the learning rate, while a small gradient leads to a larger adjustment in the learning rate. AdaGrad is designed to work well with convex cost function, which is relevant in our case study. The algorithm is shown in Algorithm 1.

However, AdaGrad is prone to premature convergence, stopping it from reaching the desired minimum. This occurs as a result of the algorithm continuously adding to the r term resulting in the learning rate decreasing significantly at a high rate. Consequently, AdaGrads performance will depend on the characteristic of the data.

2. RMSprop

Root Mean Square Propagation (RMSprop) is an improvement of AdaGrad made to improve performance on non-convex functions. Unlike AdaGrad, RMSprop uses an exponential decay to discard previous history of very

Algorithm 1 AdaGrad

```
Input $\delta$, $g$ and $\epsilon$
r = 0
for each epoch do
    for each minibatch do
        $g \leftarrow \sum_{j\in k}^{n} \nabla c_j(x_j, \beta)$        ▷ Gradient for minibatch
        $r \leftarrow r + g \odot g$        ▷ Updating $r$
        $\eta_i \leftarrow \frac{\epsilon}{\delta+\sqrt{r}}$        ▷ New learning rate
```

FIG. 1. Here δ is a small constant to avoid division by zero, usually set to 10^{-7}. ϵ is the global learning rate and stays constant for the whole process. g is the gradient.

Algorithm 2 RMSprop

```
Input $\delta$, $g$, $\rho$ and $\epsilon$
r = 0
for each epoch do
    for each minibatch do
        $g \leftarrow \sum_{j\in k}^{n} \nabla c_j(x_j, \beta)$        ▷ Gradient for minibatch
        $r \leftarrow \rho r + (1-\rho) g \odot g$        ▷ Updating $r$
        $\eta_i \leftarrow \frac{\epsilon}{\delta+\sqrt{r}}$        ▷ New learning rate
```

FIG. 2. Here δ is a small constant to avoid division by zero, usually set to 10^{-7}. ϵ is the global learning rate and stays constant for the whole process. g is the gradient. ρ is the decay rate set to 0.9.

steep gradients. By introducing a constant decay rate ρ, the algorithm assigns greater weight on the more recent gradients compared to the older ones [3]. The result is an algorithm that converges faster. The full algorithm is displayed in Algorithm 2.

3. Adam

Adaptive Moment Estimation Algorithm (Adam), from Adaptive moments, can be interpreted as a combination of RMSprop and momentum. A key feature is that momentum is built into the first-order moment of the gradient. Additionally, Adam corrects both the first and second order moment by applying bias correction. This is an improvement from RMSprop, as the second-order moment can exhibit high bias during the early stages of training. Refer to Algorithm 3 for the complete algorithm.

E. Logistic regression

In linear regression, the primary objective is to determine the optimal coefficients, β, for the best fit to the observed values. In classification tasks, we focus on identifying the discrete group or category to which each data point belongs. The functions that categorize data points into different classifications are known as *classifiers*. A common application of logistic regression involves analyzing binary outcomes, represented as 0 or 1, which may correspond to true or false, positive or negative, or yes or no. The accuracy of such a model is given by equation (6)

Algorithm 3 Adam

```
Input $\delta$, $g$, $\rho_1$, $\rho_2$ and $\epsilon$
r = 0
s = 0
for each epoch do
    t = 0
    for each minibatch do
        $t \leftarrow t + 1$
        $g \leftarrow \sum_{j\in k}^{n} \nabla c_j(x_j, \beta)$        ▷ Gradient for minibatch
        $s \leftarrow \rho_1 s + (1-\rho_1) g$        ▷ Updating $s$
        $r \leftarrow \rho_2 r + (1-\rho_2) g \odot g$        ▷ Updating $r$
        $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$        ▷ Bias in first moment
        $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$        ▷ Bias in second moment
        $\Delta\theta \leftarrow -\epsilon \frac{\hat{s}}{\delta+\sqrt{\hat{r}}}$        ▷ Updating learning rate
```

FIG. 3. Here δ is a small constant to avoid division by zero, usually set to 10^{-7}. ϵ is the stepsize, suggested value is 0.001 [3]. g is the gradient. ρ_1 and ρ_2 are the decay rates suggested to be 0.9 and 0.999, respectively.

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n} \tag{6}$$

Here, $I(t_i = y_i)$ stands for the indicator function, which equals 1 if the predicted class t_i matches the true class y_i, and 0 otherwise. This function allows us to count the number of correct predictions when calculating the accuracy of the model.

Classifiers are divided into soft and hard classifiers. Hard classifiers only assigns the data point to a category without providing any further information about certainty of the prediction. Soft classifiers provide the probability of a specific category. For example, the probability of a data point x_i is in the category $y_i \in 0, 1$ given by the *Sigmoid function* (7),

$$p(t) = \frac{1}{1+e^{-t}} = \frac{e^{-t}}{1+e^{t}}. \tag{7}$$

In the case of $y_i \in 0, 1$ and wanting to fit β, we can define the probabilities as follows:

$$p(y_i = 1 | x_i \beta) = \frac{e^{\beta x_i}}{e^{\beta x_i}}$$

and

$$p(y_i = 0 | x_i \beta) = 1 - p(y_i = 1 | x_i \beta).$$

F. Feed-Forward Neural Network

A *Feed-Forward Neural Network* (FFNN) is an artificial neural network where information flows in a single direction; from the input layer, through any hidden layers, and finally to the output layer, without forming cycles or feedback loops. This architecture is particularly suitable for tasks such as classification and regression. As mentioned, the FFNN consists of several layers; an input layer, one or more hidden layers, and an output layer. Each layer l has parameters $\Theta^l = (w^l, b^l)$, consisting of weights and biases that connect it to other layers through affine transformations, typically involving matrix-matrix or matrix-vector multiplications.

The *input layer* includes one node for each feature in the data set. Each *hidden layer* contains weights, biases, and activation functions that process the input data, enabling the network to learn complex patterns. The final layer, or *output layer*, generating the network's prediction based on the processed information. The FFNN operates two steps: *Feed-Forward Stage:* The input data flows through the network, producing an output that is compared to the target values using a loss function. *Backpropagation*, the models parameters are updated by optimizing the gradient of the loss function. Using the chain rule, gradients are computed from the cost function derivatives, allowing the network to iteratively refine its parameters and improve its predictions until a stopping criterion is reached.

Our simple FFNN will consist of an input layer, a single hidden layer and an output layer. The activation y of each neuron is a weighted sum of its inputs, transformed by an activation function. Specifically:

$$z = \sum_{i=1}^{n} w_i a_i,$$
$$y = f(x),$$

where f is the activation function, a_i is input from neuron i in the previous layer and w_i is the corresponding weight. The weights are initialized with small values distributed around zero, drawn from a normal distribution. Adding a bias term to the weighted sum enhances the networks ability to represent a broader range of values. A bias unit, having an output of 1, has a weight for each neuron j, denoted as b_j

$$z_j = \sum_{i=1}^{n} w_{ij} a_i + b_j.$$

The bias b will de set to 0.01 initially to ensure all neurons have some output which can be backpropagated in the first training cycle.

1. Forward propagation

In forward propagation, input data flows through the network, layer by layer, to generate an output. Each input node sends data to every node in the first hidden layer, where weights and biases are applied. The output from the i^{th} node, denoted as z_i^l, is transformed by an activation function f. The process is repeated as this output becomes the input for the subsequent layer:

$$z_i^l = \sum_j w_{ij}^l a_j^{l-1} + b_i^l, \tag{8}$$

with

$$a_i^l = f(z_i^l). \tag{9}$$

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, a_j^{l-1} is the output from the previous layer, and b_i^l is the bias term, and a_i^l is the activation function of the i^{th} node in the l^{th} layer. The whole algorithm is outlined in 4.

Algorithm 4 Forward Propagation

```
$a \leftarrow X\_data$
for $i = 1$ to len(hidden_weights) do
    $z \leftarrow a \times$ hidden_weights[$i$] + hidden_biases[$i$]
    Append $z$ to $z_h$
    $a \leftarrow$ activation (z)
    Append $a$ to $a_h$
$z_o \leftarrow a \times$ output_weights + output_bias
$a_o \leftarrow$ output_activation($z_o$)
```

FIG. 4. Forward propagation algorithm for neural networks.

2. Backpropagation

Backpropagation computes the partial derivatives (gradients) of the cost function C with respect to each weight w and bias b. To achieve this: initialize input data x and compute the activations z_1 of the input layer. Perform the feed-forward propagation until reaching the output layer, coalculating all z^l and corresponding activations a^l for $l = 1, 2, 3, \ldots, L$. Calculating the output error δ^L:

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

For each preceding layer, propagate the error:

$$\delta_i^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l).$$

Finally, update the weights and biases using gradient descent:

$$w_{ij}^l \leftarrow w_{ij}^l - \eta\delta_i^l a_j^{l-1}, \tag{10}$$

$$b_i^l \leftarrow b_i^l - \eta\delta_i^l, \tag{11}$$

where η is the learning rate.

Algorithm 5 Backpropagation

```
$e_o \leftarrow \frac{a_o - Y}{Y.shape[0]}$
$W'_o \leftarrow a_h[-1]^T \times e_o$
$b'_o \leftarrow \sum e_o$
$e_h \leftarrow e_o$
for $i \leftarrow \text{len}(W_h) - 1$ to 0 do
    $a' \leftarrow$ derivative of $a_h[i]$
    $e_h \leftarrow e_h \times W \times a'$
    $W'_h \leftarrow a_{i-1}^T \times e_h$
    $b'_h \leftarrow \sum e_h$
    Update $W_h[i]$ and $b_h[i]$ with $\eta$
Update $W_o$ and $b_o$ with $\eta$
```

FIG. 5. Backpropagation algorithm for neural networks.

The backpropagation algorithm 5 calculates the output error e_o, output weight gradient W_o, output bias gradient b_o, hidden error e_h, hidden weights W_h, and the derivative of the activation function a', which collectively update each layers weights and biases. This iterative process of error propagation and parameter updates allows the network to learn and refine its predictions.

3. Activation function

A property that characterizes a neural network is the choice of activation functions. In a neural network with only linear activations functions, each layer will perform a linear transformation of its inputs. Regardless of the number of layers, the output will be a linear function of the inputs. To enable the neural network to fit non-linear functions, we must introduce a non-linearity, with the Sigmoid function (7). Another option is to use the *Rectified Linear Unit* (ReLU) activation function, which outputs zero for negative values and returns positive values unchanged:

$$f_{\text{ReLU}}(z) = \max(0, z). \tag{12}$$

The ReLU function is efficient to compute and avoids saturation for large values of z. However, if all the nodes in a layer produce negative values, the network fails to learn since the gradient becomes zero across all nodes. To address this issue, a modified version known as the *leaky* ReLU is used, which is defined by:

$$f_{\text{Leaky ReLU}}(z) = \max(cz, z). \tag{13}$$

This function applies a linear mapping to negative values, while positive values remain unchanged. The constant c is set to $c = 0.01$. For the output layer, the activation function is often switched to the softmax function, which:

$$f(z_i^l) = \frac{\exp(z_i^l)}{\sum_{m=1}^{K} \exp(z_m^l)}, \tag{14}$$

as it is a good choose for classification tasks, while for regression tasks, one can simply use no activation function.

G. Data set

The regression methods are tested on a generated second order polynomial

$$y = 2 + 3x + 1.5x^2. \tag{15}$$

A normal noise term was added to the function $\epsilon \approx N(0, 0.1)$ for the gradient descent, giving $\sigma^2 = 0.01$. Equation (15) was used to generate a dataset with 1000 points. For the classification task, the *Wisconsin Breast Cancer* dataset will be applied. This dataset was downloaded using `Scikit-learn` from the `Python` library. The dataset comprises 569 observations of cell nuclei, classified as either benign or malignant, representing two distinct categories. It includes 212 malignant cells and 357 benign cells, with each cell nucleus characterized by 30 real-valued features [1].

III. RESULTS AND DISCUSSION

We test the theory by evaluating the performance of linear and logistic regression models, examining how each approach handles different datasets and outcomes. We start by looking at the linear regression of the function (15). We set $n = 1000$ data point and add noise $\epsilon = N(0, 0.1^2)$. All our data has been split 70/30 for the training and testing, respectively. Note that when referring to iterations, we are specifically discussing the number of updates made to the model during training.

Additionally we explored with using `Autograd` to find the gradient. `Autograd` is a Python library that automatically differentiates native `NumPy` functions, allowing for easy gradient computation. It creates a computation graph and uses the chain rule to provide gradients needed for training a neural network [2]. However, this method proved to be less time-efficient and produced results nearly identical to those obtained through our initial approach. Consequently, we opted to revert to the original method (the one we implemented ourself), as we had a deeper understanding of it, the results from this method are presented here.

A. Linear Regression

Gradient descent

We start with a simple analysis of the MSE for a plain gradient decent as a function of iterations, shown in Figure 6. Here, we observe the MSE gradually converging to approximately 0.01. Since we introduced noise of $\sigma^2 = 0.01$, this is our target MSE. A general trend is that higher learning rates lead to faster convergence, up to a threshold of $\eta = 0.2$. This is what we would expect as higher learning rates can accelerate convergence, although only to a certain threshold before potential instability occurs. Further we observe that adding momentum to the gradient descent causes oscillations to the MSE, before eventually stabilizing. This behavior is likely due to the increased "inertia" introduced by momentum, which can overshoot the target before settling. Additionally, we see that the $\eta = 0.01, \gamma = 0.9$ curve shows a similar convergence trend to the $\eta = 0.1$ curve, highlighting how momentum can effectively increase convergence rates. A notable difference appears between the $\gamma = 0$ and $\gamma = 0.9$ curves when $\eta = 0.01$, further demonstrating the impact of momentum. Lastly, after around 900 iterations, all the curves have stabilized around the target MSE.

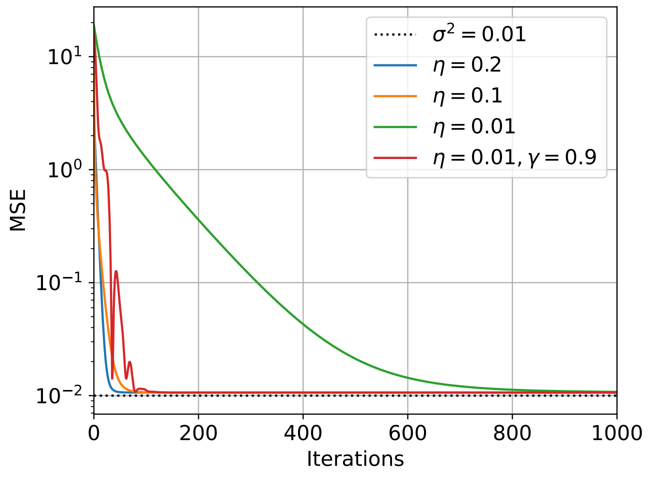

FIG. 6. MSE for each iteration with plain gradient decent. Here we compare how learning rate and momentum affect how the MSE converges toward the target value on 0.01. We see that $\eta = 0.2$ converges the fastest.

Stochastic Gradient descent

To further examine the effects of different learning rates (η) and compare plain gradient descent with stochastic gradient descent, we present the MSE after 1000 iterations and epochs as a function of η in Figure 7. This is done for no momentum and $\gamma = 0.9$, resulting in four curves. We observe that with momentum, the MSE reaches the target value at higher learning rates. This is expected as momentum accelerates convergence. Further we see that a learning rate of $\eta = 10^{-2}$ consistently produces the lowest MSE across all cases.

Among the approaches, stochastic gradient descent with momentum demonstrates the best overall performance, achieving the target MSE for most learning rates, with exceptions at $\eta = 10^{-5}$ and $\eta = 10^{-1}$. The higher MSE for $\eta = 10^{-1}$ can be a result of instability as a high learning rate can cause oscillations around the optimal point, preventing stable convergence. Overall, our results align well with theoretical expectations. Stochastic gradient descent typically achieves faster convergence as discussed in Section II.

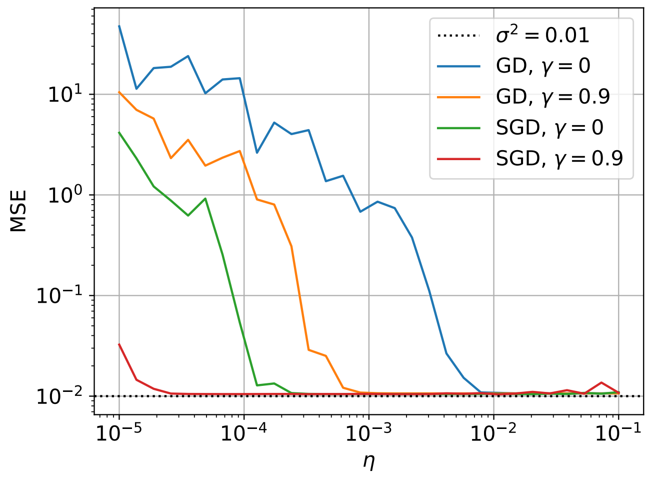

FIG. 7. MSE after 1000 iterations vs learning rate. Here we compare the result with and without momentum for plain GD and SGD. The momentum coefficient is $\gamma = 0.9$. We see that SGD with momentum performs the best with convergence for almost all learning rates.

Moreover, we search for optimal parameters for stochastic gradient descent. First, we examine the optimal number of epochs and mini-batch size, using a learning rate of $\eta = 0.1$, as shown in Figure 8. The results indicate that 1000 epochs with a mini-batch size of 5 yields the lowest MSE. However we fount that this result is highly sensitive to the learning rate as slight variations in η significantly affecting the configuration of the figure. Nonetheless, all results indicated optimal parameters around these values, so we will use them for further investigation.

Up to this point, we have only considered ordinary least squares, but now we turn to ridge regression. To identify the optimal λ value for ridge regression, we plot the MSE as a function of both η and λ in Figure **??**. While we previously examined learning rates, we revisited them here due to the high sensitivity of MSE to η.

Our results showed the optimal values to be $\eta = 10^{-2}$ and $\lambda = 10^{-4}$. A general trend observed is that larger λ values, above 10^{-3}, result in increased MSE. This may be because the regularization term penalizes the model too much, leading to under fitting. Similarly, η values smaller than 10^{-4} also yield higher MSE, perhaps due to insufficient update steps, which slow down convergence.

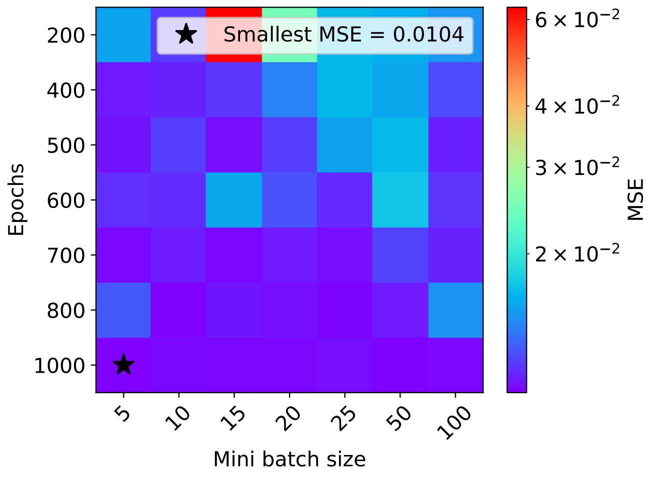

FIG. 8. The MSE for SGD for different combination of amount of epochs and size of mini batches. The learning rate is set to $\eta = 0.1$. From this case the optimal number of epochs and batch size is 1000 and 5, respectively, giving the smallest MSE.

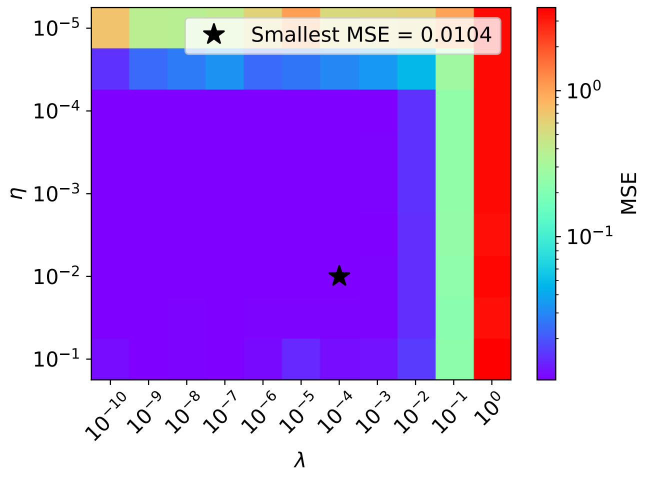

FIG. 9. The MSE for SGD for different combination of λ and η values. The values are extracted after 1000 epochs with the size of the mini batches being 5. We see that the values $\eta = 10^{-2}$ and $\lambda = 10^{-4}$ gives the smallest MSE.

Optimization

We conclude this analysis of gradient descent by examining different methods for updating the learning rate in stochastic gradient descent. In Figure 10 we compare AdaGrad, RMSprop and Adam with SGD with initial learning rate of $\eta = 0.01$, no momentum. Although it may be difficult to see, SGD performs very similarly to RMSprop in terms of the rate of convergence. However, we notice that RMSprop does not seem to stabilize, as its results remain noisy throughout. Adam converges slightly slower than both SGD and RMSprop, but maintains stability at our target value. In contrast, AdaGrad has not reached the target value after 1000 epochs. When we increased the number of epochs, we found that it needed about 2000 iterations to match the performance of the other methods.

These results are very sensitive to the learning rate. For example, with $\eta = 0.1$, RMSprop became more erratic and failed to stabilize over the iterations. In this case, AdaGrad converged almost immediately. Notably, the initial learning rate did not significantly affect Adam's results. From this, we observed that RMSprop and AdaGrad are highly sensitive to the choice of initial learning rate.

According to the theory presented in Section II, Adagrad is expected to be the most efficient and accurate optimization method. However, our findings indicate otherwise: the best-performing methods in our experiments were SGD with a learning rate of $\eta = 0.01$ and Adam. As illustrated in Figure. 10, both RMSprop and SGD initially show similar behavior, apart from SGD starting off at a lower MSE of right below $MSE = 10^0$, wheras RMSprop starts at above $MSE = 10^1$. However, as training progresses, RMSprop becomes increasingly noisy and erratic, leading to less reliable results compared to SGD. The best explanation for this deviation from theory is that the dataset may have favored the strengths of SGD and Adam, as well as the model being hyper-sensitive to learning rates. Nonetheless, this could have been a computational error in our implementation, such as improper parameter tuning, a miscalculation in the gradients, or even an inconsistency in the batch selection process.

FFNN

The neural network model for approximating a simple second-order polynomial function (15) involved tuning several hyperprarameters. To find optimal hyperparamters, an initial setup used a batch size of 32, 100 epochs, learning rate $\eta = 0.1$ and regularization $\lambda = 0.0001$. The weights were initialized using a standard initialization scheme, and the cost function were chosen as MSE. From this baseline, we explored the effect of network architecture, learning rate, activation functions and regularization.

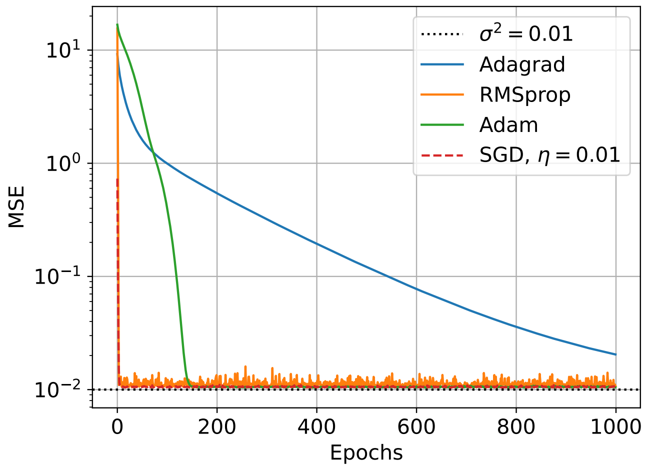

FIG. 10. The MSE as a function of epochs for SGD (without momentum) with constant learning rate compared to optimization methods for updating the learning rate. Here we used $\eta = 0.01$ and size of minibatch is 5. Here we see RMSprop converges the fastest and Adam has not converged for the 1000 iterations.

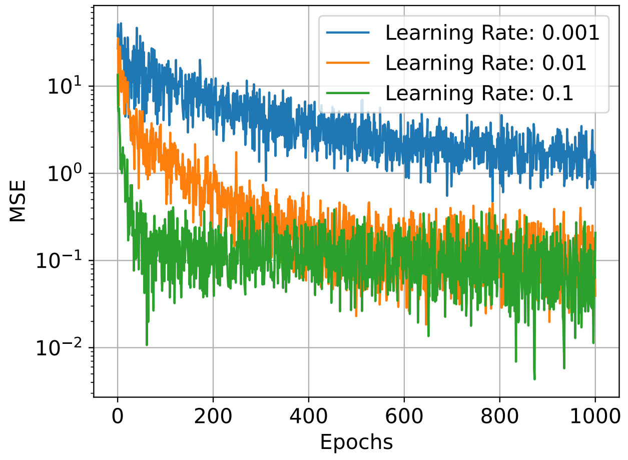

FIG. 11. The MSE as a function of epochs with $\eta = 0.1, 0.01, 0.001$. The network is with 1 layers of 50 neurons.

Learning rate selection played a crucial role in convergence and stability of the model. Hence, we start of by looking at the MSE as a function of epochs for different learning rates in Figure 11. We observed that a small increase in the learning rate, from $\eta = 0.001$ to $\eta = 0.01$, markedly improved descent speed and reduced MSE. Further refinement around $\eta = 0.1$, showed the optimal values, making the range for the polynomial. We also note that for $\eta = 0.001$, there seem to be large oscillations (as this is a logarithmic scale) and the model has a specially harder time stabilizing. This could be due to the simple nature of the polynomial

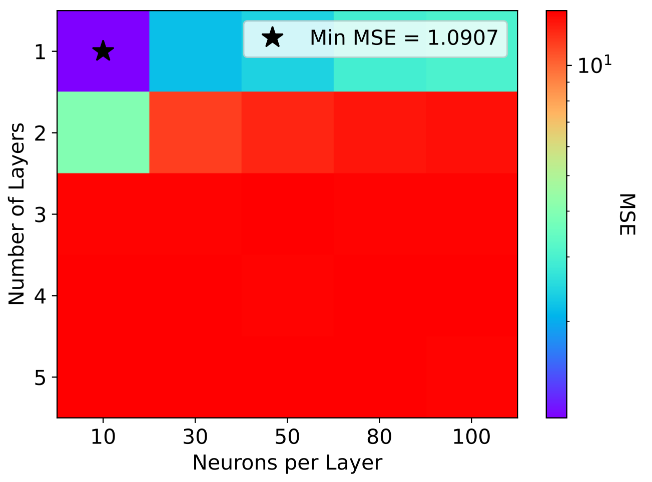

FIG. 12. MSE computed using a grid search of different neural network architectures. Choose to investigate 1-5 layers with 10, 30, 50, 80 or 100 neurons in each hidden layers. The lowest MSE was found at 1 layer with 50 neurons giving a MSE value of 1.09

and the complexness of the neural network, a small learning rate means that the training process is long and the parameters can get ”stuck” bouncing back and forth without making meaningful progress toward the minimum of the loss function, hence it can lead to some sort of oscillations.

A key finding was that simpler network architectures, with fewer hidden layers and neurons per layer, outperformed more complex setup as one can see in Figure 12, where the smallest MSE is 1.091. Increasing the network’s complexity resulted in higher MSE values especially for more than 2 hidden layers and the number of neurons increased. This outcome can be attributed to overfitting risks and the larger number of parameters to optimize. The optimal architecture was found at a single layer with a moderate number of neurons, 10, suited to the simple, polynomial regression tasks. This architecture reduced both computational cost and overfitting, showing that for simpler problems, a less complicated network is sufficient. Again we will mention that this is a very large MSE value, likely due to the simplicity of the model, as mentioned before, but it can also stem from the fact that we have not yet tuned our parameter.

Different activation function-Sigmoid, ReLU and Leaky ReLu- were evaluated for their impact on performance and stability in Figures 13, 14 and 15 respectively, their MSE values are also showcased in table I and R^2-score in II. Sigmoid activation required an overflow prevention method due to large inputs; this was achieved by

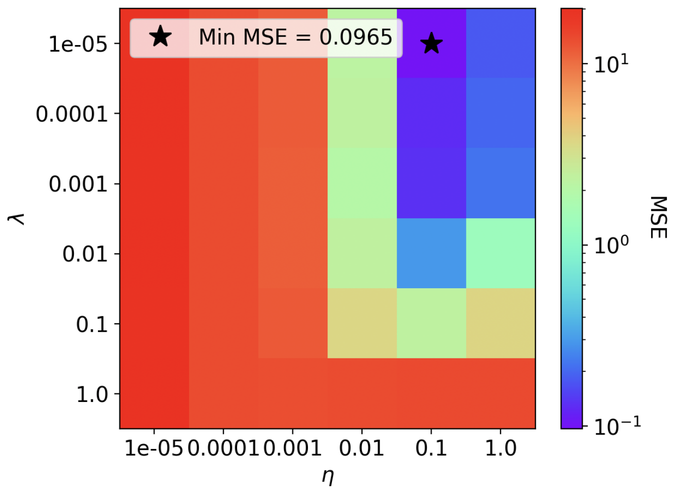

FIG. 13. MSE for grid searching for optimal learning rate and regularization on a neural network. The activation function was sigmoid with standard normal initialization of the weights. We see the smallest MSE of 0.097 for $\eta = 0.1$ and $\lambda = 10^{-5}$.

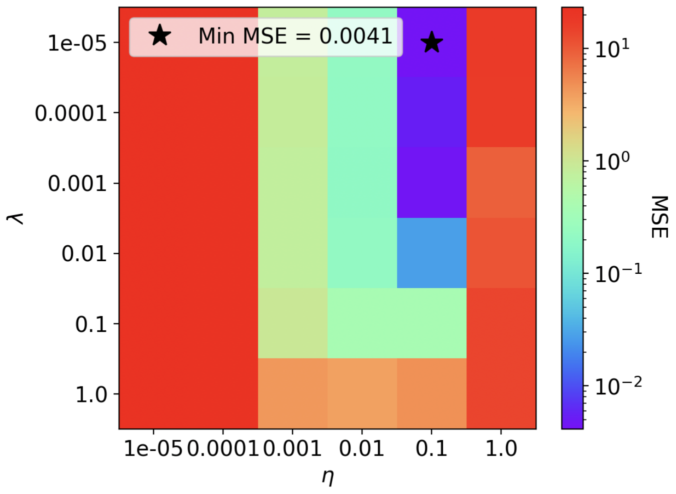

FIG. 14. MSE for grid searching for optimal learning rate and regularization. The activation function was ReLU with standard normal initialization of the weights. The smallest MSE was 0.0041 with the parameters $\eta = 0.1$ and $\lambda = 10^{-5}$.

splitting computations based on the sign of x, which prevented gradient issues in high-magnitude inputs. ReLu and leaky ReLU functions performed comparably, especially when paired with normalized weights initialization to prevent exploding gradients. Sigmoid activation was particular sensitive to learning rate changes, showing optimal results around $\eta = 0.1$. In contrast, ReLu and leaky ReLU tolerated a broader range of learning rates, contributing to smoother training and overall model stability. Leaky ReLU achieved a low MSE at 0.0021, likely due to its non-zero gradient for negative inputs, which helps avoid neuron "dying" and supports consistent gradient flow through the network. However, the MSE of

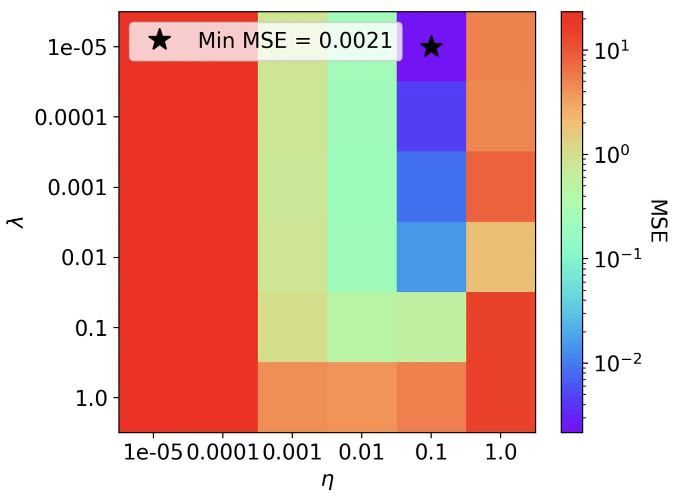

FIG. 15. MSE for grid searching for optimal learning rate and regularization on a neural network with 50 neurons and 1 hidden layer. The activation function was leaky ReLU with standard normal initialization of the weights. The smallest MSE was 0.0021 with the parameters $\eta = 0.1$ and $\lambda = 10^{-5}$

ReLU was at 0.0041. This could be because the activation sets all negative input values to zero, resulting in sparse activations, so only a subset of neurons are active at any given time. This could lead to simpler and more robust representations for the simple polynomial without introducing unnecessary complexity by keeping neurons alive. Regularization influenced MSE significantly, particularly at higher values, where it lead to MSE increases. This effect underscored the models sensitivity to hyper parameter tuning.

TABLE I. Comparison of MSE for neural network with different activation functions; Sigmoid, ReLU and Leaky ReLU with Scikit-learn.

Method	**MSE**
Neural Network with Sigmoid	0.097
Neural Network with ReLU	0.041
Neural network with Leaky ReLU	0.021
Scikit-learn	0.014

TABLE II. Comparison of R^2 score for Neural network with different activation functions; Sigmoid, ReLU and Leaky ReLU with Scikit-learn

Method	**MSE**
Neural Network with Sigmoid	0.86
Neural Network with ReLU	0.98
Neural Network with Leaky ReLU	0.98
Scikit-learn	0.99

Effective FFNN performace in polynomial regression required careful tuning of network architecture, learn-

TABLE III. Optimal hyperparameters for minimum MSE for a second order polynomial.

Hyperparameters	**Optimal values**
Layers	1
Batch size	32
Neurons per Layer	10
Epochs	32
Initialisation	Normalized
Learning rate	0.1
Regularization	10^{-5}
Activation function	Leaky ReLU

ing rate and initialization methods. The optimal configuration can be found in Table III- featuring a single hidden layer, a moderate learning rate, small regularization and RELU activation function-provided robust across grid searches, achieving competitive MSE, where a comparison can be found in Table I. Simpler architectures and moderate learning rates and smaller regularization performed best. For both the R^2- score that can be found in table II and the MSE, the MLPRegressor in `scikit-learn` yielded the best results, likely due to optimized training routines that maintain stability and reduce the risk of gradient-related issues. Based on these results, further optimization could include exploring adaptive learning rates and alternative regularization techniques, although a simpler linear model may still be more suitable for polynomial approximations. However, the R^2-score for both ReLU and leaky ReLU were close to the `Scikit` performance showcasing that our network regression model has a high level of predictive accuracy. Hence the two activation functions capture almost all the variability of the simple polynomial and generalizes well to unseen data.

B. Logistic Regression and Classification

In this section, we show the results of our modeling for logistic regression using gradient descent and a feed-forward neural network (FFNN) on the Wisconsin Breast Cancer dataset. We looked at how well each model classified the data by measuring accuracy. We also checked how changing the learning rate and regularization affected accuracy to see what worked best for each model.

Gradient descent

Similar to the case of linear regression discussed in Section III A, we begin by examining the plain gradient descent method.

First and foremost, examining the accuracy plot in Figure. 16 for different learning rates η, we see that all cases reaches an accuracy above 0.95 relatively quick,

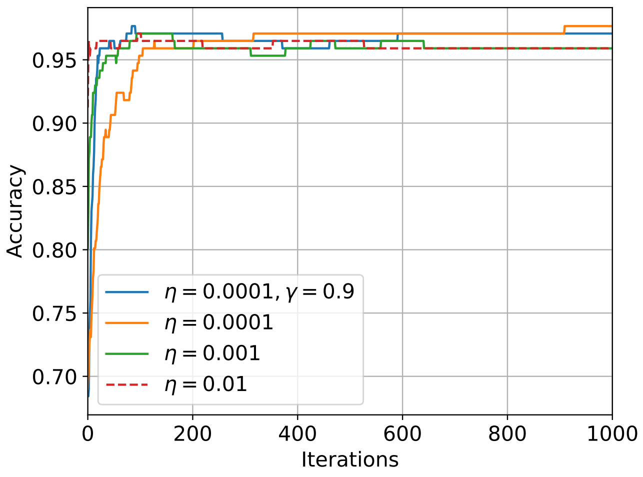

FIG. 16. How the accuracy is updated as a function of iterations. Here we see how different learning rates and momentum affect how the MSE converges.Here we see how adding momentum speeds up the converges.

the $\eta = 0.01$ being the fastest. However we note that $\eta = 0.0001$, with and without momentum reaches the highest accuracy of 0.975 after around 100 and 900 iterations, respectively.

The curves display discrete behavior due to the nature of the accuracy score. This is a result of the accuracy always being a fraction of a natural number over our number of data. Since our test dataset is not to large, we are able to see these discrete values. Overall, $\eta = 0.001$ and $\eta = 0.01$ performs the worst, especially as the iterations progress, likely due to the learning rates being too high, causing the optimization process to overshoot the minimum. This may lead to oscillations around the optimal solution, preventing proper convergence and resulting in poorer performance. However, this is not by much as almost all values are around 0.96.

Stochastic Gradient descent

Now we include stochasticity by looking at Figure. 17, showcasing an accuracy vs. learning rate (η) graph for both Gradient Descent (GD) and Stochastic Gradient Descent (SGD), with and without momentum, after 1000 iterations and epochs. We observe that for larger learning rates, SGD without momentum outperforms the others, even though it had the lowest accuracy of approximately $0.70-0.75$ for smaller learning rates. However, there is a general trend of the models performing better for learning rates at $\eta = 10^{-3}$ and larger. All displayed curves exhibit sharp fluctuations. This behavior could be attributed to the sensitivity of the learning process to the chosen learning rate, where small changes in η can lead to larger variations in the results.

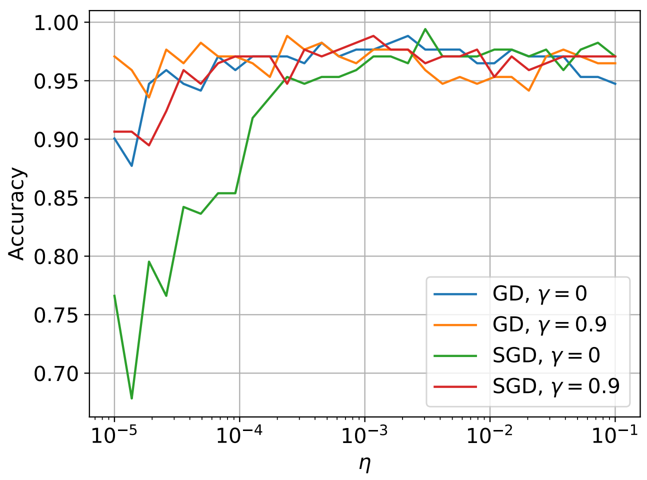

FIG. 17. Accuracy vs learning rate after 1000 iterations for GD and 1000 epochs for SGD. Here we compare the result with and without momentum for plain GD and SGD. The momentum coefficient is $\gamma = 0.9$

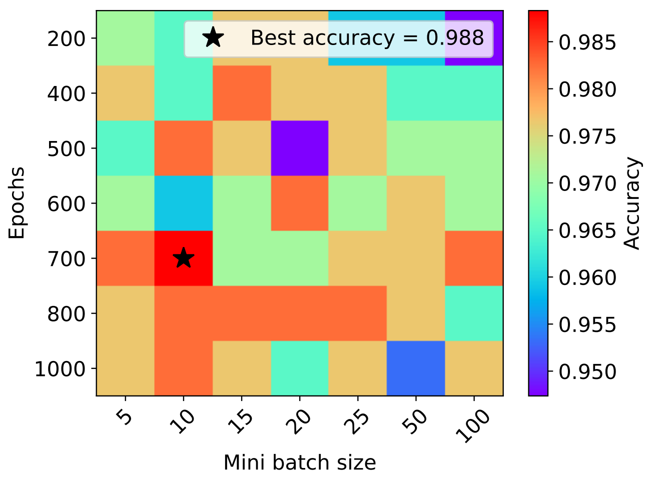

FIG. 18. The accuracy for SGD for different combination of amount of epochs and size of mini batches. The learning rate is set to $\eta = 0.001$. From this case the optimal number of epochs and batch size is 700 and 10, respectively, giving the best accuracy.

Nonetheless, we see that GD with momentum mostly outperforms GD without momentum, as previously observed in similar graphs. This is expected because adding momentum helps smooth out updates and can accelerate convergence. The lack of a similar improvement for SGD with momentum might be due to the stochastic nature of SGD, which can introduce noise that offsets the benefits of momentum, especially in the initial stages of training.

Moving onto the color maps, which use shades from red (higher accuracy) to purple (lower accuracy) to quickly show model performance across parameters. This slope highlights accuracy trends efficiently, though the plots may vary due to stochastic effects or sensitivity to the learning rate, discussed further below.

Again, we want to investigate the optimal parameters for our model, this time to improve accuracy. We start off by looking at the accuracy as a function of epochs and mini batch sizes in Figure . 18. First off, we see a randomness in the parameters yielding the different accuracies. However, one thing to note is a red (good accuracy) section for smaller mini batch sizes and larger number of epochs indicating a general trend of values for good performance. This is also where we have pinpointed the best accuracy, measuring at 0.988, occurring at 700 epochs with a mini-batch size of 10. Below 700 epochs, the accuracies fluctuate the most, perhaps due to under-fitting, where the model has not had enough epochs to learn the data effectively.

As we increase the mini-batch size, accuracies remain relatively high below the threshold of 700 epochs. This could be explained by larger mini-batch sizes usually providing more stable estimates of the gradients, allowing for more consistent updates to the model parameters during training. Conversely, when moving above 700 epochs, the accuracies begin to decrease slowly, which could be due to insufficient training time for the model to converge and fully capture the patterns in the data.

We observe accuracies that go below 0.98, especially when we decrease the epochs and increase the mini batch size, which coincides with the theory presented in Section II. However, we also detect random patterns of fluctuating accuracies, despite the margin of accuracy being small (0.95-0.988). For example, at 500 epochs with a mini-batch size of 20, we achieve an accuracy of 0.95, which is surrounded by accuracies around 0.98. This could be as a result of for instance outliers or noise. Moreover, we see that a low mini batch size and a rather high epoch size yields the best values of accuracy, closing in on 1.0. Nonetheless, due to the previously mentioned small interval of accuracies, all accuracies staying 0.95 and above, we can conclude that these parameters are not determining factors for our results. Nevertheless we will use 700 epochs and size 10 for the mini batches for the future simulations.

In Figure. 19, we examine the accuracy as we vary the learning rate, η, and the regularization parameter,λ, using 700 epochs with a mini-batch size of 10. Here we see a general trend of sections with higher and low accuracy. In the middle of the high accuracy section we see the best accuracy of 0.988, marked by a black star, is at $\eta = 10^{-3}$ and $\lambda = 10^{-2}$, the same accuracy we got for Figure. 18.

From out plot, we observe there is a broad range higher accuracies (0.96 and above). We see that the interval $\eta \in [10^{-3}, 10^{-2}]$ and $\lambda \in [10^{-9}, 10^{-2}]$ provides the best accuracies, especially around $\lambda = 10^{-2}$. This can be

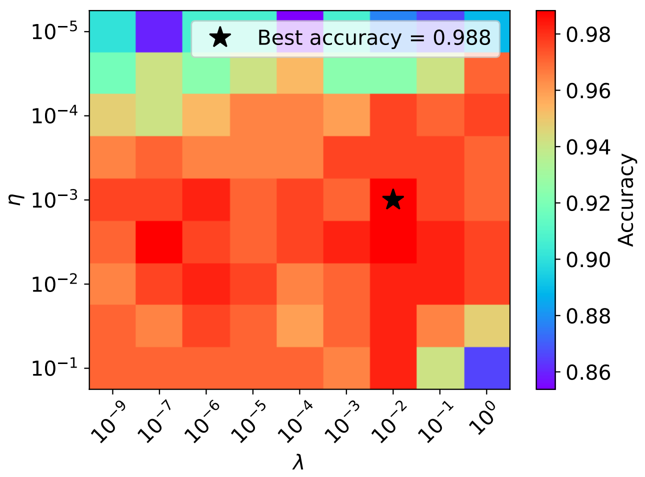

FIG. 19. The accuracy for SGD for different combination of λ and η values. The values are extracted after 700 epochs with the size of the mini batches being 10. We see that the values $\eta = 10^{-3}$ and $\lambda = 10^{-2}$ gives the best accuracy. The results indicate that model accuracy is highly sensitive to both the learning rate and minibatch size.

explained by how the learning rate and regularization strength work together in the training process.

A learning rate η between 10^{-3} and 10^{-2} helps the model learn effectively. This range allows the model to make good-sized steps toward finding the best solution without jumping too far and missing the target. If the learning rate is too high, the model might skip over the best solution, and if it is too low, the model learns too slowly and may not converge.

Additionally, while $\lambda = 10^{-2}$ offers excellent performance, the high accuracy at both lower and higher values of λ suggests the model is robust to variations in regularization strength. Lower λ values allow more flexibility, capturing complex patterns in the training data, which can be beneficial for complex data, like our breast cancer data. However, too low a λ can lead to overfitting, especially with noisy data. Higher values of λ can also yield good accuracy by enforcing stronger penalties on complexity, promoting simplicity and helping to prevent overfitting, particularly when the dataset contains outliers. Thus, the observed accuracy across various λ values indicates that the model adapts well to different levels of regularization. The best performance around $\lambda = 10^{-2}$ likely results from its optimal trade-off, maintaining enough flexibility to learn from the data while enforcing simplicity to avoid overfitting. This allows for good performance across a range of λ values, emphasizing the importance of the tuning of both η and λ for optimal results.

Areas with lower accuracies show several minima, for example at coordinates (λ, η) of $(10^{-7}, 10^{-5})$, $(10^{-4}, 10^{-5})$, and $(10^{-1}, 10^{-5})$. These lower accuracies may be due to the very low learning rate $\eta = 10^{-5}$, which could slow convergence and reduce the model's learning ability. Interestingly, another minimum is observed at $(10^{0}, 10^{-5})$, possibly due to the interaction of a high λ with the very low η, limiting the model's updates too much to learn effectively. The high λ heavily penalizes larger weights, while the small η slows the learning process significantly. As a result, the model may struggle to escape local minima or make sufficient progress toward a more optimal solution, leading to this drop in accuracy.

Optimization

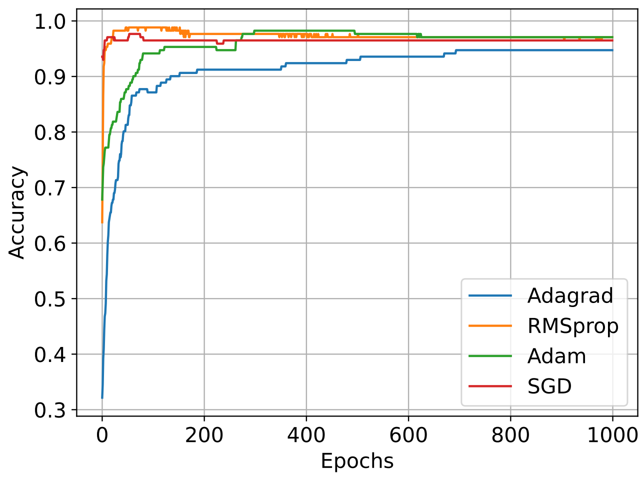

FIG. 20. The accuracy as a function of epochs for SGD (without momentum) with constant learning rate compared to optimization methods for updating the learning rate. Here, we used a learning rate of 0.01 and a minibatch size of 10, 700 epochs to compare the accuracy of different optimization methods.

As previously done in Section III A, we compare the different optimization methods—AdaGrad, RMSprop and Adam, with each other and SGD—to determine which one yields the best performance and highest accuracy. This is shown in Figure. 20. Here we observe a discrete behavior in the curve, similar to Figure 16. Initially (and throughout), AdaGrad performs the worst, with an accuracy barely above 0.3, while SGD performs the best, starting at around 0.94 accuracy. The discrepancy may arise from how each method adjusts the learning rate. SGD keeps the learning rate steady, allowing it to respond quickly to big changes in the gradients, which can give it a boost in accuracy at the start. On the other hand, AdaGrad lowers the learning rate for parameters that have already received larger updates. This can make its early steps too cautious, especially in areas with bigger gradients, leading to a slower start in accuracy. While this scaling can help later in training, it can slow AdaGrad down at the beginning compared to the other methods.

Initially, Adam outperforms RMSprop; however, this does not last for many epochs. RMSprop takes the lead, increasing almost perfectly vertically and even outperforms SGD at just under 50 epochs. This improvement could be due to RMSprop's ability to adapt its learning rate by giving more weight to recent gradients through exponential decay. This helps it adapt to the loss function more effectively, allowing it to improve more quickly in comparison to SGD, which uses a constant learning rate that may struggle with varied gradients.

RMSprop maintains its lead until it is surpassed by Adam at around 350 epochs. This shift may happen because Adam takes RMSprop's adjustable learning rate and adds momentum, which helps it remember previous gradients. This combination allows Adam to make more consistent updates and follow a smoother path toward the best solution. As RMSprop's performance begins to fluctuate or slow down, Adam's approach enables it to keep improving, ultimately leading to better accuracy in later epochs as it stabilizes.

Trailing back at the first 0-50 epochs, we observe that all methods start with a vertical progression, which deviates steeply for all but Adam. Adam exhibits a more linear progression with a smaller gradient until around 90-100 epochs. This steadiness can be attributed to Adam's momentum component, which prevents it from making abrupt changes based on potentially noisy early gradients, allowing for more consistent improvements. In contrast, AdaGrad stabilizes overall below all the other methods, remaining at lower accuracy levels throughout the epochs. This is primarily due to AdaGrad's mechanism of accumulating squared gradients, which leads to a rapid decay of the learning rate. As a result, it becomes unable to make significant updates as training progresses, leaving its performance flawed.

Overall, the highest accuracy point, almost reaching 1.0, is achieved by the RMSprop method, which performs best at lower epochs. Moreover, SGD performs well and stabilizes quickly at values above 0.95 in accuracy throughout the progression of the graph. On the other hand, Adam gradually comes out on top as the epochs increase, working best at epochs above 400. This is because Adam changes its learning rates based on two things: the average of past gradients (how the error changes) and the average of the squared gradients (how consistent those changes are). This means Adam looks at both the direction of the error and how steady the updates have been. By using this information, Adam can make smarter adjustments to the model's weights, helping it improve more as training goes on. This ability to adapt makes Adam more effective at finding the best solution as training continues.

FFNN

The FFNN was used for classifying the Wisconsin Breast cancer dataset, with 70 % of the data allocated for training and 30 % for testing. Cross-entropy was selected as the loss function due to its suitability for classification tasks, as it effectively measures how closely the predicted probabilities match the true class labels, thereby encouraging better classification accuracy.

A grid search was performed to identify the optimal network architecture, as shown in Figure 21. The results indicated that a simpler configuration- one hidden layer with 100 neurons- provided the highest accuracy of 0.96 before tuning the other hyperparameters. This outcome aligns with the relatively small, noise-free nature of the dataset, which does not require additional layers for effective learning. A single-layer architecture not only reduces the risk of overfitting by limiting the number of parameters, but also accelerates training by shortening the gradient propagation path. With only 30 features and 569 samples, the dataset benefits from a model that captures the necessary nonlinear relationships without introducing excessive complexity. The optimal value of 80 neurons achieves a balance between sufficient model capacity to capture patterns and manageable complexity, avoiding underfitting while minimizing overfitting risks.

In the hyperparameter search depicted in Figure 22, a learning rate of $\eta = 0.0$ and regularization parameter $\lambda = 0.1$ were found to yield optimal accuracy at 0.99. Given the dataset's limited size, a lower learning rate provided more controlled, incremental updates, which is particularly valuable in avoiding potential oscillations that could arise from overly large updates. Additionally, lower regularization allowed the model to learn from the data without excessively penalizing the weights, thus avoiding underfitting and ensuring generalization. Th optimal values for η and λ led to effective model adaptability without overfitting to training-specific noise, enabling the FFNN to reach stable, high accuracy.

Number of hidden layers	1
Number of hidden neurons	100
Epochs	100
Batch size	32
learning rate	0.0
Regularization	0.1
Activation function	Sigmoid

TABLE IV. Showcasing the optimal hyperparameters for the breast cancer data

Table IV summarises the optimal hyperparamters for this FFNN classification task. A grid search was also performed to find the optimal activation function which was chosen to be sigmoid. This suited the binary classification due to the fact that it maps any input to a range between 0 and 1, making it ideal for representing probabilities. Furthermore, the sigmoid introduces

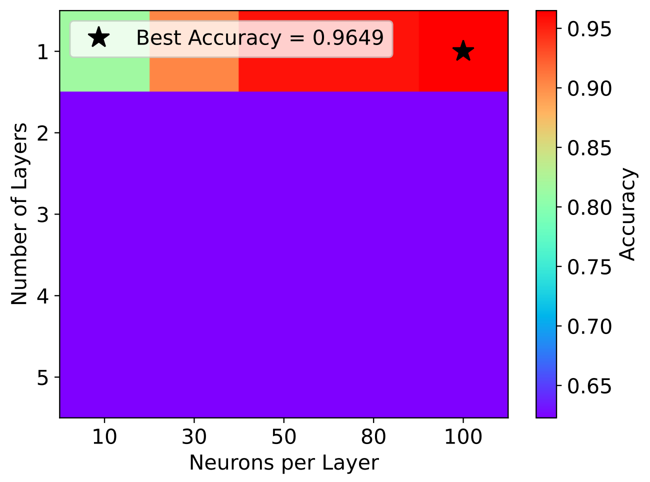

FIG. 21. Grid search of accuracy score on the FFNN architecture with varying number of neurons and number of layers. The best accuracy was found at 1 layer with 80 neurons. The activation function was sigmoid.

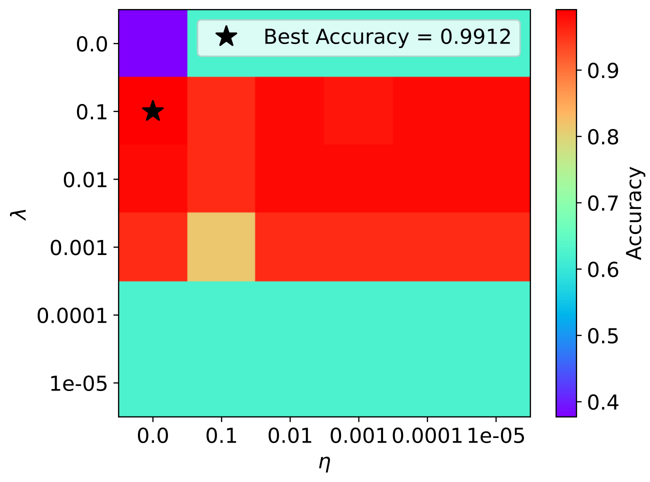

FIG. 22. Grid search to find the optimal values for the hyper parameters η and λ. The best accuracy was found at $\eta = 0.0$ and $\lambda = 0.1$, with 1 layer of 80 neurons and sigmoid as activation function.

non-linearity, which allows the model to learn complex features suitable for real-world data like the Wisconsin dataset. Notably, the required number of neurons was higher in the classification task than in the regression task shown table III, likely due to the additional complexity of mapping input features to discrete class labels. However, the single layer structure remained optimal, indicating that added layers would not improve the model performance, possibly due to the datasets simplicity.

Figure 23 illustrates how accuracy varied with epochs across different learning rates. Notably, all tested learning rates converged toward approximately 100 % accuracy by 50 epochs, with $\eta = 0.01$ reaching the highest stability and showing an average accuracy of 95 % across epochs. This rapid convergence highlights the efficiency of the selected hyperparamter for this classification task.

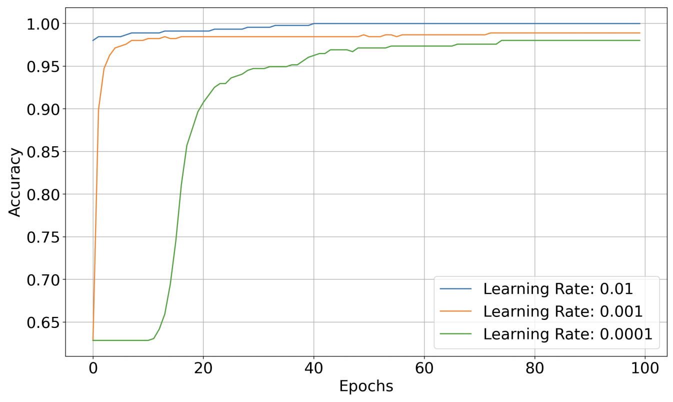

FIG. 23. Accuracy for different learning rates. Neurons = 50, epochs = 100, batch size = 32, $\lambda = 0.0$ and sigmoid as activation function.

C. Comparison

Now we have come to the meat of this project: the comparison of Gradient Descent and Feed-Forward Neural Networks for two regression tasks—linear regression on a second-order polynomial dataset and logistic regression using breast cancer data. We begin by examining the linear regression, where we evaluate the performance of both methods primarily through mean squared error (MSE) and the rate of convergence. Following this, we analyze logistic regression, assessing the same parameters along with accuracy in relation to the learning rate, η, and the regularization factor, λ. It is important to note that the simplicity of our dataset may increase the risk of overfitting in the FFNN model as a default error.

Linear regression

As previously mentioned, we tested (S)GD and FFNN methods on a simple second-order polynomial in the case of linear regression. It's important to note that these methods differ significantly in their nature and scope. Our experiments showed that gradient descent (both GD and SGD) performed more stable than FFNN for this straightforward polynomial. However, FFNN's complexity makes it better suited for handling more intricate data, such as the breast cancer dataset. Further elaboration on this is provided below.

In plain gradient descent, the gradient is calculated using the entire dataset, leading to a steady decrease in MSE toward approximately 0.01, aligning with our noise level ($\sigma^2 = 0.01$). While higher learning rates can accelerate convergence, they risk instability beyond a threshold (e.g., $\eta = 0.2$). Adding momentum results in initial oscillations but ultimately enhances convergence rates, particularly with $\gamma = 0.9$ compared to $\gamma = 0$, smoothing the approach to the target MSE after about 900 iterations.

In contrast, SGD updates parameters for each data point, introducing randomness that can destabilize convergence but often leads to faster optimal values than GD. Momentum enhances SGD's performance, achieving target MSE for most learning rates, especially at $\eta = 10^{-2}$, where it minimizes fluctuations and maximizes convergence.

The FFNN trained on the polynomial regression task involved tuning parameters like batch size, learning rate, regularization, and architecture. A simpler configuration with one hidden layer of 10 to 50 neurons yielded the lowest MSE, while complex architectures caused overfitting. A learning rate of $\eta = 0.1$ enabled stable convergence, especially with Leaky ReLU, achieving an MSE of 0.0021. Leaky ReLU and ReLU outperformed Sigmoid in MSE and R^2 scores, with Leaky ReLU effectively mitigating inactive neurons. Regularization successfully prevented overfitting, with an optimal parameter of $\lambda = 10^{-5}$.

In comparing the findings from both methods, we observed that while both SGD and FFNN achieved effective MSE values, they exhibited distinct strengths and limitations in the context of polynomial regression. SGD was straightforward and fast, particularly with the application of momentum, achieving moderate MSE scores quickly. The simplicity of the task likely favored such gradient-based methods. Conversely, FFNN demonstrated more power but required extensive tuning, including the careful selection of activation functions, learning rates, and architecture. When optimized, particularly with the Leaky ReLU activation function, FFNN achieved MSEs comparable to Scikit-learn's MLPRegressor, showcasing its ability to approximate the polynomial effectively.

In a nutshell, for this second-order polynomial regression, a linear model using SGD is simpler to implement and can achieve acceptable accuracy without extensive parameter tuning. However, the FFNN, once optimized, demonstrated comparable performance with low MSE and high R^2, indicating effective generalization to unseen data. Given the inherent complexity of an FFNN, it may be more suited to problems involving non-linear patterns or more complex datasets, as this task could be adequately addressed with simpler models.

Logistic regression

This analysis underscores the effectiveness of FFNN for classifying datasets with simple structures, such as the Wisconsin Breast Cancer data. Comparatively, in linear regression tasks, the FFNN's higher complexity and parameter count may increase the risk of overfitting, especially on simple data. For more complex data, where a neural network's learning flexibility could be advantageous, the FFNN's additional layers and neurons might prove beneficial. In this context, comparing the FFNN with logistic regression showed that while logistic regression is simpler to tune, a carefully optimized FFNN can achieve similarly high accuracy and stability with proper parameter selection.

For the classification case the trend is different with both methods giving great accuracy for several cases. Notebly, the FFNN manage to reach perfect accuracy which the (S)GD did not. This can be a result of that FFNNs are designed to learn more expressive feature representations through multiple layers, which likely enhances their effectiveness in classification tasks compared to the straightforward optimization approach of (S)GD.

The FFNN, with a single hidden layer of 80 neurons, achieved an impressive accuracy of 0.99 after 50 epochs. The ReLU activation function enabled effective pattern recognition in the data. Its rapid convergence was due to a learning rate of 0.0, minimizing updates, and a regularization parameter of 0.1 to prevent overfitting. The FFNN maintained high accuracy across epochs, demonstrating robustness and stability. Additionally, it was relatively insensitive to parameter variations, making it a straightforward and reliable choice for classification.

In contrast, the SGD model achieved a maximum accuracy of approximately 0.975, indicating a trade-off between simplicity and effectiveness. It was more sensitive to parameter selection, particularly learning rate, mini-batch size, and momentum. Smaller learning rates led to slower convergence and required more epochs, while larger rates caused oscillations in the loss function, affecting stability. An optimal setup with a learning rate of 0.01 and a mini-batch size of 10 yielded an accuracy close to 0.988, but this performance varied significantly with different parameter configurations.

The FFNN not only outperformed the SGD in terms of speed but also demonstrated reliability, achieving near-perfect classification with minimal tuning. Conversely, the SGD model required extensive experimentation with parameters to reach comparable results. The FFNN's architecture also allowed for easy adjustments and extensions, such as adding layers or neurons, potentially improving performance on more complex datasets. In contrast, the tuning complexity of SGD posed challenges, especially for practitioners lacking extensive experience in parameter optimization.

In summary, while both models demonstrated strong classification capabilities on the Wisconsin Breast Cancer dataset, the FFNN outperformed SGD in accuracy and stability. The robustness and adaptability of the FFNN architecture stand in contrast to the variability and complexity associated with tuning the SGD model. This comparison underscores the significance of model selection and parameter management in machine learning tasks, particularly when addressing datasets of varying complexity. That said, both models performed well and produced similar results, demonstrating their adequacy for both linear and logistic tasks.

IV. CONCLUSION

In this project, we looked into the performance of the gradient descent and FFNN method for performing linear and logistic regression. Our analysis included synthetic datasets such as a simple one-dimensional function, more specifically a second order polynomial, as well as real-world data; the Wisconsin Breast Cancer dataset.

For the linear regression we found that including momentum and stochasticity GD, improved the performance with faster converges and less sensitivity to the learning rate. Here we reached best MSE of 0.0104 which is very close to the 0.01 we aimed for. These result where dependent on having the optimal parameters for learning rate $\eta = 0.01$, $\lambda = 10^{-4}$, number of epochs (1000) and size of mini batch (5) values. We also found that for method optimizing the learning rate, Adam was more stable but RMSprop converged faster, both reaching a MSE value of 0.01. When looking at the same cases performed by FFNN, the MSE took larger values before parameter optimization, around 0.1 after 1000 epochs for $\eta = 0.1, 0.01$. However, after implementing optimizations and looking at different activation function, we got the smallest MSE of 0.0021 with leaky ReLU as activation function in a 1 hidden layer with 10 neurons, $\eta = 0.1$, $\lambda = 10^{-5}$ network architecture.

For the logistic regression we got the best accuracy of 0.988. This was achieved by setting learning rate η = 0.001, $\lambda = 10^{-2}$, number of epochs to 700 and size of mini batches to 10. After 1000 epochs the methods for optimizing the learning rate performed very similar, but Adam was the winner by a small margin. Here, FFNN performed way better by reaching an accuracy of 0.99 with a network architecture with 1 hidden layer with 100 neurons, $\eta = 0.0$ and regularization $\lambda = 0.1$ and sigmoid as activation function.

This lead to the conclusion that even though GD showed more stability, FFNN provided the best MSE. The same goes for accuracy. However the FFNN needed more tuning of the parameters and activation functions to deliver the nice results. It's worth noting that the difference between the two performances wasn't dramatic. However they both showed strengths and weaknesses.

Notes for future work

Future improvements could focus on testing FFNN and regression models on a wider variety of datasets, both synthetic and real-world, to better assess model generalizability. Initially, we tested the methods using a simple second-order polynomial, which has only one extreme point; this characteristic does not reflect the complexity typically found in real data. Increasing the order of the polynomial could lead to better results in capturing more intricate patterns. Expanding the training data volume may help address overfitting and enhance accuracy, particularly for complex datasets like the Wisconsin Breast Cancer dataset. Additionally, exploring alternative metrics beyond MSE and varying optimization parameters such as the learning rate, minibatch size, and regularization terms could provide deeper insights. A more detailed examination of how these metrics change with different data parameters would further validate model performance and robustness.

REFERENCES

[1] University of California Irvine. *Breast Cancer Wisconsin (Diagnostic) Data Set.* 2017. URL: https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data.

[2] Educative. *What is Autograd?* Accessed: 2024-11-04. 2021. URL: https://www.educative.io/answers/what-is-autograd.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* Optimization for Training Deep Models. MIT Press, 2016. Chap. 8. URL: https://www.deeplearningbook.org/contents/optimization.html.

[4] CompPhysics Group. *Project 2 - Part D: Classification Analysis using Neural Networks.* Accessed: 2024-11-03. 2024. URL: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/project2.html#part-d-classification-analysis-using-neural-networks.

[5] Gareth James et al. *An Introduction to Statistical Learning: with Applications in R.* Springer, 2013. URL: https://www.statlearning.com/.

[6] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. *Introduction to Linear Regression Analysis*. 5th. John Wiley & Sons, 2012.

[7] R. Sasirekharameshkumar. "Deep Learning Basics: Part 10 - Feed Forward Neural Networks (FFNN)". In: *Medium* (2021). URL: https://medium.com/@sasirekharameshkumar/deep-learning-basics-part-10-feed-forward-neural-networks-ffnn-93a708f84a31.

[8] Ilya Sutskever, Oriol Vinyals, and Quoc Le. "On the Importance of Initialization and Momentum in Deep Learning". In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013* (2013). *Beta values for momentum commonly set around 0.9.*, pp. 1139–1147.

[9] Shahid Tufail et al. "Advancements and Challenges in Machine Learning: A Comprehensive Review of Models, Libraries, Applications, and Algorithms". In: *Electronics* 12.8 (2023), p. 1789.

We used ChatGPT to help with fine tuning some parts of the writing of the report.