FYS-STK4155 Project 2

Tiril Trondsen, Trude Halvorsen, Lars-Martin Gihle & Jonas Båtnes

University of Oslo, Department of Physics

(Dated: November 4, 2024)

I. ABSTRACT

The volume of health and diagnostic data is rapidly increasing, necessitating more effective health treatments. As large datasets become more prevalent in healthcare, it's important to evaluate how well machine learning models perform in these complex scenarios. Relying solely on synthetic data may not reveal the challenges posed by real-world medical data structures. Therefore, understanding model performance on actual healthcare data is crucial for advancing treatment effectiveness. Selecting the right models and optimization techniques is crucial in machine learning, especially when dealing with varying data characteristics. This study examines how two common models perform on different tasks: regression using synthetic data and classification with a real-world dataset. Our findings show significant differences in performance based on activation functions and optimization choices, highlighting the importance of careful model tuning for optimal results across diverse datasets. These insights deepen our understanding of machine learning model design and their adaptability to complex data structures.

II. INTRODUCTION

In recent years, machine learning techniques such as neural networks and logistic regression have been widely adopted for both regression and classification tasks, as they offer a flexible framework for understanding complex data structures [1]. In particular, Feedforward Neural Networks (FFNNs) and Logistic Regression are foundational models that leverage nonlinear transformations to capture relationships within data [2]. Activation functions like sigmoid, ReLU, and LeakyReLU in hidden layers play a critical role in introducing non-linearity and have significant implications for model performance across tasks [3].

For this study, we apply an FFNN and a Logistic Regression model to address two distinct data challenges. The first task involves a regression analysis on synthetic data, where we investigate the effectiveness of the models in capturing continuous data patterns. The second task is a classification exercise on real-world Wisconsin breast cancer data, a well-known dataset for evaluating binary classification models [4]. In the FFNN, we experiment with different activation functions (sigmoid, ReLU, and LeakyReLU) in the hidden layers, and we vary the final activation function among sigmoid, softmax, and linear to assess how these configurations affect model outputs. To measure performance, we employ Mean Squared Error (MSE) and R-squared (R^2) for regression tasks, while accuracy is used to evaluate classification effectiveness.

Our optimization approach utilizes Adam, Gradient Descent, and Stochastic Gradient Descent (SGD) for updating model parameters, allowing us to compare the impact of each optimization method on training efficiency and convergence [5]. This comparison will provide insights into the optimization dynamics of FFNN and Logistic Regression under different configurations.

This article is structured as follows: we begin with an overview of the data sources and preprocessing steps. Subsequently, we discuss the model architectures and the activation functions used in this study. We then detail the experimental setup, including the optimization methods and performance metrics. Following this, we present and analyze the results, highlighting the influence of activation functions and optimizers on model performance. Finally, we conclude with insights gained and potential directions for future research.

III. METHODS

In this study, we employed several machine learning techniques to model and analyze synthetic second-order polynomial data and real Wisconsin breast cancer data. The primary focus was on implementing and comparing the performance of a specific Feed Forward Neural Network (FFNN) with different activation functions, Logistic Regression (LR), and the inbuilt functions in `scikit-learn`. To ensure the robustness and reliability of our results, we utilized data preprocessing and optimization techniques, including train-test splitting, feature scaling, gradient descent (GD), and stochastic gradient descent (SGD). To tune the learning rate we used the methods: AdaGrad, RMSprop, and Adam. For optimal computation of the gradients in the optimizers, we used Autograd, JAX, and the analytical expressions of the gradients.

A. Data Generation and Preprocessing

The synthetic dataset was generated using the second-order polynomial equation 1 with 100 samples of values from the uniform distribution. The coefficients a_0, a_1, and a_2 were drawn from a uniform distribution between 0 and 1. To simulate real-world data imperfections, normally distributed noise was added to the output $f(x)$. We used the seed "2024" to create consistent data. The resulting dataset was then prepared for regression analysis. The real Wisconsin breast cancer data was downloaded to the file `Wisconsin.csv` from the webpage Kaggle [6]. We picked out `"radius_mean"` and `"texture_mean"` as features.

$$f(x) = a_0 + a_1 x + a_2 x^2 \tag{1}$$

1. Train-Test Split

We split the data the same way and for the same reasons as in FYS-STK4155 project 1 [7], and explain it in the same way: "To evaluate the predictive performance of the models, the dataset was split into training and testing sets using the `train_test_split` function from the `scikit-learn` library [8]. The data was partitioned such that 80% was used for training and 20% for testing. This split ensures that the models are trained on a substantial portion of the data while reserving a separate set for unbiased evaluation".

2. Feature Scaling

We scale the data in the same way and for the same reasons as in FYS-STK4155 project 1 [7], and explain it in the same way: "Feature scaling is crucial for algorithms that are sensitive to the magnitude of the features, especially when regularization is involved. Without scaling, the features with larger values will be penalized more than those with small values. We scaled the variables using the `StandardScaler` from `scikit-learn`. Standardization transforms the data to have zero mean and unit variance, computed as:

$$z = \frac{x - \mu}{\sigma} \tag{2}$$

where x is the original feature value, μ is the mean, and σ is the standard deviation of the feature values in the training set. By centering the data, we reduce the multicollinearity of the columns with higher powers in the design matrix. This makes our model more numerically stable. It can also reduce the risk of our design matrix becoming nearly singular, which makes computation easier [9].

A downside of using the standard scaler is that it doesn't provide a minimum and a maximum value for our data set [10]. However, it is both easy to understand and implement, which we weighed more. The scaler was fit on the training data and then applied to both training and test sets to prevent data leakage between the two.

Although real data often benefits more from scaling than "perfect" data generated from a uniform distribution with a mean of zero, we scaled both the synthetic and real cancer data to be consistent throughout the project.

B. Gradient and Regression Techniques

The following gradient and regression methods were implemented:

1. Gradient Descent

Gradient Descent is an iterative optimization algorithm that aims to minimize a cost function $C(\boldsymbol{\beta})$ by updating the parameters $\boldsymbol{\beta}$ in the direction of the steepest descent. The update rule follows:

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \gamma \nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}_k) \tag{3}$$

where γ is the learning rate, and $\nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}_k)$ is the gradient of the cost function evaluated at $\boldsymbol{\beta}_k$.

Starting with an initial estimate $\boldsymbol{\beta}_0$, the algorithm iteratively updates the parameters until a stopping criterion is met, such as reaching a maximum number of iterations or when the change in $C(\boldsymbol{\beta})$ drops below a threshold. The choice of the learning rate γ is crucial; if γ is sufficiently small, each iteration ensures that $C(\boldsymbol{\beta}_{k+1}) \leq C(\boldsymbol{\beta}_k)$, meaning the algorithm consistently moves towards smaller function values and thus towards a minimum [10].

The gradient descent algorithm has several limitations. For the non-convex problems that are often encountered in machine learning, the algorithm may converge to a local minimum, instead of a global one. Gradient descent is also sensitive to the choice of initial conditions β_0, where different starting points can result in convergence to different local minima. The choice of the learning rate γ is crucial, as the wrong values may lead the algorithm to converge too slowly, or even fail to converge at all. Lastly, computing the gradient on large datasets is extremely computationally expensive [10].

2. Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) extends the standard Gradient Descent algorithm to handle large datasets more efficiently by introducing randomness into the gradient estimation [10]. Instead of computing the gradient over the entire dataset, SGD approximates the gradient using a randomly selected mini-batch of data points, which reduces the computational cost and can help escape local minima due to its stochastic nature.

The full gradient of the cost function is given by:

$$\nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}) = \sum_{i=1}^{n} \nabla_{\boldsymbol{\beta}} c_i(\boldsymbol{x}_i, \boldsymbol{\beta}) \tag{4}$$

where n is the total number of data points, and $c_i(\boldsymbol{x}_i, \boldsymbol{\beta})$ is the individual cost for data point $\boldsymbol{x}_i$.

In SGD, the gradient is approximated using a mini-batch B_k of size M:

$$\nabla_{\boldsymbol{\beta}} C_{B_k}(\boldsymbol{\beta}) = \sum_{i \in B_k} \nabla_{\boldsymbol{\beta}} c_i(\boldsymbol{x}_i, \boldsymbol{\beta}) \tag{5}$$

The parameter update rule becomes:

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \gamma \nabla_{\boldsymbol{\beta}} C_{B_k}(\boldsymbol{\beta}_k) \tag{6}$$

where γ is the learning rate, and B_k is the mini-batch selected at iteration k. An iteration over all mini-batches is referred to as an epoch. Typically, multiple epochs are performed to train the model effectively.

By using mini-batches, SGD introduces stochasticity into the optimization process, which can help the algorithm avoid getting stuck in local minima and reduce computational cost [10].

The limitations of stochastic gradient descent are as follows: Unlike standard Gradient Descent, SGD often requires the learning rate γ to decrease over time to ensure convergence because the noise introduced by random sampling does not vanish near a minimum [1]. Additionally, the stochastic nature of the technique can lead to fluctuations around the minimum, necessitating careful tuning of hyperparameters.

3. Automatic Differentiation Tools: Autograd and JAX

Efficient computation of gradients is crucial for optimization algorithms like Gradient Descent (GD) and Stochastic Gradient Descent (SGD). Automatic differentiation tools such as Autograd and JAX facilitate this process by computing derivatives automatically, eliminating the need for manual gradient calculations.

a. Autograd is a Python library that enables automatic differentiation by dynamically building computational graphs during code execution [11]. Autograd employs reverse-mode differentiation to efficiently compute gradients of scalar outputs with respect to high-dimensional inputs, making it suitable for optimization in machine learning models. By automating the differentiation process, Autograd simplifies the implementation of gradient-based optimization algorithms like GD and SGD, allowing practitioners to focus on model design rather than manual gradient computations.

b. JAX is a high-performance numerical computing library that combines automatic differentiation with just-in-time (JIT) compilation, optimized for modern machine learning workloads [12]. JAX provides forward and reverse-mode differentiation through the `grad` function, enabling efficient computation of gradients for complex functions. JAX also supports automatic vectorization via the `vmap` function, allowing for efficient batch computations without manual loop vectorization, and facilitates parallel computations across multiple devices with the `pmap` function. This combination of automatic differentiation and high-performance execution makes JAX well-suited for implementing and scaling optimization algorithms like GD and SGD, particularly in environments requiring speed and efficiency.

4. Momentum

Momentum is an enhancement technique applied to optimization algorithms like Gradient Descent and SGD to accelerate convergence and reduce oscillations, particularly in regions with complex curvature. It does so by incorporating a velocity term that accumulates the gradient information from previous iterations.

The momentum update rules are:

$$\mathbf{v}_{k+1} = \mu \mathbf{v}_k - \gamma \nabla_{\boldsymbol{\beta}} C_{B_k}(\boldsymbol{\beta}_k) \tag{7}$$

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k + \mathbf{v}_{k+1} \tag{8}$$

where μ is the momentum coefficient (typically between 0 and 1), controlling the influence of past gradients. $\mathbf{v}_k$ is the velocity vector at iteration k, representing the accumulated gradient information, and γ is the learning rate.

Applying momentum helps smooth out the updates, allowing the algorithm to navigate through shallow minima and noisy gradients more effectively [1].

The advantages of momentum are as follows: It speeds up the training process by allowing larger learning rates without the risk of divergence. Additionally, it helps dampen oscillations, especially in the presence of high curvature or noisy gradients.

By integrating momentum with SGD, we combine the benefits of both methods, achieving faster and more stable convergence.

5. *Adaptive Learning Rate Algorithms*

Standard optimization methods like Stochastic Gradient Descent (SGD) can struggle with selecting an appropriate global learning rate, potentially leading to slow convergence or overshooting minima in complex optimization landscapes [1]. Adaptive learning rate algorithms address this issue by adjusting the learning rate for each parameter individually based on historical gradient information.

a. AdaGrad (Adaptive Gradient) modifies the learning rate for each parameter by scaling it inversely proportional to the square root of the cumulative sum of its past squared gradients [1]. This adaptation allows for larger updates for parameters associated with infrequent features and smaller updates for those with frequent features, effectively adjusting the learning rate during training.

Let g_t be the gradient at iteration t, and let G_t be the cumulative sum of squared gradients up to iteration t:

$$G_t = G_{t-1} + g_t^2 \tag{9}$$

The parameter update rule is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\gamma}{\sqrt{G_t + \epsilon}} \odot g_t \tag{10}$$

where γ is the initial learning rate, ϵ is a small constant to prevent division by zero, and $\odot$ denotes element-wise multiplication.

While AdaGrad performs well for convex problems, it may suffer from aggressive learning rate decay in non-convex settings, causing convergence to slow down before reaching a minimum [1].

b. RMSprop (Root Mean Square Propagation) addresses AdaGrad's rapid learning rate decay by using an exponentially decaying average of past squared gradients instead of a cumulative sum [1]. This modification prevents the learning rate from decreasing too quickly, allowing the algorithm to remain effective throughout training.

The running average of squared gradients is calculated as:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1 - \beta) g_t^2 \tag{11}$$

The parameter update rule becomes:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\gamma}{\sqrt{E[g^2]_t + \epsilon}} \odot g_t \tag{12}$$

where β is the decay rate (typically $\beta = 0.9$).

c. Adam (Adaptive Moment Estimation) combines the advantages of RMSprop with momentum by maintaining running averages of both the gradients (first moment) and their squared values (second moment) [1]. This results in adaptive learning rates for each parameter and smoother updates, making it well-suited for non-stationary objectives and noisy gradients.

The first-moment estimate (mean) and the second-moment estimate (uncentered variance) are computed as:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{13}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{14}$$

Bias-corrected moment estimates are calculated to adjust for initialization at zero:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{15}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{16}$$

The parameter update rule is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\gamma}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t \tag{17}$$

where β_1 and β_2 are exponential decay rates for the moment estimates (typically $\beta_1 = 0.9$ and $\beta_2 = 0.999$).

Adam is widely adopted due to its efficiency and effectiveness in handling sparse gradients and noisy data.

6. Logistic Regression

Logistic regression is used for binary classification problems, predicting the probability that an input $\boldsymbol{x}_i$ belongs to class $y_i \in \{0, 1\}$. It models this probability using the sigmoid

function, which outputs values between 0 and 1:

$$p(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}) = \frac{\exp(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta})}{1 + \exp(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta})} \tag{18}$$

The probability of belonging to class $y_i = 0$ is:

$$p(y_i = 0|\boldsymbol{x}_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}) \tag{19}$$

To estimate the parameters $\boldsymbol{\beta}$, Maximum Likelihood Estimation (MLE) is used. Assuming independent data points, the likelihood function for the dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}$ is:

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{n} [p(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta})]^{y_i} [1 - p(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta})]^{1-y_i} \tag{20}$$

The log-likelihood function is then:

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left[y_i \log p(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta}) + (1 - y_i) \log \left(1 - p(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta})\right) \right] \tag{21}$$

The cost function to minimize is the negative log-likelihood:

$$C(\boldsymbol{\beta}) = -\ell(\boldsymbol{\beta}) \tag{22}$$

The gradient of the cost function, used for optimization algorithms like gradient descent, is expressed in matrix form as:

$$\nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}) = -\mathbf{X}^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{p}) \tag{23}$$

where $\mathbf{X}$ is the design matrix with rows $\boldsymbol{x}_i^{\mathrm{T}}$, $\boldsymbol{y}$ is the vector of observed labels, $\boldsymbol{p}$ is the vector of predicted probabilities $p(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\beta})$.

Parameters are estimated by minimizing $C(\boldsymbol{\beta})$ using iterative methods since there is no closed-form solution for logistic regression coefficients [10].

C. Model Training and Neural network algorithms

1. Feed Forward Neural Network (FFNN)

In artificial neural networks, neurons communicate by sending signals represented as mathematical functions, with each connection weighted to affect the output. Signals must

exceed an activation threshold for neurons to respond [10]. The FFNN passes data forward through an input layer, hidden layers, and an output layer, with each node connecting to all nodes in the next. During feed-forward, input data flows through the network via matrix multiplications and activation functions, to make a prediction, while back-propagation adjusts weights and biases to minimize errors. The architecture of a simple FFNN is visualized in Figure 1.

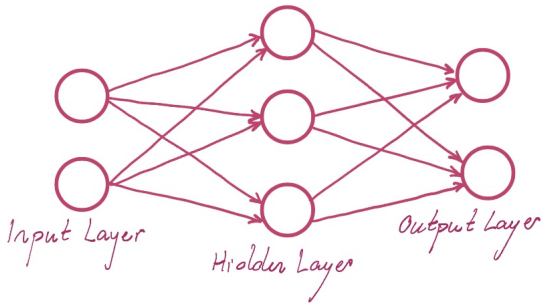

FIG. 1: An illustration of the architecture of a simple Feed Forward Neural Network.

To get our results we used a neural network with 2 nodes in the input layer and 12 nodes in two hidden layers. When using softmax as an activation function for the last layer, we used 2 output nodes as it is a requirement. In all other cases, we used 1 output node.

2. Mathematical Model

The functioning of an artificial neuron can be described mathematically. Within each hidden layer, the weighted sum of the inputs is calculated and passed through an activation function, generating outputs that proceed through the network until we get a final prediction. This process is expressed as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(z),$$

Where the output y is produced via the activation function f, where $z = (\sum_{i=1}^{n} w_i x_i + b_i)$ is the activation, b is the bias which is normally needed when we have zero activation weights or inputs, w_i are the weights and the inputs x_i are the outputs of the neurons in the preceding layer. A Multi-Layer Perceptron (MLP) is fully connected, meaning each neuron receives a weighted sum of outputs from all neurons in the previous layer [10], allowing for learning

more complex relationships. We can represent the entire MLP model as a sequence of nested functions in a generalized mathematical form:

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\dots\right) + b_k^2\right) + b_1^3\right] \tag{24}$$

Where w_{ij} are the weights for the connections between neurons, b_i represents biases at each layer, and N_l denotes the number of nodes in layer l.

3. Back progagation algorithm

Backpropagation is essential for training neural networks, as it enables us to understand how changes in weights and biases affect the cost function [13]. Neural networks learn by updating weights and biases through gradient descent. After making a prediction, the network calculates the error, which is then propagated backward, updating weights to reduce future errors. The loss function measures the prediction error, while the gradient indicates the direction and size of needed adjustments to reduce the error. Backpropagation uses the chain rule to compute these gradients, passing error corrections backward through each layer [14]. With this framework, we can introduce the key equations used in backpropagation, as presented in Nielsen's book [13].

The equation for the error gradient for the output layer, written in a matrix-based form, is:

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

Where $\odot$ is the Hadamard product, $\nabla_a C$ is a vector whose components are the partial derivatives $\partial C/\partial a_j^L$, representing how the cost function changes with the output activations. If C doesn't depend much on a particular output neuron j, then δ_j^L will be small. The $\sigma'(z_j^L)$ measures how fast the activation function σ is changing at a given activation value z_j^L.

The error δ^l in terms of the error in the next layer δ^{l+1} is:

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

This provides an effective way to calculate the error in each layer, by starting with the output error and propagating backward through the network layer by layer.

The rate of change of the cost with respect to any bias in the network is:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l.$$

Indicating that the error directly represents the rate of change of the cost function C with respect to any bias b. Lastly, the rate of change of the cost with respect to any weight in the network is:

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

By updating weights in proportion to these gradients, we can minimize the cost function.

These equations assist us in minimizing the prediction errors by adjusting the weights and biases, enhancing network performance through backpropagation. The algorithm is presented with pseudocode in Algorithm 1.

Algorithm 1 Backpropagation Algorithm [13]

procedure BACKPROPAGATION($network, X, y$)
    Initialize weights and biases in the network
    for each training example (x_i, y_i) in X, y **do**
        Forward Pass:
        Compute activations a^l for all layers l
        Backward Pass:
        Compute output error δ^L
        for each layer l from $L-1$ down to 1 **do**
            Compute error δ^l for layer l
            Update weights w^l and biases b^l using δ^l

4. Activation functions

The selection of activation functions is important for neural networks to effectively make complex predictions. The activation functions used in this project include Sigmoid, Softmax, ReLU (Rectified Linear Unit), and leaky ReLU, which will be described below.

Sigmoid

The sigmoid function is a commonly used activation function, defined as:

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

A notable feature of the sigmoid function is its biological plausibility, as inactive neurons produce an output of zero [10]. The sigmoid function features an S-shaped curve that gives a smooth transition between output values, adding non-linearity to the model.

Softmax

The softmax activation function is often effective for classification tasks, denoted by:

$$f(z_i^l) = \frac{e^{z_i^l}}{\sum_{m=1}^{K} e^{z_m^l}}$$

The softmax function converts output scores into probabilities by applying the exponential to each score and dividing by the sum over the exponentials across output neurons, ensuring the output activations sum to 1. This normalization creates a probability distribution, which is convenient in many problems for interpreting the output activation as the network's estimated probability that the correct output is i [13], and probabilities are often more intuitive to analyze.

ReLU and Leaky ReLU

ReLU is a much-used activation function in deep neural networks because of its simplicity and efficiency, as it does not saturate for positive values and is generally faster to compute than other activation functions [10]. The ReLU function is defined as:

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases} \tag{25}$$

However, ReLU can suffer from the "dying ReLU" problem, where certain neurons become inactive and consistently output zero. This issue often arises with high learning rates, which can lead to weight updates that make the neuron's weighted input sum negative [10]. Leaky ReLU resolves this problem by introducing a small positive constant α of 0.01, which helps prevent neurons from becoming inactive. The Leaky ReLU is denoted by:

$$\text{Leaky ReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{if } x \leq 0 \end{cases} \tag{26}$$

D. Implementation

Gradient descent and Stochastic gradient descent were implemented with our own code. The neural network was implemented firstly with our own code and after with the `PyTorch` library [15] for comparison. We used Mean squared error as the loss function and Adam as the Optimizer for the regression task. For classification, we used the Adam optimizer in all cases, and binary cross entropy as the loss function when sigmoid was the last layer's activation function. When Softmax was the last layer's activation function we used cross entropy as loss function. Lastly, the logistic regression was implemented with our own code before we created the same model with `scikit-learn` library [8]. Based on initial experiments, we selected a learning rate range from 10^{-5} to 10^{0} and epoch counts between 900 and 1000 as a standard for all models, as these parameters yielded the most favorable balance between model accuracy and computational efficiency. The following steps summarize the implementation process:

1. **Data Generation:** Generate synthetic second-order polynomial data with added Gaussian noise. Load real data and extract `"radius_mean"` and `"texture_mean"` as features.

2. **Data Splitting & feature scaling:** Split the data into training and testing sets using `train_test_split`. Apply `StandardScaler` to standardize the features.

3. **Model Training:** Train FFNN and Logistic Regression on the training data.

4. **Parameter Tuning:** Use a range of epochs and learning rates to find optimal MSE, R2, and Accuracy scores. Test various λ values for the Logistic Regression model.

5. **Testing:** Assess the final model performance on the testing set.

E. Performance Metrics

The performance of the regression problem was evaluated using the Mean Squared Error (MSE) and the coefficient of determination (R^2), as defined in equations 27 and 28, respectively.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{27}$$

where y_i are the observed values, $\hat{y}_i$ are the predicted values, and n is the number of observations.

The R^2 value measures the proportion of the variance in the dependent variable that is predictable from the independent variables:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}, \tag{28}$$

where $\bar{y}$ is the mean of the observed values. These metrics were calculated for both the training and testing sets to assess the models' ability to generalize to unseen data.

To measure the performance of the classification problem, we use the accuracy score:

$$Accuracy = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}, \tag{29}$$

where the number of correctly guessed targets t_i is divided by the total number of targets.

F. Large Language Models

With inspiration from FYS-STK4155 project 1 [7] we summarize our use of large language models: We have been encouraged in the group sessions to use ChatGPT [16] in writing this report. We have done so by first writing the whole paragraph, then sending it to ChatGPT with the prompt "Can you rewrite this with better and more concise language? Keep the references as they are and only keep the most important equations to explain the methods:". We have then read through the suggestion closely to make sure the values and content are the same as before. We hope that this makes it easier for the reader to follow our discussion, especially when we discuss the figures. Screenshots from conversations with ChatGPT are

uploaded in a folder on our GitHub. We have also used Github Copilot as an integrated tool [17].

G. Other Tools

We used the software Overleaf to write this report. To create the dataset along with doing basic mathematical operations we used the NumPy package [18]. Plotting our results was done with the Matplotlib package [19]. We implemented the methods with inspiration from the lecture notes in FYS-STK4155 [10]. The code for the project can be found at: https://github.com/MrSBR/FYS-STK4155_project2.

IV. RESULTS AND DISCUSSION

1. *Gradient descent and stochastic gradient descent: the relation between the loss function, momentum, optimizers, and performance*

We begin by examining the performance of the OLS and Ridge regression models trained with gradient descent as a function of learning rate and the ridge hyperparameter, λ. In figure 2 we use a learning rate of 0.12 for the Loss vs ridge parameter curve, and a ridge parameter of 0.8 for the Loss vs learning rate (ridge) curve, and train both models for 100 epochs. The figure shows that the OLS and Ridge models handle higher learning rates well, which could be explained by the relatively simple dataset. Additionally, our design matrix has the same number of parameters as the polynomial used to generate the data, which could lead to a convex optimization landscape that allows for high learning rates.

The ridge model consistently has a higher loss than the OLS model. This, along with the fact that the loss increases strictly monotonically for increasing values of the Ridge parameter suggests that a Ridge model is worse than an OLS model for training on this dataset. This could, again, be because of the dimensions of our design matrix, which largely removes the risk of over-fitting, which is what Ridge could normally help counteract. Thus, the regularization introduced by Ridge increases bias without a corresponding decrease in variance, making the model less accurate.

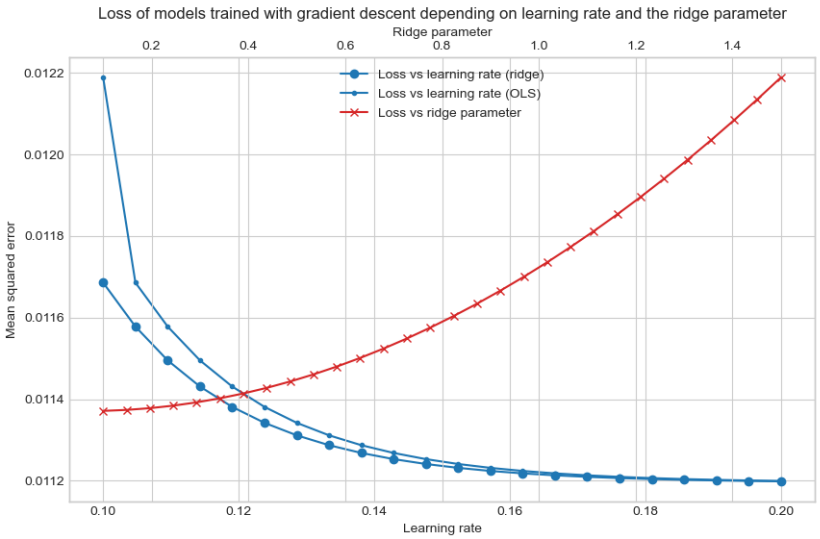

FIG. 2: Loss of OLS and Ridge models trained with gradient descent as a function of learning rate and the ridge parameter. All models were trained over 100 epochs.

We then compared the speed of convergence of OLS models using standard and momentum gradient descent training approaches, both using a learning rate of 10^{-2}. In Figure 3 we can see that momentum gradient descent, converging after around 30 epochs, converges significantly faster than standard gradient descent, which converges as it approaches 100 epochs, on our dataset. While standard gradient descent has a smooth, strictly monotonically decreasing curve, the momentum curve oscillates several times before convergence. This oscillation is likely caused by the model gaining speed as it approaches the minimum due to the momentum term, resulting in overshooting. The model then gradually corrects itself by redirecting its trajectory and letting the velocity decay. Evidence for this behavior is seen in that the steep drop in loss initially, which leads to a high velocity, is followed by a large oscillation. Successive oscillations become smaller and smaller, which is expected as each drop in loss is smaller than the previous one, leading to lower velocities and less overshooting.

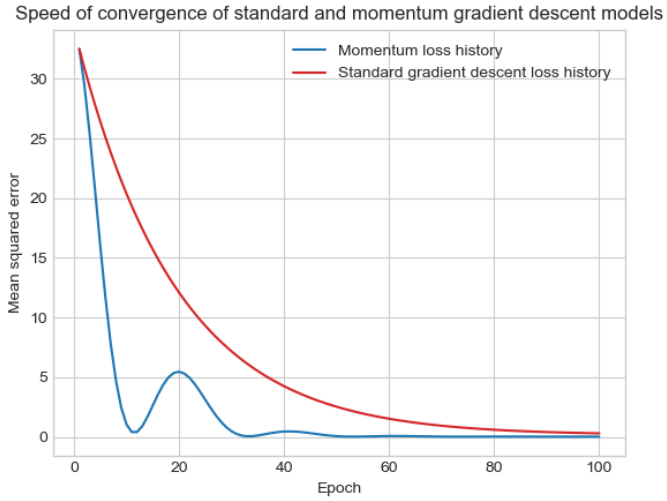

FIG. 3: Speed of convergence of standard and momentum gradient descent models. The learning rate used was 10^{-2}, and the momentum decay parameter was 0.9.

Subsequently, we tested the loss of a model using stochastic gradient descent as a function of batch size and number of epochs. In figure 11 (appendix) we can see that as batch size increases, so does the loss. As our data and optimization landscape is relatively simple and with low noise, even relatively small batches can be used to find a gradient that is highly representative of the larger dataset. As such, the higher amount of gradient updates with lower batch sizes leads to faster convergence and lower loss without being counteracted by inaccurate gradient updates. Additionally, loss smoothly and monotonically decreases as the amount of epochs increases, which is to be expected as this allows for more gradient updates.

Figure 4 shows the speed of convergence of models using standard gradient descent, along with the AdaGrad, RMSprop, and Adam optimizers. Standard gradient descent converges the fastest, RMSprop is significantly slower and Adam is slightly slower than this, while AdaGrad is much slower than the rest. Because of the simple dataset, the advantages of these alternative optimizers are largely inconsequential, leading to standard gradient descent creating the best results. AdaGrad decreases the learning rate for parameters that often have large gradients. However, in this case, that simply slows down convergence as the optimization landscape is simple enough to navigate with simpler optimizers. Unlike AdaGrad, which accumulates the square of past gradients, RMSprop and Adam implement a

moving average of the square of past gradients. This prevents the rapidly decaying learning rates observed with AdaGrad and thus leads to faster convergence.

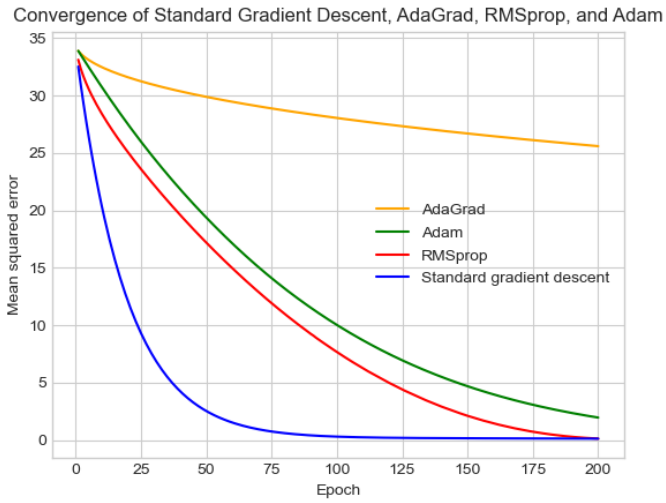

FIG. 4: Speed of convergence of model optimized with Standard gradient descent, AdaGrad, RMSprop, and Adam. A learning rate of 10^{-2} was used.

2. Feed Forward neural network: regression

The results presented in Figures 5 and 6 provide a comparison of the performance of our own FFNN and PyTorch implementation across different learning rates and epoch counts.

In both implementations, optimal performance is achieved with learning rates in the range of 10^{-2} to 10^{-1}, as indicated by a significant reduction in MSE and an increase in R^2 score across all epoch counts. However, differences emerge in the response of each model to varying learning rates. Notably, our own implementation (Figure 5) demonstrates more stability at high learning rates (around 10^0), achieving relatively lower MSE and higher R^2 scores compared to the PyTorch model (Figure 6). This suggests that our model is more resilient to large learning rates, which may be due to specific aspects of our implementation that help stabilize training in such conditions. Conversely, we only loaded 200 samples with a mean noise of 0.5, so the data may be too simple for PyTorch to demonstrate its full potential. This is supported by the marginal difference of about 1.7 in MSE value between epochs 900 and 975, respectively.

For lower learning rates (in the optimal range of 10^{-2} to 10^{-1}), the PyTorch model

generally achieves slightly lower MSE values across different epoch counts. This difference could be attributed to PyTorch's optimized functions for gradient descent, which might result in more precise gradient updates. While our own implementation achieves comparable R^2 scores, the PyTorch model's lower MSE in the optimal range suggests a slight edge in convergence efficiency under more standard learning rate settings.

Both models benefit from increasing the number of epochs, with improvements in MSE and R^2 scores observed up to around 1000 epochs. However, diminishing returns become apparent beyond 950 epochs, with performance gains becoming marginal, indicating that both models reach a saturation point in learning capacity around this range. This observation supports our choice to limit the epoch range to 900–1000 based on preliminary experiments, achieving an optimal balance between convergence and computational efficiency.

We observe similar results when using ReLU and LeakyReLU as activation functions in the two hidden layers shown in figure 12, 13, 14 and 15 in the appendix. The only difference is that they converge faster in all instances with optimal learning rates 10^{-3} to 10^{-2}.

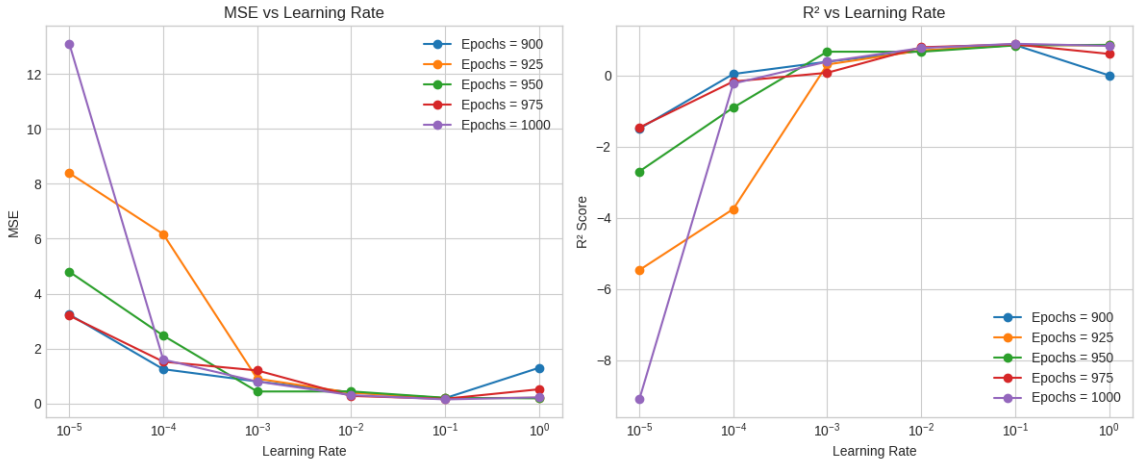

FIG. 5: Own Neural Network Regression with Sigmoid as activation function in the hidden layers.

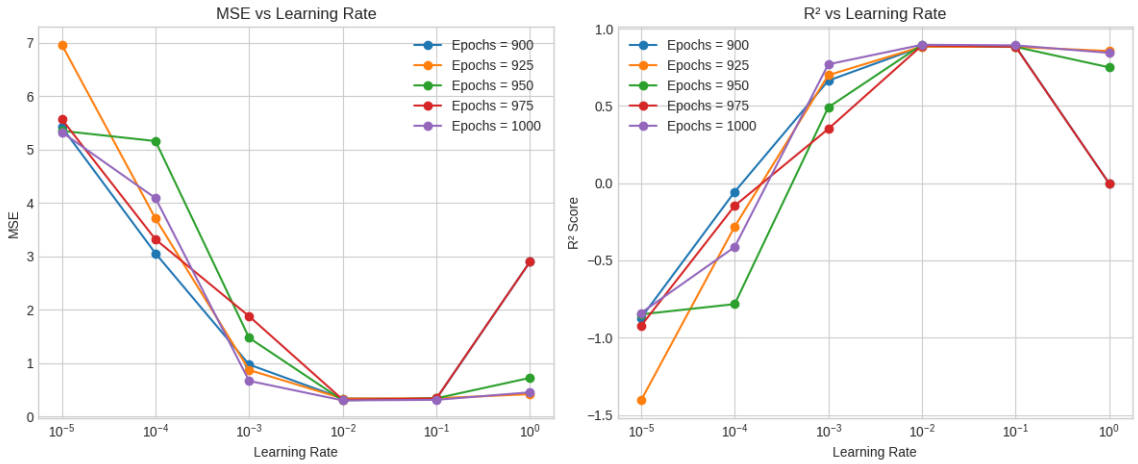

FIG. 6: Pytorch Neural Network Regression with Sigmoid as activation function in the hidden layers.

3. Feed forward neural network: classification

The results depicted in Figures 7 and 8 illustrate the accuracy of a FFNN classification model across various learning rates and epoch counts, utilizing both our own implementation (Figure 7) and the PyTorch framework (Figure 8).

In both implementations, the optimal performance is achieved with a learning rate between 10^{-3} and 10^{-1}, where the accuracy consistently reaches around 90 % or higher across the different epoch counts. This indicates that both models converge effectively within this learning rate range, suggesting it is ideal for achieving high classification accuracy with stability. At extremely low learning rates (10^{-5}), both models exhibit suboptimal performance, as seen by significantly lower accuracy, reflecting insufficient gradient updates that impede effective learning. On the other end of the spectrum, very high learning rates (10^{0}) cause a drastic drop in accuracy for the PyTorch model, indicating instability in training likely due to oscillations or divergence in the optimization process.

The impact of epochs on performance is generally positive across both models. Increasing epochs improves classification accuracy, particularly as learning rates approach the optimal range. Both models achieve near-maximum accuracy at 1000 epochs, indicating sufficient training time for convergence. However, beyond 950 epochs, the accuracy improvements are marginal, suggesting that the model's learning has saturated, making further increases

in epochs computationally inefficient. This observation supports our choice to limit the epoch range to between 900 and 1000, ensuring a balance between optimal performance and computational efficiency.

In terms of overfitting, neither model exhibits significant signs of it within the selected range of learning rates and epochs. The accuracy remains stable across increased epochs and optimal learning rates, suggesting robust generalization in both implementations.

We observe the same when using Softmax as the activation function in the last layer as shown in figure 16 and 17 in the appendix.

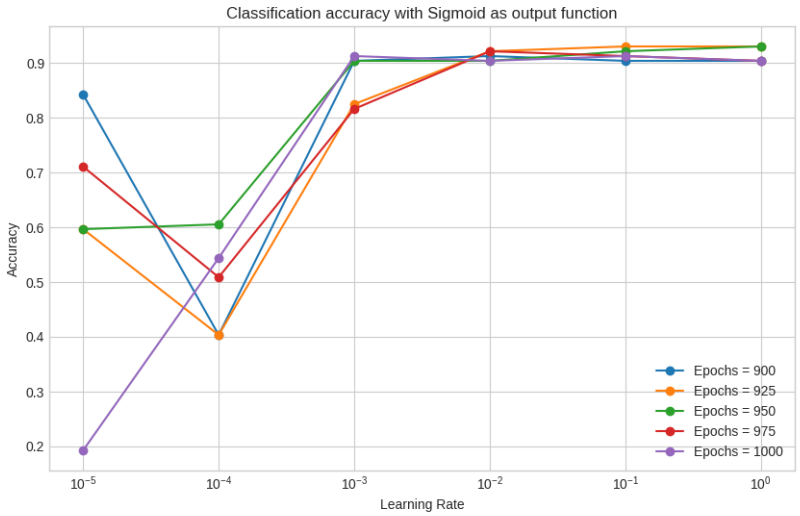

FIG. 7: Own Neural Network Classification with Sigmoid as activation function in the last layer.

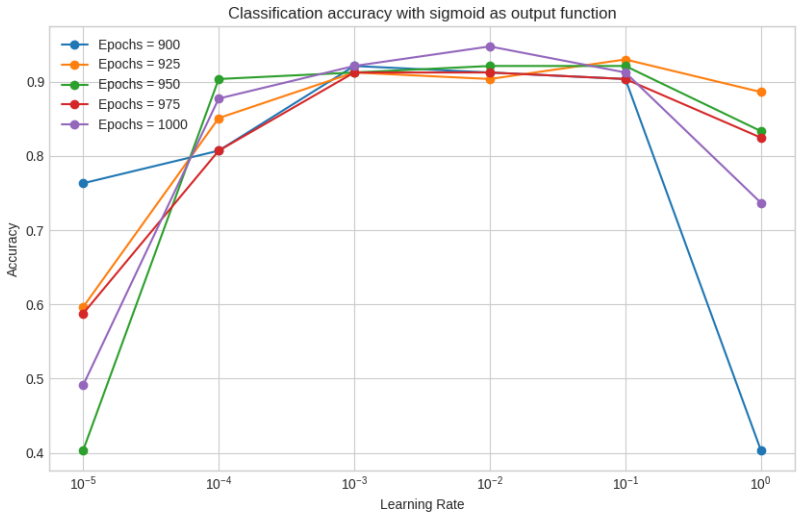

FIG. 8: Pytorch Neural Network Classification with Sigmoid as activation function in the last layer.

4. Logistic Regression: own code vs Sklearn

The results presented in Figures 9 and 10 display the classification accuracy of a logistic regression model across various learning rates and epoch counts, using both our own neural network implementation (Figure 9) and the scikit-learn logistic regression implementation (Figure 10). Both implementations incorporate a regularization parameter ($\lambda = 0.001$), which helps prevent overfitting by penalizing large model weights and encouraging simpler, more generalizable models. This specific value was chosen based on preliminary tests which showed the best results.

Optimal performance is observed at learning rates around 10^{-1} for our own model and between 10^{-3} to 10^{-2} for the scikit-learn model. In this range, classification accuracy consistently reaches approximately 90% or higher across different epoch counts, indicating effective convergence without training instability. At lower learning rates (10^{-5}), both models exhibit lower accuracy, but still high a little under 88%. Conversely, higher learning rates, particularly at 10^{0} result in a marked drop in accuracy especially for the scikit-learn model. Our own model actually has the highest accuracy value for epoch 950 at 10^{0}.

The influence of epoch count on accuracy is generally difficult to evaluate, as there is no clear trend. However, we can see that more epochs are not always better. Epochs 950 and

975 seem to be the best for both models.

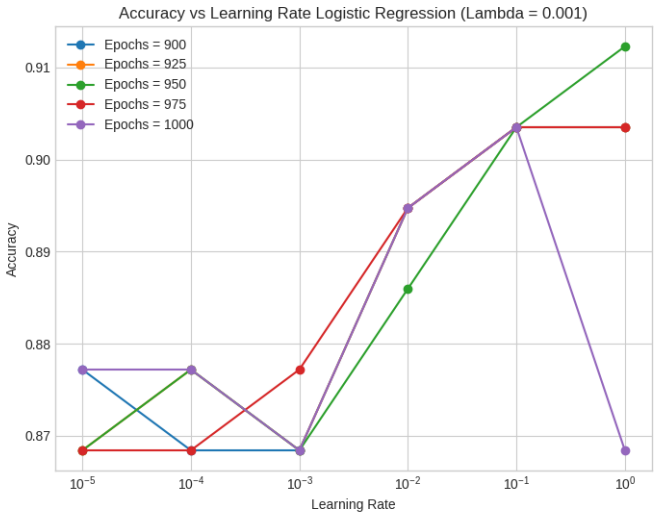

FIG. 9: Own Neural Network Logistic Regression

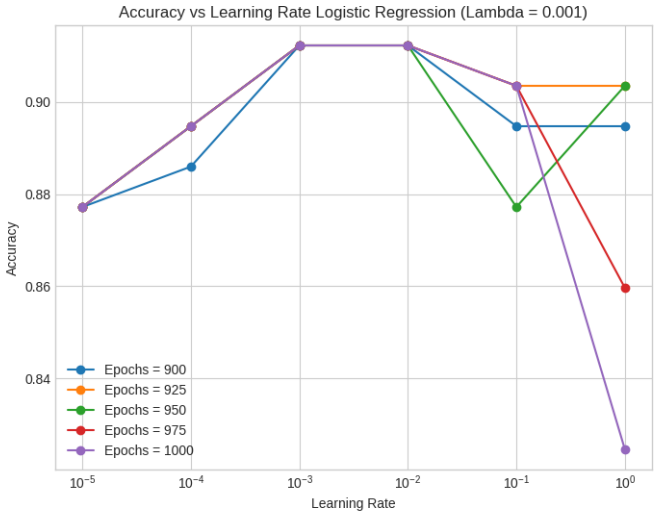

FIG. 10: Logistic regression using Sklearn

5. Summary of results

In our analysis of regression models, we analyzed the performance of Ordinary Least Squares (OLS) and Ridge regression models trained with gradient descent, focusing on the

effects of learning rate and the Ridge hyperparameter λ. We observed that both models handled higher learning rates well, likely due to the simplicity of the dataset and the convex optimization landscape created by our design matrix matching the polynomial used to generate the data. Notably, the OLS model consistently outperformed the Ridge model, as indicated by lower loss values across varying learning rates and Ridge parameters. This suggests that the regularization introduced by the Ridge parameter was unnecessary for this dataset, possibly because overfitting was not a significant concern. Additionally, momentum gradient descent significantly accelerated convergence compared to standard gradient descent, despite initial oscillations in loss values. When employing stochastic gradient descent, we found that smaller batch sizes led to lower loss, as the increased number of gradient updates outweighed any inaccuracies from less representative gradients. Finally, among various optimizers tested—including AdaGrad, RMSprop, and Adam—standard gradient descent converged the fastest, highlighting that alternative optimizers offered no significant advantage for this particular dataset, likely because of its simplicity.

Additionally, we explored various algorithms for regression and classification tasks, utilizing both custom code and implementations from libraries like PyTorch and scikit-learn. For regression problems, we employed feed-forward neural networks (FFNNs) and tested their performance using different activation functions in the hidden layers, specifically Sigmoid, ReLU, and Leaky ReLU.

In classification tasks with FFNNs, we experimented with different activation functions for the output layer, namely Sigmoid and Softmax, and compared our results with those obtained using PyTorch. Additionally, we implemented logistic regression to address a classification problem involving the Wisconsin Cancer dataset. To prevent overfitting, we introduced a regularization parameter $\lambda = 0.001$ and experimented with various learning rates and epochs to determine the optimal parameters. Our logistic regression implementation was also benchmarked against scikit-learn's version.

Regarding the advantages and disadvantages, FFNNs are beneficial due to their ability to model complex and nonlinear relationships through multiple layers and nonlinear activation functions, making them suitable for both regression and classification tasks. With the right parameters, they offer high accuracy in classification and effective convergence within optimal learning rate ranges, demonstrating robustness across different implementations. However, a notable downside is the computational cost of training these networks,

especially with larger datasets.

In contrast, logistic regression is more computationally efficient than FFNNs, making it more suitable for large datasets. It also benefits from the inclusion of a regularization parameter, which helps prevent overfitting. Like FFNNs, logistic regression is sensitive to the choice of hyperparameters such as the learning rate and regularization parameter.

For the regression case, we found that the code that we implemented for the feed-forward neural network was the optimal method. We also saw that the activation in the hidden layers did not greatly affect the performance, and we achieved low MSE results for all our implementations.

For the classification case, the feed-forward neural network and logistic regression both yielded good results, with optimal accuracy scores around 90%. However, while logistic regression is more computationally efficient, the neural network achieved slightly higher accuracy, making it the preferred choice when maximizing accuracy is the priority.

In summary, although logistic regression is more efficient, the neural network's improved accuracy makes it the better option when accuracy outweighs computation speed.

V. CONCLUSION

In conclusion, this study found that standard gradient descent converged faster than the alternative optimizers. The OLS regression model consistently achieved lower loss values compared to the Ridge regression, indicating that regularization was unnecessary due to the dataset's simplicity and minimal risk of overfitting. Additionally, momentum gradient descent accelerated convergence compared to standard gradient descent, despite initial oscillations in loss values, highlighting the benefits of incorporating momentum in optimization algorithms. Additionally, this study provides a comparison of Feedforward Neural Networks and Logistic Regression, applied to both synthetic regression tasks and real-world classification using the Wisconsin breast cancer dataset. The findings emphasize the importance of model choice, activation functions, and optimization techniques, with optimal configurations yielding significant improvements in performance. For both tasks, a learning rate range of 10^{-3} to 10^{-2} consistently provided the best results, balancing convergence and efficiency. Our own neural network showed enhanced stability at higher learning rates compared to PyTorch, while scikit-learn's logistic regression demonstrated reliable performance within optimal learning rates. The study highlights the necessity of careful hyperparameter tuning, model selection, and regularization for achieving reliable performance across different data types and complexity levels. Future work should aim to refine these models further by leveraging greater computational resources, enabling the exploration of higher model complexities and more sophisticated architectures. Additionally, applying these optimized models to larger and more diverse datasets will enhance our understanding of their adaptability to real-world data applications.

APPENDIX A

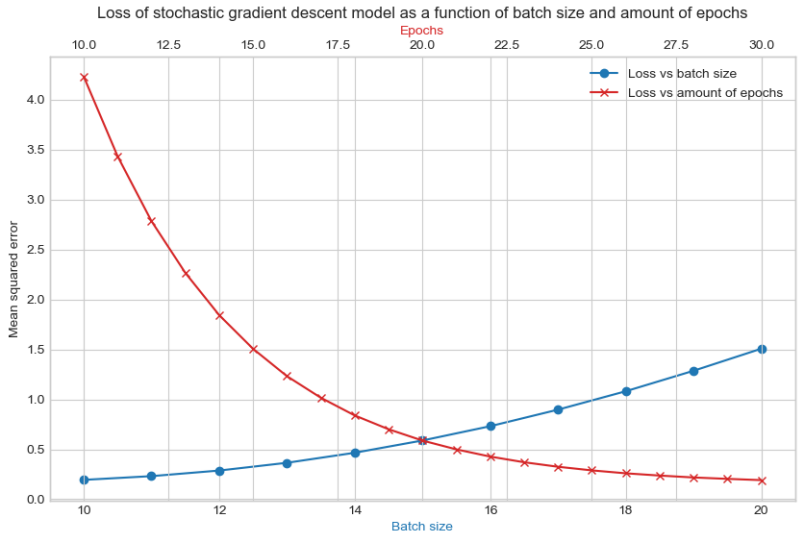

FIG. 11: Performance of stochastic gradient descent model as a function of batch size and epochs. Batch size = 20 used for the Loss vs amount of epochs curve, epochs = 15 used for the Loss vs batch size curve, and learning rate = 10^{-2} used for both.

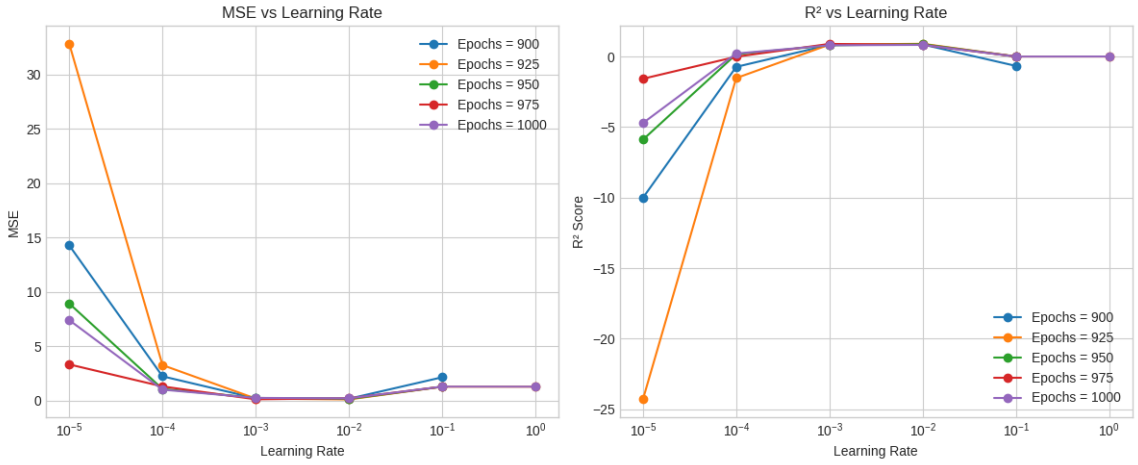

FIG. 12: Own Neural Network Regression with ReLU as activation function in the hidden layers.

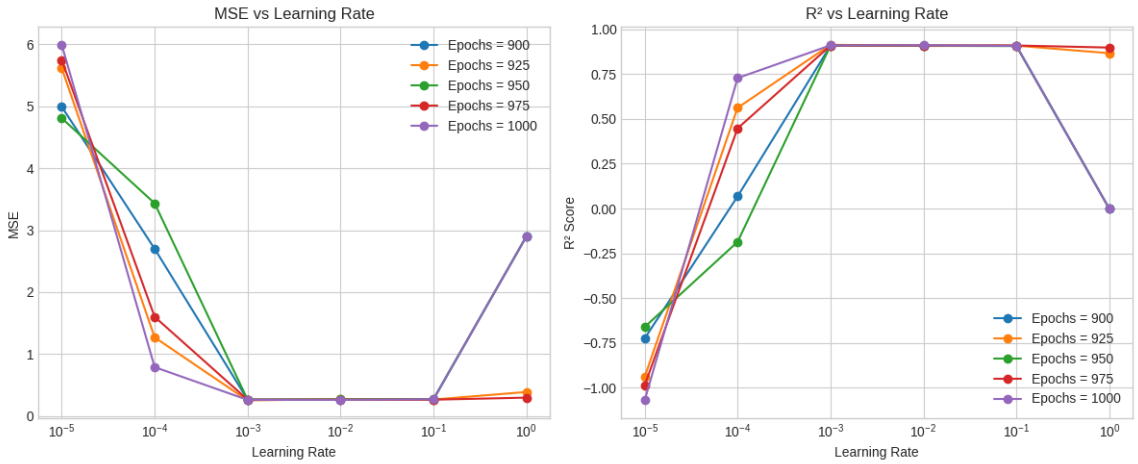

FIG. 13: Pytorch Neural Network Regression with ReLU as activation function in the hidden layers.

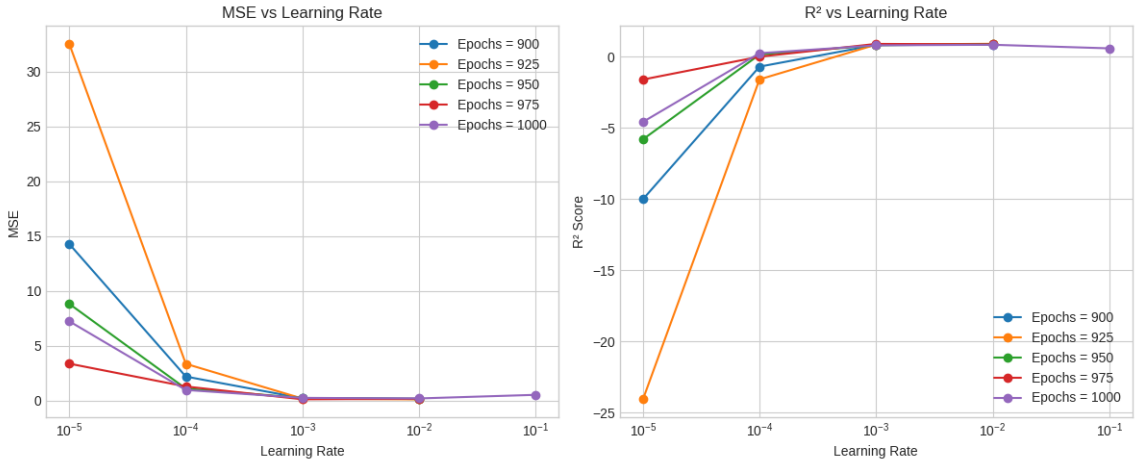

FIG. 14: Own Neural Network Regression with LeakyReLU as activation function in the hidden layers.

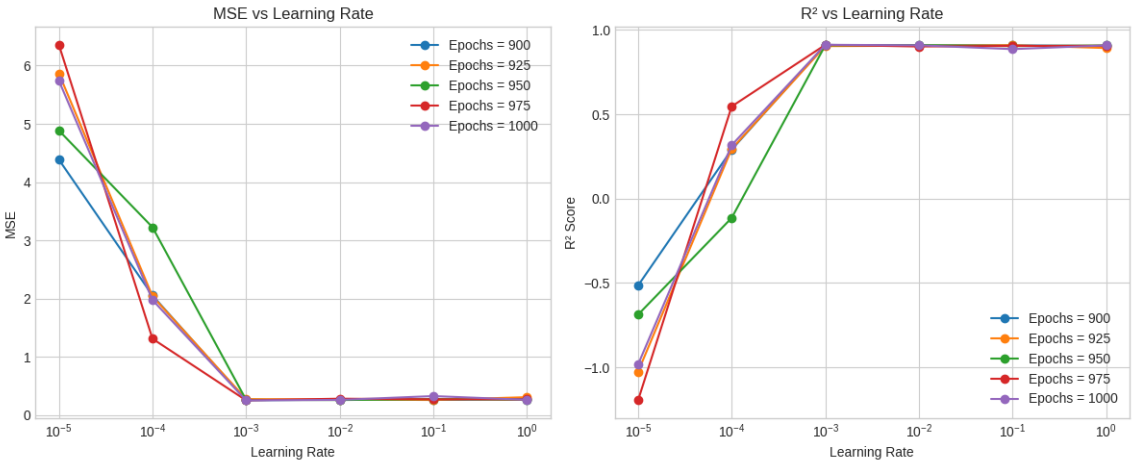

FIG. 15: Pytorch Neural Network Regression with LeakyReLU as activation function in the hidden layers.

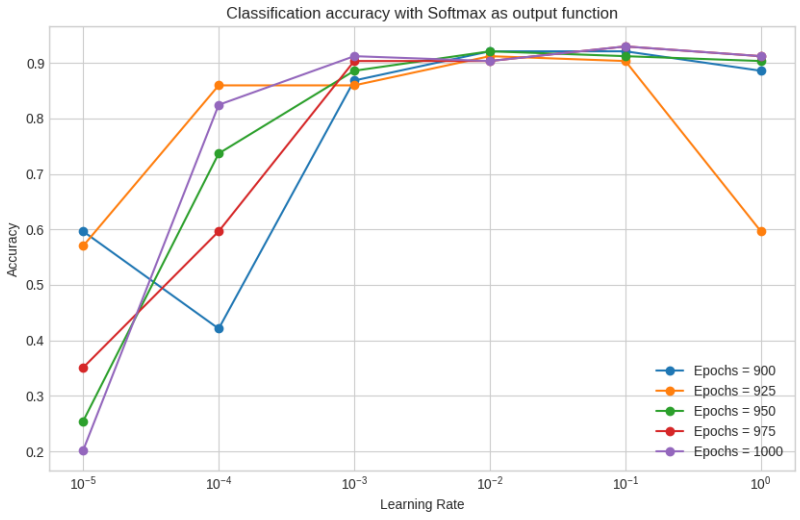

FIG. 16: Own Neural Network Classification with Softmax as ativation function in the last layer.

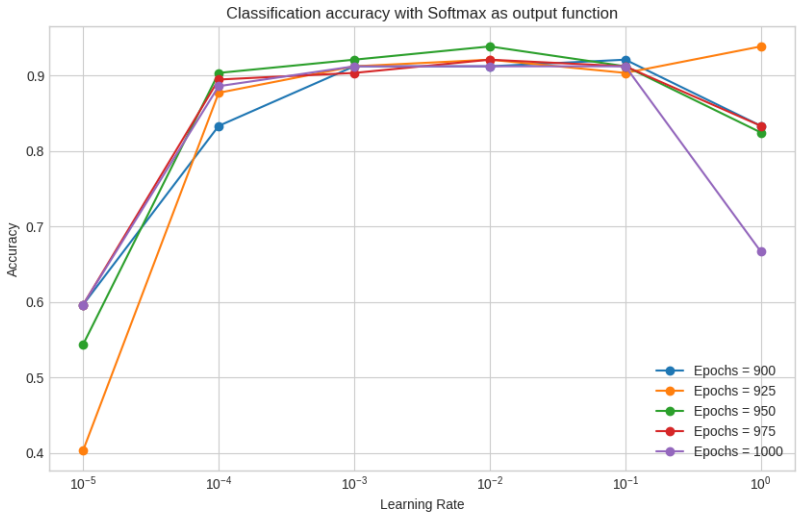

FIG. 17: Pytorch Neural Network Classification with Softmax as activation function in the last layer.

REFERENCES

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[2] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[4] William H Wolberg and Olvi L Mangasarian. Multisurface method of pattern separation for medical diagnosis applied to breast cytology. *Proceedings of the National Academy of Sciences*, 87(23):9193–9196, 1990.

[5] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2014.

[6] Kaggle and UCI Machine Learning Repository. Breast cancer wisconsin (diagnostic) data set, 2023. Accessed: 2024-10-31.

[7] Iris Bore. FYS-STK4155 Project 1. https://github.com/irisbore/FYS-STK4155_project1, 2023.

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

[9] Budescu DV. A note on polynomial regression. *Multivariate Behav Res*, 1980.

[10] Morten Hjorth-Jensen. Machine learning lecture notes. https://github.com/CompPhysics/MachineLearning/tree/master/doc/pub, 2023. GitHub repository.

[11] Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Autograd: Reverse-mode differentiation of native python. In *ICML Workshop on Automatic Machine Learning (AutoML)*, 2015.

[12] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: Composable transformations of Python+NumPy programs. http://github.com/google/jax, 2018.

[13] Michael Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/, 2019.

[14] Sasirekha Cota. Deep learning basics: Part 7 - feed forward neural networks (ffnn), 2023. Accessed: 2024-10-31.

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pages 8024–8035, 2019.

[16] OpenAI. Chatgpt: Openai language model. https://openai.com/chatgpt, 2023. Accessed: 2024-09-22, 27-09-22.

[17] GitHub. GitHub Copilot. https://github.com/features/copilot, 2023. Accessed: October 2, 2024.

[18] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[19] John D Hunter. MatPlotLib: a 2D Graphics environment. *Computing in Science and Engineering*, 9(3):90–95, 1 2007.