Classification and Regression, from linear and logistic regression to neural networks

Sheuly Debnath

November 4, 2024

Abstract

In this project, we explored how well a Feed-Forward Neural Network (FFNN) handles classification and regression problems. We compared it to linear and logistic regression using two datasets: the Franke function and the Wisconsin Breast Cancer dataset. Our main aim was to see which method was more efficient in terms of computation time and accuracy.Our results show that the neural networks code is better suited to the classification problem, achieving a higher accuracy than our logistic regression code and scikit learn's logistic regression function. Meanwhile, for the regression problem the logistic regression code performed slightly better. Various activation functions point to- wards the Sigmoid function producing the best results in this case, but the logistic regression code still performed better.

1 Introduction

Artificial neural networks are computational systems that can learn to perform tasks by considering examples, generally without being programmed with any task-specific rules. It is supposed to mimic a biological system, wherein neurons interact by sending signals in the form of mathematical functions between layers. All layers can contain an arbitrary number of neurons, and a weight variable represents each connection. It is also inspired by the brain, many different tasks such as image recognition, speech recognition, and natural language processing [8], whether it be continuous (regression) or discrete (classification).

In this project, our main aim is to implement a Feed-Forward Neural Network (FFNN) with a flexible number of hidden layers and neurons(nodes) and different activation functions. We then compare FFNN results with other methods for regression and classification. For the continuous case, we will fit a second order polynomial using both a neural network and classic linear regression. For classification, we will look at the Wisconsin Breast Cancer data set[11], applying both the neural network and logistic regression.

In order to test the optimization methods discussed in this project, we test the models on two datasets. For the regression model we use the Franke function[2], while for the classification model we use the Wisconsin Breast Cancer Data [11]. For both datasets we split the data into a train and test set, where 80% of the data will be used to train the set.

2 Theory

2.1 Regression models

2.1.1 Linear regression

Linear regression is a statistical method used to model the relationship between a dependent variable (target) and one or more independent variables (features) by fitting a linear equation to observed data. The goal is to find the line (or hyperplane in higher dimensions) that best represents this relationship by minimizing the error, typically using the Ordinary Least Squares (OLS) method. The following equation is used in the calculation of the linear regression

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

where the matrix $\mathbf{X}$ is the design matrix and $\boldsymbol{\beta}$ are the unknown parameters we want to determine. The model is fitted by finding the values of $\boldsymbol{\beta}$ which minimize the cost function $C(\mathbf{X}, \boldsymbol{\beta})$ where the cost function is a function which allows us to judge how well the model $\boldsymbol{\beta}$ fits the matrix $\mathbf{X}$. The minimum is usually found using numerical methods, as analytical methods are generally not possible.

A common linear regression model is the OLS, where we assume a cost function[4]

$$\mathcal{C}_{\text{OLS}}(\boldsymbol{\beta}) = \frac{1}{n}\left\{(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})\right\}, \tag{1}$$

which, when minimized, yields the OLS expression for the optimal parameter $\hat{\boldsymbol{\beta}}$. Another common model is Ridge regression, where we include a regularization parameter λ, and for which the cost function becomes

$$C(\mathbf{X}, \boldsymbol{\beta})_{\text{Ridge}} = \left\{(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta})\right\} + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta}. \tag{2}$$

For the linear regression analysis our main interest was around leading the coefficients of a functional fit in order to be able to predict the response of a continuous variable on some unseen data. Linear regression resulted in analytical expressions for standard OLS or Ridge regression for several quantities, ranging from the variance and thereby the confidence intervals of the parameters β to the MSE [4]. By inverting the product of the design matrices we could fit our data.

2.1.2 Logistic regression

Logistic regression is a model used for predicting binary outcomes, such as 1 or 0, to classify whether an input falls into a specific category. It does this by calculating the probability that the input belongs to one category, then assigning it either a 1 or 0 based on this probability. This can be achieved by using the sigmoid function. This function transforms the model's output into a probability, making it easy to classify results as either true or false.

The probability that a data point x_i belongs to a category $y_i = \{0, 1\}$ is given by the logistic function, also known as the Sigmoid function,

$$p(t) = \frac{1}{1 + \exp -t} = \frac{\exp t}{1 + \exp t}, \tag{3}$$

which is meant to represent the likelihood of a given event [10]. Assuming that we have two categories with $y_i \in \{0, 1\}$ and that we only have two parameters β in the fit of the Sigmoid function, we define the probabilities

$$p(y_i = 1|x_i, \boldsymbol{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)} \tag{4}$$

$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta}), \tag{5}$$

where x is an input set and $\boldsymbol{\beta}$ are the weights we wish to extract from data, in this case β_0 and β_1 which are the coefficients we use to estimate the data [10].

Our aim is now to maximize the probability of seeing the observed data. Using the Maximum Likelihood Estimation (MLE), we define the total likelihood for all possible outcomes from a dataset $\mathcal{D} = \{(y_i, x_i)\}$ with the binary labels $y_i \in \{0, 1\}$:

$$P(\mathcal{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} [p(y_i = 1|x_i, \boldsymbol{\beta})]^{y_i} [1 - p(y_i = 1|x_i, \boldsymbol{\beta})]^{1-y_i}, \tag{6}$$

which then is an approximation of the likelihood in terms of the individual probabilities of a specific outcome y_i [10].

We minimize the cross entropy cost function with respect to the two parameters β_0 and β_1, keeping in mind that this is a convex function of the weights $\boldsymbol{\beta}$, thereby making any local minimizer a global minimizer. By defining a vector $\boldsymbol{y}$ with n elements y_i, an $n \times p$ matrix $\mathbf{X}$ which contains the x_i values and a vector $\boldsymbol{p}$ of fitted probabilities $p(y_i|x_i, \boldsymbol{\beta})$, we find that the first derivative of the cost function becomes

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{p}). \tag{7}$$

By defining a diagonal matrix $\mathbf{W}$ with elements $p(y_i|x_i, \boldsymbol{\beta})(1 - p(y_i|x_i, \boldsymbol{\beta}))$, we obtain an expression for the second derivative

$$\frac{\partial^2 \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}^T} = \boldsymbol{X}^T \boldsymbol{W} \boldsymbol{X}. \tag{8}$$

When performing the linear regression analysis we solved for the best value for $\boldsymbol{\beta}$ by taking the inverse. However, this is not always possible, and in such cases we can apply a method which takes advantage of numerical optimization, called gradient descent.

2.2 Gradient descent

Gradient descent is an optimization technique used to find the minimum of a function, which, in machine learning, often translates to finding the best parameters for a model. In gradient descent, we compute the cost function and its gradient for all data points we have. By reducing this cost, we improve the model's accuracy. Limitation of gradient descent [5]:

Gradient descent (GD) finds local minima of our function and it is sensitive to initial conditions and to choices of learning rates. Gradients are computationally expensive to

calculate for large datasets. For the regression problems that were solved in project 1, the solutions were determined analytically by performing matrix inversions to minimize the chosen cost function.

For regression problems with higher numbers of features (columns in our design matrix), these matrix inversions become computationally expensive. For this reason, we have to instead use a method for approximating the minimum of a cost function. The family of gradient descent methods is A set of popular choice methods for performing this task.

The idea of these methods is to determine the minimum of a function iteratively by using the value of a function $f(\boldsymbol{x})$ in a given point $\boldsymbol{x} = (x_1, \ldots, x_n)$ decreases fastest in the direction of the negative gradient of the function. This can be formalized by the iteration process

$$\boldsymbol{x}_{(k+1)} = \boldsymbol{x}_{(k)} - \eta_{(k)} \nabla f\left(\boldsymbol{x}_{(k)}\right), \tag{9}$$

where $\eta_{(k)}$ is the step length, often called the learning rate, used to update the position in each iteration. In the next section, we will present different models for the time (iteration number) dependence of the learning rate.

In our regression models, the goal is to find the regression parameters $\boldsymbol{\beta}$ that minimize the cost function. By using mean squared error as cost function, the cost function reads

$$C(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{\beta}) = \frac{1}{n}||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}||_2^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \boldsymbol{X}_{i,*}\boldsymbol{\beta})^2, \tag{10}$$

where $\boldsymbol{X}$ is the design matrix generated from the input data, $\boldsymbol{y}$ is the correct output and n is the sample size. The gradient of the cost function with respect to the regression parameters is

$$\nabla_{\boldsymbol{\beta}} C(\boldsymbol{\beta}) = -\frac{2}{n}\boldsymbol{X}^T\left(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\right). \tag{11}$$

By replacing $\boldsymbol{x}$ and $f(\boldsymbol{x})$ with $\boldsymbol{\beta}$ and $C(\boldsymbol{\beta})$, respectively, in Equation 9, we have an iterative method for approximating the regression parameters that minimize of the cost function. However, the method might converge to a local minimum, and not the global minimum of the cost function.

2.2.1 Steepest gradient descent

The basic idea of gradient descent is that a function $F(\mathbf{x})$, $\mathbf{x} \equiv (x_1, \cdots, x_n)$, decreases fastest if one goes from $\mathbf{x}$ in the direction of the negative gradient $-\nabla F(\mathbf{x})$.

It can be shown that if

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k),$$

with $\gamma_k > 0$.

For γ_k small enough, then $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$. This means that for a sufficiently small γ_k we are always moving towards smaller function values, i.e a minimum. Ideally the sequence $\{\mathbf{x}_k\}_{k=0}$ converges to a global minimum of the function F. In general we do not know if we are in a global or local minimum. In the special case when F is a convex function, all local minima are also global minima, so in this case gradient descent can converge to the global solution.

2.2.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an optimization algorithm that attempts to reduce the minimization process's computational cost by dividing the data set into minibatches of a specified number of data samples. For this method, the update of $\boldsymbol{\beta}$ in each iteration is only determined by the gradients of the data samples in a randomly drawn minibatch, decreasing the computational cost associated with calculating gradients significantly. The stochasticity of this method may also make the process less likely to become stuck at a local minimum. This makes it faster and more efficient for large datasets. It requires less memory compared to computing gradients for the entire dataset. Hopefully avoid Local Minima. [5]

The underlying idea of SGD comes from the observation that the cost function, which we want to minimize, can almost always be written as a sum over n data points $\{\mathbf{x}_i\}_{i=1}^n$,

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

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

Stochasticity/randomness is introduced by only taking the gradient on a subset of the data called minibatches. If there are n data points and the size of each minibatch is M, there will be n/M minibatches. We denote these minibatches by B_k where $k = 1, \cdots, n/M$.

Approximate the gradient by replacing the sum over all data points with a sum over the data points in one the minibatches picked at random in each gradient descent step

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

Thus a gradient descent step [5] now looks like

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta)$$

where k is picked at random with equal probability from $[1, n/M]$. An iteration over the number of minibathces (n/M) is commonly referred to as an epoch. Thus it is typical to choose a number of epochs and for each epoch iterate over the number of minibatches.

The stochastic gradient descent (SGD) is almost always used with a *momentum* or inertia term that serves as a memory of the direction we are moving in parameter space. This is typically implemented as follows

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t,$$

where we have introduced a momentum parameter γ, with $0 \leq \gamma \leq 1$, and for brevity we dropped the explicit notation to indicate the gradient is to be taken over a different mini-batch at each step. We call this algorithm gradient descent with momentum (GDM). An equivalent way of writing the updates is

$$\Delta\boldsymbol{\theta}_{t+1} = \gamma\Delta\boldsymbol{\theta}_t - \ \eta_t \nabla_\theta E(\boldsymbol{\theta}_t),$$

where we have defined $\Delta\boldsymbol{\theta}_t = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$.

2.2.3 Batches and mini-batches

In gradient descent we compute the cost function and its gradient for all data points we have.

In large-scale applications such as the [ILSVRC challenge] [1] , the training data can have on order of millions of examples. Hence, it seems wasteful to compute the full cost function over the entire training set in order to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over batches of the training data.

2.2.4 When do we stop?

To decide when to stop searching for the minimum in gradient descent, one approach is to check the gradient's size after a set number of epochs. If it's very small, we might be near a minimum. However, this could be a local or global minimum. To be more certain, we can also keep track of the cost function's value at these points. If a later check shows a lower cost, we can update our parameters to the ones that gave the lowest value so far[5].

2.2.5 Learning Rate Tuning Methods

In the section on gradient descent we described a simple procedure of setting the learning rate to a constant value. However, there exist many methods to iteratvely tune the learning rate. These methods can lead to better convergence to the minima.

This time decay method for the learning rate is often chosen as

$$\eta_{(t)} = \frac{t_0}{t + t_1}, \tag{12}$$

where t is the current epoch, and t0 and t1 are parameters to set the initial learning rate. For example if $t0$ is 5 and $t1$ 50, then the initial learning rate would be 5/(50+0) = 0.1. As time progresses and the epoch t becomes a larger number, the learning rate will decrease. This is seen to be advantageous since as you approach the minima, you want to take shorter steps so as to not skip over it.

Several more sophisticated tuning methods exist. One of these methods is *AdaGrad*, which means Adaptive Gradient Algorithm[5] . This algorithm works by scaling the learning rate of the different model parameters by the root mean squared of all the previous gradient values [3]

The update scheme for AdaGrad is given as

$$\boldsymbol{\theta}_{(k+1)} = \boldsymbol{\theta}_{(k)} - \eta \frac{\boldsymbol{g}_{(k)}}{\sqrt{(\boldsymbol{s}_{(k)})} + \delta}, \tag{13}$$

The next learning rate tuning method is *RMSprop*, which means Root Mean Squared Propagation. In RMS prop, in addition to keeping a running average of the first moment of the gradient, we also keep track of the second moment denoted by $\mathbf{s}_t = E[\mathbf{g}_t^2]$. The update rule for RMS prop is given by

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}) \tag{14}$$

$$\mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}},$$

In [ADAM](https://arxiv.org/abs/1412.6980) we keep a running average of both the first and second moment of the gradient and use this information to adaptively change the learning rate for different parameters. The method is efficient when working with large problems involving lots data and/or parameters. ADAM performs an additional bias correction to account for the fact that we are estimating the first two moments of the gradient using a running average (denoted by the hats in the update rule below). The update rule for ADAM is given by (where multiplication and division are once again understood to be element-wise operations below)

$$\Delta\theta_{t+1} = -\eta_t \frac{\boldsymbol{m}_t}{\sqrt{\sigma_t^2 + m_t^2} + \epsilon}.$$

Some practical tips was suggested: Randomize the data when making mini-batches.Monitor the out-of-sample performance. Adaptive optimization methods don't always have good generalization.Geron's text, see chapter 11.

2.3 Neural network

A neural network consists of a series of hidden layers, in addition to the input and output layers. Each layer l has a set of parameters $\boldsymbol{\Theta}^{(l)} = (\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)})$ which are related to the parameters in other layers through a series of affine transformations, for a standard NN these are matrix-matrix and matrix-vector multiplications. For all layers we will simply use a collective variable $\boldsymbol{\Theta}$.

It consist of two basic steps:

1. a feed forward stage which takes a given input and produces a final output which is compared with the target values through our cost/loss function.

2. a back-propagation state where the unknown parameters $\boldsymbol{\Theta}$ are updated through the optimization of the their gradients. The expressions for the gradients are obtained via the chain rule, starting from the derivative of the cost/function.

These two steps make up one iteration. This iterative process is continued till we reach an eventual stopping criterion.

The cost function is a function of the unknown parameters $\boldsymbol{\Theta}$ where the latter is a container for all possible parameters needed to define a neural network. If we are dealing with a regression task a typical cost/loss function is the mean squared error

$$C(\boldsymbol{\Theta}) = \frac{1}{n}\left\{(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})\right\}.$$

2.3.1 Forward propagation

Feed Forward Neural Networks (FFNNs) are the simplest type of artificial neural networks. They are made up of multiple nodes (also called neurons or perceptrons) in multiple layers, forming a network. Each node can be thought of as a decision-maker that takes in multiple inputs and produces a single output. The way that a node makes a decision is by applying weights and biases to the input values, and determining if the resulting output meets some threshold criterion (Nielsen, 2015). Weights are values that scale the inputs according to their relative importance, and biases are static shifts that are applied that affect how easy it is for the node to activate

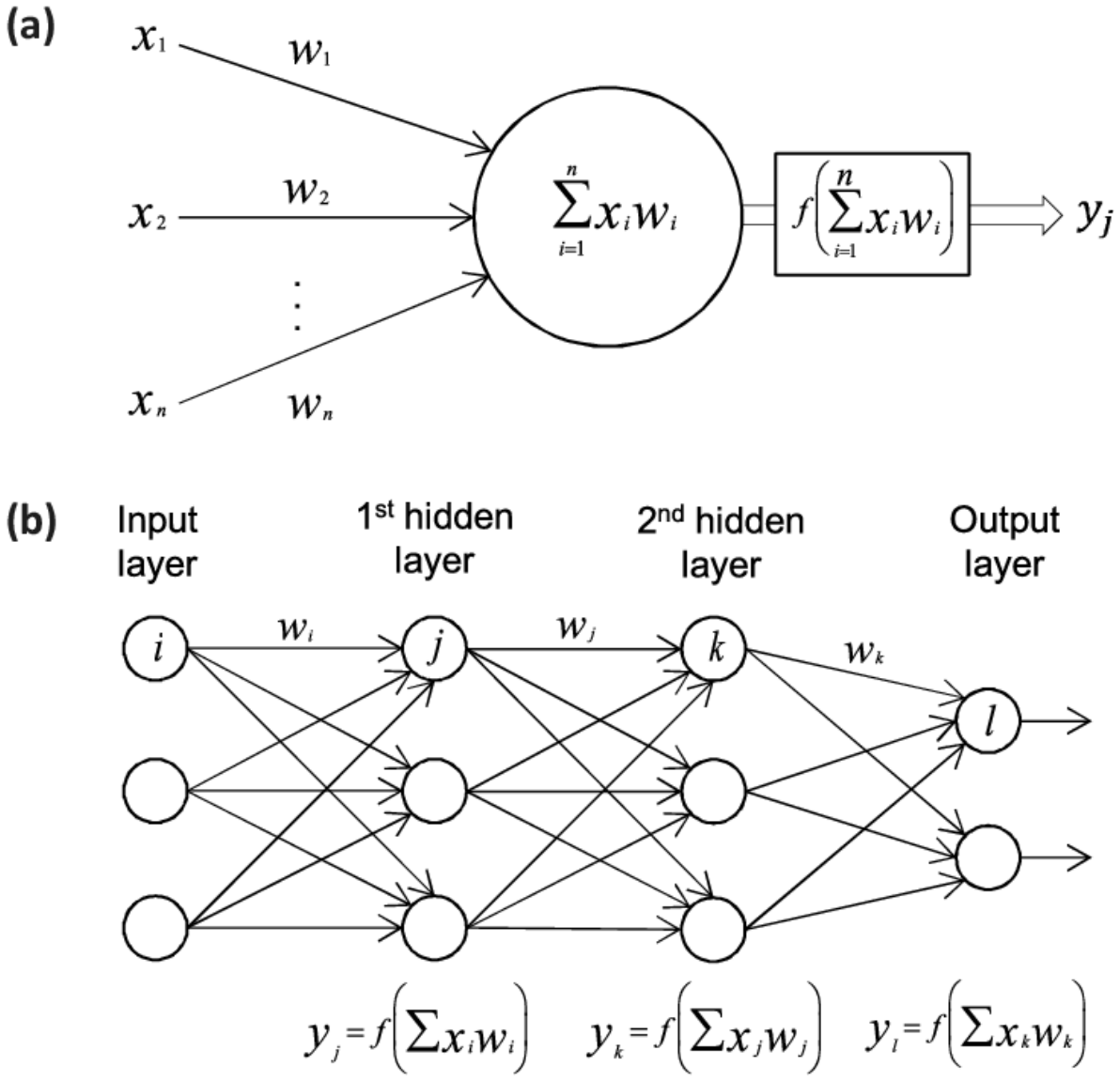

Figure 1: Illustration of a single perceptron model and a multi-perceptron model.[6]

The value for a single node k in layer i is obtained from the expression

$$a_k^{(i)} = f\left(z_k^{(i)}\right) = f\left(b_k^{(i)} + \sum_{k=1}^{N_i} a_j^{(i-1)} w_{j,k}^{(i)}\right). \tag{15}$$

Here, $w_{j,k}^{(i)}$ is the weight associated with the connection between the activation output of node j in layer $i-1$ and node k in layer i, $b_j^{(i)}$ is the weight of the bias connected to node k in layer i. The function f is known as an activation function, and is inspired from biological neural network, where a neuron only fires if the input charge exceeds an activation potential. Similarly, the activation function models the activation of the node, but in this case the output is not constrained to be either 0 or 1, and can take a continuous range of values.

2.3.2 Activation functions

The reason for applying activation functions to the output of nodes in a neural network is to introduce non-linearity in the model, making the network able to determine complex relations in the data [9]. A property that characterizes a neural network, other than its connectivity, is the choice of activation function(s).

The activation functions [7] that will be applied in this project are linear activation, sigmoid activation, rectified linear unit (ReLU) activation and leaky ReLU activation . Linear activation is the simplest activation function, which outputs the same value as the input. Similar to the linear activation function, the ReLU activation function outputs the input value for positive input values, but returns the value 0 for negative inputs, which can be expressed as

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{else} \end{cases}. \tag{16}$$

The slight modification by given small gradients for negative inputs yields the leaky ReLU activation function as

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{else} \end{cases}, \tag{17}$$

where α is a small positive constant, for example $\alpha = 10^{-3}$. The logistic sigmoid activation function (later simply referred to as sigmoid) is defined as

$$f(x) = \frac{1}{1+e^{-x}}. \tag{18}$$

It can be observed that this function moves asymptotically towards 0 for increasing negative numbers, while moving asymptotically towards 1 for increasing positive numbers.

Which activation function should we use? In general it seems that the ELU activation function is better than the leaky ReLU function (and its variants), which is better than ReLU. ReLU performs better than tanh which in turn performs better than the logistic function.

If runtime performance is an issue, then you may opt for the leaky ReLU function over the ELU function If you don't want to tweak yet another hyperparameter, you may just use the default α of 0.01 for the leaky ReLU, and 1 for ELU. If you have spare time and computing power, you can use cross-validation or bootstrap to evaluate other activation functions.

2.3.3 Backpropagation

In order to train the network to give results close to the target values, the weights and biases need to be updated. This update process is often called backpropagation, because the error of the output values is propagated backwards through the network to update the individual weights using gradient descent methods.

The idea of this method is to update the weights using the expression

$$w_{j,k}^{(i)} = w_{j,k}^{(i)} - \eta \frac{\partial C}{\partial w_{j,k}^{(i)}}, \tag{19}$$

where C is the cost function. The learning rate η may be chosen using one of the optimizers defined above in **??**.

By using the chain rule on the derivative of the cost function, the derivative with respect to the weights between node j in layer $i-1$ and node k in layer i can be expressed as

$$\frac{\partial C}{\partial w_{j,k}^{(i)}} = \frac{\partial C}{\partial z_k^{(i)}} \frac{\partial z_k^{(i)}}{\partial w_{j,k}^{(i)}}. \tag{20}$$

From the definition of $z_k^{(i)}$ in Equation 15, we obtain that

$$\frac{\partial z_k^{(i)}}{\partial w_{j,k}^{(i)}} = a_j^{(i-1)}. \tag{21}$$

We define the remaining factor as the error $\delta_k^{(i)}$, which by using the chain rule can be rewritten as

$$\delta_k^{(i)} = \frac{\partial C}{\partial z_k^{(i)}} = \sum_m \left(\frac{\partial C}{\partial z_m^{(i+1)}} \frac{\partial z_m^{(i+1)}}{\partial a_k^{(i)}} \right) \frac{\partial a_k^{(i)}}{\partial z_k^{(i)}}. \tag{22}$$

From our definition of $\delta_k^{(i)}$, we observe that $\frac{\partial C}{\partial z_m^{(i+1)}} = \delta_m^{(i+1)}$ is the error from node m in the next layer. The two remaining factors can easily be determined from Equation 15 to be

$$\frac{\partial z_m^{(i+1)}}{\partial a_k^{(i)}} = w_{k,m}^{(i+1)} \tag{23}$$

and

$$\frac{\partial a_k^{(i)}}{\partial z_k^{(i)}} = f'(z_k^{(i)}). \tag{24}$$

The resulting derivative of the cost function with respect to the weight between node j in layer $i-1$ and node k in layer i becomes

$$\frac{\partial C}{\partial w_{j,k}^{(i)}} = \sum_m \left(\delta_m^{(i+1)} w_{k,m}^{(i+1)} \right) a_j^{(i-1)}, \tag{25}$$

which makes it possible to propagate the error backward through the network in order to update the weights using the scheme in Equation 19.

The bias is updated using a similar expression as in Equation 19, by replacing $w_{j,k}^{(i)}$ with $b_k^{(i)}$. The derivative of the cost function with respect to the bias becomes

$$\frac{\partial C}{\partial b_k^{(i)}} = \frac{\partial C}{\partial z_k^{(i)}} \frac{\partial z_k^{(i)}}{\partial b_k^{(i)}} = \frac{\partial C}{\partial z_k^{(i)}} = \delta_k^{(i)}. \tag{26}$$

For the output layer, the derivative of the cost function with respect to the output values can be found directly.

3 Method

3.1 Regression

We will start by solving a regression problem using ordinary least squares and Ridge regression. The data set used to perform the regression is defined by sampling points from the function 4 + 3*x + 2*x**2

$$f(x) = 4 + 3x + 2x^2 + \epsilon, \tag{27}$$

where ϵ is a stochastic noise term. The linear regression will be performed using plain and stochastic gradient descent, with and without momentum, on a design matrix created from the three features $[\mathbf{1}, \boldsymbol{x}, \boldsymbol{x}^2]$. This introduces two hyper parameters into the models, the learning rate η and the momentum parameter γ, that need to be tuned, as well as the regularization parameter λ in Ridge regression. The loss of the trained model will be analyzed for different values of these parameters.

In addition to the mentioned gradient descent methods, the adaptive optimization methods AdaGrad, RMSProp and ADAM will also be applied to the regression problem to compare the benefits and shortcomings of different solver methods.

Since the analytical expressions for the gradients of ordinary least squares and Ridge regression are known, we utilize these to verify that automatic differentiation using Autograd gives the same expressions as the analytical gradients.

A neural network with a single hidden layer will also be trained to solve the same regression problem. Similar to the analysis for the linear regression methods, the neural network will also be considered for different gradient descent methods and for different values of the related hyperparameters. The number of nodes in the hidden layer, and the activation functions applied to the hidden layer and the output layer will also be analyzed, as the architecture of the network can heavily impact the accuracy of the model. The activation functions that will be tested for the hidden layer are the sigmoid function, ReLU, and leaky ReLU. The linear activation function will be applied at the output layer.

The mean squared error cost function in **??** will be used as the evaluation metric for the trained linear regression models and the neural networks.

3.2 Classification

We will also study a classification problem using logistic regression and a neural network. The data set that will be considered is the Wisconsin breast cancer data set[11]. Each

sample in this data set represents a tumor described by 30 features, and is labeled as either malignant or benign. The goal of the classification problem is to predict the correct label of the tumor, based on the 30 features.

The method of logistic regression is similar to linear regression, except that logistic regression uses the logistic function expressed in Equation 18 to confine the output to the range $(0, 1)$, while the target output is either 0 or 1. The classification of the breast cancer data is a binary classification problem, so we will use the binary cross entropy given in **??** as a cost function. We will analyze the classification accuracy for different values of the regularization parameter λ and the learning rate η. The accuracy score of the predictions will be given by the fraction of tumors for which the model predicts the correct label, given by

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I\left(t_i = y_i\right)}{n}, \tag{28}$$

where I is the indicator function,

$$I = \begin{cases} 1, & t_i = y_i \\ 0, & \text{else} \end{cases}, \tag{29}$$

that is simply counting the number of correct predictions, and normalizing by the sample size. The logistic regression will be performed using stochastic gradient descent.

The neural network will also use the binary cross entropy cost function to measure the accuracy of the model predictions. Since the target values are either 0 or 1 (benign or malignant), we apply the sigmoid activation function in the output layer. The accuracy of the model will be analyzed with respect to the number of hidden layers and nodes in the layers, the activation functions used in the hidden layers, and the values of the hyper-parameters introduced in the model.

4 Results and Discussion

Regression Analysis

plain gradient descent with a fixed learning rate

The convergence plot indicates that there is an optimal learning rate,

$\gamma_{\text{optimal}} = 0.728$, where the number of iterations required for convergence is minimized.

or lower learning rates, convergence is slow, requiring up to 1000 iterations or more. For higher learning rates (approaching 0.8 and above), the number of iterations required increases significantly, suggesting instability and possible divergence.

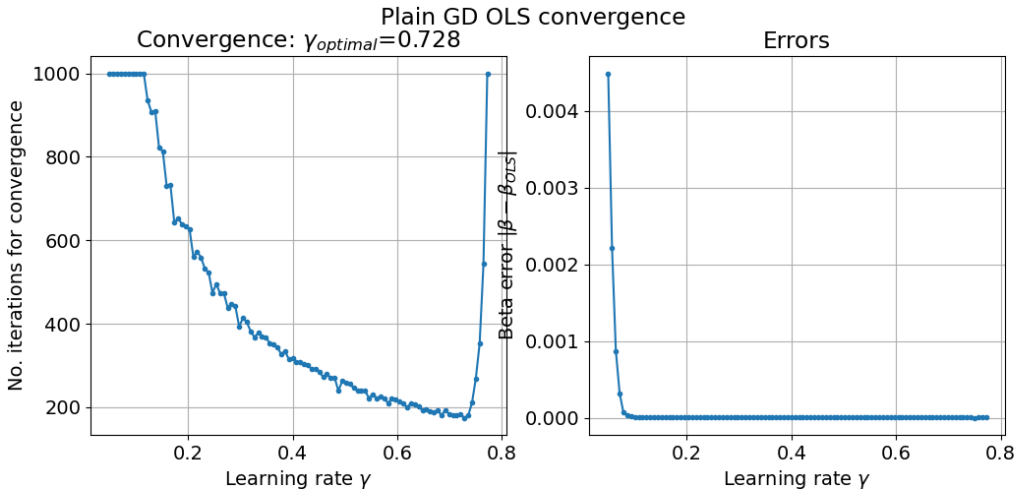

Figure 2: Convergence behavior of plain gradient descent (GD) for OLS regression as a function of the learning rate γ.

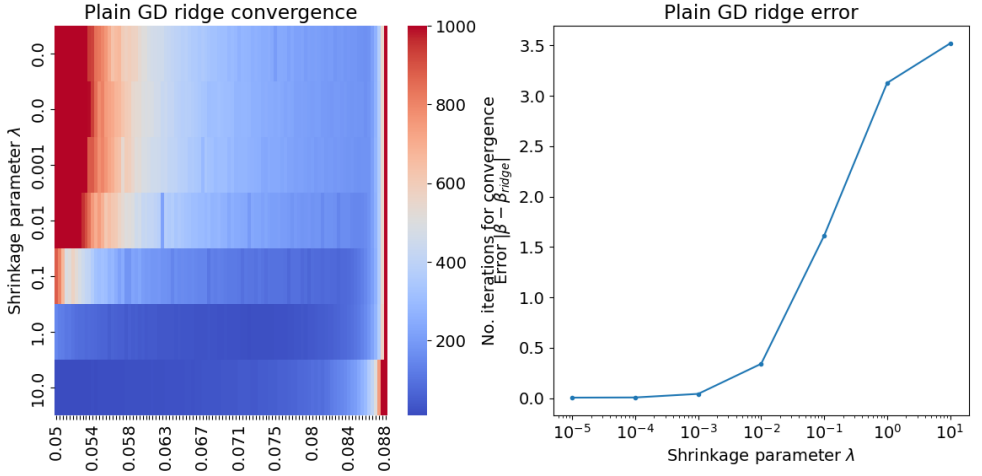

Figure 3: Convergence and error analysis of Plain Gradient Descent (GD) for Ridge regression across different values of the shrinkage parameter λ.

The left heatmap illustrates the error $|\beta - \beta_{\text{ridge}}|$ as a function of the learning rate and shrinkage parameter, showing that the error remains largely independent of the learning rate for different values of λ. The right plot shows the number of iterations required for convergence as a function of the shrinkage parameter, indicating that higher λ values increase the number of iterations needed.

Fastest convergence combination (gamma, lambda)=(0.050, 10.000) with error=3.521

Gradient descent with momentum

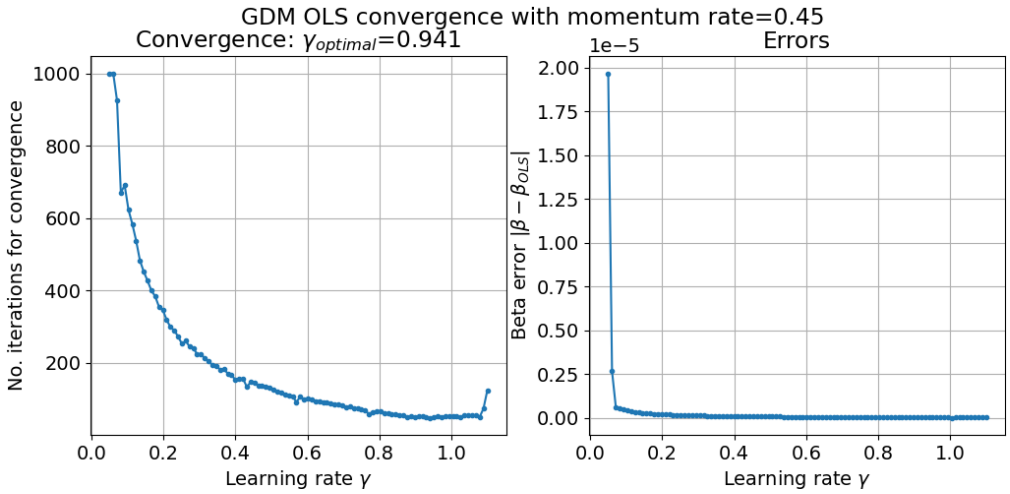

Figure 4: Convergence and error analysis of Plain Gradient Descent (GD) with momentum across different values of the shrinkage parameter λ.

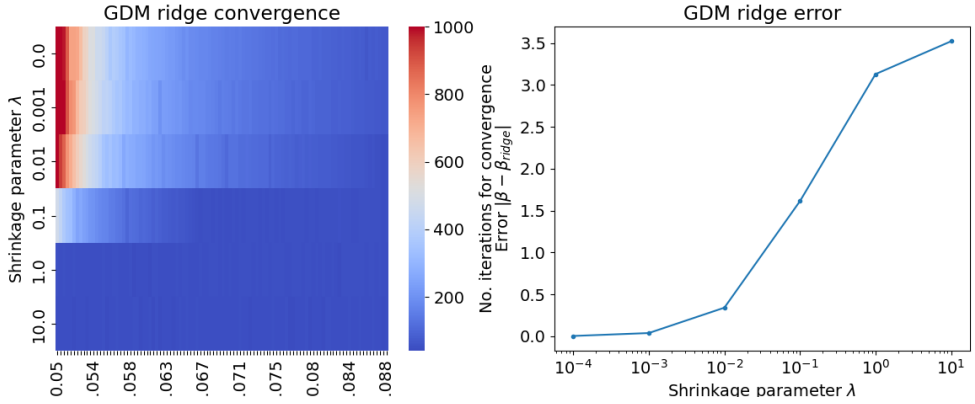

Figure 5: Convergence and error analysis of Plain Gradient Descent (GD) Ridge with momentum across different values of the shrinkage parameter λ.

Fastest convergence combination (gamma, lambda)=(0.059, 1.000) with error=3.128

Stochastic gradient descent with mini batches

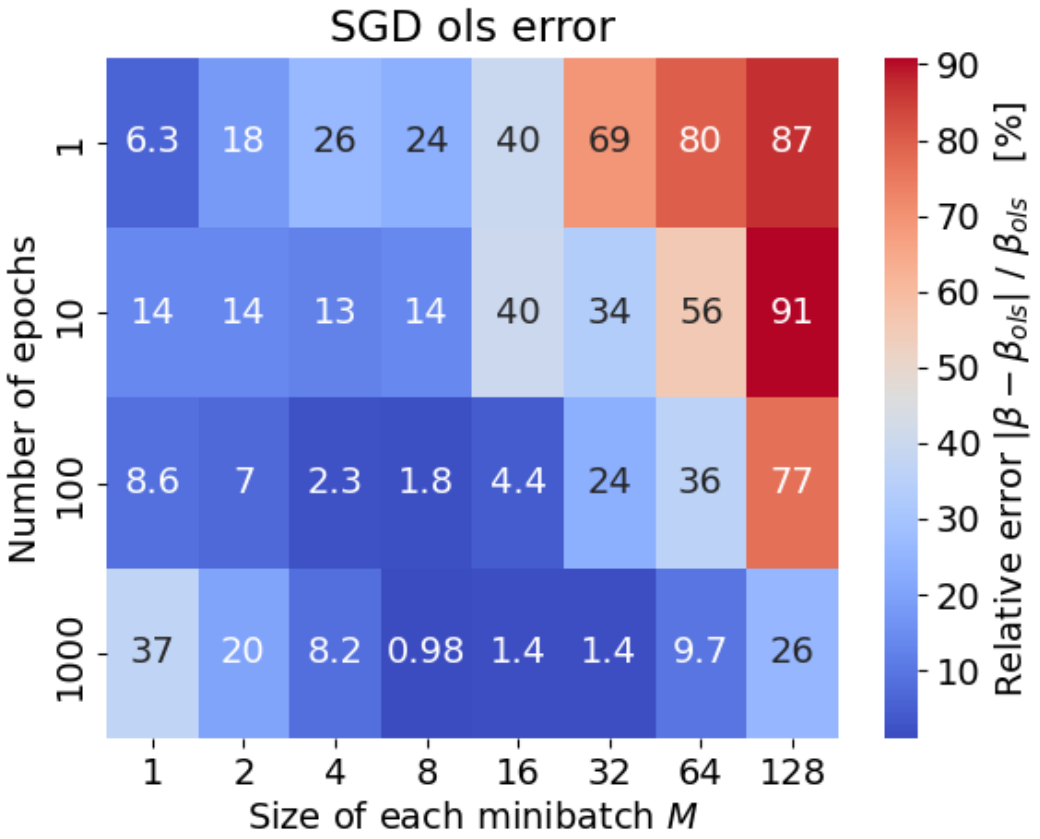

Figure 6: Stochastic Gradient Descent (SGD) across different values epochs and size of minibatch

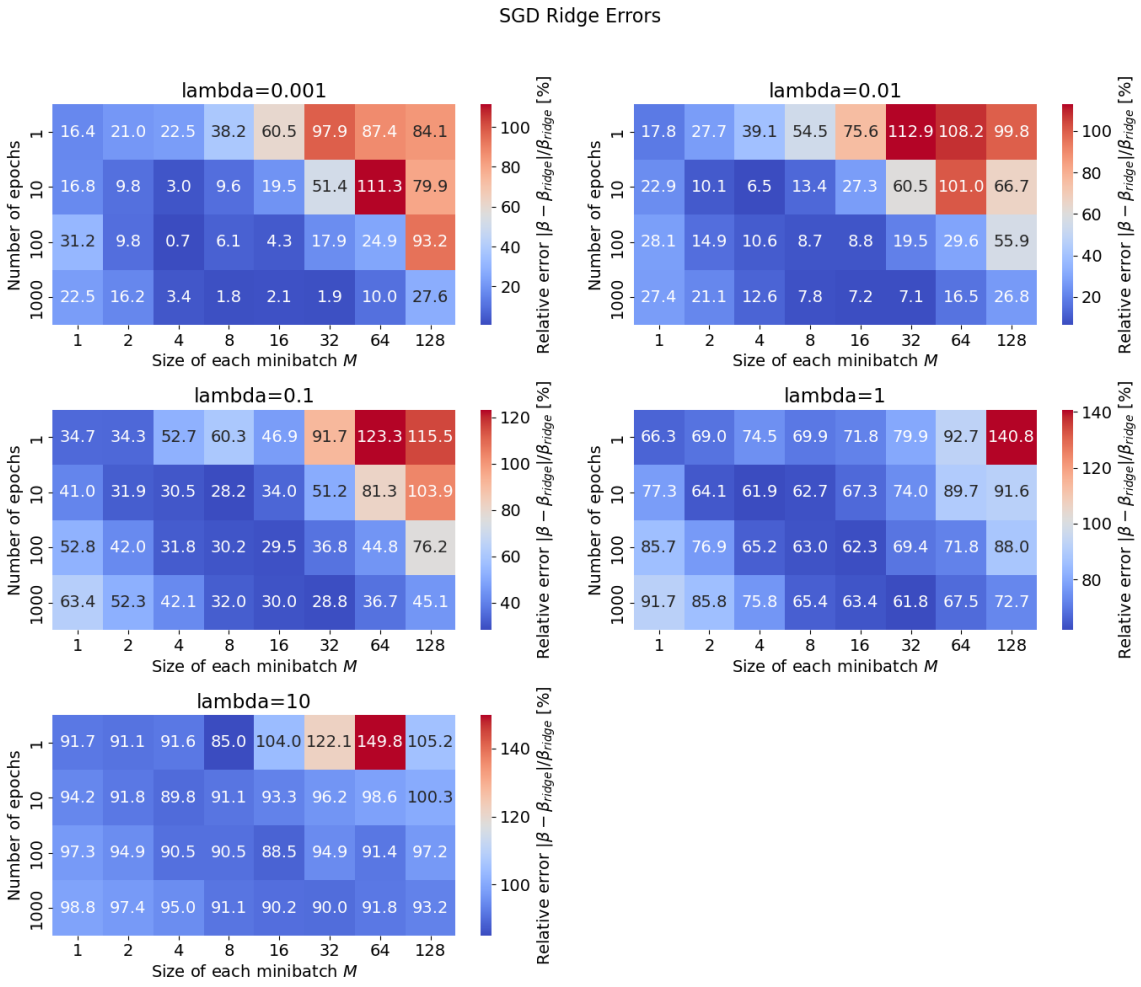

Figure 7: Stochastic Gradient Descent (SGD) across different values epochs and size of minibatch for lambda values of 0.001, 0.01, 0.1, and 1, respectively

Stochastic gradient descent with momentum (SGDM) as function of batch size M and number of epochs N_{epochs}

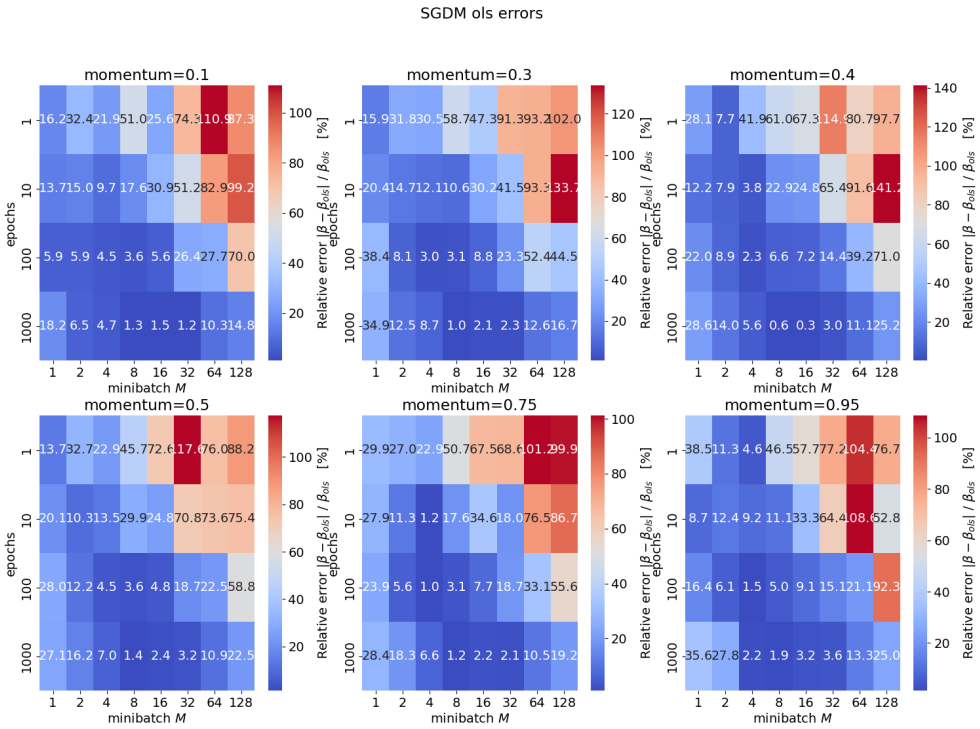

Figure 8: Stochastic Gradient Descent (SGD) across different values epochs and size of minibatch for momentum rate = 0.1,0.3,0.4,0.5,0.75,0.95 respectively

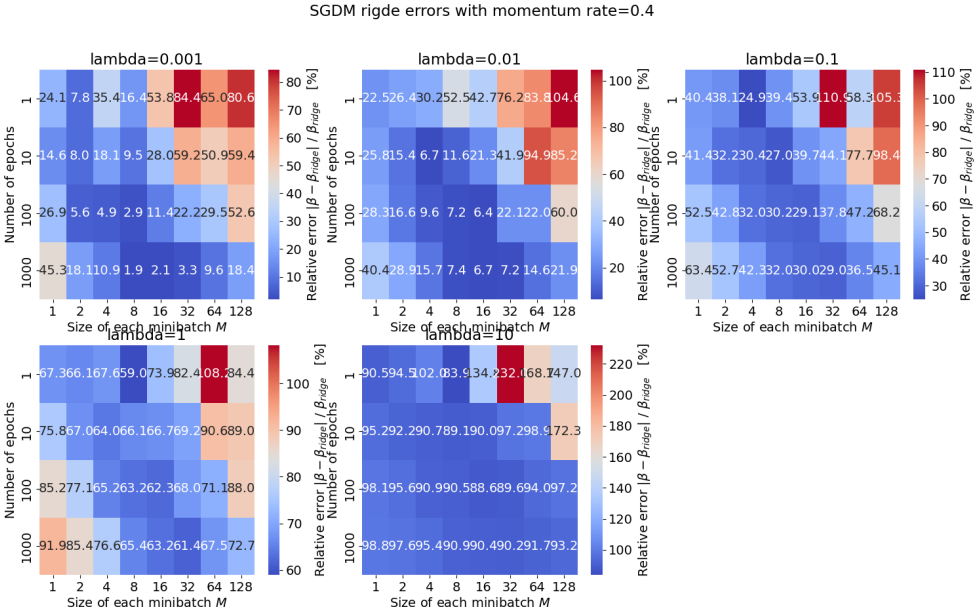

Figure 9: Stochastic Gradient Descent (SGD) for ridge across different values epochs and size of minibatch for momentum rate = 0.4

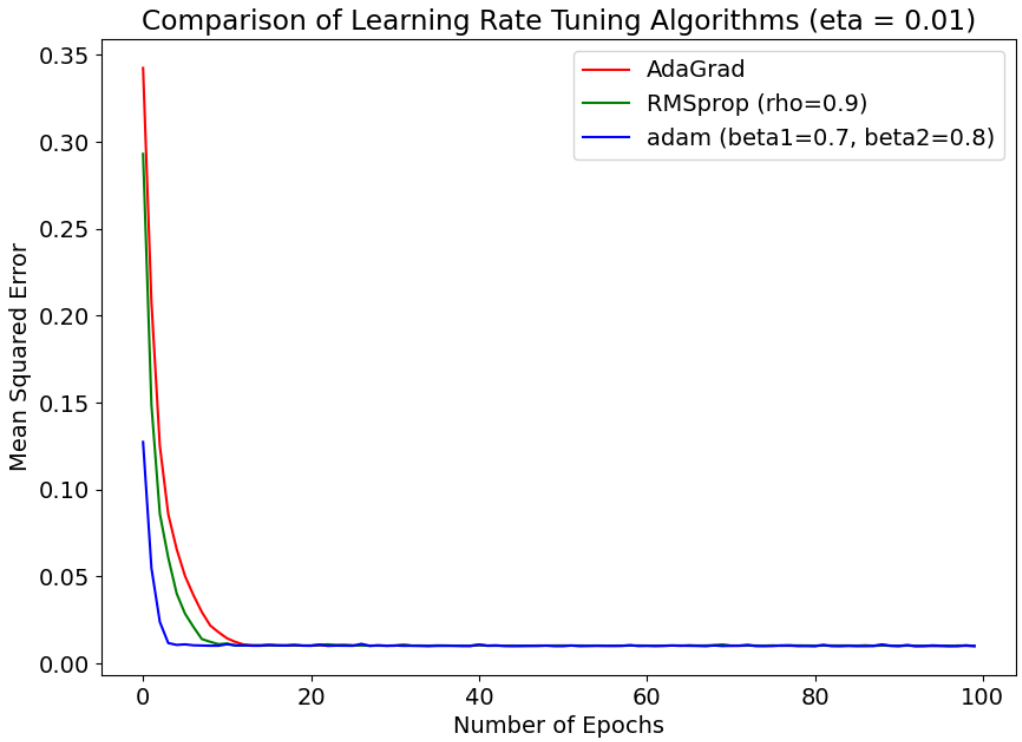

Figure 10: Comparison of Learning Rate Tuning Algorithms (eta = 0.01)

AdaGrad starts with a very high mean squared error, but it converges rapidly and achieves the lowest error among the three algorithms. RMSprop has a relatively smoother learning curve and achieves a lower error compared to Adam, but it does not perform as well as AdaGrad. The Adam algorithm, with its parameters set to beta1=0.7 and beta2=0.8, has the highest mean squared error throughout the training process compared to the other two algorithms.

Regression analysis using Feed Forward Neural Network

We analyze the same regression problem for the Franke function using the FFNN implementation. We begin by using the Sigmoid function as an activation function for one hidden layer 10 nodes, initializing the weights using a normal distribution.

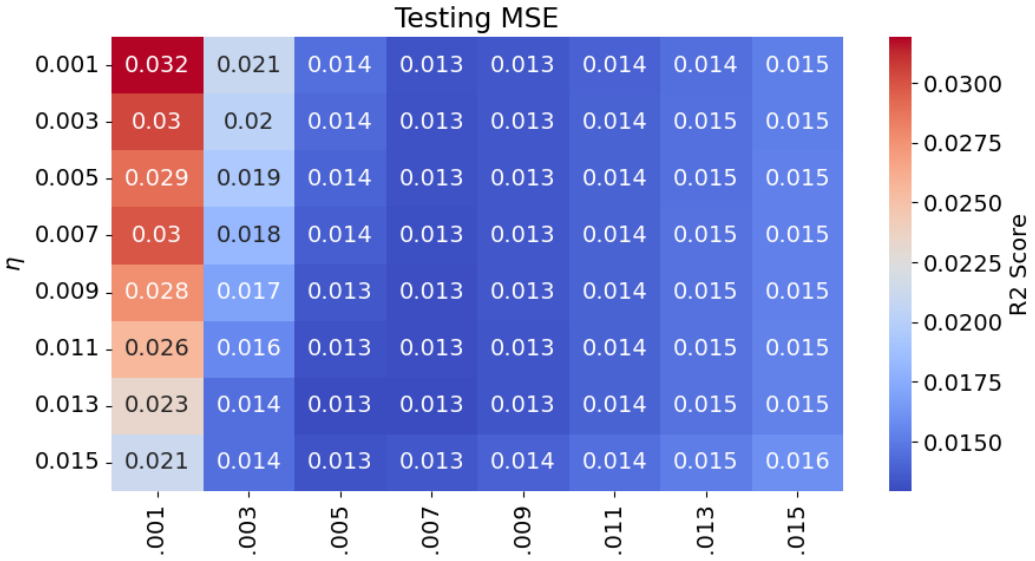

Figure 11: Test MSE using one hidden layer with 10 nodes and sigmoid activation function

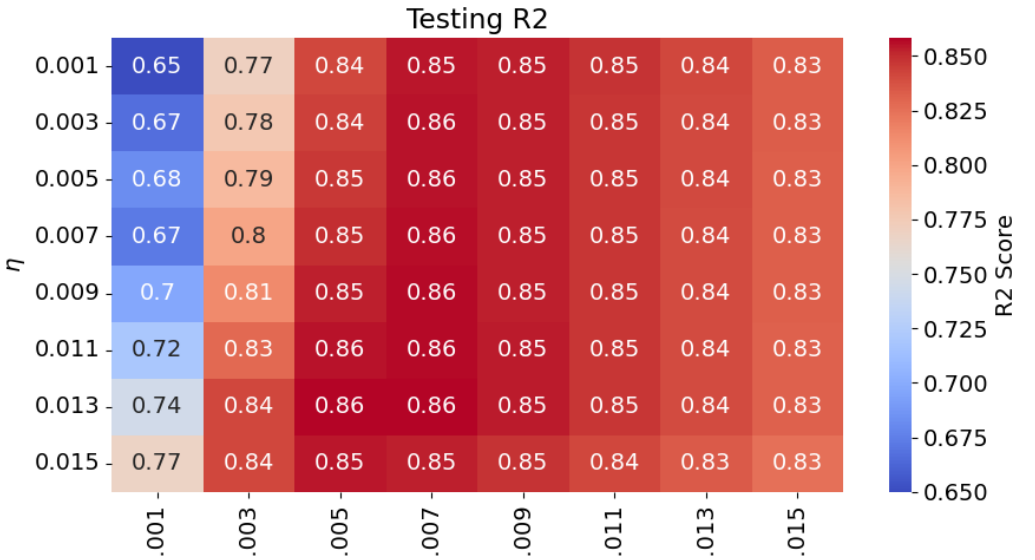

Figure 12: Test R2 using one hidden layer with 10 nodes and sigmoid activation function

The best parameters are: Lambda = 0.005 Eta = 0.013 6,2 The results with these parameters are: MSE: 0.012927600895358438 R2: 0.8582902971746211

I have also used LeakyRelu and Relu in the hidden layer and here is the result:

Table 1: Comparison of Activation Functions

Activation Function	Lambda	Eta	Hidden Layer Sizes	MSE	R^2
ReLU	0.005	0.001	(10, 2)	0.0136	0.8505
Leaky ReLU	0.005	0.005	(2, 2)	0.0126	0.8620
Sigmoid	0.005	0.013	(6, 2)	0.0129	0.8583

The LeakyReLU function still shows the best performance among the three activation functions tested

Classification Analysis

Classification analysis using neural networks

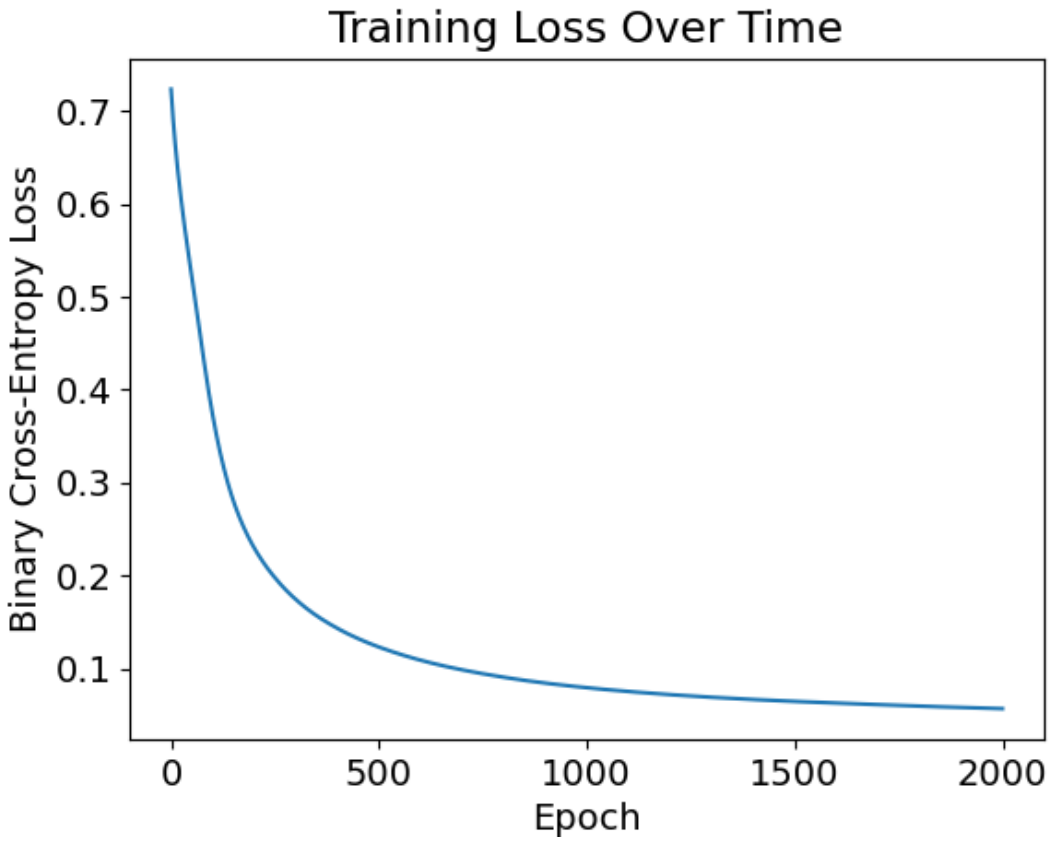

Figure 13: The binary cross-entropy loss of the custom neural network model decreasing over the course of 2000 training epochs.

Our code neural network(from lecture notes) implemented in this study achieved a strong classification performance on the Wisconsin Breast Cancer dataset. The training loss decreased steadily over the 2000 epochs, reaching a final binary cross-entropy loss of 0.0569

The model's accuracy on the training dataset also improved significantly, starting at 46.75 in the first epoch and reaching 98.77 by the end of training. This high accuracy demonstrates the neural network's ability to effectively learn the underlying patterns in the data and make accurate predictions.

To further evaluate the performance, the model was compared to Scikit-Learn's MLPClassifier, a well-known neural network implementation for classification tasks. The Scikit-Learn model achieved an accuracy of 97.37 on the test set, which is lower than the 98.77 accuracy attained by the own code neural network.

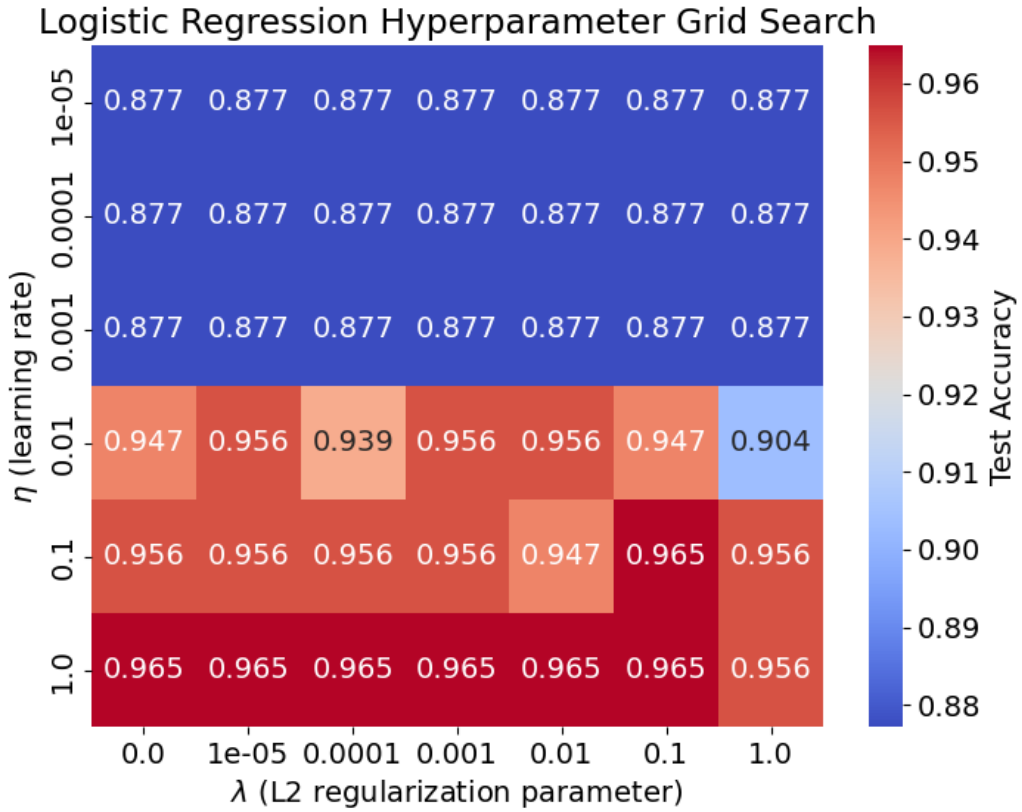

Figure 14: This heatmap shows the test accuracy results from a grid search over different combinations of the L2 regularization parameter λ and the learning rate η

This means that for this logistic regression model, a learning rate of 0.1 coupled with a wide range of L2 regularization values (from 0.0 to 0.1) all produce the same optimal test accuracy of 0.965.

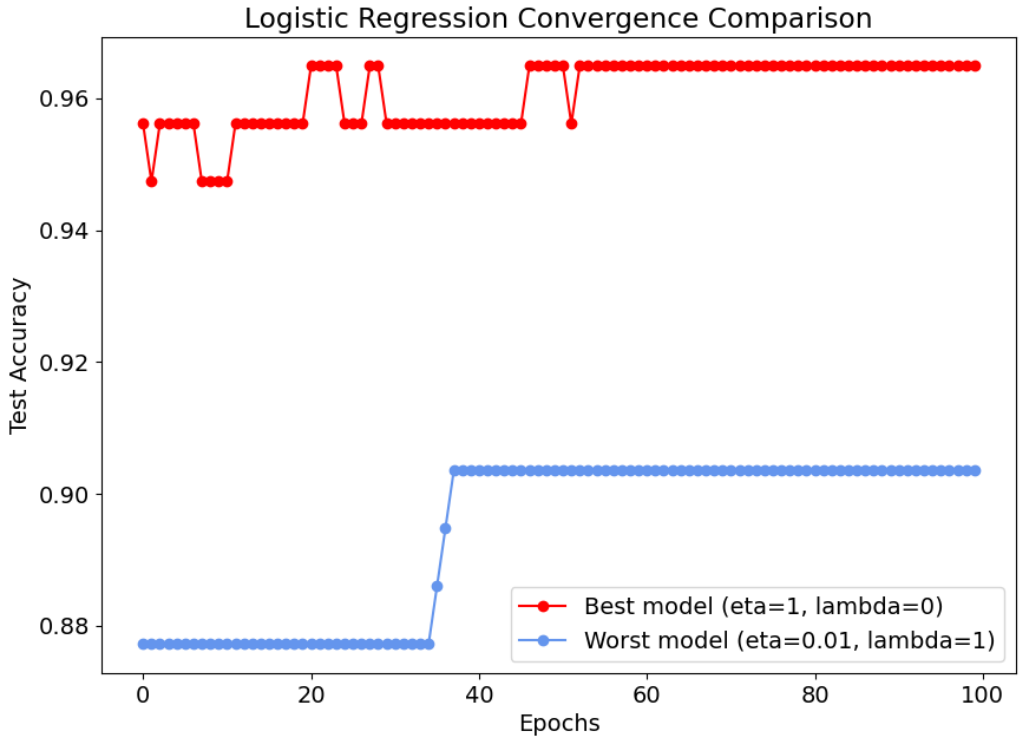

Figure 15: Logistic Regression Convergence Comparison

The best model converges much faster, reaching its peak performance within the first 20 epochs, while the worst model takes around 80 epochs to stabilize. The difference in performance between the best and worst models is quite substantial, with the best model achieving a test accuracy of 0.965 compared to 0.92 for the worst model.

5 Conclusion

In this report, we looked at different methods for solving classification and regression problems. Overall, each method produced decent accuracy, but we found that neural networks performed best for classification tasks. The accuracy of our neural network model was higher than both our own logistic regression code and the standard logistic regression algorithm from scikit-learn.

For regression tasks, the Sigmoid function produced the best results, though only slightly better than the ReLU function. Interestingly, the neural network didn't perform as well on regression as it did on classification, but the difference in accuracy between our logistic regression and neural network models was smaller in regression than in classification.

We believe that with more time to test and fine-tune various parameters, we could have achieved even more precise results. In our experiments with the regression problem using the Franke function and a feedforward neural network (FFNN), we fixed the number of hidden layers and nodes. This approach relied on a grid search to find the

best combination of layers and nodes, but our choices were based on parameters like batch size, epochs, and learning rate, which might not have been optimal.

In short, while our mean squared error (MSE) values indicate the models were reasonably accurate, future studies could explore a more detailed analysis of each parameter to see if fine-tuning would yield even better results.

References

[1] Imagenet large scale visual recognition challenge (ilsvrc), 2024. `https://www.image-net.org/challenges/LSVRC/`.

[2] Richard Franke. *A critical comparison of some methods for interpolation of scattered data.* No. NPS53-79-003. Naval Postgraduate School Monterey CA., 1979.

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

[4] Morten Hjorth-Jensen. Applied data analysis and machine learning, 2024. `https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week34.ipynb`.

[5] Morten Hjorth-Jensen. Applied data analysis and machine learning, 2024. `https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week40.ipynb`.

[6] Morten Hjorth-Jensen. Applied data analysis and machine learning, 2024. `https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week41.ipynb`.

[7] Morten Hjorth-Jensen. Applied data analysis and machine learning, 2024. `https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week42.ipynb`.

[8] Micheal A. Nielsen. *Neural Networks And Deep Learning*. Determination Press, 2015. `http://neuralnetworksanddeeplearning.com`.

[9] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, 4(12):310–316, 2022.

[10] Jerome Friedman Trevor Hastie Robert Tibshirani. The elements of statistical learning: Data mining, inference, and prediction.). *Springer*, 14(2nd ed), 2009.

[11] William Wolberg. Breast Cancer Wisconsin (Original). UCI Machine Learning Repository, 1992. DOI: https://doi.org/10.24432/C5HP4Z.