Project 2 FYS-STK4155

Ingebrigt Kjæreng

November 13, 2023

I explored the use of Gradient Descent and Stochastic Gradient Descent in order to determine which model gave the predicted data values that as closely as possible mimicked the real data. I derived the expression for the cost function using both Analytical Gradients and Automatic Differentiation and implemented the RMSProp, AdaGrad and ADAM methods for learning rate tuning, along with Momentum, using a one-dimensional polynomial model in 4 degrees. The data was also split into several minibatches using Stochastic Gradient Descent, and the combinations of parameters giving the best results were identified. Then I implemented a Feed-Forward Neural Network for the same function, experimenting with the same parameters in addition to different schedulers, activation functions and number of hidden layers. I then implemented a Neural Network for a binary classification problem for the Wisconsin Breast Cancer dataset, where the goal was to correctly identify Benign and Malicious tumors. The result was displayed in a confusion matrix. I then created a Logistic Regression code for the same problem, where the Logistic Regression was able to give the same accuracy as the Neural Network with a much smaller computational cost. For the Gradient Descent and SGD, the best results were largely varying. For the Neural Network using the polynomial function, the best results occured when using Glorot initialization, the LeakyRELU activation function, ADAM, Cyclical Learning Rates and Momentum, and for the Logistic Regression, getting good results yet again required different strategies.

1 Introduction

In this report I will show different techniques to approximate a cost function, namely a simple one-dimensional model with higher-order polynomials, then a dataset with continuous features, the Wisoncins Breast Cancer Dataset. The techniques I will use are Gradient Descent and Stochastic Gradient Descent, the latter whom will be split into minibatches. I will use the models with and without Nesterov Momentum, compare Analytical Expressions of the Gradients (Derivatives) vs. Automatic Differentiation, and look at the properties of the object-oriented optimizers Adagrad, RMSProp and ADAM. I will look at the properties of the so-called learning rates and analyze which parameters has which behaviours and tune the model to the one giving the best results in terms

of various scores such as MSE and R2 and in doing this I will also be using cross-validation and bootstrapping in further analysis. The MSE is the Mean Squared Error, which I will use as a cost function as a template for a specific problem. I will attempt to optimize this cost function by optimizing the MSE and the R2. Then I will implement a Neural Network code, and compare this to a classification code i will implement using Logistic Regression on a Breast Cancer dataset of images representing various tumors. Comparison will be done by an accuracy score of the Classification data. In the Neural Network code I will look at backpropagation, explain what it is and how it is being used, and the choice of cost function. I will explain the feed-forward Neural Network of choice and the sigmoid function, initialization of biases and selection of activation function for the final output layer. Then I will test the different activation functions RELU and Leaky RELU against the sigmoid function, then I will change the cost function to perform a classification analysis on the aforementioned Breast Cancer dataset from Wisconsin. This dataset will have a single output, 0 or 1. I will assess the accuracy score and look at the different parameters such as regularization strength with Ridge and learning rates, activation functions, number of hidden layers and nodes. Finally I will create my own Logistic Regression code, comparing the Neural Network Feed-Forward code to this, based on the Stochastic Gradient Descent algorithm. I will look at the differences when utilizing different learning rates and regularization strength.

2 Mathematical Models

2.1 Using Gradient Descent and Stochastic Gradient Descent with different properties on the Cost Function

2.1.1 Gradient Descent and Stochastic Gradient Descent

A technique being used in large parts of the project is the gradient descent, and an advanced version called the stochatic gradient descent. Gradient Descent (GD) and Stochastic Gradient Descent (SGD) are optimization algorithms used in machine learning and deep learning to minimize a cost or loss function. They both aim to find the optimal parameters of a model by iteratively adjusting them. I will lay out the mathematical models and assumptions for each of them and highlight their key differences: The core idea of Gradient Descent is to update model parameters by moving in the direction of the negative gradient of the cost function. The update rule for GD is as follows:

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$$

Where:

θ_t is the parameter vector at iteration t. α is the learning rate, which controls the step size of the update. $\nabla J(\theta_t)$ is the gradient of the cost function J with respect to the parameters θ_t. Assumptions:

GD assumes that the entire dataset is available for each iteration. It calculates the gradient using the entire dataset, which can be computationally

expensive for large datasets. GD typically converges to the global minimum of the cost function if the cost function is convex and the learning rate is appropriately chosen. Stochastic Gradient Descent (SGD):

To calculate the Gradient I have to sum over all n data points. Doing this at every Gradient Descent step becomes extremely computationally expensive. A solution is to calculate the gradients using sall subsets of the data called mini-batches, which introduces stochasticity(randomness) to the algorithm. Mathematical Model:

In contrast to GD, SGD updates the parameters using a single training example (or a small random mini-batch) at each iteration. The update rule for SGD is as follows: $\theta_{t+1} = \theta_t - \alpha \nabla J_i(\theta_t)$ Where: θ_t) is the parameter vector at iteration t. α is the learning rate, which controls the step size of the update. $\nabla J_i(\theta_t))$ is the gradient of the cost function J with respect to a randomly selected training example i. Assumptions:

SGD assumes that the dataset can be divided into smaller mini-batches or individual training examples. It calculates the gradient using only one or a few data points at a time, making it more computationally efficient, especially for large datasets. SGD may not always converge to the global minimum due to the randomness introduced by the stochastic updates. However, it can escape local minima and explore the cost landscape more effectively. Key Differences:

GD uses the entire dataset for each iteration, while SGD uses only a subset (mini-batch) or a single training example. GD can be computationally expensive for large datasets, as it computes gradients for the entire dataset in each iteration. In contrast, SGD is computationally more efficient. GD typically converges to the global minimum for convex cost functions. In contrast, SGD may not converge to the global minimum due to its stochastic nature, but it can converge to a good solution and is more suitable for non-convex cost functions. SGD introduces randomness due to its stochastic updates, which can be beneficial for escaping local minima and improving exploration of the cost landscape. Important properties of the Gradient Descent is that it is sensitive to initial conditions, gradients are computationally expensive to calculate for large datasets. The Gradient Descent is very sensitive to choices of learning rates, and it treats all directions in parameter space uniformly.

When implementing the linear regression using a gradient descent optimization method, I use a convergence criterion; this is based on the largest eigenvalue of the 2nd derivative of the design matrix. The Hessian matrix of the cost function characterizes the curvature of the cost function's surface. The purpose of the convergence criterion is to determine a useful choice of the learning rate. If the learning rate is too large, the optimization process may diverge, and if its too small it may converge too slowly. Setting the appropriate learning rate is done by making sure it is below this convergence criterion threshold. This choice is based on the properties of the problem and the Hessian matrix. Since I am dealing with linear regression, the optimization problem is convex, and therefore I know that the cost function has a single global minimum. The convergence criterion is set as $2/\lambda MAX$, believed to be applicable to convex optimization problems such as this one. Specifically, the derivative of the cost/loss function

is $dC(\beta)/d(\beta 0)$ and $dC(\beta)/d(\beta 1)$, the gradient can be written as $\nabla\beta C(\beta) = 2/n \begin{bmatrix} \sum i = 1to100of(\beta 0 + \beta 1(xi) - yi) \\ \sum i = 1to100of(\beta 0 + \beta 1(xi) - yixi) \end{bmatrix} = 2/n(x^T)(X\beta - y)$, where X is the design matrix $H = \begin{bmatrix} (d^2C(\beta)/d(\beta)02) & (d^2C(\beta)/d(\beta 0)d(\beta 1)) \\ d^2C(\beta)/d(\beta)d(\beta 1) & d^2C(\beta)/d(\beta 1)^2 \end{bmatrix} = 2/n(X^T(X))$. This implies that $C(\beta)$ is a convex function, since the matrix $(X^T * (X))$ is always positive semidefinite. I am then calculating this using a simple program minimizing $C(\beta)$ using the Gradient Descent method with a constant learning rate according to $\beta(k+1) = \beta_k - y(\nabla)\beta C(\beta(k)), k = 0, 1, ...$ I can use the expression I computed for the gradient and let $\beta 0$ be chosen randomly and let $\gamma = 0.001$. I then stop the iteration when $||\nabla\beta C(\beta(k))|| <_\epsilon = 10^-8$. Then I compare the solution for β with the analytic result given by $\beta = (X^T(X))^-1(X^T)y$. I am computing the derivatives of the cost function with respect to the model parameters $\beta 0$ and $\beta 1$, and checking that the Hessian matrix is diagonal, which means it is a convex optimization problem. The above will eventually allow me to estimate the optimal values of $\beta 0$ and $\beta 1$ for a given dataset. Here, the Cost function is the MSE: $1/n(\sum(yi - y\ i)^2$, where n is the number of data points, yi is the actual target value for the ith data point, and y^i is the predicted value for the ith data point.

2.1.2 Learning Rates

The Gradient Descent and Stochastic Gradient Descent moves the gradients towards its next local minima, and the learning rates determine the size of the steps taken in the parameter space during each iteration. It controls the magnitude of the parameter updates. A larger learning rate results in larger steps, leading to faster convergence, but a learning rate too large is likely to cause the optimization process to overshoot the optimal solution and diverge, creating instability or oscillations. A high learning rate allows for more exploration of the parameter space, which can help escape local minima and discover better solutions. Adaptive learning rates dynamically adjust, so the adaptation takes place as the model iteratively processes through the data, which is split into training in test. This happens when it processes the training, and it then remembers the past gradient data and adjusts the learning rate for the next iterations continuously. The Maximum Learning Rate is set by the steepest direction and this can slow down training significantly. I want to take large steps in flat direction and small steps in the steep directions. This requires to keep track of the 2nd derivative aswell as the Hessian. The ideal scenario is to calculate the Hessian but this is computationally expensive. The Learning Rates in optimization algorithms is critical for the convergence. Learning rates control the step size during parameter updates, and different learning rate strategies can significantly impact the training process. When using Gradient Descent and Stochastic Gradient Descent, I set a constant learning rate, or more, in the case of Stochastic Gradient Descent, with the purpose of having a constant that is simple and works well when the cost function has relatively consistent gradients, but they might not adapt to changes in gradient magnitude during training. When using Adaptive

Learning Rate methods, such as AdaGrad, RMSProp or ADAM, the learning rate is adaptively adjusted for each parameter. They contain additional variables based on the historical information of the gradients. They aim to adapt the varying scales of gradients for different parameters. They help in handling complex cost landscapes, speeding up convergence and improving optimization robustness. An important note in order to create robust and valid results, In my experimentation and analysis, the same learning rate, or in some cases the initial learning rate = 0.1, was applied in all the variants of the Neural Network, both for the one-dimensional function, in the classification analysis and the logistic regression tested.

2.1.3 Nesterov Momentum

Nesterov Momentum, also known as Nesterov Accelerated Gradient (NAG), is an optimization algorithm used in machine learning and deep learning, often as an enhancement to stochastic gradient descent (SGD). It incorporates a momentum term that allows the algorithm to accelerate convergence. The assumptions of Nesterov Momentum are as follows:

Gradient Descent as the Base Algorithm:

Nesterov Momentum is typically applied on top of the standard gradient descent (GD) or stochastic gradient descent (SGD) algorithm. It assumes that you are already using one of these gradient-based optimization techniques as the base algorithm for parameter updates.

Gradient Information Availability:

Like GD and SGD, Nesterov Momentum assumes that you have access to the gradient information of the cost function with respect to the model parameters. This gradient is used to compute the parameter updates.

A Decreasing Learning Rate:

Nesterov Momentum requires a decreasing learning rate schedule. A common choice is to use a learning rate that decreases over time, often referred to as a learning rate schedule. This schedule can be tuned based on the problem and the dataset.

Smooth and Continuous Cost Function:

Nesterov Momentum works well with smooth and continuous cost functions. It is particularly effective in optimizing non-convex cost functions, where it helps in escaping local minima.

Reasonable Hyperparameter Choices:

To make Nesterov Momentum effective, you need to choose appropriate hyperparameters, such as the momentum coefficient μ and the learning rate α. The choice of these hyperparameters may depend on the specific problem and data. Common values for the momentum coefficient are in the range of 0.9 to 0.99, and the learning rate should be chosen carefully.

Implications on OLS and Ridge: OLS doesnt have any effect with Stochastic Gradient Descent nor Momentum, because it has a closed-form solution. For Ridge regression, using Stochastic Gradient Descent may introduce more variability in the Ridge regression due to the stochasticity. Gradient Descent does

not have this stochastic component and may be more useful in some cases. Nesterov Momentum can be used for optimizing Ridge Regression. Nesterov Momentum can accelerate the convergence of Ridge Regression, especially in cases where there are multiple local minima, as it provides better exploration capabilities compared to traditional gradient-based methods. What it does Mathematically is:

The standard gradient descent algorithm calculates the gradient and updates the parameters directly. This can lead to oscillations in the cost landscape, especially in regions with multiple local minima. Nesterov Momentum introduces the concept of "momentum" which allows the algorithm to incorporate information about the past gradient. The Nesterov Momentum update rule is as follows:

$$v_{t+1} = \mu v_t - \alpha \nabla J(\theta_t + \mu v_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1}$$

Where:

μ is the momentum coefficient (typically a value close to 1).vt is the momentum vector at iteration t.

Here, in each iteration, the gradient of the cost function is calculated at the current position, θ t, as in the standard gradient descent.

Momentum Update (v_t): The momentum vector vt is updated based on the gradient at $+\theta$t$+\mu$vt, which is a prediction of where the parameters will be in the next step. The momentum term μvt accounts for the accumulated momentum from previous iterations.

Parameter Update (

$$+1\theta t + 1)$$

: The parameter vector θt is then updated using the momentum vector $+1vt+1$, which effectively adjusts the position for the next iteration.

2.1.4 Analytical Gradients vs. Automatic Differentiation

I tested two different approaches to compute the gradients, the derivatives of the cost functions. The purpose of using gradients is to see how the cost function changes as I make small adjustments to each parameter. When using analytical expressions for the gradients, an assumption that must be met is using a mathematical, closed-form expression for the cost function. In mathematical terms, in a linear regression model using MSE as the cost function which I am doing here, is: $\nabla J(\theta) = 1/NX^T(X\theta - y)$, where θ is the parameter vector, N is the no. of data points, X is the design matrix and y is the target vector. Automatic Differentiaton, on the other hand, uses the concept of computational graphs and reverse-mode differentiation to calculate gradients effectively. It constructs a computational graph while evaluating the objective function and its subcomponents. The computational graph records the operations performed during the evaluation. The gradients are then calculated by traversing the graph backward, applying the chain rule to compute derivatives. This allows the computation of

gradients for complex and non-differentiable functions. Automatic differentiation is especially useful for optimizing complex models with nontrivial objective functions and is mainly useful where manually deriving gradients can be error-prone and impractical, but sometimes it may have a higher computational cost so one needs to assess whether it is necessary. I first tested all of the techniques in this section using analytical gradients, and then I redid the same techniques using automatic differentiation, specifically by using the JAX-library in Python (see References).

2.1.5 Adagrad, RMSProp and ADAM

AdaGrad is an optimization algorithm adapting the learning rate for each parameter individually. The update rule for AdaGrad is: $\theta t+1 = \theta t-(\alpha/\sqrt{Gt+\epsilon}\nabla J(\theta t)$, where

1. θt is the parameter vector at iteration t, α is the learning rate, which can be a fixed value or adaptively adjusted.

2. Gt is a diagonal matrix where each diagonal element is the sum of the squares of past gradients for the corresponding parameter.

3. $\nabla J(\theta t)$ is the gradient of the cost function J with respect to the parameters θt,

4. ϵ is a small constant added for numerical stability.

AdaGrad assumes the cost function may have different scales for different parameters. It adapts the learning rate to compensate for this difference. AdaGrad adapts the learning rate for each parameter based on the historical information of the gradients. Parameters that have received large gradients in the past will have a smaller learning rate, which helps stabilize the optimization process. This adaptiveness can lead to very small learning rates for some parameters, potentially causing slow convergence. AdaGrad has an aggressive learning rate decay. RMSProp is adaptive and mitigates this. The update rule for RMSProp is $\theta t + 1 = \theta t$ - α $\sqrt{Eg^2t+\epsilon}$, where the terms are the same as in AdaGrad, and in addition I have Eg^2t, the exponentially moving average of the squared gradients, where the decay factor ρ is applied to update it. RMSProp assumes the cost function may have different scales for different parameters, similar to AdaGrad. It assumes the gradients are provided for each parameter during optimization. RMSProp adapts the learning rate, preventing it from decreasing too aggressively. It maintains an exponentially moving average of the squared gradients and scales the learning rate by the square root of this moving average. This helps stabilize and accelerate the optimization process by providing a more balanced learning rate for all parameters. ADAM combines momentum and RMSProp, and the update rule is: $\theta t+1 = \theta t-(\alpha)/\sqrt{v^t}+\epsilon * m^t$, the terms are as with AdaGrad and RMSProp in addition to v^t, an exponentially moving average of squared gradients, m^t, an exponentially moving average of gradients,

and $\beta 1$,$\beta 2$, controlling the exponential decay rates of m^t and v^t. ADAM assumes the cost function may have different scales for different parameters and that gradients are provided for each parameter during optimization. It adapts the learning rates for each parameter based on the first moment and the second moment of the gradients. The difference in the 3 methods is how they adapt the learning rates and how they handle decay; AdaGrad implicitly decays the learning rates by accumulating the squared gradients in Gt, so the learning rate effectively decreases over time as more gradients are accumulated. RMSProp has an exponentially moving average term Eg^2t in addition to the decay factor ρ determining how quickly the moving average forgets past squared gradients. Smaller values of ρ imply a more rapid decay of past gradients and thus a more adaptive learning rate. In ADAM, $\beta 1$ and $\beta 2$ control the exponential decay rates. An important distinciton is that RMSProp and ADAM provide explicit mechanisms for controlling the decay of past gradient information, but they involve more complex operations and additional memory.

2.2 Neural Network

The Neural Network goes deeper into the process in the gradient descent. During the process of iterating over the data, creating training and test, I can adjust the parameters, the weights and biases for a single gradient descent step. Each "neuron" has a weight and a bias. I then choose which neurons to modify the weights and biases. I want to modify the neurons that are far off from the target. A certain weighted sum of all the activations in the previous layer, plus a bias, the connections with the brightest neurons from the preceding layer have the biggest effect. I want to look at what relative proportions to those changes cause the most rapid decrease to the cost. What Stochastic Gradient Descent does is it randomly subdivides the data into mini-batches and compute each step with respect to a minibatch, which is why in Stochastic Gradient Descent I edit the batch size. The goal of a feed forward network is to approximate some function $f*$. For example,for a classifier,$y = f*(x)$ maps an input x, to a category y. The feed forward network defines a mapping y=f(x;θ) and learns the value of the parameters θ that result in the best function approximation.These models are called feed forward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. There are no feedback connections in which outputs of the model are fed back into itself. In any Neural Network code, initial conditions are very important and in this case they are given by weights and biases for hidden layers, where the initial weights and biases for the hidden layers are generated using np.random.randn() and np.zeros() functions, respectively, in the constructor. These are stored in the self.weights and self.biases lists. Initial weights and biases for the output layer are also generated using np.random.randn() and np.zeros(), and they are stored in the self.weights and self.biases lists. Below is the relevant code section containing the initial conditions:

```
def __init__(self, input_dim, hidden_layer_sizes, output_dim):

    # Initialize weights and biases for hidden layers
    prev_layer_size = input_dim
    for layer_size in hidden_layer_sizes:
        weight_layer = np.random.randn(prev_layer_size, layer_size)
        bias_layer = np.zeros(layer_size)
        self.weights.append(weight_layer)
        self.biases.append(bias_layer)

        prev_layer_size = layer_size

    # Initialize weights and biases for the output layer
    weight_output = np.random.randn(prev_layer_size, output_dim)
    bias_output = np.zeros(output_dim)
    self.weights.append(weight_output)
    self.biases.append(bias_output)
```

2.2.1 Feed-Forwarding

During the first phase I am using the input data, and the information coming through has a unique weight associated to it. For example,for a classifier,$y = f*(x)$ maps an input x to a category y. A feedforward network defines a mapping $y = f(x;\theta)$ and learns the value of the parameters θ that result in the best function approximation. These models are called feed forward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. The model is associated with a directed a cyclic graph describing how the functions are composed together. For example, If I have three functions $f(1), f(2)$, and $f(3)$ connected in a chain, to form $f(x) = f(3)(f(2)(f(1)(x)))$. These chain structures are the most commonly used structures of neural networks. In this case,$f(1)$ is called the first layer of the network, $f(2)$ is called the second layer, and so on. The overall length of the chain gives the depth of the model. The name "deep learning" arose from this terminology. The final layer of a feed-forward network is called the output layer. During neural network training, we drive f(x) to match f*(x).The training data provides us with noisy, approximate examples of f*(x) evaluated at different training points. Each example x is accompanied by a label $y = f*(x)$. The training examples specify directly what the output layer must do at each point x; it must produce a value that is close toy. The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data do not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation off. Because the training data does not show the desired output for each of these layers, they are called hidden layers. Here, the function I am using is the Cost function of a simple one-dimensional model of higher-order variables.

2.2.2 Architecture of the Model

1. 1. Cost Function: MSE, measuring the average squared difference between predicted and actual values.
2. 2.Learning Rate Schedulers: Step Decay, Exponential Decay and Cyclical Learning Rates.
3. 3. Activation Function: Sigmoid function.
4. 4. Network Architecture: - Input Layer: 1 neuron for 'x', - Output Layer: 1 neuron for the predicted value 'y', - Hidden Layers: Multiple neurons added to capture the linearity of the 5th-degree polynomial relationship, experimenting with different number of neurons.
5. 5. Neural Network Depth: Experimented with different number of hidden layers to determine the best model architecture.

2.2.3 Choice of Cost Function and Optimization

Since I am using a regression problem, not a classification problem at this point, I am trying to predict a continuous numerical value 'y' based on the input 'x'. For this regression problem, an appropriate cost function is the Mean Squared Error (MSE) cost function, which measures the average squared difference between the predicted values and the actual values: $MSE(\theta) = 1/n(\sum(y - f(x,\theta))^2)$, where n is the number of data points, y is the actual target value, and $f(x,\theta)$ is the predicted value from the model. I am using the MSE over the MLE, because using the sigmoid activation function and aiming to predict a continuous numerical value 'y' based on input 'x', MSE is a reasonable choice as it aligns with the assumption of constant variance in a Gaussian distribution. To mitigate gradient saturation, I am applying careful weight initialization (Xavier/Glorot initialization, which I apply here) and employing non-saturating activation functions like ReLU(I apply this in the next section). The derivative of the Cost Function with respect to the weights is given by

$$\frac{\partial C(WL)}{\partial wL_{jk}} = (a_{Lj} - t_j)\frac{\partial a_{Lj}}{\partial wL_{jk}} \tag{1}$$

, where the last partial derivative reads

$$\frac{\partial a_{Lj}}{\partial wL_{jk}} = \frac{\partial a_{Lj}}{\partial z_{Lj}}\frac{\partial z_{Lj}}{\partial wL_{jk}} \tag{2}$$

$$= a_{Lj}(1 - a_{Lj})a_{L-1k} \tag{3}$$

.

2.2.4 Schedulers

I experimented with step decay, exponential decay and cyclical learning rates in combination with optimization algorithms AdaGrad, RMSProp and ADAM as mentioned earlier. Step Decay reduces the learning rate by a fixed factor after a certain no. of epochs. Exponential Decay reduces the learning rate exponentially after each epoch or batch. Mathematically given as $lr = lr_0 * e^(- k * epoch)$, where lr is the learning rate, lr_0 is the initial learning rate, k is a decay constant, and epoch is the current epoch number. I then evaluated which one worked best.

2.2.5 Activation Functions

Feedforward networks have introduced the concept of a hidden layer, and this requires choosing the activation functions that will be used to compute the hidden layer values. During the training process, Gradient Descent updates the weights and biases of the neural network to minimize a loss function. Activation Functions introduce non-0 gradients, needed to compute the gradients for back-propagation without non-linear activation functions, the chain rule in calculus would lead to a product of constant values, making it impossible to learn from the data effectively. I will here use the Sigmoid function, Mathematically Given by: $\sigma(z) = 1/(1 + e^(- z))$ Where: $\sigma(z)$ is the output of the sigmoid function. z is the input to the function, which can be any real number. e is the base of the natural logarithm. I later used the identity function, the ReLU activation function and other activation functions, to see which one fits best.

2.2.6 Backpropagation

The network consists of input layers, output layers and hidden layers. Each layer is represented as a vector-to-vector function, and the network's weights and biases are represented by θ. The activation function is the sigmoid function, used for the neurons in the hidden layer. The loss at a given data point (x,y) is: $L(\theta) = 1/2(y - f(x, \theta))^2$, where f(x,$\theta$) is the predicted value. The backpropagation algorithm initialized the network parameters θ with small random values, it defines the learning rate(α), controlling the step size during gradient descent. Given input data x, it computed the activations of the neurons in the hidden layers and output layer using the sigmoid activation function: $a^(1) = \sigma(z^(1))$, $a^(2) = \sigma(z^(2))$, where $z^(1) and z^(2)$ are weighted sums of the inputs to the hidden and output layers. It then calculated the error in the output layer: $\gamma^(2) = a^(2) - y$, then it computed the gradient of the loss with respect to θ(the weights and biases): $dL/d\theta^(2) = a^(1) * \gamma^(2)$, $dL/d\theta^(1) = x\gamma^(2) * \theta^(2) * a^(1) * (1 - a^(1))$. It then updated the weights and biases in each layer based on the computed gradients: $\theta^(2) = \theta^(2) - \alpha * dL/d\theta^(2)$, $\theta^(1) = \theta^(1) - \alpha * dL/d\theta^(1)$, and repeated these steps for a number of epochs until convergence. The back-propagating equation is mathematically given by:

(replacing L with a general layer l)

$$\delta_{lj} = \frac{\partial C}{\partial z_{lj}}.$$

I can express this in terms of the equations for layer $l+1$. Using the chain rule and summing over all k entries, I have

$$\delta_{lj} = \sum_k \frac{\partial C}{\partial z_{l+1k}} \frac{\partial z_{l+1k}}{\partial z_{lj}} = \sum_k \delta_{l+1k} \frac{\partial z_{l+1k}}{\partial z_{lj}},$$

, and recalling that

$$z_{l+1j} = \sum_{i=1}^{M_l} w_{l+1ij} a_{li} + b_{l+1j},$$

, with M_l being the number of nodes in layer l, I obtain

$$\delta_{lj} = \sum_k \delta_{l+1k} w_{l+1kj} f'(z_{lj}),$$

.

2.2.7 Layers, Nodes, Neurons, Hidden Layers, Output

What is a neuron, and why does it use them? Neurons in the input layer serve as feature representations. They take the raw input data and convert it into a form that can be processed by the neural network. However, the input data may not be suitable for directly capturing complex patterns or relationships. The transformation into a hidden layer neuron allows for the creation of more abstract and high-level features. It enables the network to learn and represent relationships and patterns in the data that are not readily apparent in the raw input. Neurons in the input layer serve as feature representations. They take the raw input data and convert it into a form that can be processed by the neural network. However, the input data may not be suitable for directly capturing complex patterns or relationships. An input neuron represents a feature or input value. It doesn't perform any computation but directly passes the input value to the subsequent layers. The transformation into a hidden layer neuron allows for the creation of more abstract and high-level features. It enables the network to learn and represent relationships and patterns in the data that are not readily apparent in the raw input. When an input neuron becomes a hidden layer neuron, it goes through a transformation that involves the weighted sum; The input value is multiplied by a weight (w), and then a bias (b) is added: $z^(l) = w^(l) * a^(l-1) + b^(l)$. The activation function; The weighted sum is passed through an activation function (σ), in this case the sigmoid function, to introduce non-linearity. In the hidden layer, the neurons role is to capture complex patterns or features in the data. I am experimenting with the number of hidden layers as part of determining if there is little or many complex features in the data.

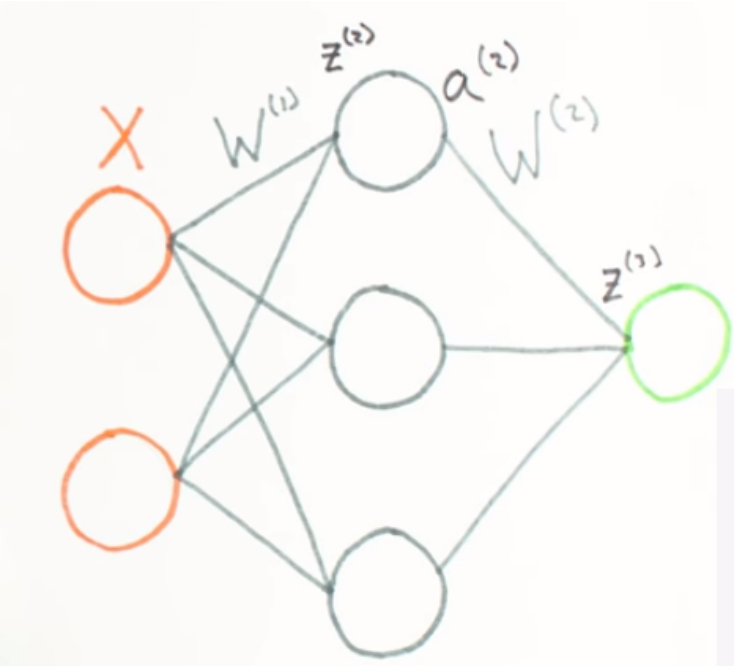

Figure 1: **2 inputs, creation of neurons, 1 output and backpropagation, using an activation function**
1

A simple example of a Neural Network using feed-forwarding, backpropagation and activation function:

A more complex example of using feed-forwarding with several layers, running through the activation function for every gradient descent step, and adjusting the weights and biases of neurons:

An important thing in this illustration is that its more important to change the weights of the neurons who are more far off from their target than the ones who are already close, and this is done during the back-propagation to change the weights with the connections with the neurons with larger errors neurons from the preceding layer, as they have bigger effect.

In my case, from the equations I used to set up the gradients, I go from the 1st layer to the final layer, these are the functions I would calculate. Automatic Differentiation works whenever I have a function that can be written as a computational graph, often elementary functions. I pick the activation functions so they have a specific mathematical property, but also so the derivatives are easy to compute. Every Neuron is connected to the subsequent layer of neurons in folds, and then every neuron is a unique bias. The dimensionality of these hidden layers determines the width of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than think-

Figure 2: **More inputs, creation of several layers of neurons, running through activation function for every gradient descent step, and adjustment of weights and biases.**
2

ing of the layer as representing a single vector-to-vector function,we can also think of the layer as consisting of many units that act in parallel,each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value. The aim is that the network will automatically learn to approximate the relationship between 'x' and 'y', including the polynomial relationship, during training. Since I am using a one-dimensional data input where the highest order is the 4th order of x, and the data is generated based on a 5th-degree polynomial relationship, I used only 1 neuron in the input layer, which is directly connected to the input feature 'x'. I used 1 neuron in the output layer, connected to the output feature 'y'. In the hidden layer, i added multiple neurons to allow the network to capture the nonlinearity of the 5th-degree polynomial relationship. I began testing for a smaller amount of neurons and gradually tested for more neurons.

2.2.8 Initializing the Biases

I initialized biases to zero in conjunction with Xavier/Glorot weight initialization. This method helps balance the scale of gradients and is often used with the sigmoid activation function in hidden layers.

2.3 Using different Activation Functions

I used the RELU function, mathematically given as: $f(x) = x, if x > 0, 0, if x <= 0$. The ReLU function is piecewise linear. For input values less than or equal to zero, the function outputs zero, and for input values greater than zero, it outputs a linear function of the input. The ReLU function is piecewise linear. For input values less than or equal to zero, the function outputs zero, and for

input values greater than zero, it outputs a linear function of the input. one of the challenges with ReLU is the "dying ReLU" problem, where neurons can get stuck in an inactive state if they consistently receive negative inputs during training. This is attempted mitigated by using the LeakyRELU activation function, $f(x) = x, if x > 0, alpha * x, if x <= 0$. Where x is the input to the function. alpha is a small positive constant, which determines the slope of the leak for negative inputs.

2.4 Classification Analysis using Neural Networks

I ran a classification analysis using Neural Networks on the Classification data Wisconsin Breast Cancer dataset (see References). This is a binary classification case where there are two possible outcomes, Benign(B) or Malignant(M) tumors. Here, I experimented with using the Negative log-likelihood cost function. I justify this with the fact that negative log-likelihood and cross-entropy are usually applied when dealing with probability distributions over discrete categories (e.g., classification problems). The NLL cost function and cross-entropy are generally defined for probabilistic models as follows:

Negative Log-Likelihood (NLL): $\text{NLL}(\theta) = -\sum(logp(y|x,\theta))$ Cross-Entropy: $\text{Cross-Entropy}(\theta) = -\sum(p(y|x,\theta) * logq(y|x))$, or more expansively

$$C(\theta) = -\ln P(D \mid \theta) = -\sum_{i=1}^{n} y_i \ln[P(y_i = 0)] + (1 - y_i) \ln[1 - P(y_i = 0)] = \sum_{i=1}^{n} L_i(\theta).$$

Where:

$p(y|x,\theta)$ represents the true data distribution. $q(y|x)$ represents the model's predicted distribution. θ are the model parameters.

This last equality means that I can interpret the cost function as a sum over the loss function for each point in the dataset, $L_i(\theta)$. The negative sign means the algorithm is minimizing a positive number, rather than maximizing a negative number.

Unregularized maximum likelihood will drive the model to learn parameters that drive the softmax to predict the fraction of counts of each outcome observed in the training set:$softmax(z(x;\theta))i = mj = 11y(j) = i, x(j) = xmj = 11x(j) = x$. Because maximum likelihood is a consistent estimator, this is guaranteed to happen as long as the model family is capable of representing the training distribution. In practice, limited model capacity and imperfect optimization will mean that the model is only able to approximate these fractions. Many objective functions other than the log-likelihood do not work as well with the softmax function. Specifically, objective functions that do not use a log to undo the exp of the softmax fail to learn when the argument to the exp becomes very negative, causing the gradient to vanish. In particular, squared error is a poor loss function for softmax units and can fail to train the model to change its output,even when the model makes highly confident incorrect predictions. The softmax activation function is normally used for multi-class classification tasks. In this case I have a binary classification task with just one output, and

therefore I keep the sigmoid activation function from earlier. The performance for the classification problem with this dataset was measured using the so-called accuracy score. The accuracy is the number of correctly guessed targets divided by the total number of targets; Accuracy=$\sum(i(yi)/n)$, where I is the indicator function, 1 if ti=yi and 0 otherwise, in the case of a binary classification problem like this one, ti represents the number of targets, in this case the variable 'Diagnosis', and yi the outputs of the Neural Network code, with n the number of targets ti. The design matrix in this case consists of the continuous features, 'radius1', 'texture1', 'perimeter1', 'area1', 'smoothness1', 'compactness1', 'concavity1' and $'concave_points1'$. In order to perform the classification analysis from the existing code, I loaded the dataset, preprocessed the dataset to extract features and labels, updated the input dimension to match the number of features, replaced the data generation part with my loaded dataset, updated the output dimension to match the structure of the dataset and changed the activation function to the negative log-likelihood/cross-entropy from the current MSE cost function. I then adjusted the number of hidden layers and neurons. The architecture of the Neural Network in this case looks as follows:

I visualized the accuracy using Confusion Matrices, a table used to evaluate the performance of the classification model. It shows the true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) for a binary classification problem, where true Positives (TP) are cases where the model correctly predicted the positive class, in this case correctly identifying a healthy person as being healthy(having a Benign tumor). True Negatives (TN) are cases where the model correctly predicted the negative class, in this case correctly identifying an unhealthy person being unhealthy(having a Malignant Tumor), False Positives (FP) are cases where the model incorrectly predicted the positive class when it should have been negative, in this case falsely claiming a person healthy despite having a Malignant Tumor, and False Negatives (FN) are cases where the model incorrectly predicted the negative class when it should have been positive, in this case attributing a Malignant Tumor to a person who had a Benign tumor. Now the Accuracy was calculated as: $Accuracy = TP + TN + FP + FNTP + TN$. I attempted to optimize the results by using different activation functions, modifying the number of hidden layers, and using He/Glorot initialization.

2.5 Logistic Regression

Activation Function: The logistic regression model using the sigmoid function can be expressed as: $P(Y = 1|X) = \sigma(\beta 0 + \beta 1X1 + \beta 2X2 + ... + \beta_n Xn)$

Where: $P(Y = 1|X)$ is the probability that the target variable Y is 1 (positive class) given the input features X. $\sigma()$ is the sigmoid function. $\beta 0, \beta 1, \beta 2, ..., \beta_n$ are the coefficients (weights) of the model. $X1, X2, ..., Xn$ are the features. In this model, the sigmoid function models the probability that Y is 1 based on the linear combination of the input features and their corresponding weights. If the probability exceeds a certain threshold (e.g., 0.5), the model predicts the positive class (Y = 1); otherwise, it predicts the negative class (Y = 0).

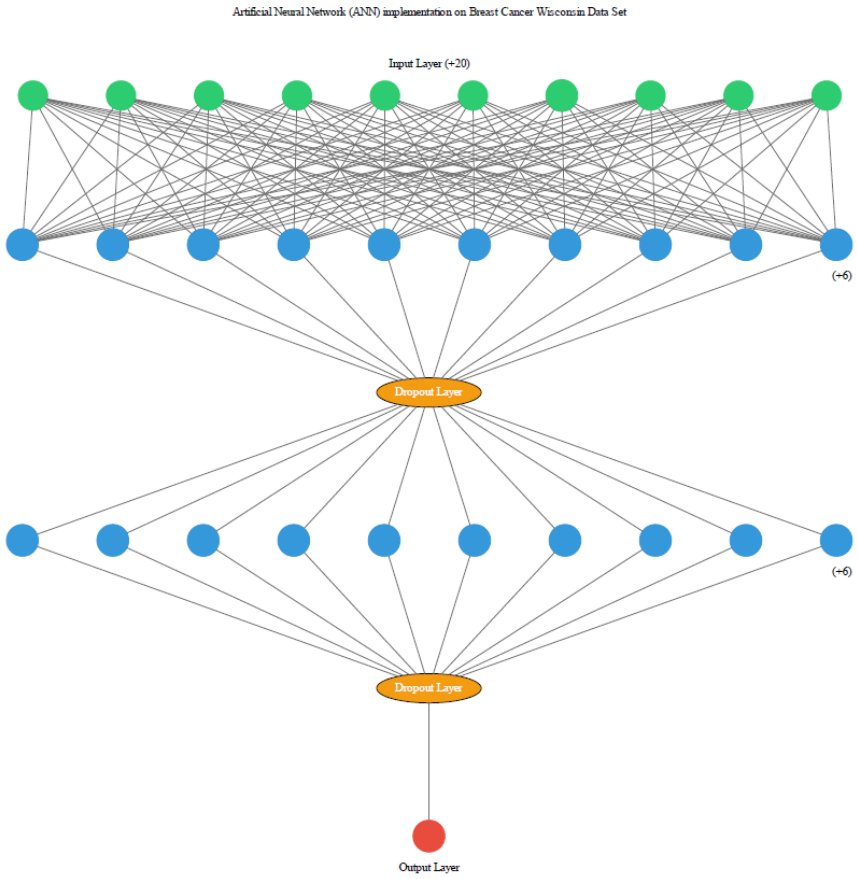

Figure 3: **Neural Network Architecture of the Binary Classification Analysis on the Wisconsin Breast Cancer dataset**
3

2.6 Reproducibility

The importance of the general mathematics and analysis above is in helping determine the applicable methods with huge datasets, containing possibly millions of parameters. It is possible to use the Neural Network code for this particular problem on another dataset, by modifying the $input_dim$, $output_dim$, x, y, $learning_rate$, num_epochs, $training_errors$, $test_data$, $predictions$, $plt.plot()$ and $print$ in the current code. It would also, to generate good results, require the modification of the activation functions, cost functions, schedulers, and network architecture including hidden layers to fit the specific dataset. Assuming these adaptations were done correctly however, it is entirely possible to use the rest of the framework from the Python code. The logistic regression model is more

reproducible, as it uses the same activation function for binary classification problems, the binary cross-entropy loss function, it does not have hidden layers, and the optimization algorithm will normally not have a large effect on the variability. There could be a few changes required depending on the problem, for instance in the case of a multiclass classification problem, you would change the activation function into the Softmax rather than the Sigmoid. The reproducibility can also be further improved by putting random seeds into the code and applying Cross-Validation to assess the reliability of the code.

3 Results

I now ran Python codes with the intention of looking at all the different techniques, modifications and parameters analytically.

3.1 Gradient Descent Results

3.1.1 Without Momentum

I began using Plain Gradient Descent using Fixed Learning rate, using automatic differentiation with JAX (using the chain rule, no tensorflow or pytorch), as a function of epochs and batch size (This is using Ridge regression). Then I Compared to using the analytical gradients. I then used plain gradient descent, analytical gradients, fixed learning rate, with the OLS. Then I used plain gradient descent, automatic differentiation, fixed learning rate, OLS. Using Automatic Differentiation, the MSE converged towards 0 using fewer iterations than with the analytical gradients, even though the MSE initially started at higher levels and R2 started at lower levels with the Automatic Differentiation (at the 1st iteration). With analytical expression for gradients, the larger batch sizes gave results of MSE and R2 very close to 0. With automatic differentiation, the MSE started far off 0 but converged even faster than Analytical Gradient. With higher batch sizes, the MSE and R2 actually converged a bit slower towards 0 than with the lower batch sizes. This was using OLS. When using Ridge, the automatic differentiation contained spikes in the MSE scores for high batch sizes and as the number of epochs grew, the MSE increased and the R2 decreased. This was not the case for the analytical gradients, which had a very good MSE, but not a very good R2. The R2 was stable in the case of analytical gradients. When adding momentum to the plain gradient descent (OLS) with analytical expressions the R2 score was very good, but both the test MSE and test R2 were diverging, so there was numerical instability when a lower number of epochs were used. When using Automatic Differentiation(JAX) the results for MSE and R2 were worse. For Ridge, when using analytical expressions for the gradients the MSE and R2 were similar for all values of λ. With higher regularization strength the numerical instability at lower numbers of epochs increased. When using Automatic Differentiation(JAX) the MSE and R2 performed significantly worse than with analytical expressions for the gradients. For low regularization strength, driving the R2 and MSE towards

a consistent value across epochs happens more slowly. For λ=1, the MSE and R2 actually fared worse, meaning high regularization strength here had a negative effect. Now, looping back to Gradient Descent without momentum, for OLS, I then used Stochastic Gradient Descent with a range of learning rates and automatic differentiation(JAX), and also plotting results as a function of minibatches of the SGD. With low learning rate and low batch size, the MSE took on a range of values for the low amount of iterations. As iterations became larger, the MSE values were more stable. For batch size 50 however, the learning rate had little effect on the results as they were close to the same for learning rates =0.01 and =0.1. Also, for these two learning rates the results were the same also for batch sizes 10 and 20, meaning the learning rate actually having an effect on the results was around 0.001, or somewhere between 0.001 and 0.01, with batch size between 10 and 20 where the learning rate stopped having an effect in this case. Comparing this to the analytical expressions from the Gradients for OLS Stochastic Gradient Descent without momentum, with Automatic Differentiation, the numerical stability was better. With the lowest learning rate, also using Analytical Gradients the results were the same across batch sizes, which was not the case with automatic differentiation. When using Stochastic Gradient Descent without momentum, both for analytical gradients and automatic differentiation, the regularization did not have any effect. When adding momentum, the increased learning rate here has little effect for the large batch size. But for batch sizes 10 and 20, increased learning rate means the numerical instability grows for batch size 10. For both learning rate = 0.001, batch size 10 and learning rate = 0.1, batch size = 20 the MSE initially starts off with a very high value but quickly converges to a value between 2 and 4. R2 exhibits the same behavior.

Note: In the following 4 subsections, all results are using JAX(Automatic Differentiation).

3.1.2 Plain Gradient Descent, Ridge Regression, With and Without Momentum, also using AdaGrad and RMSProp

When using Gradient Descent with Ridge without momentum, when using smaller batch sizes, the MSE and R2 converged to an MSE close to 0 and R2 close to 1 after a low number of epochs. When the batch size was larger, the MSE and R2 started off more distant from the value it eventually converged to, and it took a larger number of epochs to get to a good value. When using AdaGrad, after 1000 iterations the MSE and R2 was still converging towards 0 and 1, respectively, and increasing the regularization strength too much had a negative effect on the MSE and R2 scores, and the effect was quite significant. Using a medium regularization strength, λ=0.01 gave the best MSE and R2 score. When using RMSProp, regularization strength had the same effect, but the MSE converged more quickly than with AdaGrad, meaning took fewer iterations to get to a good score, meaning after around 100 iterations, the MSE and R2 had stabilized. RMSProp also had a little bit of numerical instability at a lower number of iterations, something AdaGrad did not have. Using a

medium regularization strength, λ=0.01 gave the best MSE and R2 scores, and they were slightly better than with AdaGrad, using fewer iterations.

When adding momentum, there was numerical instability for both the least amount of regularization applied and the most amount of regularization applied. The MSE and R2 scores were not good for any values of λ. Using AdaGrad, there was numerical instability for all values of λ, at the initial iterations. But the MSE and R2 had a faster convergence, and the results were very similar to RMSProp. For RMSProp with momentum, the results were exactly equal to RMSProp without momentum, meaning momentum had no effect on RMSProp.

3.1.3 Plain Gradient Descent, OLS, With and Without Momentum, also using AdaGrad and RMSProp

MSE and R2 converged towards 0 quickly within few iterations without momentum. Using AdaGrad, MSE and R2 was still converging after 1000 iterations, and the scores were very bad for the first 1000 iterations. Using RMSProp, the convergence was quicker, like with Ridge, and stabilized after 200-400 iterations. With momentum, there was some numerical instability for lower number of epochs on both the MSE and R2. There was also some numerical instability when using AdaGrad with momentum, but the convergence was very quick. Interestingly, with momentum, AdaGrad and RMSProp here yielded the same results. Gradient Descent without Momentum, and AdaGrad and RMSProp with momentum gave the best MSE and R2 scores here.

3.1.4 Stochastic Gradient Descent, Ridge, With and Without Momentum, AdaGrad, RMSProp and ADAM

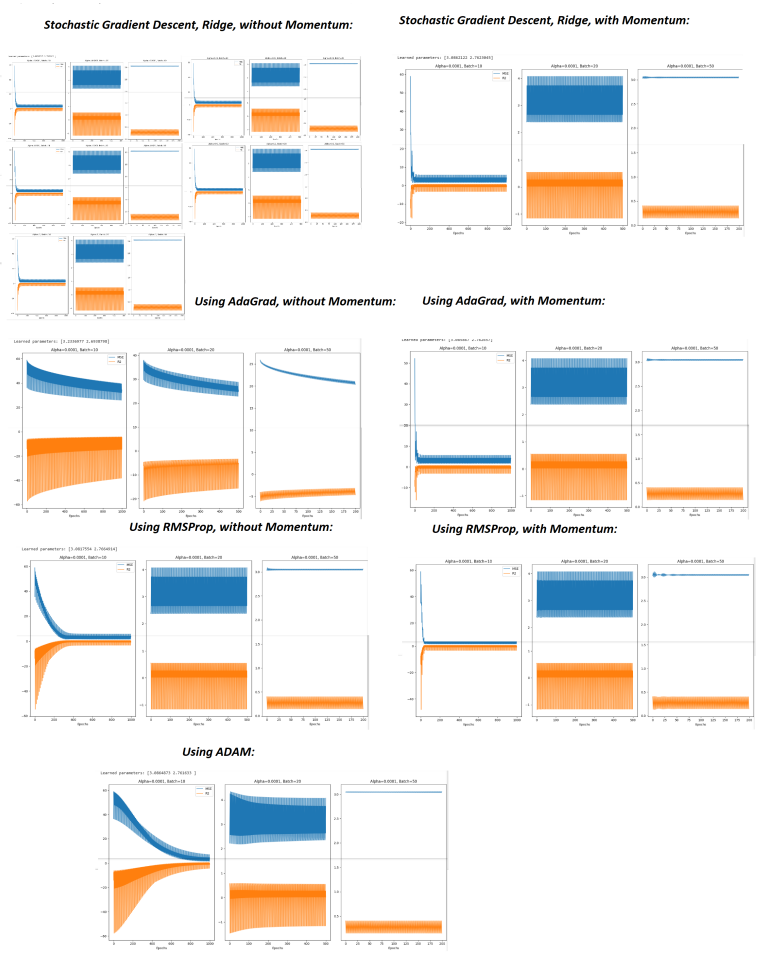

Figure 4: **Stochastic Gradient Descent, Ridge, Automatic Differentiation, also using AdaGrad and RMSProp, without and with Momentum**
4

Stochastic Gradient Descent using Ridge regression without Momentum(Figure 4) quickly converged to a value close to 0, and the regularization had no effect. When using AdaGrad, increasing the batch size increased the numerical stability, but the MSE and R2 scores were far off from the target. When using RMSProp, the numerical stability also increased for larger batch sizes. For

low batch size it took looping over many epochs to drive the MSE and R2 to a desirable size. The ADAM optimizer largely exhibited the same behavior as for RMSProp without momentum, but it took looping over even more epochs in order to get a desirable MSE and R2. The numerical instability was also a bit higher. When adding momentum, the results were similar to without momentum, and adding momentum to AdaGrad had similar results as Stochastic Gradient Descent with and without momentum without using AdaGrad or RMSProp. When adding momentum to RMSProp, the convergence was quicker than with AdaGrad. The Stochastic Gradient Descent with and without momentum as mentioned produced the same result, giving the best MSE and R2 score, and using RMSProp with momentum also achieved the same results.

Here, I am taking into account the learning rate:

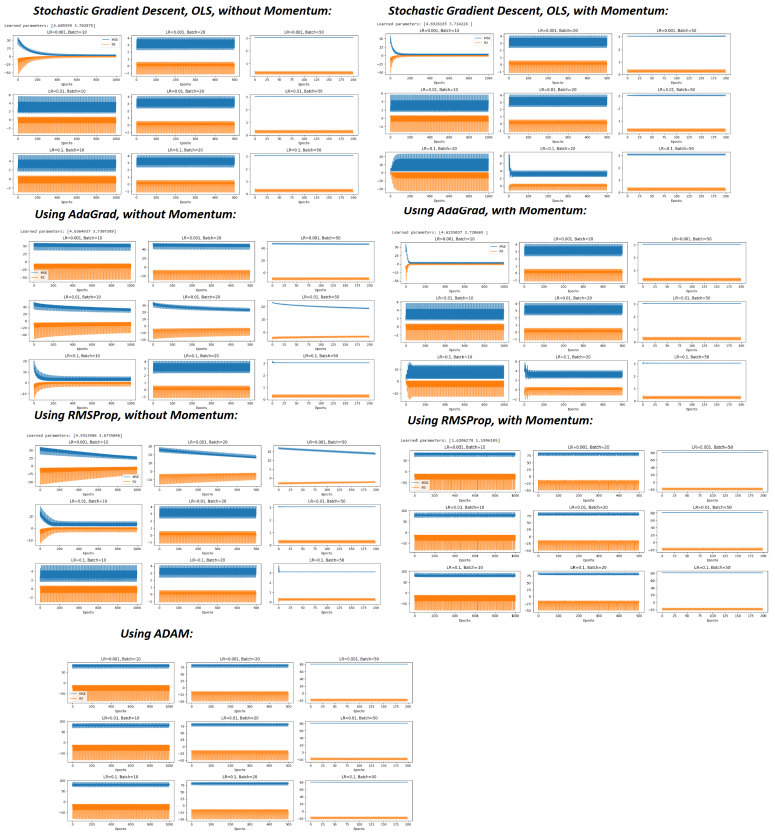

Figure 5: **Stochastic Gradient Descent, OLS, Automatic Differentiation, also using AdaGrad and RMSProp, without and with Momentum**

5

For Stochastic Gradient Descent with OLS(Figure 5) without momentum for the lowest learning rate gave an MSE and R2 beginning far off from the target, with a lot of numerical instability. It took many iterations to stabilize the results. When the learning rate increased, numerical stability increased, and it gave very good MSE and R2 scores after the 1st iteration. In general, the numerical stability increased as the batch size increased, but single instances of better MSE occured for lower batch sizes. When adding momentum, the convergence towards a good MSE score was quicker. Using AdaGrad, for a low learning rate the MSE was very far off. Increasing the learning rate gave a better MSE, and convergence of the MSE to a better score did not use many iterations. Adding momentum to AdaGrad, the convergence towards a good MSE was even quicker than without AdaGrad, with momentum. The numerical instability actually increased when the learning rate increased for AdaGrad. Using RMSProp without momentum the convergence was quicker, and the best MSE score came from RMSProp without Momentum. When adding momentum, the MSE was actually very bad for RMSProp for all learning rates. The R2 score did not change for either AdaGrad or RMSProp. The results here highlights the fact that an appropriate learning rate is crucial to achieving both good scores and numerical stability. This is consistent with the fact that Stochastic Gradient Descent applied to nonconvex loss functions are sensitive to initial parameters, indicating the cost/loss function may be nonconvex. Adding momentum is not necessarily consistent with the results or convergence without momentum and must be managed carefully. Using ADAM, there was no difference between applying different learning rates and the MSE was very bad. The best MSE score came from some values in the range of MSE scores produced using RMSProp without Momentum, for learning rate = 0.1.

3.2 Neural Network Results

I implemented the Neural Network using 3 different schedulers, Step Decay, Exponential Decay and Cyclical Learning Rates. I used 3 optimization algorithms, AdaGrad, RMSProp and ADAM. I used separate Neural Network implementation for the combinations of schedulers and optimization algorithms. I used a different number of hidden layers for all combinations, while preserving the total number of neurons. In one instance, I got a marginally better result by applying a larger number of total neurons. I tried different values for the learning rate, for which several interesting patterns occured. When using ADAM, splitting into more hidden layers produced better results than one or two hidden layers. Using AdaGrad and RMSProp, the opposite was the case. A larger initial learning rate was not always what yielded the best results. When using exponential decay, a larger learning rate gave worse results on some runs. And in other attempts, the relationship in step decay was not straightforward, meaning that sometimes a lower learning rate would give a better result and sometimes the opposite was the case. In this case, adding momentum made the MSE worse in all cases applied. Setting the learning rate very low however always gave results that were significantly worse. I then applied cross-validation and boot-

strapping to the combination of scheduler, optimization algorithm and hidden layer quantity giving me the best result, to further strengthen its interpretative robustness. This combination was using AdaGrad with Exponential Decay, as shown below:

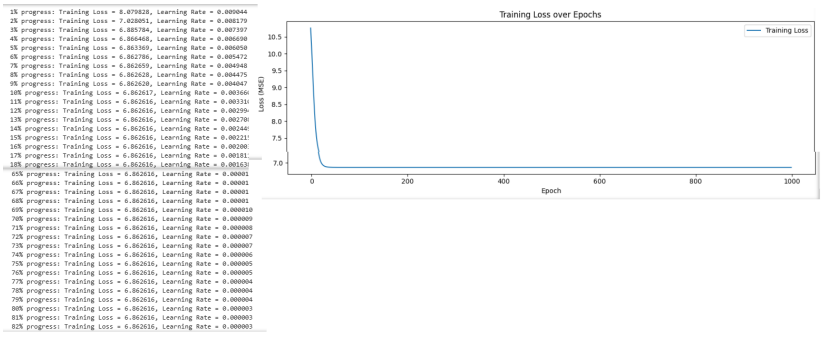

Figure 6: **Neural Network using AdaGrad and Exponential Decay with the Sigmoid Activation Function**
6

The loss function optimization seemed to take place in the early percentages of the progress in the Neural Network, and the initial values of the model's weights and biases play a critical role. Proper weight initialization can speed up convergence. Techniques like He initialization or Xavier initialization, used here, was likely influential to helping the model start closer to an optimal solution. SGD updates the model's parameters using a subset (mini-batch) of the training data at each iteration. Here, the randomness in selecting mini-batches was likely what lead to rapid progress early in training. After applying cross-validation and bootstrapping:

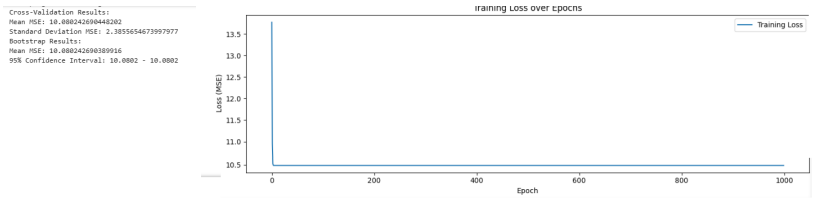

Figure 7: **Neural Network using AdaGrad and Exponential Decay with the Sigmoid Activation Function, Cross-Validation and Bootstrapping**
7

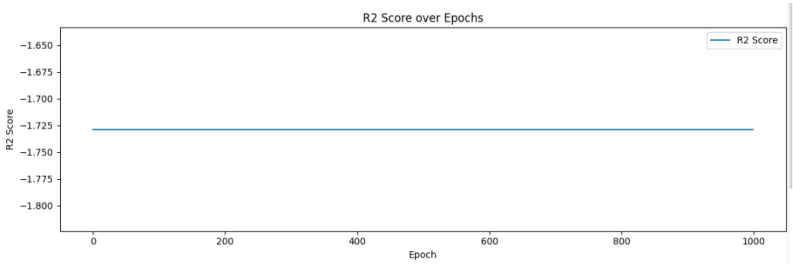

Figure 8: **Neural Network using AdaGrad and Exponential Decay with the Sigmoid Activation Function, R2 Score**
8

The MSE fared worse after applying cross-valiation and bootstrapping, but the results were consistent, and therefore more reliable in the general case. However using the OLS method without a Neural Network gave an MSE score close to 1, significantly better than using the Neural Network. When using Scikit-Learn, the results were quite a lot better. The R2 Score was quite a bit better using Scikit-Learn, and now it was close to 1:

```
Cross-Validation Results:
Mean MSE: 3.945555083114988
Standard Deviation MSE: 1.4979823175232645
Bootstrap Results:
Mean MSE: 3.9628696854298466
95% Confidence Interval: 3.2708 - 4.6996
```

Figure 9: **Neural Network using AdaGrad and Exponential Decay with the Sigmoid Activation Function, Cross-Validation and Bootstrapping, Scikit-Learn's MLP Regressor**
9

```
Cross-Validation Results:
Mean MSE: 4.781676394279167
Standard Deviation MSE: 0.9937887039042625
Mean R2 Score: 0.7254657672966907
Standard Deviation R2 Score: 0.05400327514609671
Bootstrap Results:
Mean MSE: 4.830061154138733
95% Confidence Interval MSE: 4.0046 - 5.6416
```

Figure 10: **R2-Score using Scikit-Learn**

10

3.3 Using different Activation Functions

A number of different combinations with and without momentum were tested using RELU as the Activation Function rather than the sigmoid function, and for the specific combination ADAM with Momentum and Exponential Decay, I got a very good MSE and R2-score, which was also much better than the other combinations. The results follow:

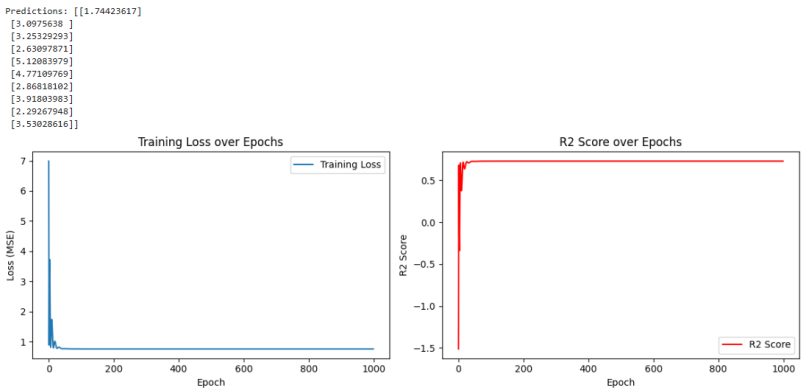

Figure 11: **RELU ADAM with Exponential Decay**

11

I then applied Cross-Validation and Bootstrapping to these results to check the stability and reliability, where only 1 of the 5 folds yielded good results, and the other 4 gave very bad results, indicating that the reliability in the general case is not good:

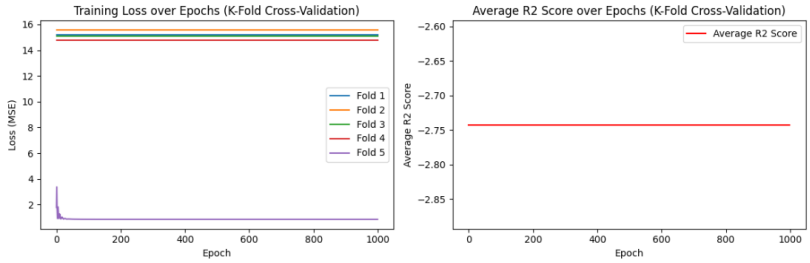

Figure 12: **RELU ADAM with Exponential Decay, Cross-Validation and Bootstrapping**
12

LeakyRELU was then applied to
the same codes as with RELU, and LeakyRELU generated very good MSE scores, sometimes after close to 100 percent progress. Sometimes it happened earlier in the progress. Sometimes the MSE scores were cyclical, meaning they would go up a lot and then down a lot in percentages of progress, before the pattern would repeat itself.
Notably with many different combinations here rather than with just one selected combination as in the case of RELU.
Below is a figure showing one of the results:

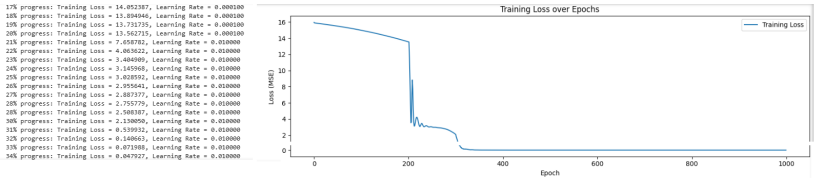

Figure 13: **LeakyRELU ADAM with Cyclical Learning Rates and Momentum**
13

Now, the same code was cross-validated:

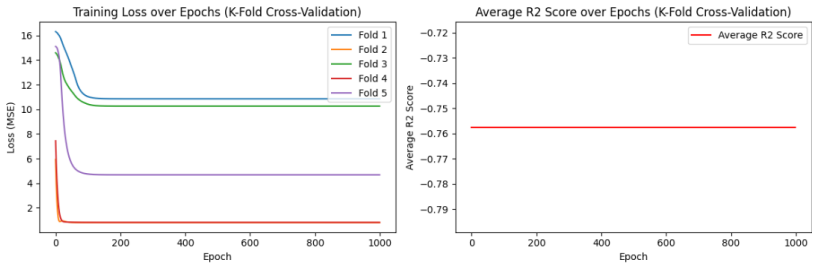

Figure 14: **LeakyRELU, ADAM, Cyclical Learning Rates, Momentum, Cross-Validated**
14

Only a few of the folds generated the same results, indicating again a less than satisfactory reliability. I then changed the way the weights and biases were initialized, switching the He-initialization with the Glorot initialization, assumed to be suitable for networks with different layer sizes and to help improve convergence during training. The best result I got using He-initialization was the above, using the LeakyRELU activation function, ADAM, Cyclical Learning Rates and Momentum, eventually achieving an MSE=0.02. When using Glorot-initialization I was able to achieve an MSE=0.01, thus yielding the best MSE result overall.

3.4 Classification Analysis Using Neural Networks

The confusion matrix with the best accuracy using the Cross-Entropy Cost Function and the sigmoid activation function, using the Wisconsin Breast Cancer Dataset(Using Analytical Expressions for the Gradients) on the left, and then adding ADAM and He/Glorot initialization on the right gave me an accuracy score of 0.96 and the following confusion matrix:

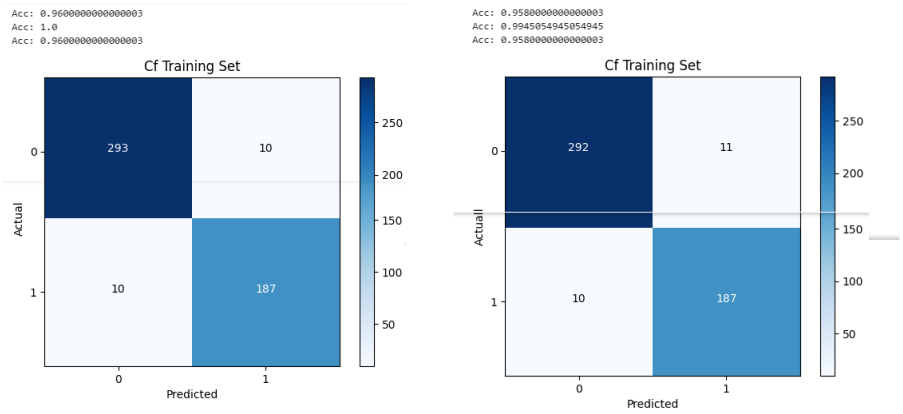

Figure 15: **Cross-Entropy cost function, Sigmoid activation function, analytical expressions for the gradients on Binary Classification Problem on the left, adding ADAM and He/Glorot Initialization on the right**
15

Adding L1 and L2 regularization had very little effect on the results in this case, and the same was true for ADAM and He/Glorot Initialization. The results were close to the results gained using MLPRegressor with Scikit-Learn(16), but still there was 4 percent accuracy difference when using Scikit-Learn, which would make a big difference in the case with a very large dataset. I then added Dropout, but it had no effect.

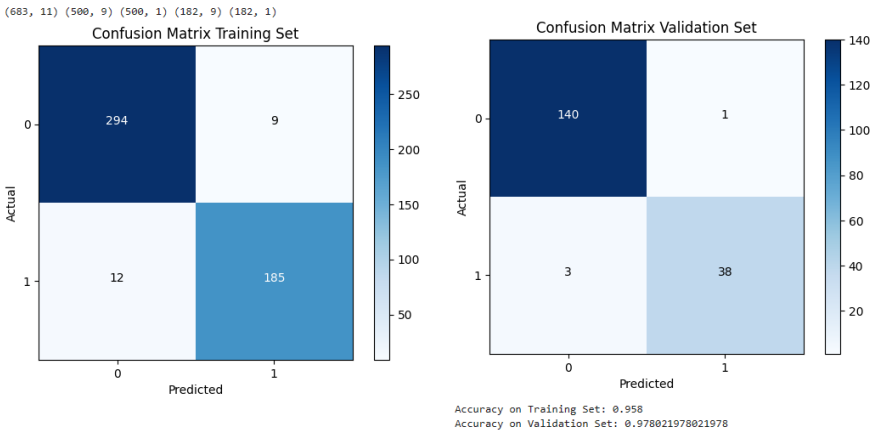

Figure 16: **Cross-Entropy cost function, Sigmoid activation function, analytical expressions for the gradients on Binary Classification Problem using Scikit-Learns MLPRegressor.**
16

3.5 Logistic Regression

Logistic Regression was implemented using the Cross-Entropy Cost function and the Stochastic Gradient Descent algorithm from earlier, using also L2 regularization, and the fixed learning rate of 0.01, gave an accuracy score around 4 percent worse than Scikit-Learn. The size of the L2 regularization term did not have any effect on the results here, neither the Accuracy Score or the MSE and R2.

Compared to using Scikit-Learn:

```
Training Accuracy: 0.958
Validation Accuracy: 1.0
Training Confusion Matrix:
[[294   9]
 [ 12 185]]
Validation Confusion Matrix:
[[141   0]
 [  0  41]]
```

Figure 17: **Cross-Entropy cost function, Sigmoid activation function, using JAX, Logistic Regression with Stochastic Gradient Descent on the Binary Classification Problem.**
17

Using He/Glorot initialization and the Hinge Loss function, while modifying the L2 regularization term like before, the changes were barely noticable, but there were some small changes. It would be easy to conclude that the L2 regularization did not affect the results in this case, but the effect was visible when deploying Cross-Validation on a range of different learning rates and L2 regularization values. Modifying the learning rate had an effect; at learning rate 0.001 the Accuracy Score on the Training set was 0.964, and at learning rate 1 the score was 0.97. Without employing cross-validation, increasing the learning rate had a bigger effect than modifying the L2 regularization term, but this relationship turned out to be false when assessing the reliability, as when employing Cross-Validation increasing the learning rate and decreasing the L2 actually gave a worse score of 0.954. I did not test ADAM, RMSProp or AdaGrad here, as these are not expected to have any influence in this logistic regression problem. Seeing as though the Logistic Regression gives me as good a result as the Neural Network decoding the classification problem, the Logistic Regression is desirable in this case as it requires less computational power.

4 Conclusions

The aim is to strike a balance between what is a good model and what one is willing to spend when using that model. As discussed for the Gradient Descent and SGD the best results for SGD in Ridge Regression were achieved using The Stochastic Gradient Descent with and without momentum and using RMSProp with momentum. For Plain Gradient Descent with Ridge regression, Using a medium regularization strength, λ=0.01 gave the best MSE and R2 scores, and they were slightly better than with AdaGrad, using fewer iterations. I saw that adding momentum came at the cost of large numerical instability across the board for this mehtod. For Gradient Descent, using OLS, Gradient Descent without Momentum, AdaGrad and RMSProp with momentum gave the best MSE and R2 scores here. I also saw that OLS doesnt have any effect with Stochastic Gradient Descent nor Momentum, because it has a closed-form solution. For Ridge regression, using Stochastic Gradient Descent may introduce more variability in the Ridge regression due to the stochasticity. The best MSE score for Stochastic Gradient Descent using Ridge came from RMSProp without Momentum. When adding momentum, the MSE was quite bad for RMSProp and ADAM for all learning rates. The results here highlights the fact that an appropriate learning rate is crucial to achieving both good scores and numerical stability, and is consistent with the fact that Stochastic Gradient Descent applied to nonconvex loss functions are sensitive to initial parameters, linking together a part I wrote in the Mathematical Assumptions and the actual Results. The results behaved the way the mathematical assumptions would suggest. The best MSE score came from some values in the range of MSE scores produced using RMSProp without Momentum, for learning rate = 0.1. For the Neural Network, The best result I got using He-initialization was using the LeakyRELU activation function, ADAM, Cyclical Learning Rates and Momentum, eventually achieving an MSE=0.02. When using Glorot-initialization I was able to achieve an MSE=0.01, thus yielding the best MSE result overall. The fact that Glorot initialization resulted in a lower MSE suggests the neural network was able to generalize better here. It was likely able to prevent the vanishing gradient problem and learn more effectively and fit the underlying polynomial of order 5. Randomly tuning parameters and choosing methods do not change the results to a large extent, however choosing the right combinations of algorithms, methods and parameters will, as evidenced in this case by the combination of Glorot initialization on the Sigmoid activation function that allowed the network to fit the data more effectively due to better weight initialization. A major point here is that the different activation function gave huge impacts on the result, meaning I was able to get a score that was much better when switching the activation function. In the Classification Analysis of the Binary classification problem, using the Cross-Entropy Cost Function and the sigmoid activation function gave the best result, whereas a different initialization of the weights and biases did not majorly impact the results, which was also something I expected from the Mathematical Assumptions. For the Logistic Regression, I was able to achieve the same results as when using the Neural

Network, but at a lower computational cost, and thus Logistic Regression is more ideal as it requires amongst other things fewer iterations.

4.1 Appendix

The Python code used in this project, is found at https://github.com/ingebrigtkj/Project-2-FYS-STK4155

References

[1] https://jax.readthedocs.io/en/latest/notebooks/quickstart.html JAX Automatic Differentiation Method

[2] https://www.kaggle.com/datasets/uciml/breast-cancer-wisconsin-data Dataset for the Binary Classification problem analyzed in parts Classification Analysis and Logistic Regression

[3] Chapter 6,8 and 11 of Goodfellow, Bengio and Courville: https://www.deeplearningbook.org/contents/guidelines.html

[4] Videos on Neural Networks https://www.youtube.com/watch?v=bxe2T-V8XRs&list=PLiaHhY2iBX9hdHaRr6b7XevZtgZRa1PoU, https://www.youtube.com/watch?v=CqOfi41LfDw, https://www.youtube.com/watch?v=Ilg3gGewQ5U

[5] Interactive approach to weights and biases in the Neural Networks: http://neuralnetworksanddeeplearning.com/chap4.html