# Project 2: Classification and Regression, from Linear and Logistic Regression to Neural Networks

Owen Huff and Ilya Berezin

November 16, 2022

FYS-STK 4155 - Applied Data Analysis and Machine Learning

University of Oslo

## Abstract

In this project we present a practicum of classification and regression using feed forward neural networks, with comparison to traditional methods such as logistic regression. We start with exploring the behavior of several optimization methods, which we use to minimize the error in reconstructing a simple quadratic function. These methods include gradient descent and stochastic gradient descent (with and without momentum), and learning rate tuning methods such as AdaGrad, RMSprop, and Adam. In general we find that stochastic gradient descent with momentum outperforms gradient descent in terms of convergence speed, while achieving similarly low error, and also find that the Adam algorithm allows faster convergence than AdaGrad and RMSprop. We then develop a feed forward neural network code from scratch, using stochastic gradient descent as the optimizer, and use it to perform regression and classification tasks. For the regression task, we again perform reconstruction of data from the simple quadratic function, and achieve a lowest mean squared error of 0.0051 by using a single-hidden-layer network with 50 nodes and ReLU activation. To verify this result, we train a neural network of the same architecture on the same data in Tensorflow, and achieve a slightly higher error of 0.009. We then adapt our feed forward neural network to handle a classification task - the classification of tumors as malignant or benign from the Wisconsin Breast Cancer Dataset. We develop a network with a single hidden layer of 20 nodes and sigmoid activation, and find that it achieves an accuracy of 0.965 on the test dataset. To validate this result, we build a logistic regression model and a similar neural network in Tensorflow, and find that they achieve accuracies of 0.965 and 0.956, respectively. Our results demonstrate that overall, neural networks can be implemented to achieve similar results to traditional methods on simple datasets.

## Reproducibility

The code used for this report can be accessed at:
https://github.com/orhuff/FYS-STK-4155-Project2-Neural-Networks

## Introduction

Supervised machine learning problems can broadly be divided into the domains of regression and classification. Regression involves developing models that can predict continuous function values, such as a salary, velocity, or temperature. On the other hand, classification involves developing models that can predict labels (or classifications) of data, for instance whether a tumor is malignant or benign, or whether an image contains a cat or a dog.

The focus of Project 1 was on simple regression methods: ordinary least squares, ridge, and lasso regression. These methods demonstrate great performance and explainability on problems with a lower degree of complexity, namely problems with (at least roughly) linear dependence between the input and output. The analogue to these simple regression methods in the domain of classification is logistic regression. Logistic regression models can be trained to output the probability that a datapoint belongs to a given class, and works best when the data's classes are easily delineated.

However, many machine learning problems have a higher degree of complexity, due to nonlinear relationships between variables and/or nonlinear separations between data classes. This motivates the need for more sophisticated models that can handle nonlinearities, such as neural networks. Neural networks are based on the perceptron model of how neurons work in the brain, where a neuron's activation is dependent on input features and weights (Rosenblatt, 1958). By combining multiple neurons in multiple layers, in what are called "deep" neural networks, increasingly complex problems can be solved. The development and use of neural networks actually stagnated for many decades until the early 2010s, when computational power had increased enough for them to be implementable. Now there is a revolution in which neural networks are used commonly and to great success on a variety of problems. While the advantage and power of neural networks is most apparent when moving to highly complex data, they can sometimes outperform traditional methods on simple data as well.

Part of the beauty and effectiveness of neural networks is how they can be applied to both regression and classification problems. This is done by 1) changing the cost function used in its training, which can then be optimized with a variety of methods such as gradient descent. And 2) by changing the output layer of neurons, to either output a continuous value (for regression), or a set of probabilities for the various labels (for classification). Furthermore, neural networks can utilize different activation functions to determine how easily and in what manner neurons will activate.

In this project, we develop our own neural network code to perform both regression and classification tasks. Our goal is to show that neural networks can do as well as, and perhaps outperform, traditional regression and classification methods on simple datasets. We start with explaining the theory behind gradient descent, gradient descent with momentum, stochastic gradient descent, and learning rate tuning methods that are used in the optimization of training cost functions. Then we explain the theory of logistic regression and neural networks. In our tests, we first compare the performance of the various optimization methods for regression on a simple quadratic function. We then demonstrate our neural network for

regression, again on the quadratic function, but also with comparison to a neural network developed in Tensorflow. Finally, we demonstrate our neural network applied towards a classification task - the classification of tumors as malignant or benign from the Wisconsin Breast Cancer Dataset - and compare its performance to logistic regression, and a classification neural network built in Tensorflow.

# Theory

## Optimization Methods

In supervised machine learning problems, one of the most important concepts is optimizing - or finding the minimum of - a cost function that measures how well your model is performing. There exist a number of methods to iteratively solve for the minima of cost functions, as well as algorithms to tune the "learning rate" at which these methods progress. We will explore those here.

*Gradient Descent*

Gradient descent is an optimization method that is used to iteratively solve for minima or maxima in differentiable functions. It is used commonly in machine learning in order to solve for the minima of a cost function that measures the performance of a model in training. Gradient descent iteratively solves for points $x$ that approach the minima by going in the direction of steepest descent, which is the direction of the negative gradient (Boyd and Vandenberghe, 2004).

$$x^{(k+1)} = x^{(k)} - (J(f(x^{(k)}))^{-1} f(x^{(k)}) \tag{1}$$

In the above equation $x^{(k)}$ is an initial guess for the $x$ point corresponding to the minima, $J$ is the Jacobian matrix (which is a matrix of first-order partial derivatives, where the ith row is the gradient of the ith component of the function $f(x)$, $f(x^{(k)}$ is the function we want to minimize evaluated at $x^{(k)}$, and $x^{(k+1)}$ is the updated value of $x$ which is closer to the minima point. This process then repeats itself in the next iteration with the old value of $x^{(k+1)}$ set to $x^{(k)}$, and so on until the minima is reached. This equation can be written in terms of the Hessian matrix $H$ (the matrix of second order partial derivatives) and the gradient of the function f:

$$x^{(k+1)} = x^{(k)} - (H(x^{(k)}))^{-1} \nabla f(x^{(k)}) \tag{2}$$

Because calculating the Hessian is often very computationally expensive, one of the most common methods is to set it to a scalar value , which is referred to as the learning rate. This learning rate can also be thought of as the step size of this iterative solver.

$$x^{(k+1)} = x^{(k)} - \eta \nabla f(x^{(k)}) \tag{3}$$

If the learning rate is too large, the gradient descent method might miss the local functionâs extrema. On the other hand, if the learning rate is too small, it might take a lot of time before it reaches the extrema, and thus be computationally inefficient. One issue often arises in the case of complex functions with several extrema - the solver can get stuck in a local extrema instead of reaching the global minima (which is the intention). Thus gradient descent is very sensitive to the type of function it is applied on, with convex functions being preferrable, and is also sensitive to the setting of its learning rate.

*Gradient Descent with Momentum*

One improvement to gradient descent is including a momentum parameter. This parameter is analogous to momentum or inertia in an equation of motion, which represents some memory of the speed and direction being travelled in. In gradient descent, this can be thought of as memory of the direction and magnitude one is travelling in parameter space in order to find the function's minima. The implementation in equation form looks like:

$$v^{(k+1)} = \gamma v^{(k)} - \eta \nabla_\theta f(\theta_k) \tag{4}$$

where $\theta_{k+1} = \theta_k - v_k$, and $\gamma$ is the momentum parameter which should take values between 0 and 1. The updates $v^{(k+1)}$ can then be interpreted as a running average of the gradients.

*Stochastic Gradient Descent*

Stochastic gradient descent (or SGD) is another improvement to gradient descent that works by randomly sampling smaller portions of the data. The process involves first dividing the dataset randomly into portions called "minibatches", and then calculating the gradients and updating for each minibatch (Goodfellow et al., 2016). After completing this process for every minibatch, the entire dataset has been seen, and it is said that one "epoch" or "iteration" has been completed. The relation between the number of datapoints $n$ and the minibatch size $M$ is such that there are $n/M$ minibatches. If we define a cost function that we want to minimize, $C(\beta)$, then the iterative gradient descent equation then becomes the following for stochastic gradient descent:

$$\beta^{(k+1)} = \beta^{(k)} - \eta_k \sum_{i \varepsilon B_k}^{n} \nabla_\beta c_i(x_i, \beta) \tag{5}$$

where $B_k$ is the kth minibatch of the data. The main advantage of stochastic gradient descent is how it reduces the computational demands of calculating gradients, as it does this on smaller minibatches of the data. This leads to a lower computational time for each iteration compared to regular gradient descent. While estimations of the gradients may not always be as accurate

as for regular gradient descent (which is looking at the whole data), the methods are usually very close in their results, and the much faster convergence makes SGD often preferred.

*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.

One of the most simple methods is with an epoch-dependent tuning according to two parameters:

$$\eta(t) = \frac{t_0}{t_1 + t} \tag{6}$$

where $t$ is the current epoch, and $t_0$ and $t_1$ are parameters to set the initial learning rate. For example if $t_0$ is 5 and $t_1$ 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.

However, several more sophisticated tuning methods exist. One of these methods is AdaGrad, which means Adaptive Gradient Algorithm. This algorithm works by scaling the learning rate of the different model parameters by the root mean squared of all the previous gradient values (Goodfellow et al., 2016). In this way, parameters that have larger gradients experience a larger decrease in their learning rates, and parameters with smaller gradients experience a smaller decrease in their learning rates. This leads to an adaptive shift of the learning rate in order to make more progress in the parameter directions that have lower gradients. The update rule for AdaGrad is given by the equation, where $g$ represents the gradient, and $delta$ is a very small value (usually on the order of 1e-8) used to avoid potential division by 0:

$$x_i^{(k+1)} = x_i^{(k)} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{k} g_{\tau,i}^2 + \delta}} g_{k,i} \tag{7}$$

The next learning rate tuning method is RMSprop, which means Root Mean Squared Propagation. RMSprop is an extension of AdaGrad which instead of using the regular RMS average of the gradients, uses a decaying moving average of the gradients (Goodfellow et al., 2016; Hinton, 2012). The potential advantage of RMSprop in comparison to AdaGrad is that by using the moving average, it can focus less on the early values of the gradient, and instead focus more on more recent values of the gradient that are more relevant to the current status of the search for the minima. The update equation is as follows:

$$x_i^{(k+1)} = x_i^{(k)} - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_k + \delta}} g_k \tag{8}$$

Where $\mathbb{E}[g^2]_k$ is a moving average of the expectation values of the squared gradients, given by:

$$\mathbb{E}[g^2]_k = \gamma \mathbb{E}[g^2]_{k-1} + (1-\gamma) g_k^2 \tag{9}$$

A good value for $\gamma$ is typically on the order of 0.9 (Hinton, 2012).

The final learning rate tuning algorithm that we will implement is called Adam, whose name derives from adaptive moment estimation. The key concept of the Adam algorithm is that it keeps track of a running average of the squared gradients, as well as a running average of the unsquared gradients (Kingma and Ba, 2014; Goodfellow et al., 2016). In this way it is like a combination of RMSprop and momentum gradient descent. The running average of the gradients is referred to as the first moment $m$, and the running average of the squared gradients is referred to as the second moment $v$, which are given iteratively by:

$$m_{k+1} = \beta_1 m_k + (1-\beta_1) g_k \tag{10}$$

$$v_{k+1} = \beta_2 v_k + (1-\beta_2) g_k^2 \tag{11}$$

However, these moments tend to be stuck around zero when initialized as such, and thus need to be corrected with bias terms:

$$\hat{m_k} = \frac{m_k}{1-\beta_1^k} \tag{12}$$

$$\hat{v_k} = \frac{v_k}{1-\beta_2^k} \tag{13}$$

Putting this all together in terms of the bias-corrected moments, the update rule for Adam becomes:

$$x_{k+1} = x_k - \frac{\eta}{\sqrt{\hat{v}_k} + \delta} \hat{m}_k \tag{14}$$

### Linear and Ridge Regression

The focus of Project 1 was on linear and ridge regression, so we refer the reader to that project for more details. However, the main detail that is important to note is that in our testing of regression methods, we will use the mean squared error as the cost function and calculate its gradient as the following for linear regression:

$$g = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\beta - \mathbf{y}) \tag{15}$$

and as the following for ridge regression, which includes an L2 regularization parameter $\lambda$:

$$g = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\beta - \mathbf{y}) + 2\lambda\beta \tag{16}$$

In these equations, $\mathbf{X}$ is the input data, $\mathbf{y}$ is the target data, and $n$ is the number of datapoints.

### Feed Forward Neural Networks

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. This process can be visualized and mathematically represented as seen in Figure 1:

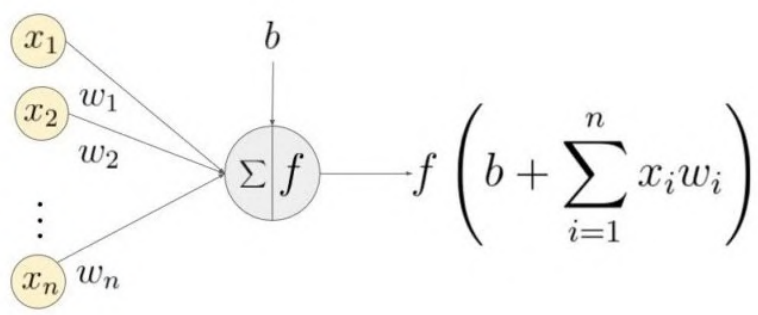


**Figure 1:** Diagram showing the process of input, weighting, and activation occurring at a neural network node. From Sharma, 2017.

The inputs are the values $x_i$, and the weights $w_i$. Within the node, the sum of the bias and the weighted sum of the inputs, $b + \sum_{i=1}^{n} x_i w_i$, is evaluated with what is called an activation function $f$. There exist several activation functions that can be used - we will implement three simple ones in this project: the sigmoid, ReLU, and leaky relu.

The sigmoid function is designed so as to take an input value, and yield an output between 0 and 1. It is defined by the following equation:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{17}$$

The sigmoid function is also known as the logistic activation function, because it is used in logistic regression as we will explore later. It is especially appropriate for classification problems, because it yields values between 0 and 1, which can represent probabilities that a datapoint belongs to a given class. However, one of the disadvantages of sigmoid activation functions is that they can lead to so-called vanishing gradients (Mehta et al., 2019). This occurs when a large change in the input to a sigmoid function produces a small change in the output (because the output is squashed from 0 to 1), yielding a very small derivative or vanishing gradient. For this reason, the hyperbolic tangent function is commonly used in place of the sigmoid, as it has a similar curve but with larger derivatives (though we do not implement tanh in this project for the sake of brevity).

The next activation function we will implement is the ReLU, or Rectified Linear Unit. The ReLU is a piecewise function defined as:

$$\sigma(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{18}$$

The main advantages of ReLU over sigmoid are twofold. 1) It is less computationally expensive because it simply involves an if statement instead of the calculation of an exponential as in the sigmoid. And 2) it does not as easily experience the problem of vanishing gradients because its derivative is usually 1. These advantages make ReLU more commonly used, and able to be implemented in deep neural networks (Mehta et al., 2019).

A modified version of the ReLU that further remediates the problem of vanishing gradients is called the Leaky ReLU. This function takes the portion of the ReLU function less than 0, and multiplies it by a (small, usually about 0.01) constant so that the derivative is not zero. It is defined as such:

$$\sigma(z) = \begin{cases} z & \text{if } z \geq 0 \\ cz & \text{if } z < 0 \end{cases} \tag{19}$$

With this understanding of a node and its activation developed, we can explain the overall construction and training of a feed forward neural network. The key aspect of a neural network is that the weights and biases at each node are trainable - they are adjusted during the learning process until a given cost function is minimized (Nielsen, 2015). In general, the more complex a problem is, the more nodes and layers of nodes are required in order to yield a good result.

The simplest FFNN consists of an input layer and an output layer. The input layer consists of a set of nodes, each of which is connected to one of the input values. The output layer consists

of a set of output nodes, each of which is connected to one of the output values. In between the input and output layers, there may be one or more hidden layers. A hidden layer is simply a layer of nodes that is not directly connected to the input or output. Hidden layers allow the network to learn more complex patterns. The nodes in each layer are fully connected to the nodes in the adjacent layer. That is, each node in the input layer is connected to every node in the hidden layer (if any), and each node in the hidden layer is connected to every node in the output layer.

The training process for an FFNN involves presenting the network with a set of training data. The training data consists of a set of input values and the corresponding output values. The network adjusts the weights of the nodes based on the error between the actual output values and the predicted output values. The error is calculated using a cost function that measures how well the network is performing. After the weights have been updated, the training data is presented to the network again, and the process is repeated. This process is continued until the error is minimized. Once the training process is complete, the network can be used to predict the output for unseen testing data.

In the training of a neural network, the cost function can be optimized with a method such as SGD. However, the calculation of the gradients of the cost function requires a more sophisticated method, because simple calculation of all the gradients with respect to all the weights and biases in the network would be far too computationally complex. This is where the backpropagation algorithm comes in.

The backpropagation algortihm is a feedback-based method, meaning that it uses feedback from the network itself to adjust the weights of the connections between the neurons. The method propagates the error in the output of a neural network back through the network, and the weights of the connections being adjusted accordingly (Nielsen, 2015). The backpropagation algorithm is composed of two parts: the forward propagation phase and the backward propagation phase. In the forward phase, the inputs to the network are fed forward through the network, and the output of the network is computed. In the backward phase, the error in the output is propagated back through the network, and the weights of the connections are adjusted accordingly. The backpropagation algorithm is able to learn the weights of the connections between the neurons in a neural network by iteratively adjusting the weights in accordance with the error in the output of the network. The backpropagation algorithm is highly efficient, and allows the gradients to be calculated in a reasonable amount of time.

Mathematically, the backpropagation algorithm can be described in four steps (Nielsen, 2015):

1) Input: Set the corresponding activation $a^1$ for the input layer.

2) Feedforward: For each layer $l$, compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$

3) Output Error: Compute the output error $\delta^{(L)} = \nabla_a C \odot \sigma'(z^{(L)})$, where $\odot$ is the hammond product

4) Backpropagate the Error: For each layer $l$, compute $\delta^{(l)} = ((w^{l+1})^T \delta^{(l+1)}) \odot \sigma'(z^{(l)})$

5) Output: Return the gradient of the cost function with respect to the weights as $\frac{\partial C}{\partial w_{jk}^{(l)}} = a_k^{(l-1)}\delta_j^{(l)}$, and with respect to the biases as $\frac{\partial C}{\partial b_j^{(l)}} = \delta_j^{(l)}$

By using an optimization method like SGD in combination with gradient calculation via backpropagation, a neural network can be trained to perform a variety of tasks. Two main tasks we will describe in this project are classification and regression, which a neural network can be designed for by changing its cost function and input/output layers. In the case of regression tasks, the cost function used is usually that of the Mean Squared Error (MSE) or the Mean Absolute Error (MAE), which we have described in Project 1 on regression. In the case of classification, the cost function is usually that of cross-entropy, which we will describe in the next section on logistic regression.

**Logistic Regression**

Logistic regression is a type of model that can be trained to predict classifications of data. It heavily relies on the sigmoid function described earlier, and is essentially the most simple version of a neural network with only one layer. Consider that we have an input observation $x$ with $n$ features: $x = [x_1, x_2, ..., x_n]$. In this project we will consider the case of binary classification, so we want to predict whether the observation belongs to class 0 or class 1. The output of the classifier, which we will call $y$, can then be 0 or 1. Our goal in logistic regression is to have this classifier output the probability $P(y = 1|x)$ that the observation is part of class 1 (Jurafsky and Martin, 2021).

The sigmoid function is the perfect choice for this task, as it outputs a value between 0 and 1. Thus we can input the value $x$ into the sigmoid function with trainable weights $w$ and biases $b$, similar to a neural network approach:

$$P(y = 1) = \sigma(wx + b) = \frac{1}{1 + e^{-(wx+b)}} \tag{20}$$

We then need to set a threshold for the output in order to determine how the sigmoid will make the classification. Traditionally this value is set at 0.5, with values at or above 0.5 corresponding to a majority-probability that the data belongs to class 1. Thus we convert the output from the sigmoid using the following rule:

$$Decision(x) = \begin{cases} 1 & \text{if } P(y = 1|x) \geq 0.5 \\ 0 & \text{if } P(y = 1|x) < 0.5 \end{cases} \tag{21}$$

In order to train a logistic regression model, we need a cost function that represents the difference between the correct output $y$ (which is 0 or 1) and the sigmoid output $\hat{y} = \sigma(wx+b)$. This cost function can be derived by first noting that the probability of a correct label given

$x$, $p(y|x)$ follows a Bernoulli distribution because there are only two outcomes (Jurafsky and Martin, 2021). It is this quantity that we desire to maximize:

$$P(y|x) = \hat{y}^y(1-\hat{y})^{1-y} \tag{22}$$

By taking the logarithm of both sides, and then taking the negative of the quantity (because we want a cost function to minimize), we obtain our cost function, which is called the cross-entropy loss:

$$C = -P(y|x) = -(y\ log(\hat{y}) + (1-y)\ log(1-\hat{y})) \tag{23}$$

From here, optimization methods like SGD can be applied to train the weights and biases of the model until the cost function is minimized. The gradient of the cross entropy cost function can be calculated as:

$$g = \frac{1}{n}(\mathbf{X}^T(\sigma(\mathbf{X}\mathbf{w}) - \mathbf{y}) - 2\lambda\mathbf{w}) \tag{24}$$

In this equation, $\mathbf{X}$ is the input data, $\mathbf{y}$ is the target data, $w$ are the weights, $\lambda$ is an optional L2 regularization parameter, $n$ is the number of datapoints, and $\sigma$ is the sigmoid activation function.

While it is beyond the scope of this report, it is important to note that there also exist modifications to logistic regression that can be made to perform multi-class (3+ classes) classification. The main modification is changing the activation function from a sigmoid to softmax, which can output probabilities corresponding to multiple classes. Models designed as such are commonly called multinomial logistic regression models (Jurafsky and Martin, 2021).

## Implementation Details

### *Data*

In our regression tests using gradient descent, OLS, and a feed forward neural network, we use a simple quadratic function of the form:

$$y = 4 + 3x + 2x^2 \tag{25}$$

We sample 100 random points in the range [0,3] in our implementation, and use 80 for the training dataset and 20 for the testing dataset.

In our classification tests using a feed forward neural network and logistic regression, we use the Wisconsin Breast Cancer Dataset. This dataset is openly available through the scikit-learn Python package, in sklearn.datasets. This data has 569 datapoints and 30 features, with the features consisting of observations like tumor radius, perimeter, and area. The diagnosis column is the output classification, and consists of 212 malignant tumors and 357 benign tumors. We apply a standard scalar (subtraction of the mean and division by the standard deviation) to the data in order to compensate for the effect of radically different feature units and values (for instance radius vs concavity). There is no missing data or large outliers, and thus no interpolation or data editing other than the scaling was applied.

*Testing Metrics*

In our regression tests we use a metric of mean squared error (MSE) in order to measure the error between our predicted values and the true values. In classification for our testing metric, we use the accuracy, which is defined as (James et al., 2013):

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{26}$$

where TP is the number of true positives (correct predictions of malignant), FP is the number of false positives (incorrect predictions of malignant), TN is the number of true negatives (correct predictions of benign), and FN is the number of false negatives (incorrect predictions of benign). It is the number of correct predictions over the number of incorrect predictions. While there exist more sophisticated metrics like the F1-score, that are used commonly on imbalanced and/or multi-class datasets, we have a very simple binary classification dataset that is relatively balanced. Therefore we use only the accuracy.

# Results

## Optimization Methods

*Gradient Descent*

We first test simple gradient descent for linear and ridge regression on our quadratic function $y = 4 + 3x + 2x^2$ with manual tuning of the learning rate. The results of the predicted data are shown below in Figure 2 for 4 values of the learning rate, eta = [0.0001, 0.001, 0.01, 0.2], after 100 iterations:

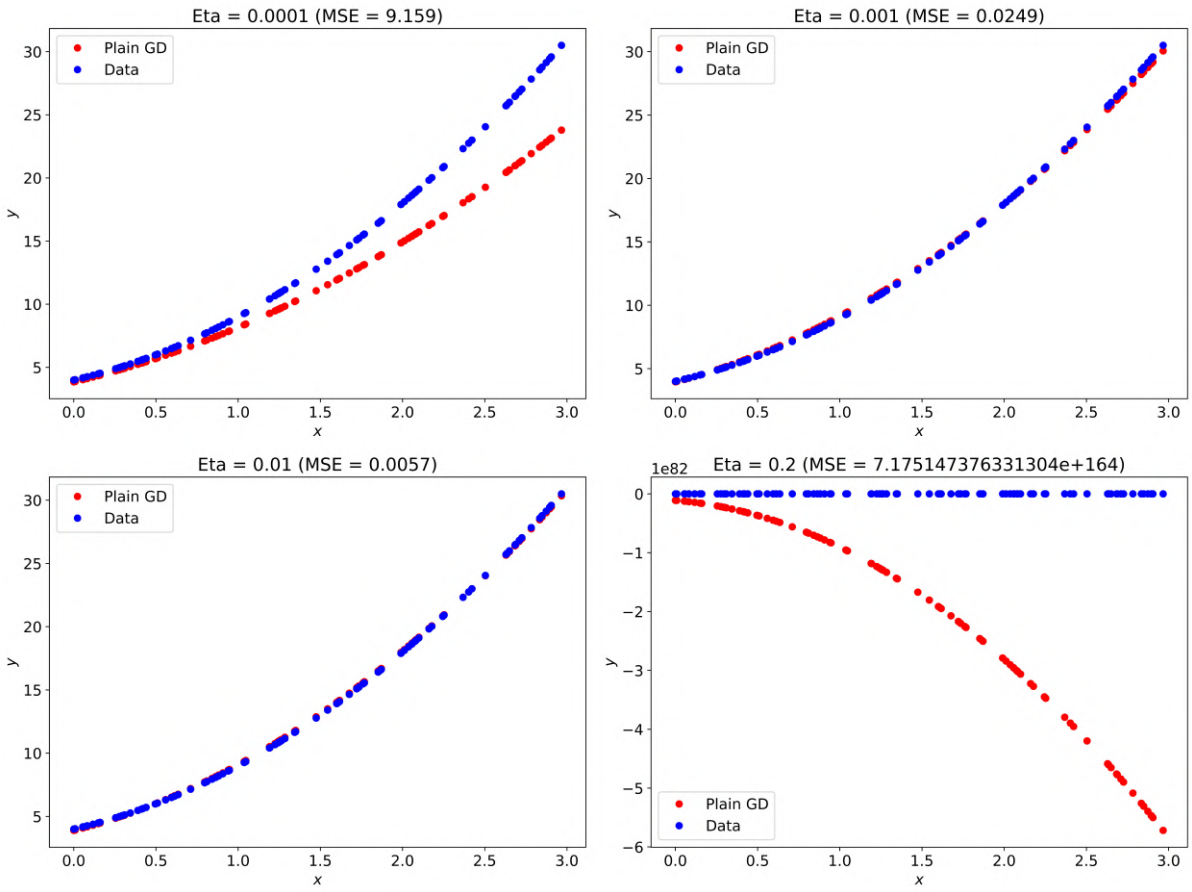


**Figure 2:** Gradient descent results plotted against the ideal quadratic function $y = 4 + 3x + 2x^2$ after 100 iterations for four different learning rates. The corresponding learning rates and MSEs are shown in the subfigure titles.

We can see that the learning rate of 0.0001 is too low and the result does not have time to converge to the true curve after 100 iterations, yielding a large MSE of 9.16. Increasing the learning rate to 0.001 results in a much more accurate curve with an MSE of 0.02, and increasing the learning rate by another factor of 10 to 0.01 results in even lower error of 0.0057. However, if we increase the learning rate significantly more, to 0.2, the error explodes and the result is not able to replicate the quadratic function at all. This is interpreted as the gradient descent overshooting the minima of the cost function because the initial step was too large. From these results it is evident that gradient descent is highly sensitive to the learning rate. In the following analyses we will often use the learning rate of 0.01, as it has been demonstrated here as a good choice for gradient descent on our considered function.

*Momentum Gradient Descent*

Next we test momentum gradient descent versus regular gradient descent for OLS and ridge regression, with a learning rate of 0.01. We investigated various values of the momentum and momentum-change parameters through trial and error, and found a momentum of 0.2 and a change of 0.01 to yield faster convergence to lower error, than for plain gradient descent. The

results are shown in Figure 3, where the momentum gradient descent curves converge faster for both OLS and ridge regression with a regularization parameter of 0.01.

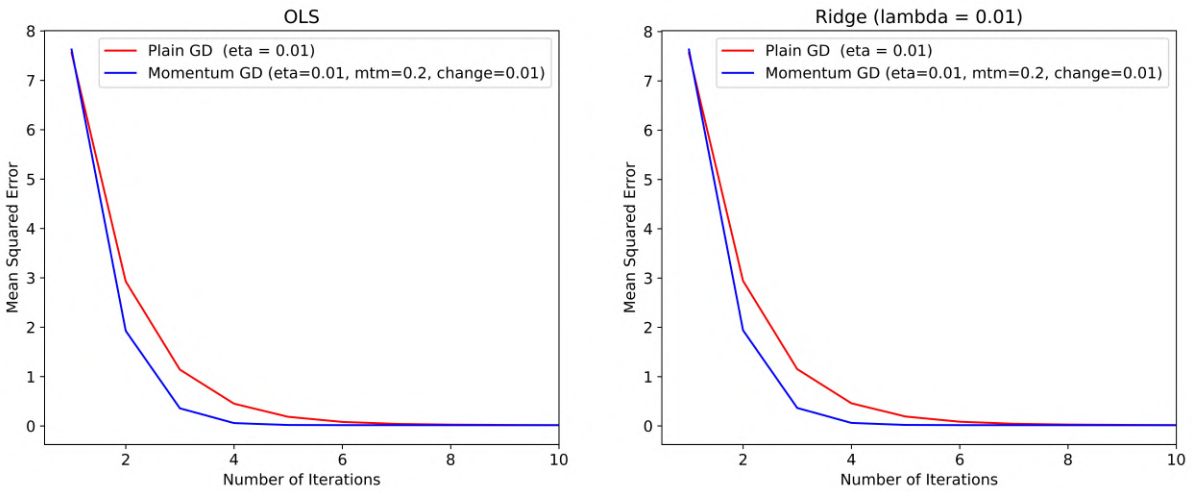


**Figure 3:** OLS (left) and ridge (right) regression comparison of plain gradient descent convergence, and momentum gradient descent convergence. For our momentum gradient descent, we use a momentum value of 0.2 and a change value of 0.01. The leraning rate used for both momentum and regular gradient descent is 0.01. In our ridge regression we use a lambda value of 0.01.

The resulting plots of the quadratic curves for this comparison of momentum and regular gradient descent are shown in Figure 4, along with the MSE of the predictions. We can see that the momentum gradient descent for OLS yields a lower MSE of 0.0048 as compared to 0.0057 for plain gradient descent. For ridge regression we achieve a lower error of 0.0039, and even lower at 0.0037 for momentum gradient descent.

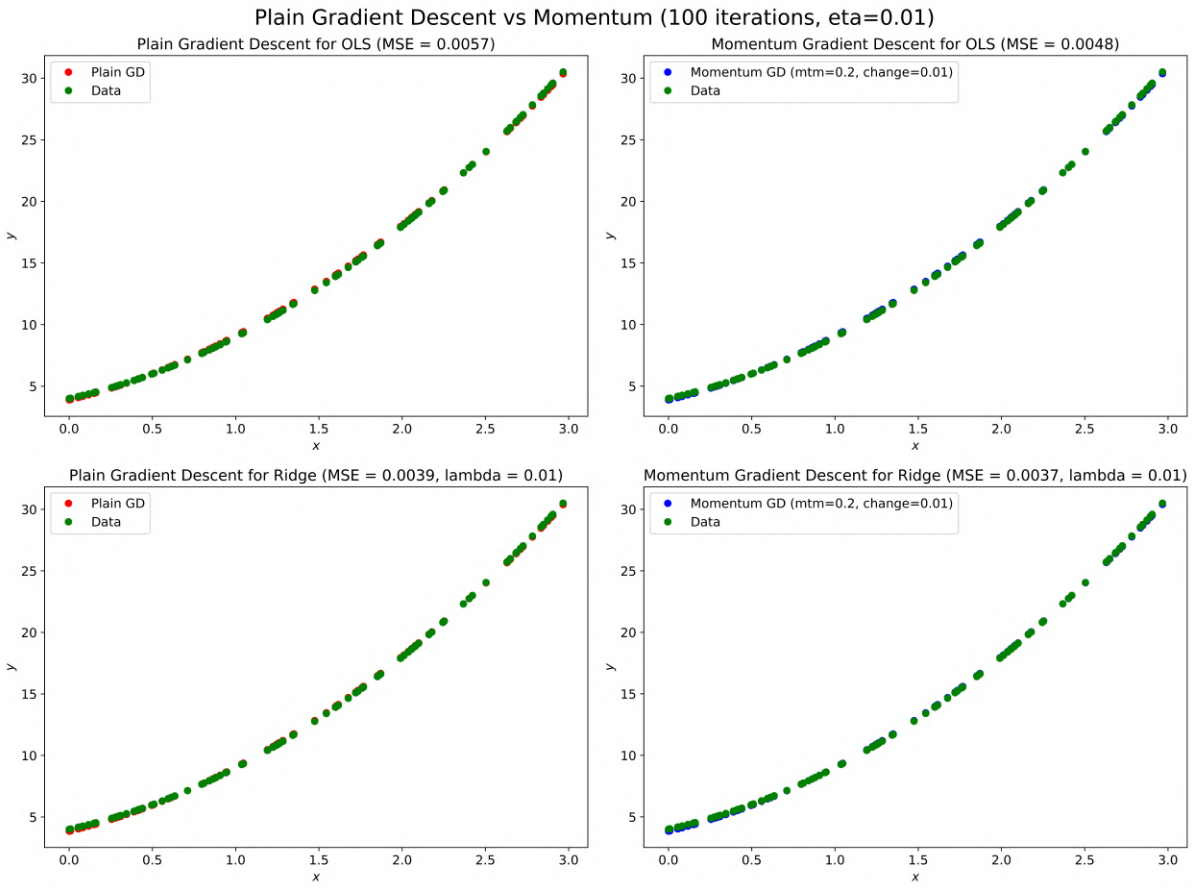


**Figure 4**

*Stochastic Gradient Descent*

Next we implement stochastic gradient descent (SGD) with adjustable minibatch size, tunable learning rate, and momentum. We first conduct a comparison of SGD with momentum GD. In the example in Figure 5, for SGD we use an initial tuned learning rate of 0.1 (t0/t1 = 5/50 = 0.1), a momentum of 0.8 with change of 0.1, and a minibatch size of 5. We compare this with momentum GD that has the same momentum and change parameters, and a learning rate of 0.01. The curves, plotted as a function of the number of epochs, show that SGD converges much more quickly (after about 20 epochs) as compared to momentum GD (after about 60 epochs) for these parameter values. Note that there is some choppiness in the curve for GD, which tends to manifest itself for several of our results, including for SGD as well. This may be because of some random fluctuations in the descent, as well as because of stochastic effects from selecting out batches in the case of SGD. But regardless a low error is still able to be achieved following the descent of these curves.

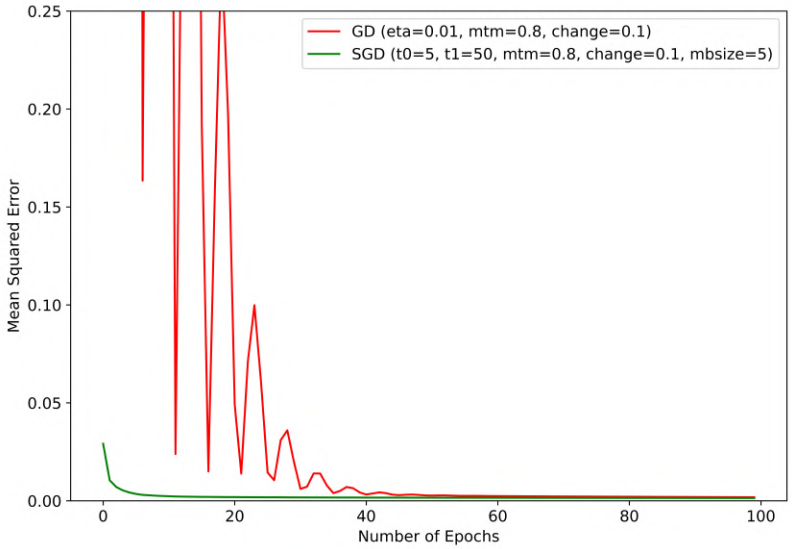


**Figure 5:** A comparison of the convergence of stochastic gradient descent (SGD) and momentum gradient descent (GD) for our quadratic function. The SGD curve converges to low error much more rapidly than for momentum GD.

Next we conduct SGD using the same parameters as for the result in Figure 5, but for varying minibatch size. The results are plotted in Figure 6. We can see that in general for lower minibatch size, the SGD converges more quickly, and for a minibatch size of 100, the results are very similar to the curve for momentum GD in Figure 5. This result is to be expected, as the minibatch size of 100 corresponds to using the entire dataset, and it is essentially the same as implementing regular gradient descent. For using the lowest minibatch size of 1, however, the error that the curve converges to is not as low, which is also intuitive since it is trying to converge based on individual noisier samples of the data. A compromise between regular GD (largest minibatch size) and a minibatch size of 1 is therefore desirable, in order to achieve faster convergence but with still lower error. This is why moderate minibatch sizes are commonly used in practice, and so we use a minibatch size of 5 in our previous example in Figure 5, and our next in Figure 7.

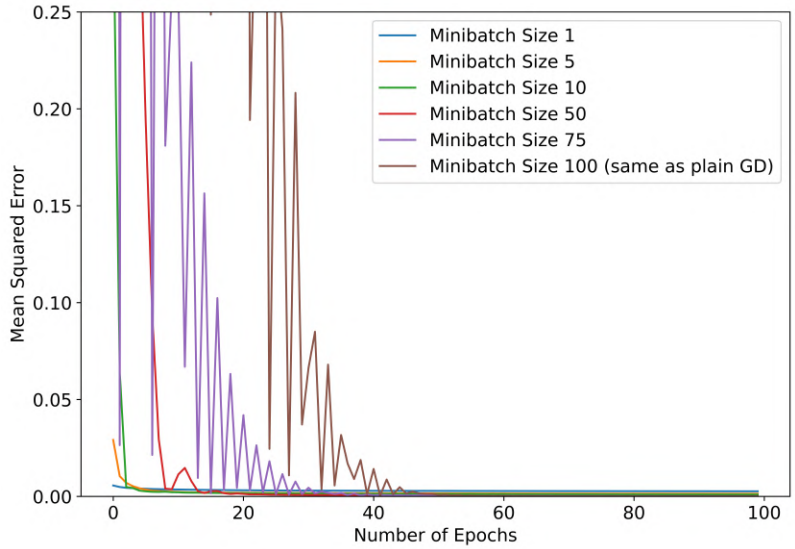


**Figure 6:** A comparison of the convergence for SGD with varying minibatch size, for OLS on our quadratic function. The parameters for our SGD in this case are t0=5, t1=50, momentum=0.8, change=0.1.

*Learning Rate Tuning Methods*

Next we implement three learning rate tuning methods: AdaGrad, RMSprop, and Adam, and compare them for SGD on our quadratic function. The convergence results to 100 epochs are shown in Figure 7. We use an initial learning rate of 0.01 for all the methods. For the RMSprop implementation, we use a rho value (the weighting of the rolling moving average for the gradients) of 0.9. And for the Adam implementation, we use a first moment value beta1 (for the momentum running average of the gradients) of 0.7, and a second moment value beta2 (for the momentum running average of the squared gradients) of 0.8. While it is difficult to say this is a general rule because different learning rate tuning algorithms may work better in different cases depending on the data and hyperparameters, we found in most of our convergence tests that Adam performed the best in terms of convergence rate on our noisy quadratic function. This makes sense as Adam is probably the most commonly used learning rate tuning method.

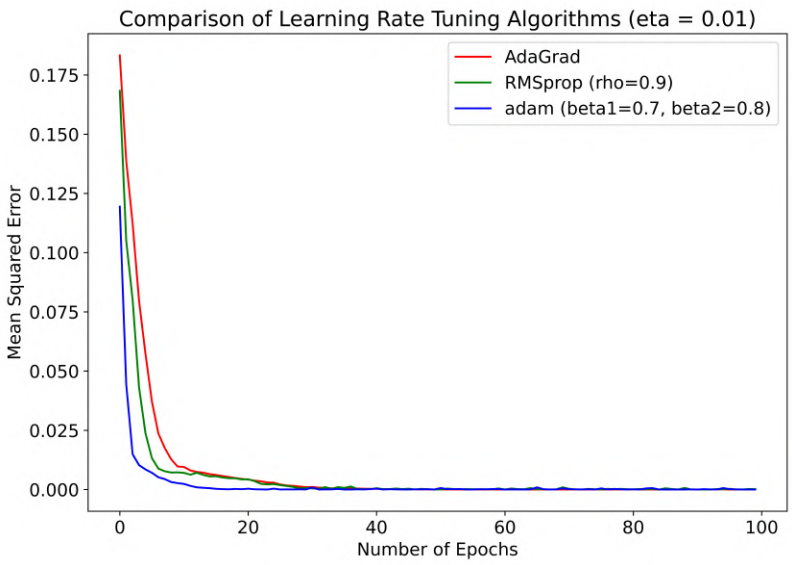


**Figure 7:** Convergence comparison using different learning rate tuning methods (AdaGrad, RMSprop, and Adam) for SGD on our noisy quadratic function. Using Adam leads to the fastest convergence when comparing on this learning rate (0.01).

## Regression on a Quadratic Function

In this section we will modify our feed forward neural network code to apply towards regression on our quadratic function, $y = 4 + 3x + 2x^2$. We use a 80/20 train test split for the data, and test various network configurations and hyperparameters to see their effect on the test mean squared error. Finally we compare our results with a neural network developed in Tensorflow. (Note: for testing error we actually use what is often considered the 'validation' error used for evaluating model hyperparameter selection, but we consider this as the testing error for simplicity since we only have 100 datapoints total and the model has not seen this data in training.)

*Feed Forward Neural Network Regression*

First we adapted our neural network to handle the task of regression. This was accomplished by first ensuring that the input layer and output layer each had 1 node (because we are dealing with a 1-dimensional function). Second, we ensured that our cost function used was that of mean squared error. Third, we decided that our output activation function should be ReLU, because we wanted a positive continuous output that could be greater than 1 like some of our data points (not one bounded by 0 to 1 as yielded by the sigmoid). Then we tested various network configurations.

These network configurations and the resulting minimum testing MSE they achieved are shown in Figure 8 below. We tested these by running the training for 1000 epochs at a learning rate of 0.01. Most of the tests were on single-hidden-layer networks with 50 nodes, and varying the

type of activation function for the hidden layer. We also tested a two-hidden-layer network with 50 nodes each, and a network with a single hidden layer of 100 nodes. From our results we determined that a network with a single hidden layer of 50 nodes and ReLU activation worked the best, achieving a lowest test MSE of 0.091 after 1000 epochs. Thus we decided to move forward with this configuration in another investigation over learning rate and regularization parameter.

| | ReLU hidden, 50 nodes | sigmoid hidden, 50 nodes | leaky ReLU hidden, 5 nodes | two ReLU hidden, 50 nodes each | ReLU hidden, 100 nodes |
|---|---|---|---|---|---|
| Minimum Test MSE | 0.091 | 0.70 | 1.85 | 0.57 | 280.46 |

**Figure 8:** Testing MSE for various regression neural network configurations, using a learning rate of 0.01 and 1000 epochs. We found that the single-hidden-layer network of 50 nodes and a ReLU activation performed the best (highlighted in green).

Here we perform a grid search over different values of the learning rate and an L2 regularization parameter, for the single-hidden-layer 50 node ReLU network. The results of the minimum test MSE after 1000 epochs is shown for each parameter combination in Figure 9. We find that using a learning rate of 0.01 and a lambda value of 1e-5 leads to the lowest MSE of 0.0494. Many of the results with an MSE of 280 do not end up converging, and stayed at that high value.

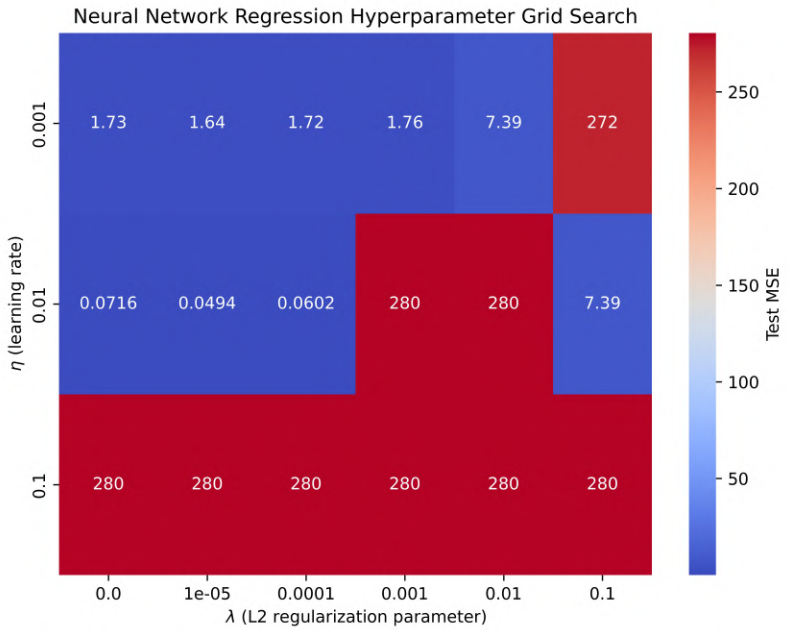


**Figure 9:** Hyperparameter grid search over lambda and learning rate for the regression neural network with one hidden layer and ReLU activation. The lowest testing MSE after 1000 epochs is displayed for each combination of lambda and eta.

Taking these hyperparameter values (learning rate of 0.01 and lambda of 1e-5) we train a neural network for 7500 epochs to investigate how low the error converges to. The results

of the testing MSE are shown in Figure 10, with the lowest value reached being 0.0051. This value is comparable to the many results of lowest MSE that we presented in the Optimization Methods section.

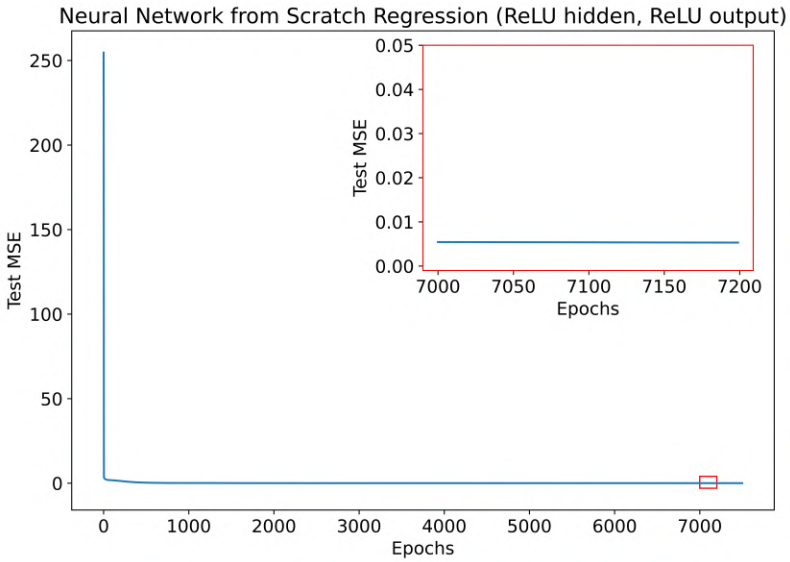


**Figure 10:** Testing MSE for 7500 epochs for our best neural network configuration (single hidden layer of 50 nodes, ReLU activation, no regularization, and learning rate of 0.01).

*Tensorflow Feed Forward Neural Network Regression*

We now take the parameters used in this best-performing network, and use them to code a similar network in Tensorflow. Again we train for 7500 epochs and output the testing MSE over time. The results are presented in Figure 11. We can see that initially the convergence is a bit slower than for our neural network from scratch, but we can see that the minimum is almost reached by 1000 epochs (this is why we used this value of 1000 epochs in our shorter tests earlier). However, zooming in on the curve late in the training, we see that the error reaches a value of about 0.09. This error fluctuates more than for our neural network trained from scratch, likely because of some complexities in Tensorflow that cause perturbations in the training process in order to avoid being stuck in local minima. While it is not important for our case - testing on a simple quadratic function - these complexities are likely important for data that are more complex. Regardless, it is good to see that the final error of about 0.009 is relatively close to the values of 0.005-0.006 that we reached with our own neural network and gradient descent methods.

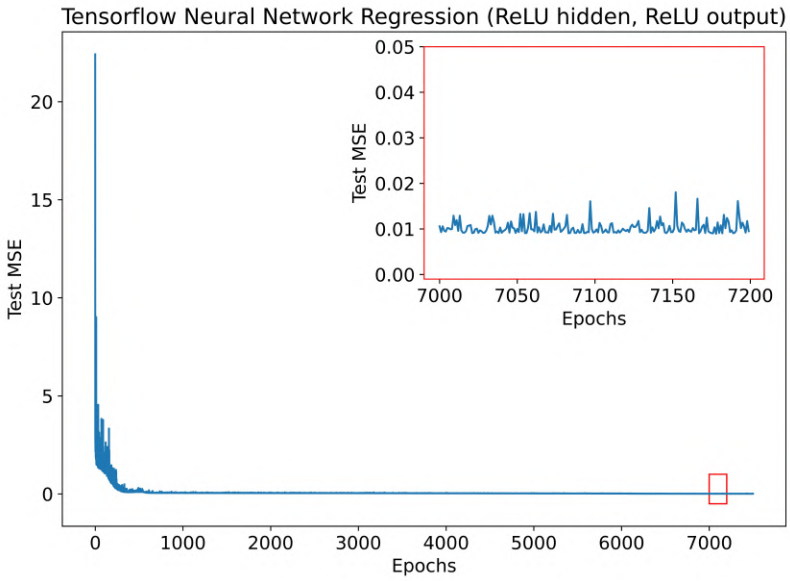


**Figure 11:** Testing MSE for 7500 epochs for a network developed in Tensorflow with the same configuration as the network presented in Figure 10. The convergence is a bit slower initially, but after 1000 epochs both networks essentially reach the minima. The final error the network reaches is about 0.009.

## Classification on Breast Cancer Data

In this section we will implement logistic regression, as well as modify our feed forward neural network code to apply towards classification on the Wisconsin Breast Cancer Dataset. We use a 80/20 train test split for a standard scaled version of the data, and test various network configurations and hyperparameters to see their effect on the test classification accuracy. Finally we compare our results with a neural network developed in Tensorflow.

*Logistic Regression*

We first implement logistic regression for a variety of learning rates and regularization parameters. We use SGD with a momentum value of 0.8 and a change value of 0.1 in order to minimize the cross entropy as the cost function, and run the training for 100 epochs. The results of our grid search are shown in Figure 12 below. We can see that the highest values of testing accuracy (0.965) are obtained with a learning rate of 1.0 and lower regularization parameter values. Therefore we choose a learning rate of 1.0, and a regularization parameter of 0 (for simplicity since it yields the same accuracy as other parameters). The values of 0.877 correspond to network configurations that never converge.

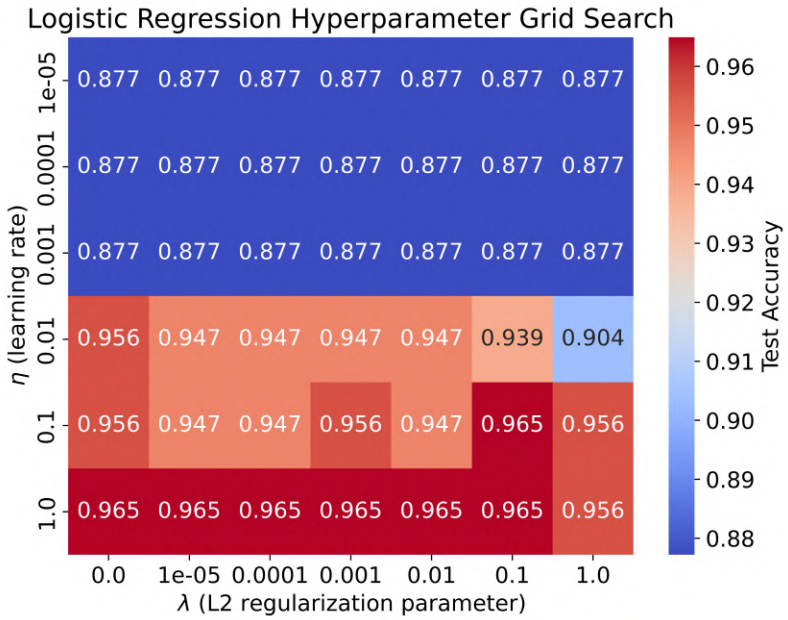


**Figure 12:** Grid search over the learning rate and regularization parameter for logistic regression. We implement SGD with momentum = 0.8 and change 0.1, for 100 epochs in order to optimize the cross entropy cost function. A learning rate value of 1.0 and lower regularization parameters yield the best accuracy.

We next compare the best performing model (eta = 1 and lambda = 0) with the worst performing model that converged (the model with an error of 0.904, eta = 0.01 and lambda = 1.0). We plot their test accuracies as a function of training epochs in Figure 13. Clearly both networks converge fairly quickly after about 40 epochs.

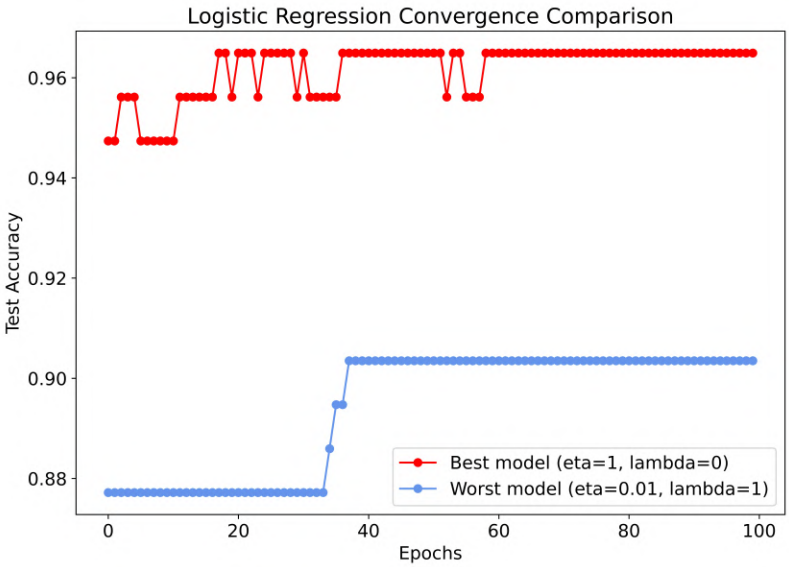


**Figure 13:** Convergence plots of the best and worst performing logistic regression models.

It is also interesting to investigate the impact that data scaling had on our classification results. We do this by training the two models in the previous example, but on unscaled data. The results are presented in Figure 14. We can see that the training curves are much noisier and do not converge well at all. This speaks to the importance of scaling the data, because in this case we have radically different features and values that need to be standardized to the same level.

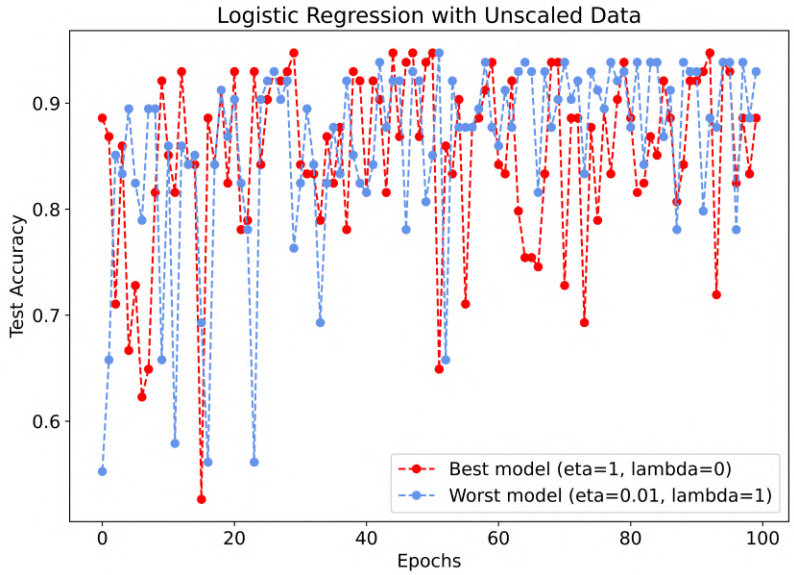


**Figure 14:** Convergence plots of the best and worst performing logistic regression models, for unscaled data.

*Feed Forward Neural Network Classification*

In this section we describe how we modified our feed forward neural network code to apply towards classification on the Wisconsin Breast Cancer Dataset. This was accomplished by first ensuring that the input layer had 30 nodes (for the number of features) and the output layer had 2 nodes (because we are solving a binary classification problem). Second, we ensured that our cost function used was that of binary cross entropy. Third, we decided that our output activation function should be a sigmoid, because we want to yield probabilities from 0 to 1. Then we tested various network configurations.

These network configurations and the resulting maximum testing classification accuracy they achieved are shown in Figure 15 below. We tested these by running the training for 200 epochs at a learning rate of 0.1. Most of the tests were on single-hidden-layer networks with 5 nodes, and varying the type of activation function for the hidden layer. We also tested a two-hidden-layer network with 5 nodes each, and a network with a single hidden layer of 20 nodes. From our results we determined that a network with a single hidden layer of 20 nodes and sigmoid activation worked the best, achieving a highest test accuracy of 0.956 after 200 epochs. Thus we decided to move forward with this configuration in another investigation over learning rate and regularization parameter.

| | ReLU hidden, 5 nodes | sigmoid hidden, 5 nodes | leaky ReLU hidden, 5 nodes | two sigmoid hidden, 5 nodes each | sigmoid hidden, 20 nodes |
|---|---|---|---|---|---|
| Max Test Accuracy | 0.939 | 0.947 | 0.921 | 0.921 | 0.956 |

**Figure 15:** Testing accuracy for various classification neural network configurations, using a learning rate of 0.1 and 200 epochs. We found that the single-hidden-layer network of 5 nodes and a sigmoid activation performed the best (highlighted in green).

Here we perform a grid search over different values of the learning rate and an L2 regularization parameter, for the single-hidden-layer 5 node sigmoid network. The results of the maximum test accuracy after 200 epochs is shown for each parameter combination in Figure 16. We find that using a learning rate of 1.0 and a lambda value of 0 (so in effect no regularization) leads to the highest test accuracy of 0.965. Note that this value is just slightly higher than the accuracy achieved with the same configuration as shown in Figure 15 (0.956) - this is likely due to some random effect in the initialization or training of our network, that was different on the second run.

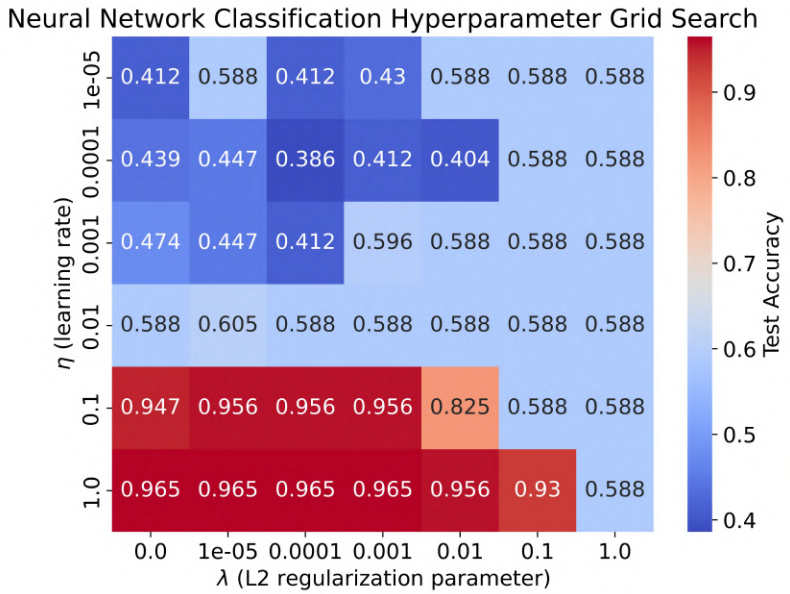


**Figure 16:** Grid search over the learning rate and regularization parameters for our classification neural network. We train each model for 200 epochs with a learning

Taking these hyperparameter values (learning rate of 1.0 and lambda of 0) we plot the testing accuracy as a function of the number of epochs. The results are shown in Figure 17, with convergence occurring past about 125 epochs, and the highest value reached being 0.965. This value is the same as the best value obtained in logistic regression.

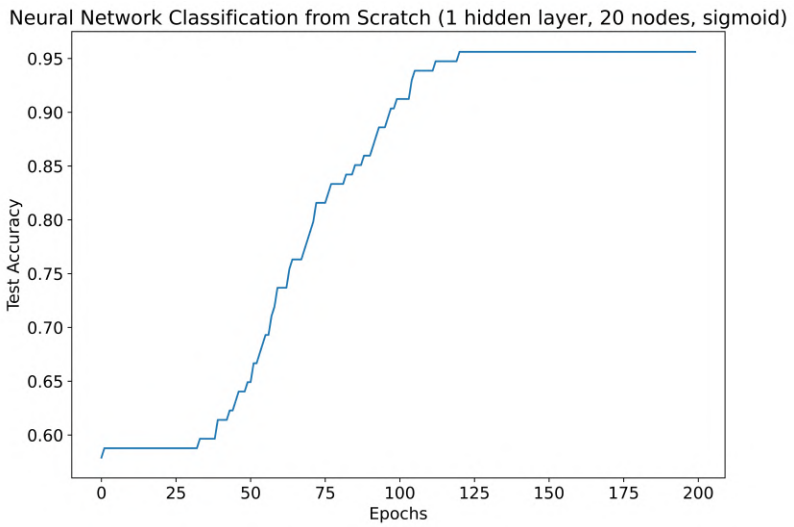


**Figure 17:** Plot of the testing accuracy vs number of epochs for our classification neural network with 1 hidden layer of 20 nodes and sigmoid activation.

*Tensorflow Feed Forward Neural Network Classification*

We also train a neural network for classification in Tensorflow, using the same configuration and training parameters. The resulting plot of its testing accuracy as a function of training epoch is shown in Figure 18. It reaches a final test accuracy of 0.956, a very similar value to the 0.965 achieved by our logistic regression and neural network from scratch.

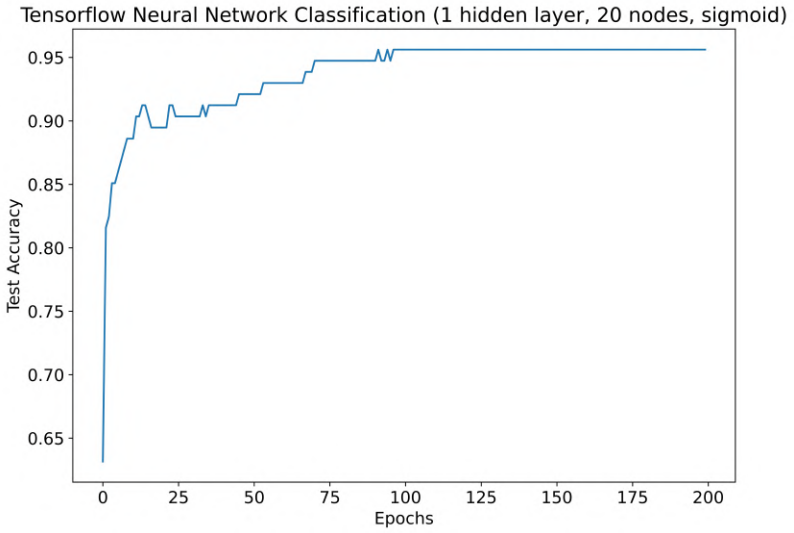


**Figure 18:** Testing accuracy as a function of epoch for a neural network trained in Tensorflow, with the same parameters as our neural network built from scratch. The accuracy converges to 0.956 after about 125 epochs.

## Discussion

We now compare the results of our regression and classification models, and put them in broader context. Our best results of various regression and classification models are presented in Figure 19.

| Regression | | | | Classification | | | |
|---|---|---|---|---|---|---|---|
| | Best NN from Scratch | Tensorflow NN | OLS | | Best NN from Scratch | Tensorflow NN | Best Logistic Regression |
| Lowest Test MSE | 0.0051 | 0.009 | ~ 0 | Highest Test Accuracy | 0.965 | 0.956 | 0.965 |

**Figure 19:** The best regression and classification models and their corresponding errors and accuracies.

When it comes to regression models, our neural network trained from scratch yielded the lowest testing MSE of 0.0051. Somewhat surprisingly, this was a slightly lower MSE than the Tensorflow neural network, which yielded 0.009. Although this is likely due to the more sophisticated code in Tensorflow that perturbs within the training to escape small local minima. Of course, the error yielded by ordinary least squares is essentially zero, since there is an analytical process and solution. In the case where someone wanted to achieve the lowest error on a simple function such as this, or a similarly simple dataset, they should obviously use OLS (or ridge or lasso) regression. Especially since the time taken to train a neural network is significantly longer. Neural networks are designed to be used on problems of greater complexity, such as those with nonlinear relationships, so on tasks such as those neural networks would definitely outperform traditional methods. However since this project is largely for educational purposes, we are happy to see that our regression neural network coded from scratch yields similar (or even slightly better) results than the Tensorflow network.

For the classification models, we found that the best performing models were our neural network from scratch and logistic regression, both with testing accuracies of 0.965. With slightly lower accuracy at 0.956 was the classification network developed in Tensorflow. It is interesting to note that we used single-hidden-layer networks with 20 nodes in our from-scratch and Tensorflow models, yet these did not perform any better than logistic regression, and logistic regression is essentially a single-node neural network. This indicates that this dataset and classification problem are simple enough to be handled by logistic regression, and it does not require a complex neural network with many nodes. Regardless it is also good to see that our from-scratch network can perform at the same level as the Tensorflow network.

## Conclusions

In this project we reviewed and implemented several optimization, regression, and classification methods. We started with implementing gradient descent, gradient descent with momentum, stochastic gradient descent, and stochastic gradient descent with learning rate tuning methods, all on a quadratic function. From this we found that stochastic gradient

descent with momentum reached the same error level as regular gradient descent, but with significantly faster convergence. We also found that the Adam learning rate tuning algorithm tended to allow faster convergence than the AdaGrad and RMSprop methods.

We then moved to implementations of a neural network from scratch on a regression problem for the same quadratic function. We found that a single-hidden-layer network with 50 nodes and ReLU activation performed the best, reaching a test MSE of 0.0051. This error was slightly lower than the error achieved by a network we built in Tensorflow with the same architecture. While this error is significantly higher than the essentially-zero error that can be achieved with OLS for this problem, we are confident that the neural networks we developed would perform signifcantly better on higher complexity datasets and tasks, which we may explore in the next project.

Finally we implemented a classification task using our neural network from scratch - the classification of tumors from the Wisconsin Breast Cancer Dataset. We found that the our custom network, which had a single hidden layer with 20 nodes and sigmoid activation, performed as well as logistic regression, with both achieving a classification accuracy of 0.965. This was slightly higher than a Tensorflow implementation of the network, which achieved a classification accuracy of 0.956.

# References


Boyd, S., and L. Vandenberghe, 2004, "Convex Optimization," Cambridge University Press, https://web.stanford.edu/ boyd/cvxbook/bv_cvxbook.pdf.

Geron, A., 2017, "Hands-On Machine Learning with Scikit-Learn and TensorFlow," O'Reilly Media.

Goodfellow, I., Y. Bengio, and A. Courville, 2016, "Deep Learning," MIT Press, https://www.deeplearningbook.org/.

Hinton, G., 2012, "RMSprop: divide the gradient by a running average of its recent magnitude," Neural Networks for Machine Learning Lecture 6e, https://www.cs.toronto.edu/ tijmen/csc321/slides/lecture_slides_lec6.pdf.

James, G., D. Witten, T. Hastie, and R. Tibshirani, 2013, "An Introduction to Statistical Learning: with Applications in R," Springer, https://doi.org/10.1007/978-1-4614-7138-7.

Jurafsky, D., and J.H. Martin, 2021, "Speech and Language Processing," 3rd edition, Chapter 5 - Logistic Regression, Prentice Hall, https://web.stanford.edu/ jurafsky/slp3/.

Kingma, D.P., and Ba, J., 2014, "Adam: A Method for Stochastic Optimization", https://doi.org/10.48550/arXiv.1412.6980.

Mehta, P., M. Bukov, C.H. Wang, A.G.R. Day, C. Richardson, C.K. Fisher, and D.J. Schwab, 2019, "A High-Bias, Low-Variance Introduction to Machine Learning for Physicists," Physics Reports, vol. 810, pg. 1-124, https://doi.org/10.1016/j.physrep.2019.03.001.

Nielsen, M.A., 2015, "Neural Networks and Deep Learning," Determination Press, https://neuralnetworksanddeeplearning.com.

Sharma, A., 2017, "Understanding Activation Functions in Deep Learning", LearnOpenCV, https://learnopencv.com/understanding-activation-functions-in-deep-learning/.