Project 2: Neural Networks

Josef Haisanawi,Higinio Joaquin Perez Parra, Mustafa Najjar *University of Oslo*

Gradient descent, stochastic GD and neural networks were used on random data generated using Franke function and the Wisonsin breast cancer dataset. The best performing SGD optimizer was the ADAM optimizer with the lowest net error of 0.001932 at 500 epochs and a batch size of 50. The neural network on the Franke function resulted in the best scores for MSE = 0.04877 using the ReLU activation function with 500 epochs, a learning rate of 10^{-3}, regularization parameter 10^{-15}, 4 hidden layers and 128 nodes per hidden layer. Neural network classification on the Wisconsin dataset showed that the accuracy was lowest for Sigmoid and higher but all equal for the rest of the activation functions of about 92.31%. Performing logistic regression on the Wisconsin dataset resulted in accuracy decreasing sharply at a learning rate of 10^{-3}, dropping from 94% to 62%, and is negatively affected by a regularization hyperparameter value larger than 10^{-10} causing the accuracy to drop from 90.9% to 89.5%.

Introduction

Artificial neural networks (ANNs), usually simply called neural networks (NNs) or neural nets, are computing systems that can learn to perform tasks by considering examples, generally without being programmed with any task-specific rules. Neurons interact by sending signals in the form of mathematical functions between layers.

An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons or not. An artificial neuron receives signals from other neurons, then processes all the inputs, and then transmit a signal to other neurons connected to it. These other neurons will repeat the process until the final output is reached.

The idea of neural networks dates as far back as the 1940's. In fact, the field of artificial neural networks has a long history of development, and is closely connected with the advancement of computer science and computers in general. A model of artificial neurons was first developed by McCulloch and Pitts in 1943 to study signal processing in the brain and has later been refined by others.This model is the foundation of most neural networks used today. Weighted inputs are evaluated by a special function to determine the state of activation of the neuron. This however posed a major drawback: with a single perceptron (artificial neuron), only linear problems can be solved.

In the following decade, the work of Bernard Widrow and Marcian Hoff, led to the creation of networks of perceptrons that are capable of solving non-linear problems. Their application of neural networks to a real-world problem (in that case was reducing noise over phone lines), is still in use today. Nowadays, neural networks are much more robust and able to solve a vast array of complex problems, from weather prediction to self-driving cars and more that are yet to come!

In this report we are not going to program a car to self-drive, but we are hoping to build a neural network that can correctly classify the state of a potential cancer patient given some features about the tumour.

We start by solving simple regression and classification problems with gradient descent (GD), mini-batch stochastic gradient descent (SGD) and RMSprop gradient descent based on momentum as well as the AdaGrad and ADAM optimizer methods. We solve the ordinary least squares, ridge and logistic regression cases. We also compare the efficiency of the different optimization methods and the update schedule for the learning rate. This part is done without any involvement of a neural network.

We then describe our implementation of a feedforward neural network (FFNN), with a flexible number of layers, nodes per layer, activation functions, and so forth. The training process is achieved by the backpropagation algorithm, for the lightweight calculation of otherwise computationally expensive gradients, in conjunction with GD or SGD.

Lastly we apply this neural network code to a famous data set amongst the machine learning community, the Wisconsin Breast Cancer data set https://archive.ics.uci.edu/ml/index.php. This data set consists of features, such as area, radius, texture, etc..., taken from the medical imaging of tumours from different patients that are either classified as malignant or benign. Our objective is to use the developed code for the neural network to predict whether or not a given individual has a malignant or benign tumour given the features of the tumour.

Theory

One of the benefits of neural networks is that they can be set up to both classification and continuous problems. We will test our implementation against logistic regression for a classification problem, and against linear regression for a continuous problem. In this section, we will describe some of the important points that are used in our implementation. As we have already gone through regression in our previous report, we will mostly focus on logistic regression and FFNNs here.

Feedforward Neural Network

The feed-forward neural network (FFNN) was the first and simplest type of ANNs that were devised. In this network, the information moves in only one direction: forward through the layers.

The action of a single neuron can be represented as follow:

$$y = f\left(\sum_{i=1}^{n} W_i x_i\right) = f(u)$$

where, the output y of the neuron is the value of its activation function f, which have as input a weighted sum of signals $x_i, \dots, x_n$ received by n other neurons.

The simplest possible neural network is the single perceptron model, where we have one node with two inputs and one output. We consider that each input has an associated weight W_x and W_y, and that the node has an intrinsic bias b and an activation function σ. Given an input $z = W_x x + W_y y + b$, the activation function determines the output of the node $a = \sigma(z)$. This simple example can be thought of as a regression problem for the three parameters x, y and the intercept b.

With this simple example we can think of the weights representing the coefficients for a linear regression problem of up to degree two and the bias as the intercept.

We will study linear models that can perform a polynomial fit to a well-behaved function. We can then build upon the simple perceptron model. If we now let the one node have n inputs, we can solve a linear regression problem of degree n, if we think of each input i as being x^i, and each weight as being a coefficient in the linear expression.

One uses often so-called fully-connected feed-forward neural networks with three or more layers (an input layer, one or more hidden layers and an output layer) consisting of neurons that have non-linear activation functions. Such networks are often called multilayer perceptrons (MLPs). With this we can solve even harder problems, for instances, image recognition or K-class classification problems, instead of just binary classification problems.

Weights and Biases

Typically weights are initialized with small values distributed around zero, drawn from a uniform or normal distribution. Otherwise, setting all weights to zero means all neurons give the same output, making the network useless and unable to learn.

Adding a bias value to the weighted sum of inputs allows the neural network to represent a greater range of values. Without it, any input with the value 0 will be mapped to zero (before being passed through the activation). The bias unit has an output of 1, and a weight to each neuron j, b_j:

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

The bias weights $\boldsymbol{b}$ are often initialized to zero, but a small value like 0.01 ensures all neurons have some output which can be backpropagated in the first training cycle.

Such a network is assigned L layers (layers 2 through $L-1$ are the hidden layers, and layer 1 is the input layer and layer L is the output layer), with n_l neurons (or nodes) in each layer. For each of the layers a weight matrix $W^{(l)}$ of size $n_l \times n_{l-1}$ is assigned, where the entry $W_{ij}^{(l)}$ represents the weight that connects node i from layer $l-1$ to node j into layer l (do not mix capital L for the total number of layer with l a dummy index). Each layer is also assigned a bias vector $b^{(l)}$ of size n_l. To evaluate the networks predictions, we apply a cost function. This will determine the accuracy or error of our model. We wish, of course, to minimize this function.

A trained network has its weights and biases in all layers set to some combination of values that, together, ideally produce results in its output layer that are close to the desired target values for any valid input of size n_0.

The process by which we can use our model to make predictions is called feedforward. We can use such a model to make predictions using the feed forward process. The inputs can be denoted as $a^{(0)}$; then, for each layer, in turn, from 1 to L, the layer's output $a^{(l)}$ (vector of size n_l) is computed as

$$\begin{aligned} z^{(l)} &= W^{(l)} a^{(l-1)} + b^{(l)}, \\ a^{(l)} &= \sigma^{(l)}(z^{(l)}), \end{aligned}$$

where $\sigma^{(l)}$ is the layer's activation function. For a regression problem, the output layer's activation function should be the identity function, $\sigma^{(L)} : z_i \mapsto z_i$, as we wish to allow all values in $\mathbb{R}$. On the other hand, for classification problems, as we desire a probabilistic output, the prediction should be contained in $[0, 1]$. This way, we can use a sigmoid function,

$$\sigma^{(L)}(z_i) = \frac{1}{1 + e^{-z_i}},$$

or any other function of the type $\mathbb{R} \to [0, 1]$. Another usual example is the softmax

$$\sigma^{(L)}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n_l} e^{z_j}}$$

or the hyperbolic tangent

$$\sigma^{(L)}(z_i) = \tanh(z_i).$$

Additionally, we explore using different activation functions in the hidden layers, amongst the sigmoid, the hyperbolic tangent, the rectified linear unit (ReLU) function

$$\sigma^{(l)}(z_i) = \begin{cases} z_i & z_i \geq 0 \\ 0 & z_i < 0, \end{cases}$$

the leaky ReLu

$$\sigma^{(l)}(z_i) = \begin{cases} z_i & z_i \geq 0 \\ \alpha z_i & z_i < 0, \end{cases}$$

and the exponential linear unit (ELU) function

$$\sigma^{(l)}(z_i) = \begin{cases} z_i & z_i \geq 0 \\ \alpha\left(e^{z_i} - 1\right) & z_i < 0, \end{cases}$$

with $0 < \alpha \ll 1$.

As discuss previously, it is common courtesy to initialize out weights and biases randomly. Typically, the weights and biases in the network are initialized randomly, corresponding to an initial guess. It is then highly unlikely to produce good predictions. In order to obtain accurate prediction with our network, the weights and biases are iteratively adjusted until the predictions given by the network are satisfactory. This process is called training. In this process we use gradient methods to find the minimum of the model's cost function. We are now faced with a problem of computing the gradients of the cost function with respect to all of the weights and biases that comprise the network. As the network has potentially hundreds of weights and biases all dependant on each other, this becomes quite computationally heavy to do naively.

Depending on the way we initialize the weights and biases, we might get a faster convergence or non-convergence of the employed gradient method. We initialize the biases to some small value ϵ (the value of which we can experiment with to find the best initial value), and the weights with uniform distribution between -1 and 1. Then, over a number of generations or epochs (iterations of our neural network), the training data, or, depending on the method employed, smaller batches taken out of the full set, is repeatedly fed forward through the network. Each time, the output error is computed as the gradient of the cost function times the derivative of the output layer's activation function.

The way we update the weights and biases each epoch is given by

$$\begin{aligned} W^{(l)} &\leftarrow W^{(l)} - \eta \frac{\partial C}{\partial W^{(l)}} \\ b^{(l)} &\leftarrow b^{(l)} - \eta \frac{\partial C}{\partial b^{(l)}}. \end{aligned}$$

Where C is our cost function. To compute the gradients of the cost function with respect to the weights and biases, we used the back propagation algorithm, which consists of computing

$$\begin{aligned}
\frac{\partial C}{\partial z^{(L)}} &= \frac{\partial C}{\partial a^{(L)}} \odot \sigma^{L\prime}(z^{(L)}), \\
\frac{\partial C}{\partial z^{(l)}} &= \sigma^{(l)\,\prime}(z^{(l)}) \odot ((W^{(l+1)})^T \delta^{(l+1)}), \\
\frac{\partial C}{\partial b^{(l)}} &= \delta^{(l)}, \\
\frac{\partial C}{\partial W^{(l)}} &= \delta^{(l)}(a^{(l-1)})^T,
\end{aligned}$$

where $\delta^{(l)} := \frac{\partial C}{\partial z^{(l)}}$.

Once all weights and biases throughout the network have been updated epoch times, the hope is that our cost function decreases.

Additionally, to prevent overfitting, we can introduce a regularization term in the cost function . For an ℓ_2 regularization term the updated gradients will take the form

$$\begin{aligned}
W^{(l)} &\leftarrow W^{(l)} - \eta \left(\frac{\partial C}{\partial W^{(l)}} + \lambda W^{(l)} \right) \\
b^{(l)} &\leftarrow b^{(l)} - \eta \left(\frac{\partial C}{\partial b^{(l)}} + \lambda b^{(l)} \right),
\end{aligned}$$

where λ is a hyperparameter for the regularization term, and C is the cost function without regularization.

Methods

As seen previously, one of the biggest challenges is to find the minimum of a certain function. In our cased it will be the cost function that we will define for our different problems. In order to find that minimum it exist different methods that we are going to discuss in this part.

In deed, when optimizing a function $f : \mathbb{R}^n \to \mathbb{R}$, we can find hard or impossible expressions to solve analytically. We are then in need of numerical method that give good approximations. If the function we are optimizing is differentiable, we can try to find where the gradient is zero.

Gradient Methods

We note that any maximization problem can be turned into a minimization problem. We are going to define the g function as $g(x) = \nabla f(x)$.

For a function $g : \mathbb{R}^n \to \mathbb{R}^n$, Newton's method can be written as

$$x_{k+1} = x_k - \big(J_g(x_k)\big)^{-1} g(x_k)$$

where $J_g(x)$ denotes the Jacobian of g at x. Two ways of deriving this are by setting a first order Taylor approximation of g equal to zero, or by finding the matrix Λ such that the fixed point iteration $x_k - \Lambda g(x_k) = x_{k+1}$ is a contraction . As our goal is to find a zero of ∇f, the function g above will now be the gradient of ∇f, and the Jacobian of ∇f is the Hessian matrix $H_f(x)$. Thus, Newton's method takes the form

$$x_{k+1} = x_k - \big(H_f(x_k)\big)^{-1} \nabla f(x_k).$$

In many cases, data sets and the number of parameters can be so large that computing the Hessian matrix and inverting it demands to much time. To handle this problem, we can use simultaneous relaxation instead. Simultaneous relaxation is defined by the recursion

$$x_{k+1} = x_k - \gamma g(x_k)$$

We can think of this as setting the Jacobian of g equal to a single number. For finding zeros of our gradient ∇f, this method is given by

$$x_{k+1} = x_k - \gamma_k \nabla f(x_k) \tag{1}$$

This method is known as gradient descent. The basic idea of gradient descent is that a function $F(\mathbf{x})$, $\mathbf{x} \equiv (x_1, \cdots, x_n)$, decreases fastest if one goes from $\mathbf{x}$ in the direction of the negative gradient $-\nabla F(\mathbf{x})$. The parameter γ_k is often referred to as the step length or the learning rate within the context of Machine Learning. For it to be a contraction, we would need γ_k to be less than the largest eigenvalue of $H_f(x)^{-1}$ for all x.

Ideally the sequence $\{\mathbf{x}_k\}_{k=0}$ converges to a global minimum of the function f. However, in general we do not know if we are in a global or local minimum. In the special case when f is a convex function, all local minima are also global minima, so in this case gradient descent can converge to the global solution. The advantage of this scheme is that it is conceptually simple and straightforward to implement. However the method in this form has some severe limitations.

In machine learning we are often faced with non-convex high dimensional cost functions with many local minima. Since GD (gradient descent) is deterministic we will get stuck in a local minima, if the method converges, unless we have a very good initial guess. This also implies that the scheme is sensitive to the chosen initial condition.

Besides that, it is not uncommon to work with so large data sets and the gradient is a function of $\mathbf{x} = (x_1, \cdots, x_n)$ which makes it expensive to compute numerically and might be considered to slow.

Batches and mini-batches

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

In large-scale applications, the training data can have on order of millions of examples. Hence, it seems wasteful to compute the full cost function over the entire training set in order to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over batches of the training data. For example, a typical batch could contain some thousand examples from an entire training set of several millions. This batch is then used to perform a parameter update.

Stochastic Gradient Descent

To solve the problems associated with gradient descent, we introduce the method called Stochastic Gradient Descent. This method consists on randomly choosing each time a data point and do gradient descent for this data point alone. This way, we are able to have much faster computations, but we will not choose an optimal direction for the updated point in our iteration. Although this sounds bad, it will also allow the iterated points to be able to move away from local minimum. This corresponds to the extreme case where we have only one batch, that is we include the whole data set.

The idea of stochastic gradient descent opens up for a similar method which we will call mini-batch Stochastic Gradient Descent (or simply SGD). In this method, we choose M data points at random and do gradient descent for these data points. M is called the batch size of the method. This method benefits from being less computationally heavy, while also allowing us the benefits of gradient descent. We note that if we have N data points, then $\lfloor N/M \rfloor$ iterations is called an epoch.

An alternative to the constant learning rate γ we suggested above would be a learning rate that depends on each iteration. This is known as the adaptive learning rate. This becomes especially important in situations where the landscape is shallow and flat, where a larger learning rate is needed, or narrow and steep, in that case a slower learning rate is required. For decreasing learning rates, one major drawback is that our learning rate might approach zero before the method has had time to converge. We will study the learning rate

$$\gamma(t) = \frac{t_0}{t_1 + t}$$

where t_0 and t_1 are parameters we can tune and t is the current epoch.

Stochastic Gradient Descent with Momentum

We can apply the concept of momentum to our vanilla gradient descent algorithm. In each step, in addition to the regular gradient, it also adds on the movement from the previous step. Mathematically, it is commonly expressed as:

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

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

where we have introduced a momentum parameter γ, with $0 \leq \gamma \leq 1$, and for brevity we dropped the explicit notation to indicate the gradient is to be taken over a different mini-batch at each step. We call this algorithm gradient descent with momentum (GDM). From these equations, it is clear that $\mathbf{v}_t$ is a running average of recently encountered gradients and $(1-\gamma)^{-1}$ sets the characteristic time scale for the memory used in the averaging procedure.

So, in what ways is Momentum GD better than vanilla gradient descent? Momentum GD simply moves faster, thanks to all the momentum it accumulates. Momentum GD has also a shot at escaping local minima because the momentum may propel it out of a local minimum. In fact, the gradient descent with momentum algorithm borrows the idea from physics. Imagine rolling down a ball inside of a friction less bowl. Instead of stopping at the bottom, the momentum it has accumulated pushes it forward, and the ball keeps rolling back and forth, or may just escape the local minima well.

Now, let's consider two extreme cases to understand this decay rate parameter better. If the decay rate is 0, then it is exactly the same as ordinary gradient descent. If the decay rate is 1 (and provided that the learning rate is reasonably small), then it rocks back and forth endlessly like the friction less bowl analogy we have mentioned. Typically the decay rate is chosen around 0.8–0.9. It's like a surface with a little bit of friction so it eventually slows down and stops.

AdaGrad Method

Instead of keeping track of the sum of gradient like momentum, the Adaptive Gradient algorithm, or AdaGrad for short, keeps track of the sum of gradient squared and uses that to adapt the gradient in different directions.

In machine learning optimization, some features are very sparse. The average gradient for sparse features is usually small so such features get trained at a much slower rate. One way to address this is to set different learning rates for each feature.

AdaGrad addresses this problem using the following idea. The more a feature has been updated, the less it will update it in the future. Thus giving a chance for the others features (like the sparse features) to catch up. In visual terms, how much you have updated this feature is to say how much you have moved in that dimension (or direction), and this notion is captured by the cumulative sum of gradient squared.

This property allows AdaGrad (and other similar gradient-squared-based methods like RMSProp and ADAM) to escape a saddle point much better. AdaGrad will take a straight path, whereas momentum gradient descent takes the approach of sliding down the steep slope first and maybe take care of the slower direction later. Sometimes, vanilla gradient descent might just stop at the saddle point where gradients if both directions are 0.

RMSProp Method

The problem of AdaGrad, however, is that it is incredibly slow. This is because the sum of gradient squared only grows and never shrinks. RMSProp which stands for Root Mean Square Propagation fixes this issue by adding a decay factor.

More precisely, the sum of gradient squared is actually the decayed sum of gradient squared. The decay rate is is meaning that only recent gradient matters, and the ones from long ago are basically forgotten. As a side note, the term decay rate is a bit of a misleading. Unlike the decay rate we saw in momentum, in addition to decaying, the decay rate here also has a scaling effect. It scales down the whole term by a factor of (1 - decay rate). In other words, if the decay rate is set at 0.99, in addition to decaying, the sum of gradient squared will be $\sqrt{1 - 0.99} = 0.1$ that of AdaGrad, and thus the step is on the order of 10 times larger for the same learning rate.

ADAM Method

Lastly but not least, ADAM (short for Adaptive Moment Estimation) takes the best of both worlds of Momentum and RMSProp. ADAM gets the speed from momentum and the ability to adapt gradients in different directions from RMSProp. The combination of the two makes it probably the best option. The method is especially efficient when working with large problems involving lots data and parameters.

Results

Gradient Descent

The primary way in which the performance of the various gradient descent, stochastic gradient descent its optimizers was measured was by taking the error between the beta values from the descent and reference beta values from ordinary least squares regression with the ridge parameter lambda. Note: most of the GD simulations have constant parameters set to the following for consistency. Momentum = 0.5, Learning rate = 0.1, Lambda = 0.001 and Batch size = 50

The performance of standard gradient descent is dependent on several parameters, including the regularization parameter, momentum, learning rate and batch sizes. The first simulation was done using varying lambda values, the results are shown in appendix A1 12. It is clear from these results that the lower values for the regularization parameter, lambda work best for gradient descent. This was also tested with most of the other methods and the results are consistent.

In standard GD with the learning rate set to 0.1 and batch size = 50, the net error between the reference OLS beta values and those from the simulation was given to be about 0.689 at the lowest lambda = 0.1 and the next lowest was 0.961 at lambda = 0.001. The value for lambda that was used throughout the simulations was selected to be 0.001 as it provides good results for both GD and SGD The learning rate is one of the largest contributors to the performance, making it necessary to tune it to the right value and the results were consistent throughout each method and simulation that makes use of a constant learning rate, as seen in figure 1 below as well as appendices A2 13 and A4 15.

A constant learning rate of 0.1 was selected due to the far superior performance as compared to all other tested values. A learning rate of 0.1 gave an error of 0.234 whereas the next lowest value was 2.698 for a

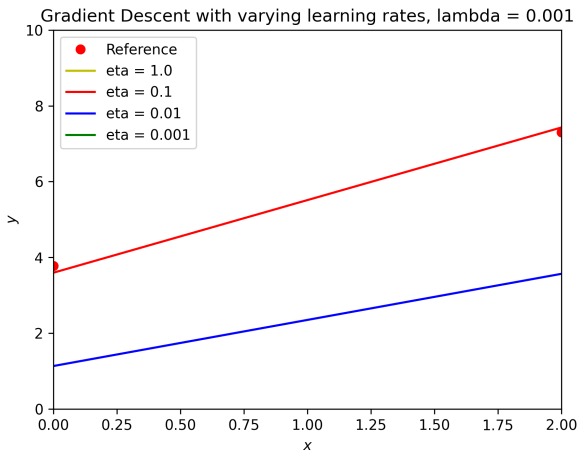

Figure 1: *Plot of standard gradient descent with varying learning rates, lambda = 0.001. Best fit lines are plotted using learning rates of 1.0, 0.1, 0.01 and 0.001.*

learning rate of 0.01.

When varying the batch sizes for standard GD, the results are as expected with the best performance yielded by lower batch sizes. This is because for standard gradient descent, a batch size of just 1 gives the maximum number of iterations based on the number of data points. A batch size of just gives an error of about 0.0106 whereas a batch size of 25 gives a much higher error of 0.6512 as shown in appendices A3 14 and A6 17. A batch size of 50 was used for the remainder of the simulations as though it does not provide the best values for standard GD, it provides better results for SGD and will be useful to use as a baseline for comparisons.

Adding momentum to standard GD results in better performance up to a certain point, about 0.1 as shown in appendix A5 16. Momentum values that are too high will result in drastically worse performance, a net error of about 2.278 when the momentum = 1.0 as compared to an error of 0.207 for a momentum of 0.1. A momentum of 0.5 was chosen however, for the remainder of the GD with momentum simulations for better comparisons to SGD. Table 1 below shows standard gradient descent with different learning rates and momentums.

Stochastic Gradient Descent

When interpreting these results, due to the random nature of stochastic gradient descent, some variation is expected when doing repeated runs without a predefined seed.

The simulations set up for SGD and all the related optimizers use a varying learning rate that decays

Table 1: *Gradient descent with varying learning rates, momentums and the associated errors. Lambda = 0.001, Batch size = 50.*

Learning Rate	Momentum	Error
0.1	0.1	0.5822
	0.5	0.3138
0.01	0.1	1.7349
	0.5	0.9335
0.001	0.1	3.8211
	0.5	3.5675

with each epoch. This ties the number of epochs to the learning rate. The unoptimized SGD simulations show a large increase in performance beyond 1 epoch and then a slower decline after about 250 epochs or so, as indicated by appendix A7 18. An SGD with an epoch of 1 is similar to a standard GD in the code used, though it is important noting that the learning rates are different due to the decay in the SGD. Using the same parameters for both GD and SGD simulations apart from using a learning rate of 0.0725 for both simulations yields a net error of 0.562 for GD and 0.194 for SGD.

Comparing the effect of momentum on SGD and GD shows that a higher momentum value of 0.5 was optimal for SGD as opposed to the much lower value of 0.1 for standard GD. Conversely, increasing the batch size follows the same trend for both GD and SGD, with a general decrease in performance as batch sizes is increased up to a point. Interestingly however, increasing the batch size beyond 50 for SGD begins to improve performance as opposed to the sharper decrease in performance between 50 and 100 batch size for GD. Appendix A9 19 shows the results of varying batch sizes for SGD. Table 2 shows the performance of SGD with varying parameters.

Optimizers

Various optimizers were used in an attempt to improve the effectiveness of GD and SGD. AdaGrad was used on both GD and SGD, RMSprop and ADAM were used solely on SGD.

The AdaGrad method on GD shows a massive increase in performance as compared to all other GD methods when including several epochs. Appendix A10 21 displays the decrease in net error from 3.13624

Table 2: *Stochastic gradient descent with varying epochs, batch sizes and the associated errors. Lambda = 0.001, Momentum = 0.5*

Epochs	Batch Size	Error
	1	0.0865
1	50	0.1936
	100	0.3395
	1	0.0091
250	50	0.0079
	100	0.0105
	1	0.0115
500	50	0.0118
	100	0.0084
	1	0.0104
1000	50	0.0116
	100	0.0113

for 1 epoch and a learning rate of 7.250e-2 to a net error of 4.400e-4 for 250 epochs and a much lower learning rate of 9.903e-4. The efficacy of the method does slowly decrease after 250 epochs as the learning rate decreases.

Using the AdaGrad optimizer on SGD shows a similar trend to when it is used on GD, though with much worse performance. Keeping all other parameters the same, AdaGrad SGD has an exceedingly poor performance at 1 epoch with a net error of 3.8033 (appendix A12 23) whereas SGD at 1 epoch results in a net error of 0.1936. The method improves significantly at higher epochs, reaching a net error minimum of 4.910e-3 at 250 epochs whereas SGD only has a error of 7.969e-3 at 250 epochs. In addition to this, AdaSGD also yields better values at higher batch sizes, 6.088e-3 at batch size 100 while SGD gives a value of 1.110e-2 for the same batch size (appendix A11 22).

The RMSprop optimizer performed worse than AdaGrad for all epochs but with a significant improvement when the batch size is 1. RMSprop did however provide better values than standard SGD for higher batch sizes as well as higher numbers of epochs as seen in appendices A14 25 and A15 26.

The final optimizer tested, ADAM, resulted in quite low net error values across most epochs and batch sizes. It performed better than RMSprop throughout each epoch range and batch size range as well as significant improvement at a batch size of 1. When compared to AdaGrad, ADAM has noticeably lower net error values for epochs above 250 as seen in appendix A16 27. It also performs exceedingly well at a batch size of 1 and slightly better at higher batch sizes than AdaGrad (appendix A17 28). Overall, ADAM seems to have provided the most results with lower net error values and proves to be one of the better optimizers. Table 3 shows the various optimizers as compared to the unoptimized SGD. The parameters have been chosen to give the lowest values for error for each of the methods.

Table 3: *Optimizers with parameters chosen to produce the best results. Lambda = 0.001, Momentum = 0.5 for Unoptimized SGD*

Optimizers	Epochs	BS	Error
Unoptimized SGD	250	1	$6.549 * 10^{-3}$
AdaGrad GD	1000	25	$2.569 * 10^{-4}$
AdaGrad SGD	250	50	$4.910 * 10^{-3}$
RMSprop	500	50	$7.482 * 10^{-3}$
ADAM	500	50	$1.932 * 10^{-3}$

Neural Network on Franke Function

In order to perform the best analysis of the data for the later classification problem, we have to tune in all the different parameters of our neural network. These parameters are the usual parameters that we have been working on, such as: the learning rate that we call η and the regularisation hyperparameter λ. Besides that, we want also to figure out the size of our neural network. For this purpose we optimize the learning rate, the regularization hyperparameter, the number of hidden layers that we are going to work on, the activation function and the number of neurons per layer, or in other words, the number of neurons that there are in each hidden layer. For this search, we are going to use the mean squared error (MSE) instead the R^2 coefficient of correlation, since we are not interested in the correlation of the different variables to the final output, specially in the test part. We are interested in how much our results deviate from the theoretical results, and for that purpose MSE is our estimator of choice.

Table 4: *MSE and R^2 for different initialization methods for the biases. Scores were averaged over 3 random seeds. Normal distribution was normed to avoid overflow. The weights were initialized using uniform distribution throughout.*

Distribution	R^2	MSE
Uniform	0.347	0.0573
Normal	0.332	0.0587
Zeros	0.385	0.0537

1. Uniform distribution.
2. Normal distribution.
3. Zeros

The weights were consistently better initialised using a uniform distribution, for that reason we chose to vary the bias initialization instead. Table 3 shows the performance of the various initialization methods.

First the optimal learning rate is found. For this task we are going to run the neural network for different η and calculate their respective MSE, and find which η minimize it. So we plot a graph showing the different MSE score against the η learning rate that we use in order to find the one that minimize it. Figure 2 shows the results of varying the learning rate on MSE, which reaches a minimum at about $10^{-2.75}$.

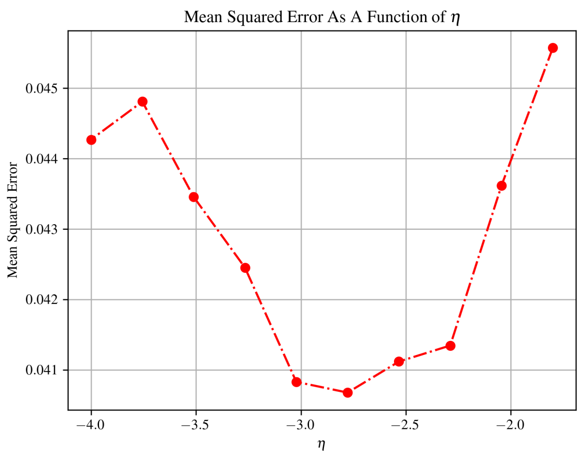

Figure 2: *Plot of the MSE against the learning rate of the NN*

Next, the regularisation hyperparamter λ is discussed and varied similarly to the learning rate. The optimal value for the MSE is found and the corresponding value dor the regularisation hyperparamter is found. Figure 3 shows the results of varying the regularisation hyperparameter λ on MSE. The lowest value MSE begins at about $\lambda = 10^{-10}$ but does not decrease much at smaller values, though there is a large performance improvement from 10^{-5}.

The number of hidden layers is also analyzed with respect to MSE and the results are plotted in figure 4. The MSE is minimized at 4 or 5 hidden layers, but then the performance decreases at higher numbers of hidden layers.

Finally, the number of hidden nodes per layer must also be analyzed and is varied in the same way, with respect to MSE. This is displayed in figure 5,

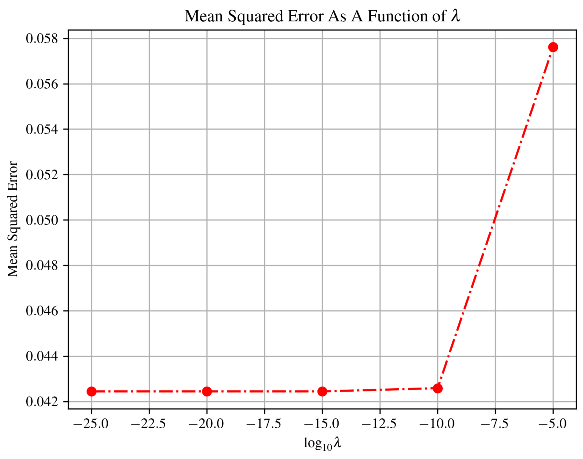

Figure 3: *Plot of MSE against hyperparameter λ of the NN*

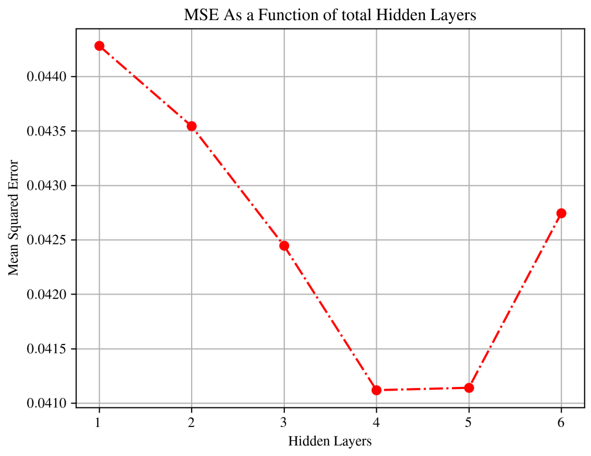

Figure 4: *Plot of MSE against the number of hidden layers in the NN*

which shows a gradual improvement of MSE as the number of hidden nodes increases up to the maximum selected value of 128.

The initialization of the weights and biases is of interest to this project as it can affect the performance of the network. The weights and biases were initialized in three different ways.

To recap, our optimal parameters would be: learning rate $\eta = 10^{-2.75}$, regularisation hyperparameter $\lambda = 10^{-10}$, the number of hidden layers is 4 or 5 and the number of nodes per layer is 128.

Finally, the various activation functions were tested to see which performed the best under the same conditions. The results are displayed in Table 6, showing the MSE and R2 scores of each of the tested activation functions.

The activation functions that performed the best were ReLU and Leaky-ReLU, while Sigmoid and Lin-

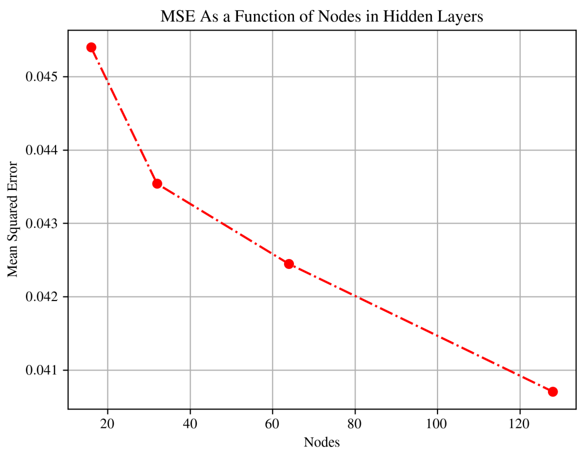

Figure 5: *Plot of MSE against the number of hidden nodes in the NN*

Table 5: *MSE for different combinations of Nodes and Hidden layers. Parameters: $\eta = 1 \cdot 10^{-3}$, $\lambda = 1 \cdot 10^{-15}$ with relu activation function in the hidden layers and linear activation function in the output*

Hidden layers	Nodes	MSE
	32	0.0478
1	64	0.0443
	128	0.0448
	32	0.0452
2	64	0.0435
	128	0.0423
	32	0.0435
3	64	0.0425
	128	0.0407

ear yielded worse results. In order to achieve the best results possible, the best performing activation function, ReLU was chosen along with the most optimal parameters. The number of epochs chosen was 500, learning rate $\eta = 10^{-4}$, regularization parameter $\lambda = 10^{-15}$, number of hidden layers = 4 and number of hidden nodes per layer = 128. The MSE value resulting from this simulation was 0.04877. It is worth noting that the learning rate value used was not actually the optimal value of $\eta = 10^{-3}$ because of NaN errors at that value.

Tensorflow was used to set a benchmark for how the results of the neural network should perform. The same optimal parameters were used and the MSE result was 0.0491, which is slightly worse than the one from out own NN of 0.04877. The MSE can also be compared to the results from project 1, the lowest of which was 0.043 for OLS with a polynomial degree of 5.

Table 6: *MSE and R^2 for different activation functions. All parameters are the same for each of the activation functions.*

Activation Function	MSE$[10^{-2}]$	$R^2 * 10^{-1}$
Sigmoid	5.672	3.846
ReLU	4.025	5.633
Leaky-ReLU	4.027	5.631
Tanh	4.540	5.074
Linear	5.707	3.808

NN Classification on Wisconsin Dataset

The optimal values for the parameters were done throughout the previous analyses, and were used to determine the accuracy of the different activation functions. The same parameters were used for each activation function to try and achieve the most comparable results as well as the highest accuracy. Table 7 shows the results, all used 500 epochs.

Table 7: *Accuracy for different activation functions. All parameters are the same for each of the activation functions.*

Activation Function	Accuracy
Sigmoid	90.91%
ReLU	92.31%
Leaky-ReLU	92.31%
Tanh	92.31%
Linear	92.31%

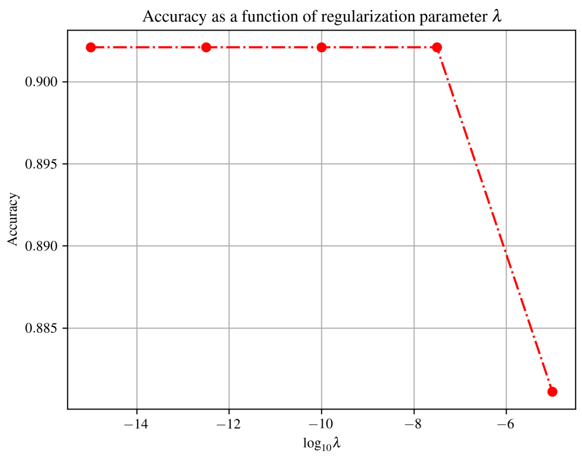

Figure 6: *Plot of accuracy as a function regularization hyperparamter*

All of the activation functions performed equally aside from Sigmoid which yielded slightly worse results. The results from tensorflow using stochastic GD with a learning rate of 0.01 yielded an accuracy

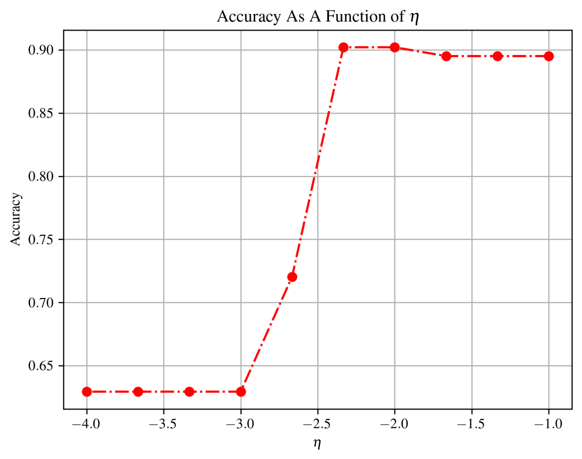

Figure 7: *Plot of accuracy as a function of learning rate*

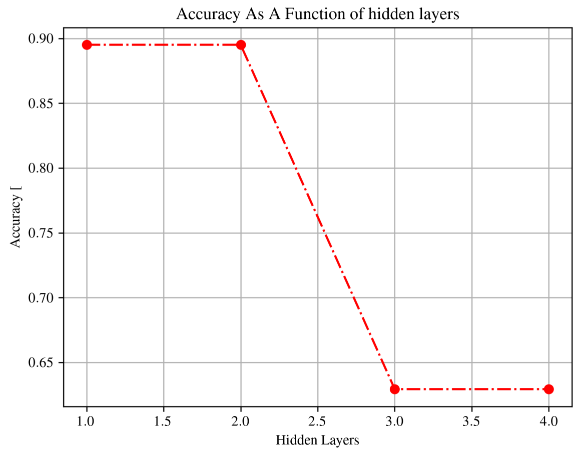

Figure 8: *Plot of accuracy as a function of hidden layers*

of 0.83. The optimal learning rate of 0.001 could not be used because the gradient got stuck (only when using tensorflow). Using the ADAM optimiser from tensorflow resulted in an accuracy of 0.937 which is an improvement over the best performance of our own code

Logistic Regression on Wisconsin Dataset

The code used was not changed, because logistic regression is simply a NN with 0 hidden layers.

Logistic regression was done on the Wisconsin breast cancer data set with varying regularization hyperparameter values as well as different learning rates to find the best results. Figure 10 shows the accuracy of the logistic regression as a function of the regularization hyperparameter. The accuracy is highest below $\lambda = 10^{-10}$ at about 0.909 but drops off dramatically at higher λ values.

Using Scikit learn's logistic regression package we

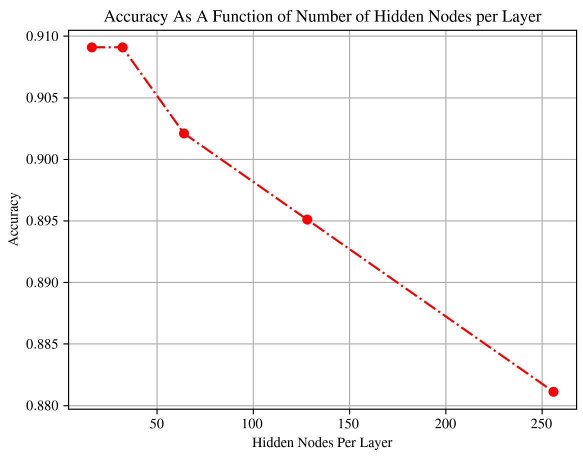

Figure 9: *Plot of accuracy as a function of hidden nodes*

obtained 76% accuracy.

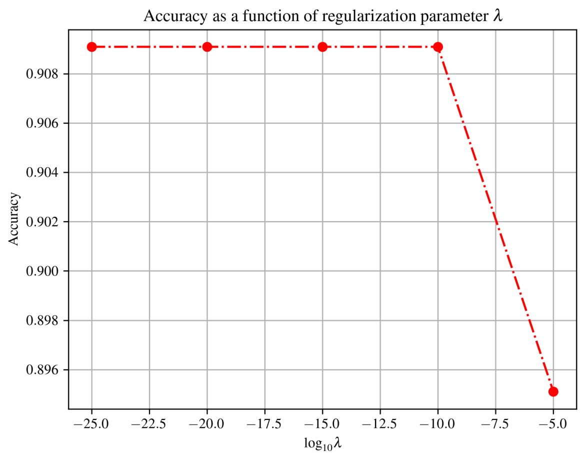

Figure 10: *Accuracy of logistic regression with respect to the regularization parameter λ*

Plotting the accuracy with respect to the learning rate shows that for values of learning rate below 10^{-3}, the accuracy is poor compared to higher learning rates. The optimal shown in the graph is at around $10^{-1.5}$ with an accuracy of about 0.930. This is shown in figure 11.

Discussion

Personal Disclaimer: The analysis in this research was hastily done. The researcher responsible for coding the neural networks (80 % of this project's code) dropped out of the research with no code to show for, two days before the deadline. The code and the analysis was done in few days under considerable pressure.

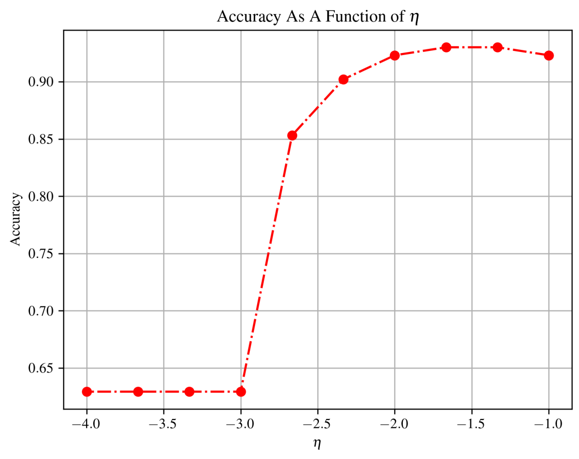

Figure 11: *Accuracy of logistic regression with respect to the learning rate η*

GD, SGD and Optimizers

Overall, the methods used in the gradient descent, SGD and the optimizers were satisfactory in the results presented. RMSprop produced results that were slightly poorer than the initial prediction, this could perhaps be due to poor code implementation. Unfortunately, that could be said for the majority of this section as there was a lack of coherence between the code used to produce results for the different methods and made it possible that some of the results were skewed. The way in which the results were displayed also left something to be desired.

The time constraints on the project made it difficult to properly display the results in an ideal way, specifically making heatmaps and altering parameters at the same time for a more robust analysis. It would have been more prudent to assess the time constraints more accurately, though this was difficult with all of the unexpected delays and reduction of group members towards the end.

Neural Network on Franke Function

To ensure more robust results, we initialized using 3 different seeds rather than relying on an outlier. However, if computation power/time is not an issue, this analysis would be more robust with more averages seed runs.

The Wisconsin Dataset has a lot of co-dependence in the features given the inconsistent ranges, it is beneficial to standardize the data such that each feature has a zero mean and unit variance.

The MSE scores for the different activation functions indicate that the ReLU activation function has the best performance of all those tested. This is not unexpected as it is one of the most widely used in the world. The downside of ReLU is that the negative values given to the activation function are set to 0, fortunately for this project, we do not pass any negative values to the activation functions. The similar performance of Leaky ReLU was predictable as it is designed to make up for the shortcoming of ReLU nulling negative values, though it was expected that the performance of Leaky ReLU would be the exact same as that of the ReLU as there are no negative value inputs.

The sigmoid functions performs poorly relative to most other functions. This could be due to the little change in output values with respect to changes on the input. The tanh function yielded better results than sigmoid but not quite as good as the ReLU function. It is worth mentioning that our analysis is speculative at best, given that machine learning of this type is mostly black box.

This analysis shows as well that more is not better. Even though there are a lot of machine learning packages out there, it is still essential to vary, among other things, the number of layers and hidden nodes per layer etc.. The optimal parameters are dataset specific and not performing this kind of analysis would be indefensible.

NN Classification on Wisconsin Dataset

The classification analysis shows that the Sigmoid function performed the worst compared to all the other methods. This is not unexpected, though the other activation functions should not have had the exact same performance bar ReLU and Leaky-ReLU. In addition to this, the accuracies are too high to be reasonable, even higher than the benchmarks of tensorflow. The most likely cause to both the too-high accuracy and the same results shared among most of the activation functions is that there is some sort of computation mistake in the code.

One important detail relevant to this and previous subsection is how computationally expensive using Neural Networks is. The dataset we have is not that complicated but it took a very long time.

This research could have been improved by showing tables of how long it took to run certain models, this way we have a quantitative description of the scale of computation expensiveness.

Logistic Regression on Wisconsin Dataset

The results from the logistic regression show that accuracy decreases with a lower value for the regularization hyperparameter, even at very low values, suggesting that it may have a negative effect on the logistic regression. This is likely an issue as some regularization should in theory improve the performance at certain points. The effect of the learning rate on accuracy is expected due to higher learning rates resulting in better performance. The results are untrustworthy however, due to the too-high values for accuracy. It is expected that logistic regression should perform worse than neural networks which was not evident from the results. The results were also higher than that of Scikitlearn's which is highly unlikely. This is again, likely due to mistakes in the code.

General discussion

This research could have been much more effective and robust had it been compared to established articles/papers. The chosen datasets are not niche and with some searching, could have been compared. Having said that, the time constraint placed on us meant some very important parts of research had to be sacrificed, such as comparison to established science (seeing if we can replicate). Standalone research (with no citations of other relevant papers) is generally bad research.

One more area of improvement is the displaying of results. Some of the parameters we were optimizing were interdependent. Meaning that a heat map would have been favorable. As an example, instead of plotting MSE of hidden layers and hidden nodes separately, we plot both on x and y axis with a heat map, indicating where the sweet spot is. Having said that, we hypothesize that the improvements would be minor at best.

Conclusion

The main takeaway from this research is the importance of analysis and optimization of parameters. Every dataset is different and we do have some general ideas in which model should work best, however, it is always worth running thorough analysis to ensure that one is using optimal values, the optimal model (and parameters).

In addition, it was observed that Neural Networks performed very well on classification compared to other methods. However, for regression problems they were computationally expensive and with not much better scores, resulting in poor performance to cost.

Knowing this, it might be wise for ML practitioners to attempt to solve a problem at hand using regression problems and build up from there to neural networks, rather than the other way around.

References

[1] T. Hastie, R. Tibshirani, and J. Friedman., *The Elements of Statistical Learning.*, Springer, New York, 2009.

[2] Süli, Endre and Mayers, David F., *An Introduction to Numerical Analysis*,Cambridge University Press, 2003.

[3] Kevin Murphy, *Machine Learning: A Probabilistic Perspective*, MIT Press, 2022.

[4] Bonnans, J.F. and Gilbert, J.C. and Lemarechal, C. and Sagastiz, *Numerical Optimization: Theoretical and Practical Aspects*, Springer Berlin Heidelberg, 2013.

[5] Bishop, Christopher M., *Pattern Recognition and Machine Learning*, Springer, New York, 2016.

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

Appendix

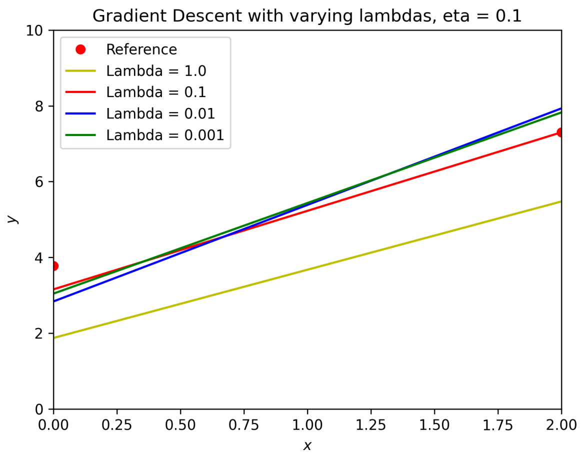

Figure 12: *A1, Plot of Ordinary Gradient Descent with varying hyperparameter λ and constant learning rate η = 0.1*

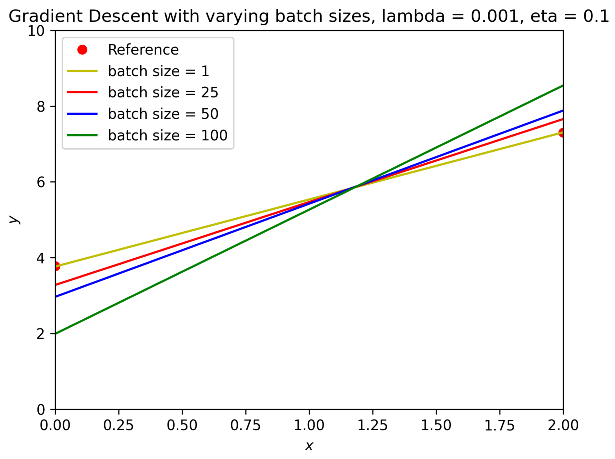

Figure 14: *A3, Plot of Ordinary Gradient Descent with varying batch size, and constant learning rate η = 0.1 and hyperparameter λ = 0.001*

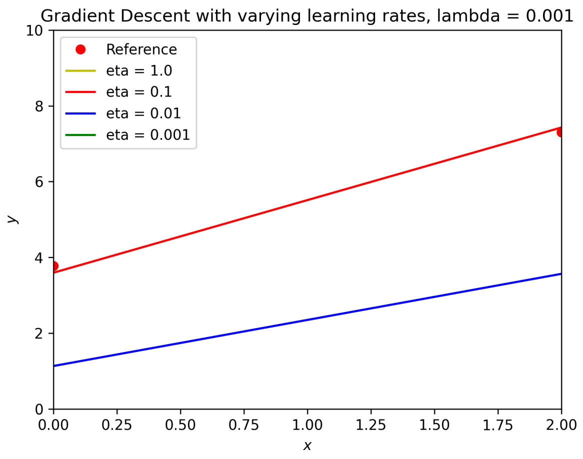

Figure 13: *A2, Plot of Ordinary Gradient Descent with varying learning rate η and constant hyperparameter λ = 0.001*

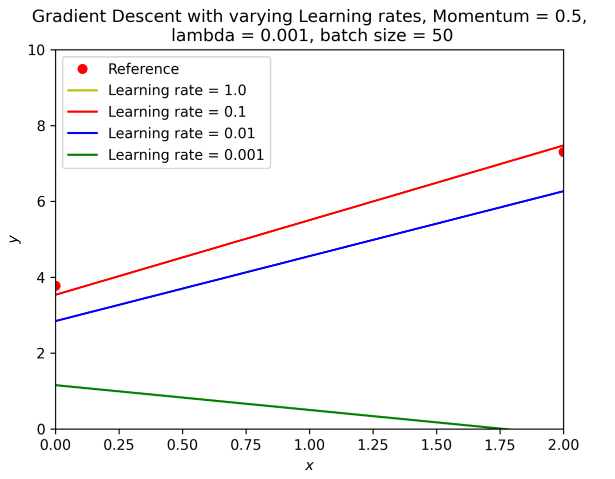

Figure 15: *A4, Plot of Ordinary Gradient Descent with momentum varying learning rate η and constant hyperparameter λ = 0.001, batch size = 50 and momentum = 0.5*

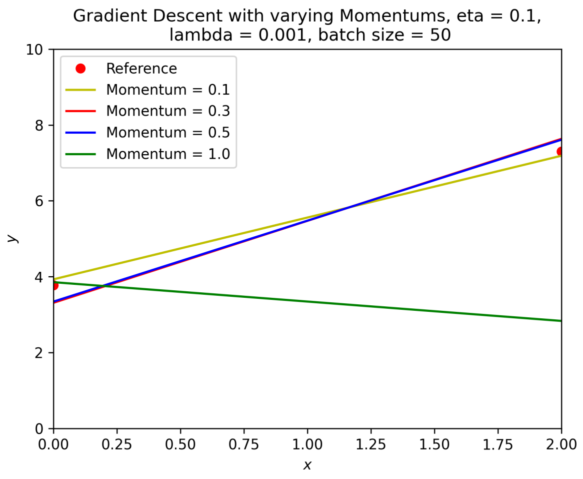

Figure 16: *A5, Plot of Ordinary Gradient Descent with momentum varying its momentum and constant hyperparameter λ = 0.001, learning rate η = 0.1 and batch size = 50*

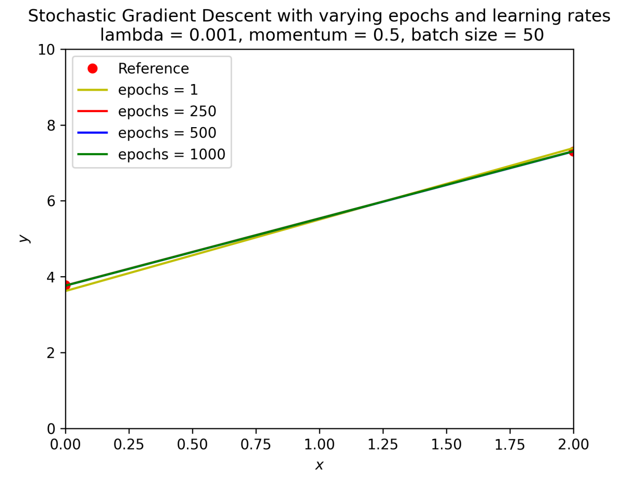

Figure 18: *A7, Plot of Stochastic Gradient Descent with varying epochs, and constant learning rate η = 0.1, hyperparameter λ = 0.001, momentum = 0.5 and batch size = 50*

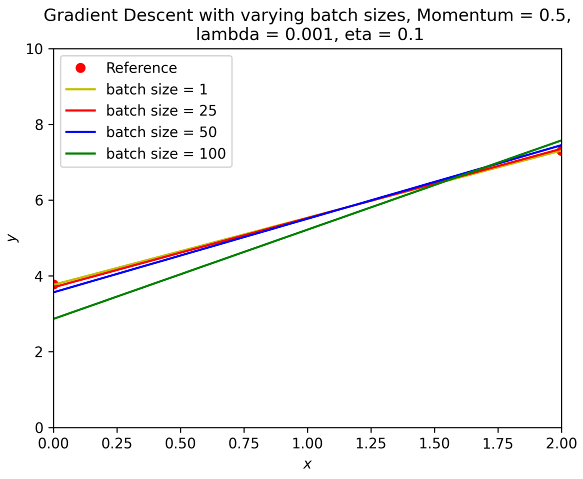

Figure 17: *A6, Plot of Ordinary Gradient Descent with moment varying the batch size, and constant learning rate η = 0.1, hyperparameter λ = 0.001 and momentum = 0.5*

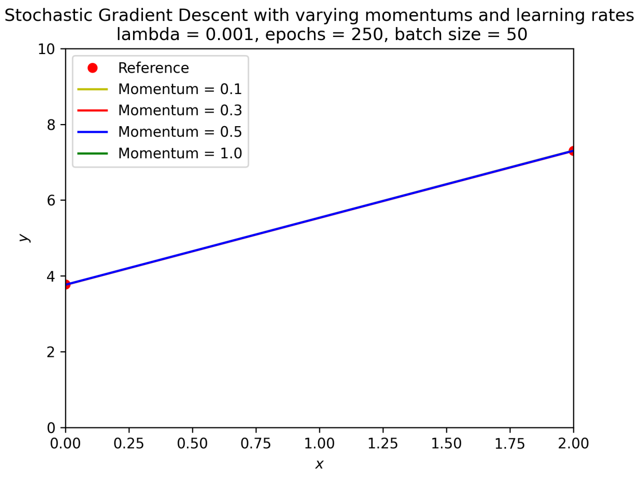

Figure 19: *A8, Plot of Stochastic Gradient Descent with varying momentum, and constant learning rate η = 0.1, hyperparameter λ = 0.001, epochs = 250 and batch size = 50*

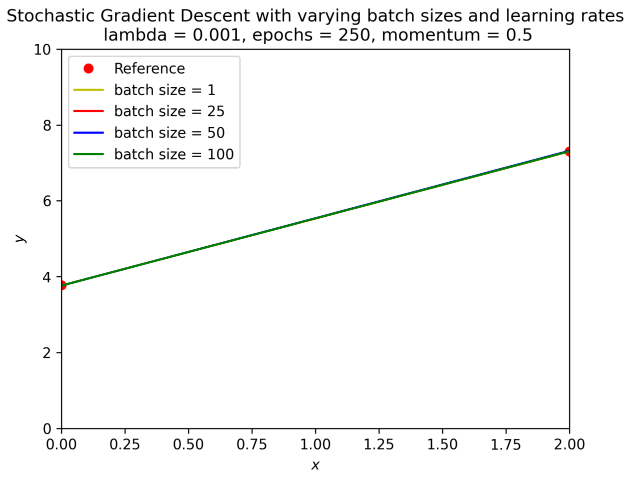

Figure 20: *A9, Plot of Stochastic Gradient Descent with varying batch size, and constant learning rate $\eta = 0.1$, hyperparameter $\lambda = 0.001$, epochs $= 250$ and momentum $= 0.5$*

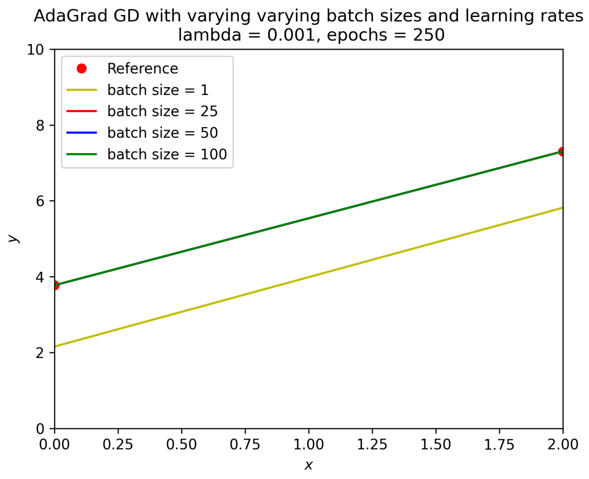

Figure 22: *A11, Plot of AdaGrad Gradient Descent with varying batch size, and constant learning rate $\eta = 0.1$, hyperparameter $\lambda = 0.001$ and epochs $= 250$*

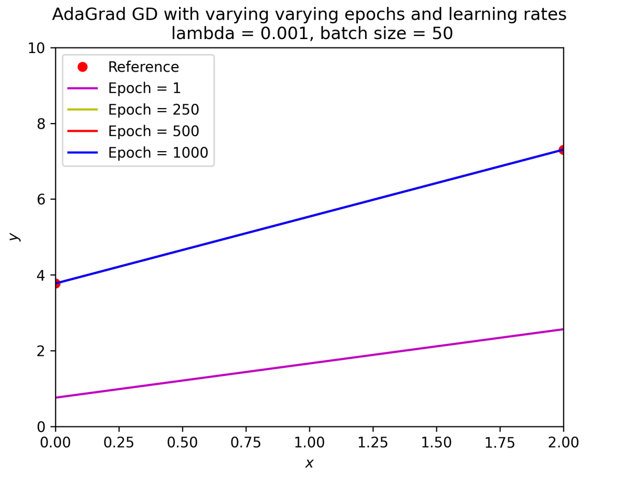

Figure 21: *A10, Plot of AdaGrad Gradient Descent with varying epochs, and constant learning rate $\eta = 0.1$, hyperparameter $\lambda = 0.001$ and batch size $= 50$*

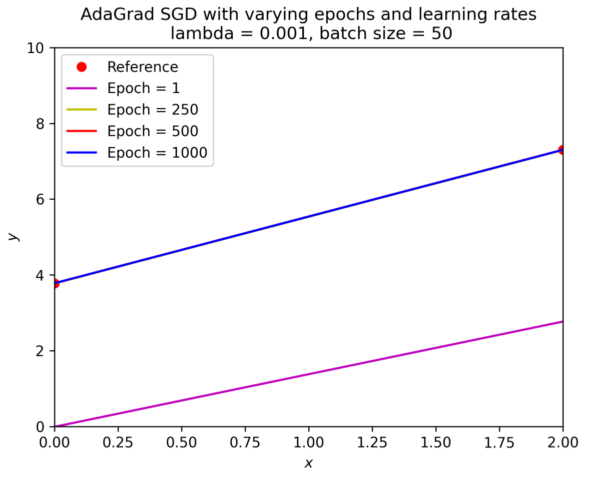

Figure 23: *A12, Plot of AdaGrad Stochastic Gradient Descent with varying epochs, and constant learning rate $\eta = 0.1$, hyperparameter $\lambda = 0.001$ and batch size $= 50$*

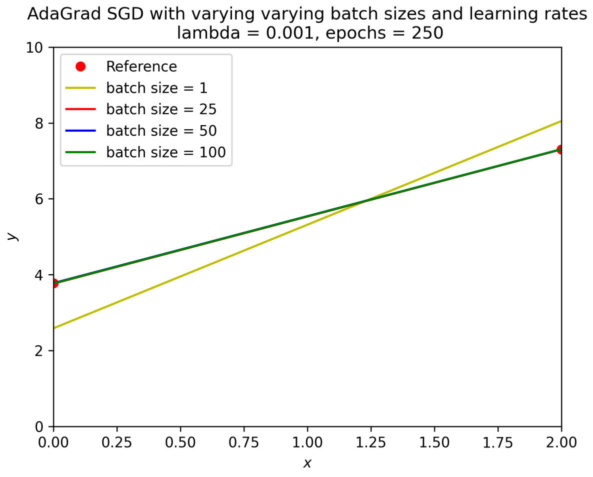

Figure 24: *A13, Plot of AdaGrad Stochastic Gradient Descent with varying batch size, and constant learning rate η = 0.1, hyperparameter λ = 0.001 and epoches = 250*

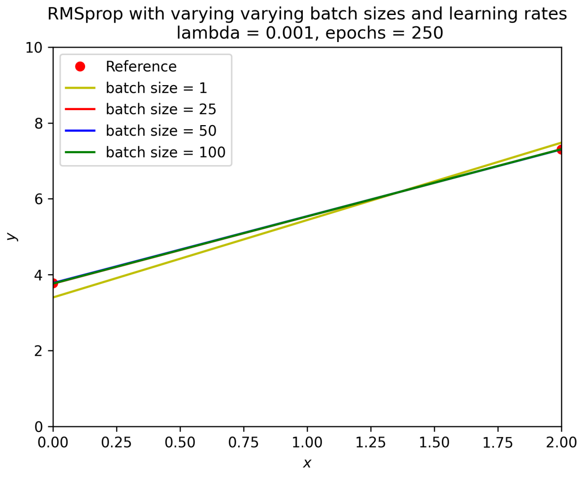

Figure 26: *A15, Plot of RMSprop Stochastic Gradient Descent with varying batch size, and constant learning rate η = 0.1, hyperparameter λ = 0.001 and epochs = 250*

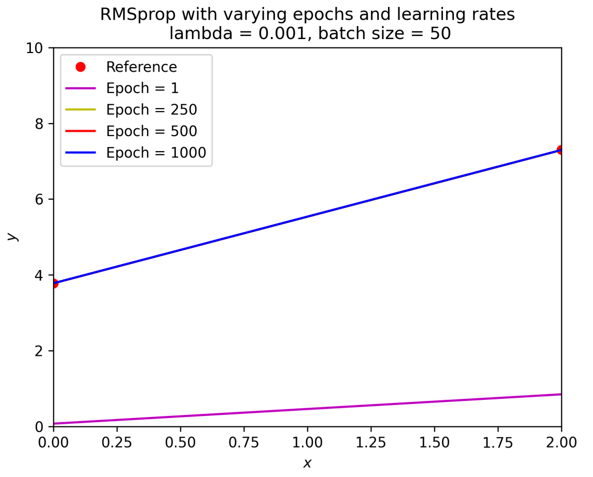

Figure 25: *A14, Plot of RMSprop Stochastic Gradient Descent with varying epochs, and constant learning rate η = 0.1, hyperparameter λ = 0.001 and batch size = 50*

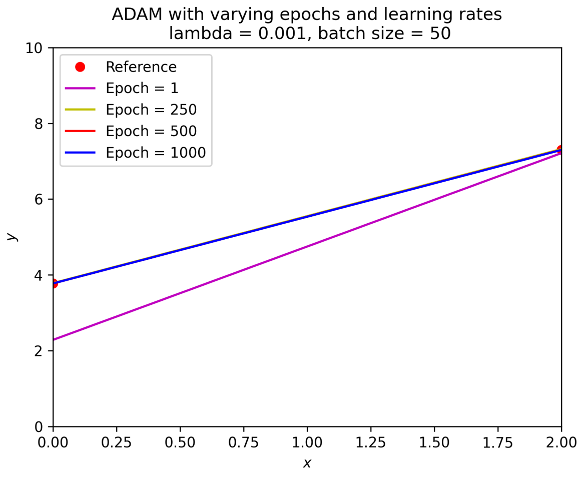

Figure 27: *A16, Plot of ADAM Stochastic Gradient Descent with varying epochs, and constant learning rate η = 0.1, hyperparameter λ = 0.001 and batch size = 250*

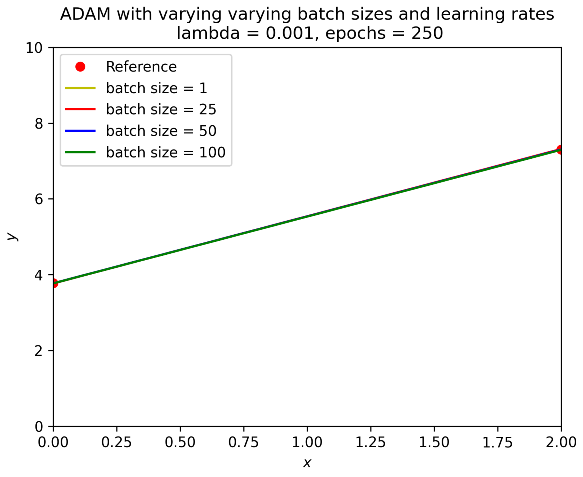

Figure 28: *A17, Plot of ADAM Stochastic Gradient Descent with varying batch size, and constant learning rate $\eta = 0.1$, hyperparameter $\lambda = 0.001$ and epochs = 50*

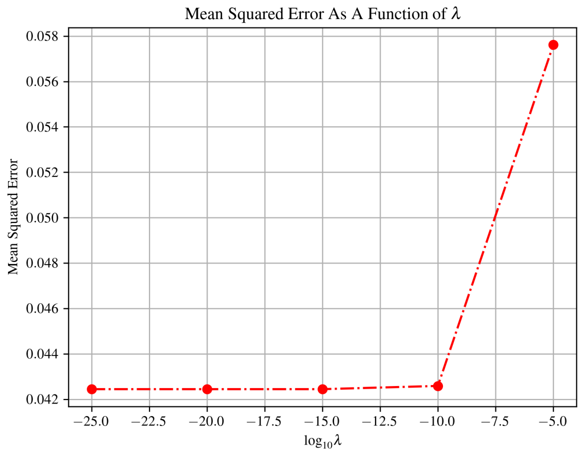

Figure 30: *B2, Plot of the MSE against the hyperparameter λ of the NN, with constant learning rate η, number of hiden layers and number of nodes per layer*

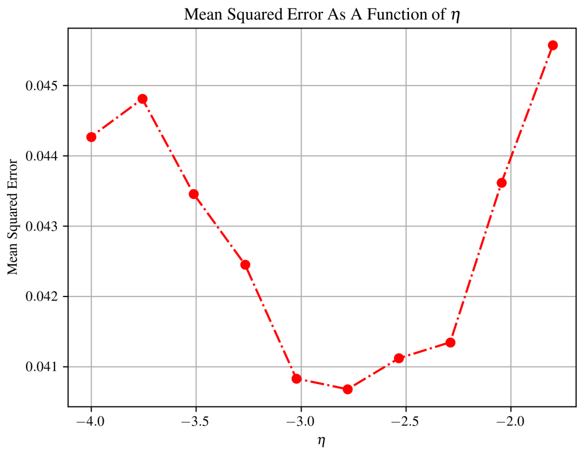

Figure 29: *B1, Plot of the MSE against the learning rate η of the NN, with constant hyperparameter λ, number of hiden layers and number of nodes per layer*

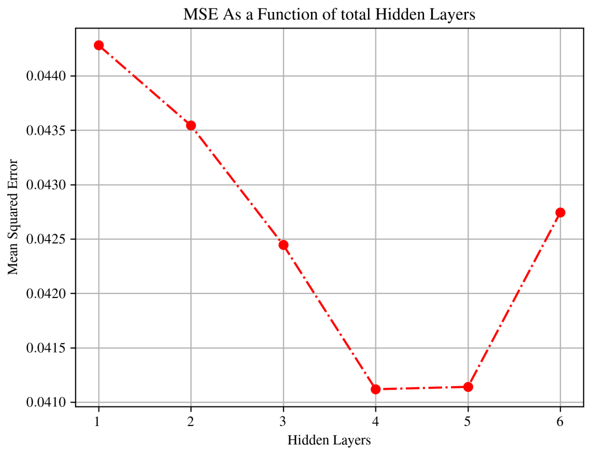

Figure 31: *B3, Plot of the MSE against the number of hidden layers of the NN, with constant learning rate η, hyperparameter λ and number of nodes per layer*

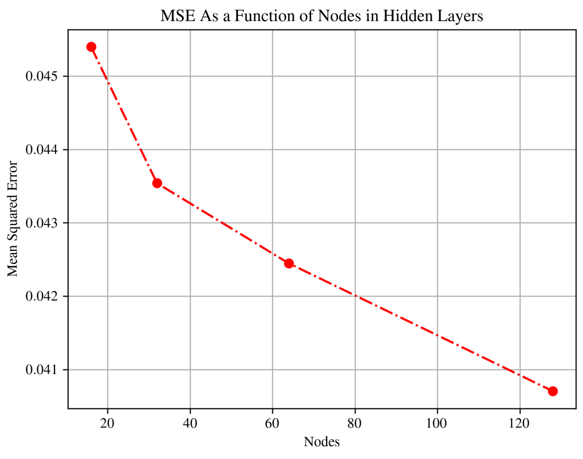

Figure 32: *B4, Plot of the MSE against the number of nodes per layer of the NN, with constant learning rate η, hyperparameter λ and number of layers*

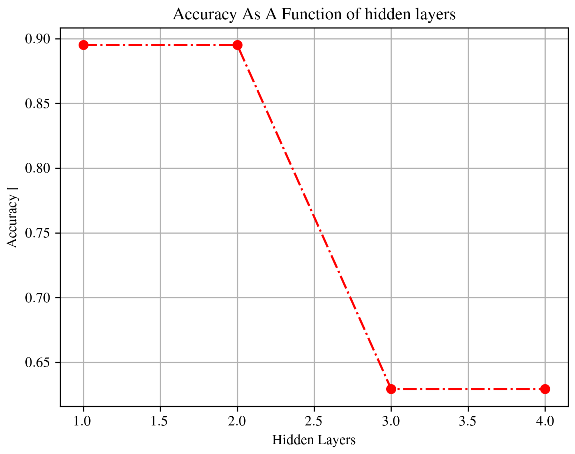

Figure 35: *D3, Plot of accuracy as a function of hidden layers.*

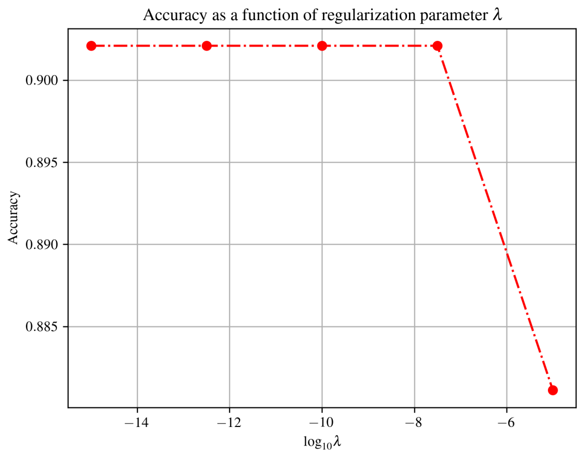

Figure 33: *D1, Plot of accuracy as a function of the regularization hyperparameter λ.*

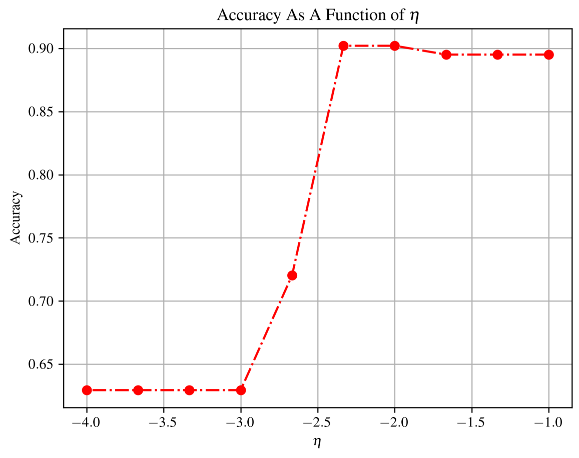

Figure 34: *D2, Plot of accuracy as a function of learning rate.*

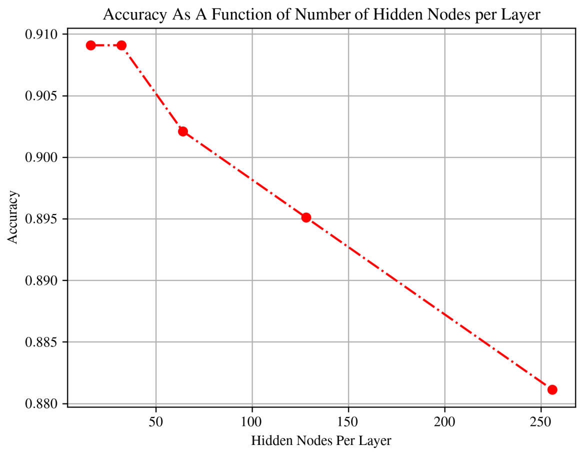

Figure 36: *D4, Plot as a function of hidden nodes.*