Comparative Analysis of Neural Networks and Traditional Numerical Methods for Solving Differential Equations and Eigenvalue Problems

Kanaparthi Sai Chandhu
saichandukanaparthi@gmail.com

Vinay Gadila
vinaygifcab0@gmail.com

***Abstract*—This paper presents a comparative study between neural network-based approaches and traditional numerical methods for solving differential equations and eigenvalue problems. We first applied a neural network to solve the one-dimensional diffusion equation and compared the results with those obtained using the forward Euler iteration scheme. While the neural network provided flexibility in grid selection and was not constrained by stability criteria, it was less accurate and slower than the iterative method. Subsequently, we employed a neural network to determine the largest and smallest eigenvalues of a symmetric, real matrix. Though the neural network successfully converged to the correct eigenvalues and eigenvectors, it exhibited challenges in convergence time and accuracy, particularly when compared to standard library routines. The findings demonstrate that while neural networks can replicate the results of traditional methods, they fall short in terms of computational efficiency and precision. Therefore, their use in these applications is limited by the trade-off between flexibility and performance.**

I. Introduction

Classification problems are pivotal in machine learning, serving as the foundation for numerous applications across diverse fields. These applications range from medical diagnostics, where models analyze clinical data to predict diseases such as cancer [1], to materials science, where machine learning enhances density functional theory predictions to determine energy-efficient atomic structures [2]. Similarly, handwriting recognition leverages classification methods for tasks like converting handwritten text into digital formats [3]. In recent years, neural networks have emerged as a flexible and powerful tool for tackling classification problems, outperforming traditional techniques in many scenarios [4].

In this work, we explore the application of classification techniques to a dataset on breast cancer, aiming to distinguish between malignant and benign cases. This analysis employs two common classification methods: logistic regression and neural networks. Logistic regression, a robust statistical method, models the probability of an outcome based on input features [5]. Neural networks, on the other hand, offer adaptability and scalability, excelling in complex and nonlinear data scenarios [6].

The foundation of both logistic regression and neural networks lies in optimization, specifically the minimization of a loss function. Optimization techniques, particularly gradient descent, have been widely studied for their convergence properties and computational efficiency [7]. To validate our methods, we initially tested our neural network implementation on linear regression using the Franke function, which is frequently employed for benchmarking numerical methods [8]. Subsequently, we applied both logistic regression and neural networks to the breast cancer dataset, comparing their performance in terms of accuracy and computational efficiency [9]. This study also contextualizes the broader implications of using neural networks in numerical problems, such as solving differential equations and eigenvalue problems. Recent research highlights both the promise and limitations of neural networks in these areas, particularly when compared to traditional numerical methods like forward Euler schemes for differential equations and library routines for eigenvalue problems [10]. These comparisons underscore the trade-offs between flexibility and performance, a critical consideration in selecting computational tools for scientific and engineering applications.

II. Theory and Methods

A. Logistic Regression

Logistic regression models the probability that a set of data features, or input variables, $\boldsymbol{x}^{(i)} = \{1, x_1, ..., x_p\}$ leads to a certain response, y_i. We organise these observations in a dataset $G = \{\mathbf{x}^{(i)}, y_i\}_{i=1}^{n}$. In logistic regression, the response is called a class, and the index k indicates which class the outcome belongs to, where k can take on the values $k = 1, 2, ..., K$, with K being the total number of classes. The simplest case is $K = 2$, with $y_i \in [0, 1]$, and the probability given by the sigmoid function (also called the logistic function)

$$\sigma(t) = \frac{1}{1 + e^{-t}}. \tag{1}$$

The probability that the outcome y_i belongs to class 1 is therefore given by the logistic model,

$$P(y_i = 1|\boldsymbol{x}, \beta) = \frac{1}{1 + e^{-\beta \cdot \boldsymbol{x}}}. \tag{2}$$

Similarly, the probability of y_k belonging to the class 0 is given by

$$P(y_i = 0|\boldsymbol{x}, \beta) = 1 - P(y_i = 1|\boldsymbol{x}, \beta). \tag{3}$$

We have used the standard probability notation; $P(y|x)$ is the probability of observing y given x. The regression coefficients of our model are given by the vector $\beta = (\beta_0, \beta_1, ..., \beta_p)$. Our model is, therefore, similar to the standard linear regression models,

$$\beta \cdot \boldsymbol{x} = \beta_0 + \sum_{i=1}^{p} \beta_i x_i. \tag{4}$$

In order to train the model, we need to define a cost function. The best candidate for logistic regression is minimizing the cross entropy. To arrive at it, we use the principle of maximum likelihood. We follow the derivation in [11, page 120] throughout this section. The likelihood, L, of all possible outcomes in G can be approximated by the product of the probability of all outcomes. In other words,

$$L(\beta) = \prod_{i=1}^{n} P_i^{y_i} (1 - P_i)^{1-y_i}, \tag{5}$$

where we have used the notation

$$P_i = P(y_i = 1 | \boldsymbol{x}, \beta). \tag{6}$$

The optimal regression coefficients β are, therefore, the ones that maximise the likelihood, L. In this case,e it is mathematically easier to work with the logarithm of the likelihood,

$$l(\beta) = \log[L(\beta)], \tag{7}$$

because the logarithm converts the products into summations. The fact that the logarithm is a strictly monotonic function assures that $L(\beta)$ and $l(\beta)$ have the same solutions for the maximization problem. We can therefore, define our cost function as the negative logarithm of the likelihood,

$$\begin{aligned} C(\beta) &= -l(\beta) \\ &= -\sum_{i=1}^{n} (y_i \log(P_i) + (1 - y_i) \log(1 - P_i)). \end{aligned} \tag{8}$$

This is known as the cross entropy.

We can also study the cost function with an additional regularisation parameter λ, similar to ridge regression. This is done by adding an L_2-term to the cross entropy, giving

$$C_\lambda(\beta) = -l(\beta) + \lambda \|\boldsymbol{\beta}\|^2, \tag{9}$$

Minimizing $C(\beta)$ is equivalent to computing the gradient and equating it to zero. Taking the gradient with respect to β and writing out the result in matrix form, we obtain,

$$\nabla_\beta C(\beta) = -\boldsymbol{X}^T (\boldsymbol{Y} - \boldsymbol{P}) \tag{10}$$

Here, $\boldsymbol{X}$ is the design matrix, with the vector $\boldsymbol{x}^{(i)}$ on row i, $\boldsymbol{y} = (y_1, ..., y_n)$, and $\boldsymbol{P} = (P_1, ..., P_n)$. The minimization equation, $\nabla_\beta C(\beta) = 0$, has no analytical solutions, and we will therefore apply gradient descent methods in order to find the optimal β parameters.

B. Gradient Descent

Gradient descent methods are widely used in machine learning problems to find the minimum of the cost function. The minimization equations rarely have analytical solutions, and numerical methods falling under the "gradient descent"-umbrella are often used to compute the minima. We will present some of them here.

We will consider a general dataset X, with some model function $f(z)$ depending on the variables z. As with the case for logistic regression, we have defined a cost function $C(X, f)$. We will fit our model by finding the parameters z that minimize C.

The standard form of gradient descent is called steepest descent. The derivation follows the lecture notes [1]. If we want to minimize the function $F(\boldsymbol{x})$ that evaluates the vector $\boldsymbol{x} = (x_1, ..., x_n)$, the function F decreases the fastest in the direction of the negative gradient. Thus, if we want to find the $\boldsymbol{x}$ that minimises the F iteratively, we employ the algorithm

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \gamma_i \nabla F(\boldsymbol{x}_i). \tag{11}$$

Here, $\gamma_i > 0$ is called the learning rate. For γ_i small enough we are guaranteed that $F(\boldsymbol{x}_{i+1}) \leq F(\boldsymbol{x}_i)$. This naive, first version of gradient descent is simple to implement but comes with significant disadvantages. Most prominently, the minimum found by gradient descent is not guaranteed to be a global minimum; the algorithm is heavily dependent on the learning rate, and a gradient is computationally expensive to compute. Therefore, several improvements to the gradient descent algorithm have been developed that aim to tackle these issues. We will go through some of them here, including the stochastic gradient descent (SGD) method that introduces randomness, momentum-accelerated methods, and methods that tune the learning rate.

1) Stochastic Gradient Descent (SGD): The crucial observation of the SGD method is that, in many cases, including Eq. (8), the cost function $C(\beta)$ can be written as a sum over n independent data points,

$$C(\beta) = \sum_{i=1}^{n} c_i(\boldsymbol{x}_i, \beta) \tag{12}$$

The implication is, therefore, that the gradient also can be written as a sum of n independent gradients.

$$\nabla_\beta C(\beta) = \sum_{i=1}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta) \tag{13}$$

The randomness in the *stochastic* gradient descent method is now introduced by dividing the n independent data points into subsets called mini-batches. With n datapoints and M points in each minibatch, there are n/M total minibatches. We denote the minibatch B_k, with $k \in [1, n/M]$. The total gradient is now approximated by only computing the gradient of one randomly chosen minibatch in each iteration step.

[1] Week 39

$$\sum_{i=1}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta) \approx \sum_{i \in B_k}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta) \tag{14}$$

The minimization step in SGD, therefore reads

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta_j). \tag{15}$$

The SGD method offers two major improvements to simple GD. Firstly, only computing the gradient of the randomly chosen minibatch in each step decreases computational cost, and cost reduction scales with system size. Secondly, randomly picking the minibatch in each step minimizes the chance of getting stuck in a local minimum.

2) Momentum accelerated methods: One of the main drawbacks of the SGD method is that it has difficulties navigating in parameter space when the gradient is much steeper in one direction. This can lead to oscillations that are hard to combat using only SGD. The solution is to introduce a momentum term, which stores the gradient computed in the previous step. This accelerates the minimization in the direction that is consistently decreasing, making rapid arbitrary oscillations less important. The minimization step is updated to

$$\beta_{j+1} = \beta_j - \eta \sum_{i \in B_k}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta_{j-1}) - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta_j), \tag{16}$$

Where η is the strength of the momentum term, usually with a value close to unity, we use a value of $\eta = 0.99$.

3) Tuning the learning rate: Momentum accelerated methods improves the SGD, but they can still do better. In particular, a lot of information is lost when we throw away the history of the computed gradients. We therefore introduce two methods that keep track of the moments of the gradients we compute; RMS prop and Adam. RMS prop keeps track of the second moment of the gradients, while Adam keeps track of both the second and the first moment.

For ease of notation, we denote the gradient at each iteration t by

$$g_t = \nabla_\beta C(\beta_t). \tag{17}$$

In RMS prop, the second moment of the gradient is given by

$$s_t = \mathbb{E}[g_t^2]. \tag{18}$$

As we want to keep track of past gradients as well, we use a running average of the second moments given by

$$s_t = \alpha s_{t-1} + (1-\alpha) g_t^2. \tag{19}$$

The coefficient α controls the averaging time of the second moment. Typically, the value is about $\alpha = 0.9$. The minimization step is given by

$$\beta_{t+1} = \beta_t - \frac{\gamma_t}{\sqrt{s_t + \epsilon}} g_t. \tag{20}$$

The coefficient γ_t is as before the learning rate, and ϵ is a small constant introduced to prevent divergences. Typical values are of the order of $\epsilon \sim 10^{-8}$. As is evident from the equation above, RMS prop effectively tunes the learning rate by virtue of the second moment. In directions where the norm of the gradient is large, the learning rate is reduced, and this speeds up convergence by allowing for the use of large learning rates in flat directions.

ADAM is an evolution of RMS prop that includes both the second and first moments. As above, we denote the gradient by g_t. The first moment is defined as

$$m_t = \mathbb{E}[g_t]. \tag{21}$$

Introducing the parameters α_1 and α_2 to control the running averages of the first and second moments, respectively, we compute them as

$$m_t = \alpha_1 m_{t-1} + (1-\alpha_1) g_t, \tag{22}$$

and

$$s_t = \alpha_2 s_{t-1} + (1-\alpha_2) g_t^2. \tag{23}$$

At the first iteration, both m_0 and s_0 are set equal to zero. This implies that the estimates are biased towards zero. This is counteracted by using bias-corrected estimates in the minimization step. The bias-corrected first and second moments are given by

$$\hat{m}_t = \frac{m_t}{1-\alpha_1^t}, \tag{24}$$

and

$$\hat{s}_t = \frac{s_t}{1-\alpha_2^t}. \tag{25}$$

Finally, the minimization step in the Adam algorithm reads

$$\beta_{t+1} = \beta_t - \gamma_t \frac{\hat{m}_t}{\sqrt{\hat{s}_t + \epsilon}}, \tag{26}$$

where again ϵ is a regularisation parameter and γ_t is the learning rate.

C. Error Metrics

In order to compare our results between different methods we will use two different error metrics; the mean squared error (MSE) and the accuracy score. The mean squared error between a prediction, $\tilde{\mathbf{y}}$, and the actual value, $\mathbf{y}$, is given by

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

We will apply MSE to the minimization problems when we test the gradient descent methods and the neural network. Meanwhile, the accuracy score will be applied when we test the classification schemes. The score measures the percentage of correctly classified labels. For instance, a score of 0.8 implies that 80% of the data was correctly labeled. The score is therefore given by

$$\text{accuracy} = \frac{1}{n} \sum_{i=1}^{n} \delta(\tilde{y}_i - y_i), \tag{28}$$

where the δ-function returns 1 if the prediction is correctly labelled, and 0 otherwise.

D. Neural Networks

Artificial neural networks (ANN) are a class of algorithms where a collection of nodes, commonly referred to as neurons, are connected by graphs. Neural networks are inspired by how brains function. There are many different versions of ANNs, and herein we will focus on (one of) the most common implementations, namely multilayer perceptrons (MLP). The algorithm will feed the input data to the first layer of neurons before it is propagated through the layers by the graphs connecting them. We employ the feedforward algorithm, implying that information only flows one way through the neuron network.

1) Propagation: The propagation of the information, $\mathbf{p}$, is controlled by a set of weights, $\mathbf{w}$, which determines the strength of the connection between the neurons. This is done in three steps, which are:

- The input information $\mathbf{p}$, which is either the initial information or the output from the previous layer, is scaled with the strength of the connection,
$$\mathbf{p} \longrightarrow \sum_{i}^{n} w_i p_i. \tag{29}$$
- The bias of the neuron is added to the weighted input:
$$\sum_{i}^{n} w_i p_i \longrightarrow \sum_{i}^{n} w_i p_i + b. \tag{30}$$
- The computed argument is processed by the activation function, f, of the neuron to compute the output,
$$\tilde{\mathbf{p}} = f\left(\sum_{i}^{n} w_i p_i + b\right). \tag{31}$$
- The output is the input of the next layer.

Summarised the algorithm can be written out in one equation,
$$\mathbf{p} \longrightarrow f(\mathbf{w}^T\mathbf{p} + b) = \tilde{\mathbf{p}}. \tag{32}$$

2) Activation functions: There are multiple choices for activation functions in the feed-forward neural network, and the choice of activation function partly determines the network behavior. For instance, in our application to regression and classification problems, different activation functions will be chosen for the output layer of the model. Activation functions are needed to introduce non-linearity in the network, as without them, the neural network would simply be a series of linear transformations.

For regression problems, there are multiple possible options. We will apply the sigmoid function, introduced in Eq. (1), the rectified linear unit function (ReLU), and the leaky ReLU functions. The last two are given by
$$\mathrm{ReLU}(t) = \begin{cases} 0 & \text{for } t < 0 \\ t & \text{for } t \geq 0 \end{cases} \tag{33}$$
and
$$\text{Leaky ReLU}(t) = \begin{cases} 0.01t & \text{for } t < 0 \\ t & \text{for } t \geq 0 \end{cases}, \tag{34}$$
respectively. As the output of the last layer in a regression model could be any real number, it does not make sense to add an activation function to this layer.

For the classification problem, we will use the sigmoid function. This choice is suitable for binary classification, where the output is labeled to either 0 or 1. For the output layer, we will simply round the output from the sigmoid to either 0 or 1. For multiclass classification, the generalized choice would be the softmax function.

3) Output and back propagation: After the completion of the feed-forward process, the output of the neural network is compared to the true values of the training data. The comparison is computed using the MSE in regression problems and the accuracy score in classification problems. Afterward, the weights and biases are adjusted, and the process is repeated. One run of the feed-forward algorithm is referred to as an epoch, while the adjustment of weights and biases is referred to as backpropagation.

Comparing the output with the real values of the training data requires a cost function. The cost functions used for the neural network are the same as the ones we have already discussed. We will use the OLS and Ridge functions, studied in Project 1, in regression problems and the cross entropy in Eq. (8) for the classification problems. The objective of the comparison is to minimize the cost function, C. We employ gradient descent methods to do this, calculating the gradient of the cost function and updating the weights and bias in the direction of the steepest descent, essentially applying the gradient descent methods already introduced. If we define the argument of the activation function, f, as
$$t \equiv \mathbf{w}^T\mathbf{p} + b, \tag{35}$$
then the gradient of the cost function of the output layer, L, is given by
$$\epsilon_L = \frac{\partial C}{\partial f}\frac{\partial f}{\partial t_L}. \tag{36}$$
We can then backpropagate this error to the other layers $l = L-1, L-2, ..., 2$ via
$$\epsilon_l = \epsilon_{l+1}\mathbf{w}_{l+1}^T\frac{\partial f}{\partial t_l} \tag{37}$$
The new weights and biases are thereafter calculated as
$$\mathbf{w}_l = \mathbf{w}_l - \gamma\epsilon_l\frac{\partial f}{\partial t_{l-1}}, \tag{38}$$
$$b_l = b_l - \gamma\epsilon_l, \tag{39}$$
where γ is the learning rate.

4) Initialisation of weights and bias: The feedforward neural network is an iterative process, and its general nature means that there is no guarantee for it to converge to a global minimum or even converge at all within the limits of computational time. Picking suitable initial conditions is, therefore, important for the performance of the model. Unfortunately, how to choose good initial conditions has been a poorly understood problem[12].

One known important aspect is that every node should be initialised differently. As the feedforward network is fully connected, two equal nodes will simply stay equal through training and the complexity of the model will be reduced. The simplest way to avoid this is to initialize all nodes randomly, as the chance of getting many equal nodes is negligible.

It has been pointed out that the variance of the input and output from a layer should be approximately equal[13]. In the case where the variance increases through the network, many of the nodes will be saturated (with an activation function like the sigmoid), meaning that the gradient is nearly vanishing. As a result, the learning process slows down. In the other case, when the variance dies out, many of the nodes become similar, and the complexity of the model decreases.

In our analysis, we tested different random distributions to see how the choice affected our network.

E. Data Preparation

Before we analyze our implementation of the neural network, we want to gain an understanding of the various parameters at play. This we do by analyzing various gradient descent methods on the Franke function. The Franke function is modeled on a 1000×1000 grid of random points spanning $x, y \in [0, 1]$. We use $p = 5$ degree polynomials in our model and seek to find the best fit of the Franke function using gradient descent methods. We use the mean squared error (MSE) as the metric to compare the different methods. We learned from work on project 1 that ordinary least squares (OLS) regression worked well on the Franke data, and we therefore compared our analysis to the MSE obtained using OLS regression. We employ scikit-learn's own implementation[2] to compute the OLS result. All data was scaled using scikit-learn's standard scaler [3] before regression and classification. We split our data into a train/test ratio of 80/20.

III. Results and Discussion

A. Gradient Descent Methods

Figures 1 and 2 show results for the SGD and SGDM methods for two different minibatch sizes (64 and 128, respectively). Both methods use the learning rates presented in the legend in the left panel of the figures.

Immediately, several interesting effects are observed. Firstly, for all instances, the general observation is that the MSE approaches the OLS result while the learning rate approaches unity. In other words, increasing the learning rate results in faster convergence. Secondly, the inclusion of momentum gives a better MSE at lower learning rates but does not improve the MSE at the high learning rates that give a good MSE without momentum. However, the inclusion of momentum also implies a faster convergence for all learning rates. Thirdly, as the learning rate increases, we observe rapid oscillations in the MSE. Finally, of the two minibatch sizes tested, the smaller minibatch size gives the smallest MSE, but the advantage is minor.

[2]Scikit-learn linear regression.
[3]Standard Scaler

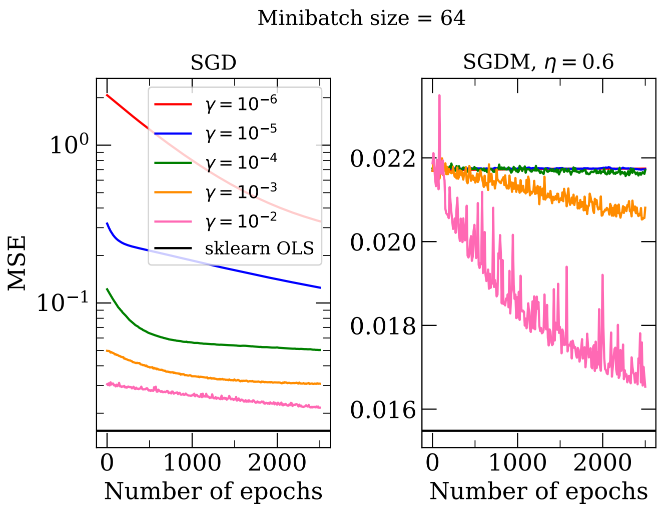


Fig. 1. Comparison of the MSE as a function of the number of epochs for the SGD and SGDM methods, using momentum $\eta = 0.6$ for the latter. The various learning rates are indicated in the legend in the left panel and compared with the OLS result obtained using library methods shown as a horizontal black line. The minibatch size is indicated in the chart title. Note that the y-axis is on a logarithmic scale in the left panel.

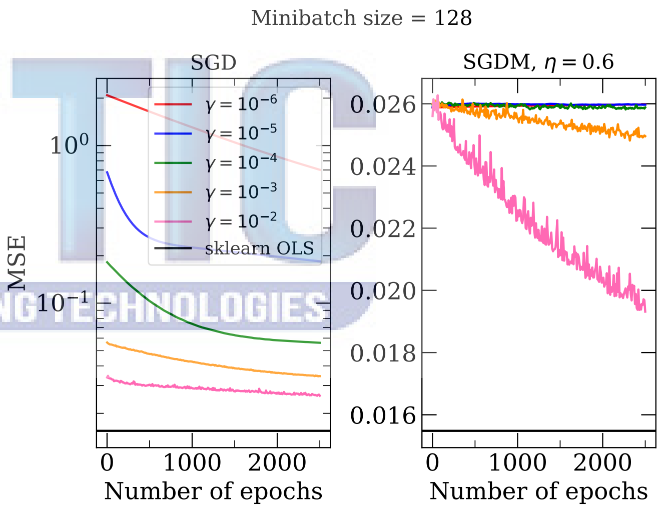


Fig. 2. Comparison of the MSE as a function of the number of epochs for the SGD and SGDM methods, using momentum $\eta = 0.6$ for the latter. The various learning rates are indicated in the legend in the left panel and compared with the OLS result obtained using library methods shown as a horizontal black line. The minibatch size is indicated in the chart title. Note that the y-axis is on a logarithmic scale in the left panel.

Figure 3 shows SGD on the Franke function where we have included an L^2 cost term in the cost function. The two panels in the Figure show the two learning rates that gave the smallest MSE in the previous example. We once again obtain the smallest MSE for the largest learning rate, but it does not outperform the MSE we found using the simple OLS cost function in the previous figures. We also observe an interesting effect in the parameter λ where the MSE initially decreases as λ increases but starts increasing again when λ becomes too large. This is true for both learning rates.

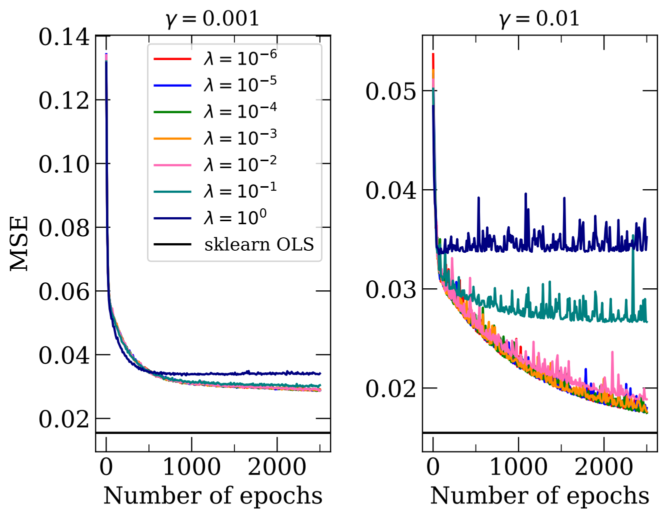


Fig. 3. The two learning rates that performed best for SGD investigated using an L2 penalty parameter λ, as indicated in the legend in the left panel. The results are compared with the OLS result obtained using library methods.

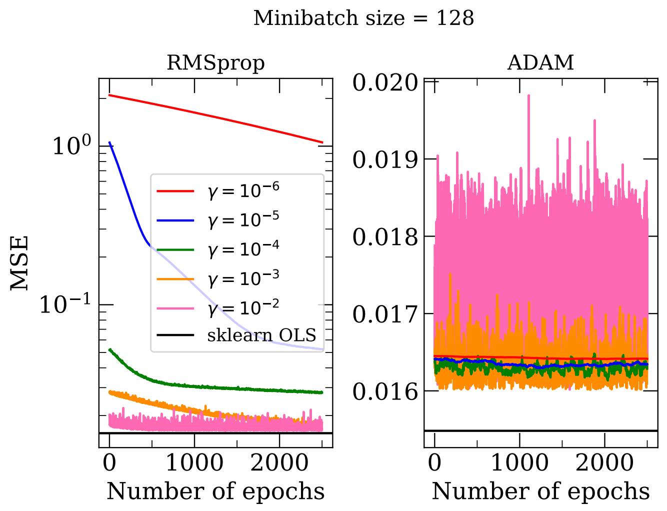


Fig. 5. Comparison of the two SGD methods that tune the learning rate: RMSprop (left) and ADAM (right). The learning rates for both panels are indicated in the legend on the left side. The methods are compared to the OLS regression result. Note that the y-axis has a log scale in the left panel.

Figure 4 investigates the adaptive gradient (adagrad) method with and without momentum. Again, we observe that adding momentum yields a faster convergence, and here, we observe that it also yields a slightly lower MSE for the larger learning rates.

Finally, Figure 5 compares two methods for tuning the learning rates, RMSprop and ADAM. These methods give the smallest MSE we observe, but they also feature the most rapid oscillations with the largest amplitudes.

In all the methods we have investigated, we find that when the gradient descent method has converged, it starts oscillating rapidly, about a minimum. Intuitively, the reasoning for the SGD and SGDM methods with a fixed learning rate is simple. Once the method locates a minimum and converges to it, it is unable to get closer to said minimum due to the finite size of the learning rate. This is often explained with the metaphor of a ball with fixed energy in a frictionless well; it is unable to settle at the bottom of the well due to its finite energy (corresponding to the learning rate), and the energy is not lost to friction. The ball will, therefore, oscillate around the bottom of the well indefinitely. This issue could, however, be addressed by tuning the learning rate, making it decrease as we approach the minimum. However, the same oscillatory behavior is also observed with the two methods that tune the learning rate, namely RMSprop, and ADAM. Curiously, for ADAM, we obtain a good, non-oscillating MSE when we start out with lower learning rates. The oscillations increase with increasing learning rates, signaling that we either are unable to combat the curvature by tuning the learning rate or that learning rates are not decreased on a fast enough time scale.

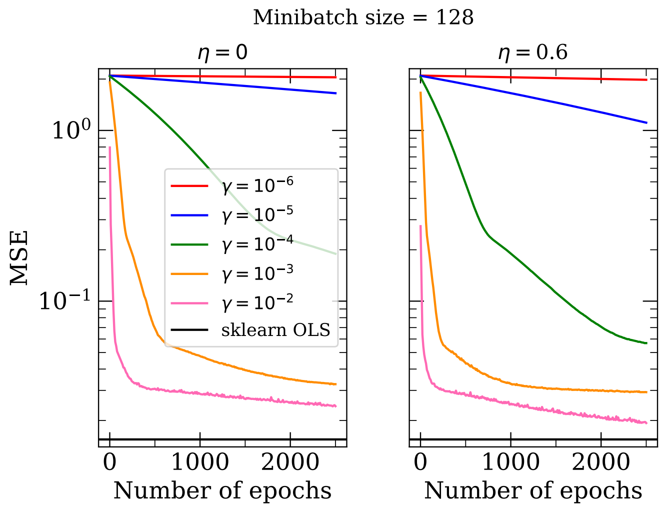


Fig. 4. The adaptive gradient, or adagrad, method tested for various learning rates as indicated in the left panel legend and compared to the OLS result. The left panel shows the adagrad method tested without momentum, while the right panel shows the result using momentum $\eta = 0.6$.

Another curious behavior we observe is shown in Figure 3, where too large penalty parameters λ are seen to increase the MSE. This can be understood from Eq. (9), where we add a constant term to the cost function, which is the product of the squared norm of the regression weights and the penalty parameter λ. This prevents the norm of the regression weightsfrom becoming too large, but it will also act as a constant addition to the weights when they otherwise are well converged. The constant addition increases with the size of λ, explaining why, in Figure 3, the converged MSE increases with the size of λ.

Summarising the analysis on gradient descent methods, we obtain good results for all investigated methods. The more complex methods, *i.e.* adagrad, RMSprop and ADAM all give good, and when good parameters are chosen, slightly better MSE than the standard SGD and SGDM methods. However,

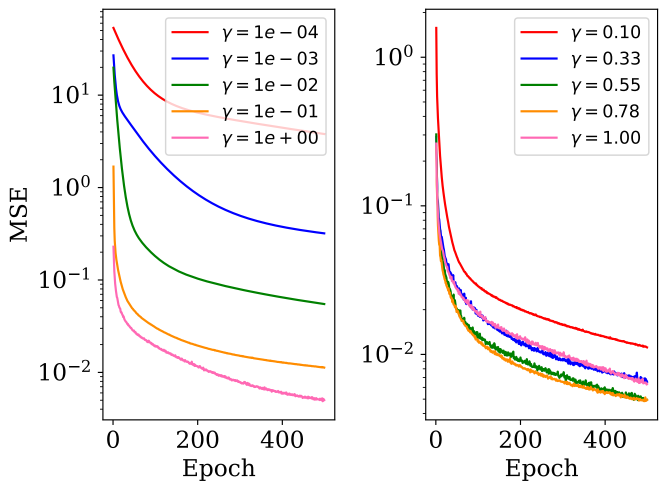
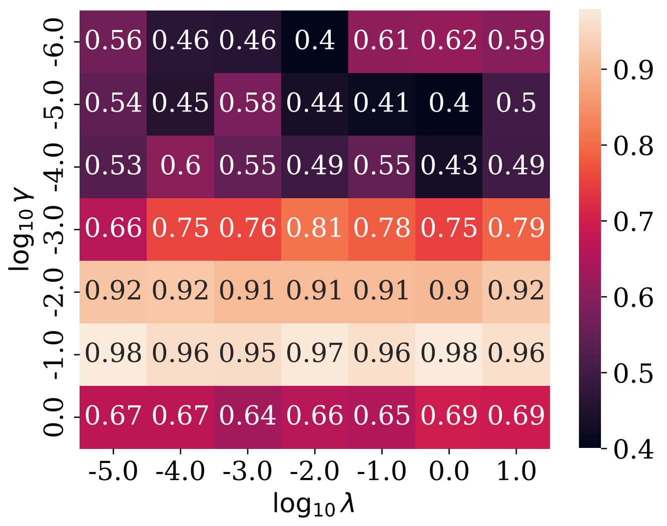


Fig. 6. Feedforward neural network tested for various learning rates. We see that the fit converges faster for higher learning rates, up to order 1.



$\log_{10}\gamma$ \ $\log_{10}\lambda$	-5.0	-4.0	-3.0	-2.0	-1.0	0.0	1.0
-6.0	0.56	0.46	0.46	0.4	0.61	0.62	0.59
-5.0	0.54	0.45	0.58	0.44	0.41	0.4	0.5
-4.0	0.53	0.6	0.55	0.49	0.55	0.43	0.49
-3.0	0.66	0.75	0.76	0.81	0.78	0.75	0.79
-2.0	0.92	0.92	0.91	0.91	0.91	0.9	0.92
-1.0	0.98	0.96	0.95	0.97	0.96	0.98	0.96
0.0	0.67	0.67	0.64	0.66	0.65	0.69	0.69

Fig. 7. Accuracy matrix for the feedforward neural classifier, tested on the breast cancer dataset. The network consists of a single layer with 50 nodes.

they are more computationally expensive, with minimal to no gain. There are a couple of likely explanations for this. Firstly, we are training our methods on a relatively small dataset, and secondly, we are using a fairly limited number of epochs. For larger datasets, and with epoch numbers orders of magnitude larger than the ones we are using, it is likely that we would observe better performance of the more complex models.

B. Neural Network

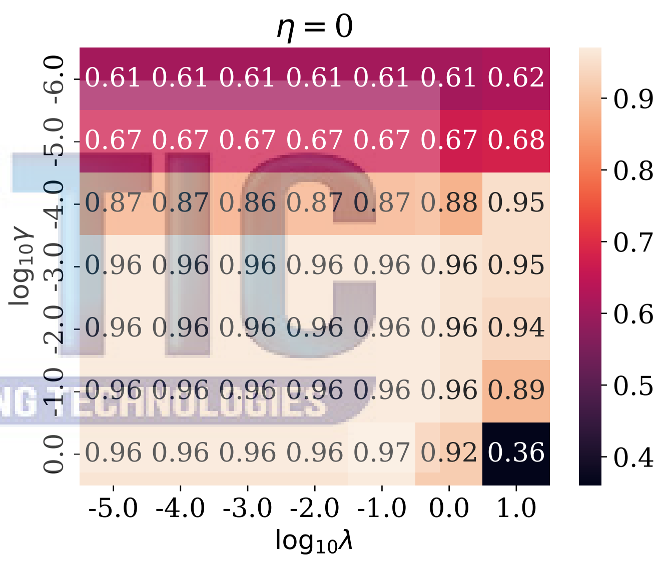
The neural network was analyzed using a simple SGD, with constant learning late. Figure 6 shows the development of the loss during training on the Franke function for different learning rates. As was the case for the linear model above, we see that for small learning rates, the convergence goes very slowly. The optimal value lies in the interval $0.5 - 0.8$. When we increase the learning rate beyond 1, the network becomes unstable.

Similar to previously, we got no improvement from including a regularisation parameter, λ, for the fit to the Franke function.

For the initial weights, we found that a normal distribution with a standard deviation of around 2, gave the best fit to the data. For a distribution with standard deviation less than 0.1, we found that the training never got going. For higher standard deviation the network became unstable.

Having fixed the parameters discussed above we adjusted the number of nodes per layer and number of layers. Starting with a single layer, the performance improved from going from 1 to around 20 layers. Further, it stabilized until we reached about 90 layers when the model started overfitting.

Having tested the network on the Franke function, we adopted it to classify the breast cancer data set. The only changes we made were changing the cost function to the cross entropy and adding the sigmoid function to the output layer. In Figure 7, we present the accuracy scores for a set of learning rates and regularisation parameters. Again, we see that the



$\log_{10}\gamma$ \ $\log_{10}\lambda$	-5.0	-4.0	-3.0	-2.0	-1.0	0.0	1.0
-6.0	0.61	0.61	0.61	0.61	0.61	0.61	0.62
-5.0	0.67	0.67	0.67	0.67	0.67	0.67	0.68
-4.0	0.87	0.87	0.86	0.87	0.87	0.88	0.95
-3.0	0.96	0.96	0.96	0.96	0.96	0.96	0.95
-2.0	0.96	0.96	0.96	0.96	0.96	0.96	0.94
-1.0	0.96	0.96	0.96	0.96	0.96	0.96	0.89
0.0	0.96	0.96	0.96	0.96	0.97	0.92	0.36

Fig. 8. Accuracy matrix for logistic regression on the breast cancer dataset for various values of the learning rate, γ, and the regularisation parameter, λ, without momentum in the gradient descent method.

performance does not depend heavily on the regularisation parameter. We see that the network performs very well, with a score of around 0.97 at a learning rate of 0.1.

C. Classification and Logistic Regression

Accuracy score matrices for the classification of breast cancer data using logistic regression are presented in Figures 8 and 9, the latter with a momentum term in the gradient descent minimization. In both cases, we obtained good accuracy in a large portion of the explored parameter space, but the momentum term yielded better accuracy in the ”worst” squares. This is in line with our observations from the initial comparison of the SGD/SGDM methods.

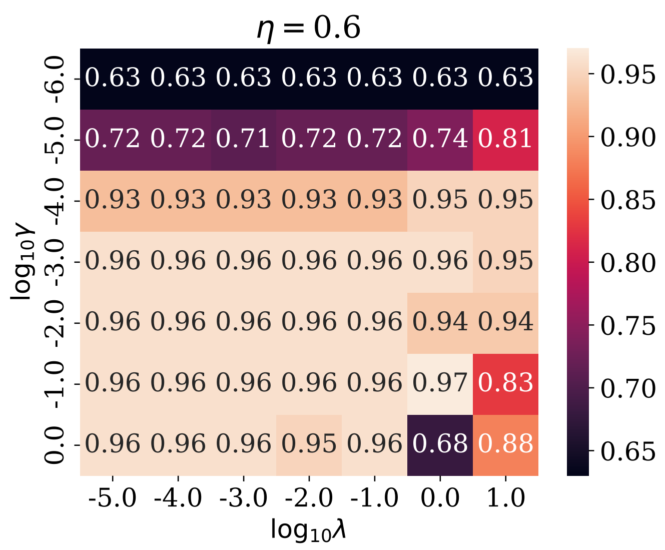


Fig. 9. Accuracy matrix for logistic regression on the breast cancer dataset for various values of the learning rate, γ, and the regularisation parameter, λ, with momentum $\eta = 0.6$ in the gradient descent method.

In both cases, we also observe the same effect for the regularisation parameter λ as we did when exploring it using SGD methods earlier. This holds true for large learning rates. We also ran Scikit-learn's implementation of logistic regression on the same dataset, obtaining an accuracy score of 0.99. Our implementation, therefore, performs well compared to the benchmark, albeit not perfect.

IV. Conclusion

In this project, we have compared gradient descent methods with neural networks applied to regression problems and neural networks with logistic regression on classification problems. For the regression problems we have extended the work of project 1 where OLS, Ridge and Lasso regression were applied to the Franke function. We find good convergence of all gradient descent methods applied but find no real advantage of using computationally expensive adaptive methods on the "simple" problem we are dealing with in this project. As the standard SGD and SGDM methods achieve just as good MSE as the more advanced methods, these are the preferred choices when handling simple regression problems. Nevertheless, more advanced methods are likely to outperform them for more complex and large data sets. The neural network also performs well for regression problems, achieving an MSE comparable to the standard deviation of the noise.

For the classification problem we have studied a breast cancer data set. The logistic regression method achieves an accuracy of 0.96 for a large portion of the parameter space, which is not too far from the benchmark accuracy of 0.99 obtained by the library method. The neural network is more dependent on the learning rate, but is able to score even better than logistic regression given the right learning rate.

References

[1] Z. Obermeyer and E. J. Emanuel, "Predicting the future—big data, machine learning, and clinical medicine," *New England Journal of Medicine*, vol. 375, no. 13, pp. 1216–1219, 2016.
[2] M. Rupp, A. Tkatchenko, K.-R. Müller, and O. A. Von Lilienfeld, "Fast and accurate modeling of molecular atomization energies with machine learning," *Physical review letters*, vol. 108, no. 5, p. 058301, 2012.
[3] A. Graves and J. Schmidhuber, "Offline handwriting recognition with multidimensional recurrent neural networks," *Advances in neural information processing systems*, vol. 21, 2008.
[4] A. Mathew, P. Amudha, and S. Sivakumari, "Deep learning techniques: an overview," *Advanced Machine Learning Technologies and Applications: Proceedings of AMLTA 2020*, pp. 599–608, 2021.
[5] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013.
[6] Y. Yang, Z. Ye, Y. Su, Q. Zhao, X. Li, and D. Ouyang, "Deep learning for," *Acta Pharmaceutica Sinica B*, vol. 9, no. 1, pp. 177–185, 2019.
[7] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*. Springer, 2010, pp. 177–186.
[8] R. Franke, *A critical comparison of some methods for interpolation of scattered data*. Naval Postgraduate School Monterey, CA, 1979.
[9] M. Zwitter and M. Soklic, "Uci machine learning repository breast cancer dataset. 1988."
[10] J. Sirignano and K. Spiliopoulos, "Dgm: A deep learning algorithm for solving partial differential equations," *Journal of computational physics*, vol. 375, pp. 1339–1364, 2018.
[11] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. New York: Springer, 2009.
[12] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org
[13] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. of the Thirteen. Int. Conf. on Artif. Intell. and Stat.*, ser. Proc. of Mach. Learn. Res., Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html