

# Feed Forward Neural Networks: Performance Analysis on Regression and Classification

## FYS-STK3155 - Project 2

Heine Elias Husdal and Frederik Callin Østern

*Department of Physics, University of Oslo*

(Dated: November 10, 2025)

We investigate the performance of feed forward neural networks (FFNNs) on regression and classification tasks. For regression, we test a dataset taken from the one-dimensional Runge function, and optimize the mean squared error (MSE) on the testing data for different numbers of layers and hidden nodes, gradient descent methods and activation functions. The best performance with  $L1$  and  $L2$  regularization is also found. We get the highest  $R^2$  score of 96.4 % with no regularization, compared to a value of 96.0 % from linear regression with the optimal polynomial fit. Next, we do a classification analysis on the multiclass MNIST dataset. We try different numbers of hidden nodes, activation functions and regularization parameters in order to maximize the accuracy on the testing data. At the end, we end up with a value of 96.9 %. Our analysis suggests that Stochastic Gradient Descent (SGD) is preferred for the training of neural networks, and that one may obtain more accurate results with the use of standard libraries, like PyTorch. With that said, our code can work as a useful benchmark for other, more specialized uses of neural networks.

## I. INTRODUCTION

Neural networks have been an indispensable tool in the modern AI and machine learning revolution. They extend the usual methods of linear regression and binary logistic classification to allow deep learning through several internal layers connected by tunable weights and biases. Thus, a neural network can be trained to incrementally combine and identify more complex features in input data. One of the main types of neural networks is the fully connected feed forward neural network (FFNN). In a FFNN, information flows only in one direction through the network, and each node in one hidden layer is connected to all nodes in the next layer.

Each layer in a FFNN consists of a certain number of nodes or "neurons", each of which is associated with a certain value. The values in the nodes are successively updated through the layers of the network with an alternating sequence of affine transformations and nonlinear activation functions, in order to produce the final output values from the input values in the first layer. A neural network may be trained by tuning the internal weights and biases of each affine transformation in order to minimize a desired cost function. Typically, the parameters are tuned with a desired gradient descent (GD) method, utilizing the backpropagation algorithm to compute the gradients.

In this work, we explore how FFNNs perform with different model architectures (given by the number of layers and nodes in each layer), training methods and activation functions. We also try with and without  $L1$  and  $L2$  regularization in the cost functions. We perform a regression analysis on data taken from the one-dimensional Runge function, using the mean squared error (MSE) as cost function. Moreover, we train FFNNs for classification on the MNIST dataset, with the Softmax cross-entropy cost. We use grid search to find the model hyperparameters and training methods with the optimal performance on the testing data, and we compare the best results with those found with the PyTorch library. The goal of our analysis is to become familiar with the behavior of neural networks, including potential pitfalls with exploding or vanishing gradients in the cost function.

In section II, we explain the theory underlying neural networks and the backpropagation algorithm. We also briefly explain how we performed our analysis in Python. In section III, we present our main results and discuss them. In particular, we critically evaluate the results we got and the methods used. Finally, we end in section IV with a conclusion and ideas for future improvements.

## II. METHODS

### A. Perceptron

Neural networks are computational models that, historically, have been inspired by the structure of the biological brain [1]. The goal is to train a model to be capable of solving a desired problem by computing an optimized output for any desired input one feeds into it. Thus, to seek a model with sufficiently "intelligent" behavior, one may take inspiration from the computational processes in the brain [1].

One of the first implementations of what we today may recognize as a neural network, is the *perceptron* model [1, 2]. As input, the model takes in a set of values  $\{x_i\}_{i=1}^n$  and produces an output  $y$  according to [3]

$$y = f(\sum_{i=1}^n w_i x_i + b), \quad (1)$$

where  $f$  is some nonlinear *activation function*,  $\{w_i\}_{i=1}^n$  are a set of tunable weights and  $b$  is a bias term. One may consider the perceptron as a model inspired by a single neuron. Originally, the activation function  $f$  of the perceptron was the (Heaviside) step function [2]. This would correspond to the fact that the neuron either fires or not, depending on whether the combined input passes a certain threshold. In modern machine learning research, however, one may use many other activation functions (which we get back to).

The perceptron provides the link between neural networks and earlier statistical optimization techniques, including linear and logistic regression. Linear regression may be considered a perceptron model where the activation function is the trivial identity function. Thus, the output is simply a linear function of the input, and the weights  $\{w_i\}_{i=1}^n$  correspond to the optimizable parameters of the model. In the case of logistic regression, the activation function is the logistic function (defined later in eq. 13). The output then corresponds to the probability that the target is an element of the positive class.

## B. Feed Forward Neural Networks

A (fully connected) feed forward neural network (FFNN) or multi-layer perceptron (MLP) extends the perceptron model by allowing the use of multiple neurons distributed in several *layers* in the network. All of the neurons in one layer may then communicate with the neurons of the next layer. Thus, the layout of a feed forward neural network is conceptually similar to the structure of the connections between neurons in a brain. By combining information in several layers, the model can learn to successively combine simpler features in the input to more complex, abstract representations [1], which significantly enhances its learning capabilities. Thus, FFNNs are at the core of modern deep learning.

To represent the forward pass of a FFNN mathematically, we let  $\mathbf{a}^{(l)}$  represent the (two-dimensional) matrix of values in the neurons in layer  $l$ . This matrix has shape  $(B, M_l)$ , where  $B$  is the number of independent samples in the input batch and  $M_l$  is the number of neurons in layer  $l$ . The first layer of the network, corresponding to  $l = 0$ , is the *input* layer. Here, the matrix  $\mathbf{a}^{(0)}$  corresponds to the input values in the forward pass. The last layer  $l = L$  is the *output* layer, so that  $\mathbf{a}^{(L)}$  corresponds to the values of the output from the model. The intermediate layers  $l$  with  $0 < l < L$  are the *hidden* layers.

The output of any one neuron in layer  $l$  is connected to the input of all other neurons in the next layer  $l + 1$  by a certain weight factor, similar to the connection between an input and output value in the simple perceptron model. Thus, we can construct a matrix  $\hat{\mathbf{W}}^{(l+1)}$

of weight values with shape  $(M_l, M_{l+1})$ . The activation values  $\mathbf{z}^{(l+1)}$  in layer  $l + 1$  can then be found from the output  $\mathbf{a}^{(l)}$  of layer  $l$  by the affine matrix transformation [3]

$$\mathbf{z}^{(l+1)} = \mathbf{a}^{(l)} \hat{\mathbf{W}}^{(l+1)} + \mathbf{b}^{(l+1)}. \quad (2)$$

Here,  $\mathbf{b}^{(l+1)}$  is a row vector of size  $M_{l+1}$  with bias values in layer  $l + 1$  (note, however, that  $\mathbf{a}^{(l)} \hat{\mathbf{W}}^{(l+1)}$  has shape  $(B, M_{l+1})$ ). The point is that when  $\mathbf{a}^{(l)} \hat{\mathbf{W}}^{(l+1)}$  and  $\mathbf{b}^{(l+1)}$  are summed together in eq. 2 above, the values in  $\mathbf{b}^{(l+1)}$  are duplicated along the columns in  $\mathbf{a}^{(l)} \hat{\mathbf{W}}^{(l+1)}$ . This equation extends the linear transformation from the perceptron in eq. 1.

The output  $\mathbf{a}^{(l+1)}$  of layer  $l + 1$  is then found from the values  $\mathbf{z}^{(l+1)}$  by passing it into the activation function  $f^{(l+1)}$  in layer  $l + 1$ :

$$\mathbf{a}^{(l+1)} = f^{(l+1)}(\mathbf{z}^{(l+1)}). \quad (3)$$

Each of the values in  $\mathbf{z}^{(l+1)}$  are passed into the activation function separately to produce each value in  $\mathbf{a}^{(l+1)}$ .

## C. Backpropagation

The model is trained for a certain task by carefully tuning the weights and biases  $(\hat{\mathbf{W}}^{(l)}, \mathbf{b}^{(l)})$  in each layer  $l$ . For a given set of training data, the model is trained to make predictions that best fit the true targets of the training samples. This is done by minimizing a *cost function*  $C(\Theta)$ , which measures the error between the final output values  $\mathbf{a}^{(L)}$  and the target values in the training data. Since the final layer output  $\mathbf{a}^{(L)}$  implicitly depends on all of the weights and biases of the network through the forward pass, the value of the cost function will depend on the full collection of parameters  $\Theta = \prod_{l=1}^L (\hat{\mathbf{W}}^{(l)}, \mathbf{b}^{(l)})$ .

We minimize the cost as a function of the weights and biases by the use of gradient descent algorithms. In a previous project [4], we have presented a detailed review of these algorithms, including the (minibatch) Stochastic Gradient Descent (SGD) and ordinary Gradient Descent (GD) versions of them. In order to apply these methods on neural networks, one must calculate the gradients  $\frac{\partial C}{\partial \hat{\mathbf{W}}^{(l)}}$  and  $\frac{\partial C}{\partial \mathbf{b}^{(l)}}$  for each layer  $l$ . We do this by the use of the *backpropagation* algorithm [3], which is the standard technique in modern machine learning.

We start at the last layer of the network, and compute the gradient  $\frac{\partial C}{\partial \mathbf{a}^{(L)}}$  using the expression for the cost function with respect to the final output values. Here,  $\frac{\partial C}{\partial \mathbf{a}^{(L)}}$  is a matrix of shape  $(B, M_L)$  with elements

$\left(\frac{\partial C}{\partial \mathbf{a}^{(L)}}\right)_{ij} = \frac{\partial C}{\partial a_{ij}^{(L)}}$ . By the chain rule, we may then compute the gradients

$$\frac{\partial C}{\partial z_{ij}^{(L)}} = \frac{\partial C}{\partial a_{ij}^{(L)}} \frac{\partial a_{ij}^{(L)}}{\partial z_{ij}^{(L)}} = \frac{\partial C}{\partial a_{ij}^{(L)}} (f^{(L)})'(z_{ij}^{(L)}), \quad (4)$$

where  $(f^{(L)})'$  is the derivative of the activation function  $f^{(L)}$  with respect to its input. Thus, by introducing a matrix  $\delta^{(l)}$  with elements  $\delta_{ij}^{(l)} = \frac{\partial C}{\partial z_{ij}^{(l)}}$  for a layer  $l$ , we may write [3]

$$\delta^{(L)} = (f^{(L)})'(\mathbf{z}^{(L)}) \circ \frac{\partial C}{\partial \mathbf{a}^{(L)}}, \quad (5)$$

where  $\circ$  is the Hadamard (element-wise) matrix product.

We can now go backwards through the network and inductively evaluate  $\delta^{(l)} = \frac{\partial C}{\partial \mathbf{z}^{(l)}}$  for smaller values of  $l$ . In particular, from the chain rule, we have

$$\begin{aligned} \delta_{ij}^l &= \frac{\partial C}{\partial z_{ij}^{(l)}} = \sum_{k=1}^{M_{l+1}} \left( \frac{\partial C}{\partial z_{ik}^{(l+1)}} \frac{\partial z_{ik}^{(l+1)}}{\partial a_{ij}^{(l)}} \right) \frac{\partial a_{ij}^{(l)}}{\partial z_{ij}^{(l)}} \\ &= \sum_{k=1}^{M_{l+1}} \left( \delta_{ik}^{(l)} \hat{W}_{jk}^{l+1} \right) (f^{(l)})'(z_{ij}^{(l)}), \end{aligned} \quad (6)$$

which we can write in matrix form as [3]

$$\delta^{(l)} = \delta^{(l+1)} \hat{W}^T \circ (f^{(l)})'(\mathbf{z}^{(l)}). \quad (7)$$

Thus, together with eq. 5 and 7, we can compute  $\delta^{(l)}$  for all layers  $l$ .

From the chain rule, we can then compute the gradients  $\frac{\partial C}{\partial \hat{W}^{(l)}}$  (defined by  $\left(\frac{\partial C}{\partial \hat{W}^{(l)}}\right)_{ij} = \frac{\partial C}{\partial \hat{W}_{ij}^{(l)}}$ ) and  $\frac{\partial C}{\partial b_i^{(l)}}$ . We end up with [3]

$$\begin{aligned} \frac{\partial C}{\partial \hat{W}^{(l)}} &= (\mathbf{a}^{(l-1)})^T \delta^{(l)} \\ \frac{\partial C}{\partial b_i^{(l)}} &= \sum_{j=1}^B \delta_{ji}^{(l)}, \end{aligned} \quad (8)$$

where  $i = 1, 2, \dots, M_l$ .

#### D. Activation functions

The activation functions introduce the required non-linearity that makes sure that the neural network model does not end up being one large affine transformation. In particular, this is important for the neural network to

satisfy the *universal approximation theorem*. In order to fulfill this theorem, the activation functions would have to, among other things, be bounded and monotonically increasing [3].

One of the most popular activation functions in modern machine learning is the *Rectified Linear Unit (ReLU)* function [1]:

$$\text{ReLU}(z) = \max(0, z). \quad (9)$$

with its associated derivative

$$\frac{d(\text{ReLU}(z))}{dz} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (10)$$

This function can be quite useful, partly because its simplicity makes it fast to compute and optimize [1].

A potential issue with the ReLU function is that the values of some neurons in the network may eventually die out, since it returns the value 0 for all negative input values. This could lead to the *vanishing gradient* problem for the parameters in the last layers, such that these values are not updated. A possible remedy is to instead use the *LeakyReLU* function [5]:

$$\text{LeakyReLU}(z) = \max(z, \alpha z), \quad (11)$$

with the derivative

$$\frac{d(\text{LeakyReLU}(z))}{dz} = \begin{cases} 1 & \text{if } z \geq 0 \\ \alpha & \text{if } z < 0 \end{cases} \quad (12)$$

where  $\alpha$  is a small positive value. The point is that this introduces a tiny non-zero slope  $\alpha$  for negative input values. A typical value of  $\alpha$  is 0.01 [3]. This is the value we will use.

Furthermore, we have the *Sigmoid* (or logistic) activation function. It is given by [3]

$$\text{sigmoid}(z) = \frac{1}{1 + e^{-z}} \quad (13)$$

and has the derivative

$$\frac{d(\text{sigmoid}(z))}{dz} = \text{sigmoid}(z)(1 - \text{sigmoid}(z)) \quad (14)$$

This function smooths out the step function, and hence it fits better with the behavior of biological neurons compared to the ReLU and LeakyReLU functions [3]. However, a significant disadvantage is that as the activation values propagate through the network, the

output of the Sigmoid function may eventually saturate (since its output quickly converges for large input values). That means the gradients of the cost function may be very small for the last layers, which could spoil the backpropagation method [3].

Finally, we have the *Softmax* function:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n_C} e^{z_j}} \quad (15)$$

Here, we consider a collection  $\{z_i\}_{i=1}^{n_C}$  of values for each of  $n_C$  classes. This activation function is used at the last output layer of a neural network, and may be used in multiclass classification to compute the probability that the output belongs to any one of several classes.

### E. Cost functions

There are two main uses of neural networks: regression and classification. In the context of regression, the objective is to fit continuous output values to continuous input values. To do this, we use a combination of *no* activation function in the last output layer, as well as the *mean squared error* (MSE) between the predicted and true output values as the cost function. Thus, the neural network is trained by minimizing this cost. The MSE is given by

$$C_{MSE}(\mathbf{y}, \tilde{\mathbf{y}}) = \frac{1}{n} \sum_{i=1}^n (y_i - \tilde{y}_i)^2. \quad (16)$$

Here,  $\tilde{\mathbf{y}}$  is a vector of the predicted output values for the  $n$  training samples in the last output layer, and  $\mathbf{y}$  is a vector of true target values.

From this expression, the gradient of the MSE cost function with respect to the final output values becomes

$$\frac{\partial C_{MSE}}{\partial \mathbf{a}^{(L)}} \equiv \frac{\partial C_{MSE}}{\partial \tilde{\mathbf{y}}} = \frac{2}{n} (\tilde{\mathbf{y}} - \mathbf{y}). \quad (17)$$

In the context of classification, we wish to fit the input to one of a set of finite, categorical classes. The model should evaluate probabilities that the output belongs to the various classes. For binary classification and logistic regression, the true target value is a binary variable, where the value 1 (corresponding to a probability of 100 %) means that the output belongs to a certain class, and the value 0 means that it does not. We can then use the *binary cross-entropy* as cost function to measure the error between the targets and predicted probabilities. It is given by

$$\begin{aligned} & C_{binary\_c\_e}(\mathbf{y}, \tilde{\mathbf{y}}) \\ &= -\frac{1}{n} \sum_{i=1}^n (y_i \ln(\tilde{y}_i) + (1 - y_i) \ln(1 - \tilde{y}_i)), \end{aligned} \quad (18)$$

where  $\mathbf{y}$  is a vector of true target values and  $\tilde{\mathbf{y}}$  is a vector of predicted probabilities that the output belongs to the class. We use the Sigmoid function in the last output layer to compute these probabilities. From eq. 18, we may compute the gradient with respect to the final output values:

$$\begin{aligned} \frac{\partial C_{binary\_c\_e}}{\partial \mathbf{a}^{(L)}} &\equiv \frac{\partial C_{binary\_c\_e}}{\partial \tilde{\mathbf{y}}} \\ &= \frac{1}{n} \sum_{i=1}^n \left( \frac{1 - y_i}{1 - \tilde{y}_i} - \frac{y_i}{\tilde{y}_i} \right) \end{aligned} \quad (19)$$

In the case of the general multiclass classification problem, the true target is a one-hot encoded vector with a size  $n_C$  equal to the number of classes. Each element is 1 or 0, corresponding to whether the output belongs to the given class or not. We generalize to use a combination of the Softmax function in the last layer to compute the probabilities and the multiclass cross-entropy as cost function. This function measures the difference between the estimated probabilities and the exact one-hot probabilities. It is given by

$$C_{c\_e}(\mathbf{y}, \tilde{\mathbf{y}}) = -\frac{\sum_{i=1}^n \sum_{j=1}^{n_C} y_{ij} \ln(\tilde{y}_{ij})}{n} \quad (20)$$

where  $\mathbf{y}$  and  $\tilde{\mathbf{y}}$  are the true and predicted one-hot probabilities, respectively. They are two-dimensional matrices of shape  $(B, n_C)$ , where  $n_C$  is the number of classes.

There is a simple formula for the gradient of the composite function consisting of the Softmax  $\sigma$  and the cross-entropy, which is to say the gradient  $\frac{\partial C_{c\_e}}{\partial \mathbf{z}^{(L)}}$ . One finds [6]

$$\begin{aligned} \frac{\partial C_{c\_e}}{\partial \mathbf{z}^{(L)}} &= \sigma'(\mathbf{z}^{(L)}) \circ \frac{\partial C}{\partial \mathbf{a}^{(L)}} \\ &\equiv \sigma'(\mathbf{z}^{(L)}) \circ \frac{\partial C}{\partial \tilde{\mathbf{y}}} = \frac{\tilde{\mathbf{y}} - \mathbf{y}}{n} \end{aligned} \quad (21)$$

Thus, instead of first finding  $\frac{\partial C_{c\_e}}{\partial \mathbf{a}^{(L)}}$  in the backpropagation algorithm, as in the case of the MSE cost, we find  $\delta^{(L)} = \frac{\partial C_{c\_e}}{\partial \mathbf{z}^{(L)}}$  directly.

#### 1. Regularization

One can also add *L1* and *L2* regularization terms to the cost functions in eqs. 16 and 20. These have the form

$$\begin{aligned} C_{L1} &= \lambda \sum_{l=1}^L \left( \|\hat{\mathbf{W}}^{(l)}\|_1 + \|\mathbf{b}^{(l)}\|_1 \right) \\ C_{L2} &= \lambda \sum_{l=1}^L (\|\hat{\mathbf{W}}^{(l)}\|_2^2 + \|\mathbf{b}^{(l)}\|_2^2), \end{aligned} \quad (22)$$

where  $\lambda$  is a regularization parameter, and the operators  $\|\dots\|_1$  and  $\|\dots\|_2^2$  give the sum of the absolute values and squares of the elements of the corresponding matrix or vector, respectively. The gradients of these terms with respect to one of the parameters  $\theta$  (which could be either  $\hat{\mathbf{W}}^{(l)}$  or  $\mathbf{b}^{(l)}$ , for some  $l$ ) becomes

$$\begin{aligned} \frac{\partial C_{L1}}{\partial \theta} &= \lambda \text{sgn}(\theta) \\ \frac{\partial C_{L2}}{\partial \theta} &= 2\lambda\theta, \end{aligned} \quad (23)$$

where the sign function  $\text{sgn}()$  is applied independently on each element of  $\theta$  in  $\text{sgn}(\theta)$ .

The potential benefit of adding regularization is that it may improve the error on testing data in the case of *overfitting* [4]. In particular, an  $L2$  regularization may reduce the effects of some parameters of the model, whereas  $L1$  regularization is useful for to fully remove the effects of some features, thus enabling feature selection.

## F. Our implementation

We implemented a class for feed forward neural networks. It can take in any number of hidden layers, with their associated number of nodes and activation functions. We used eqs. 5, 7 and 8 to manually train the model with the backpropagation algorithm, as well as eq. 2 and 3 to calculate the final forward pass output. Eqs. 10, 12 and 14 were used to calculate the derivative  $(f^{(l)})'$  of the appropriate activation function in eq. 5 and 7. Moreover, eqs. 17 and 21 (plus the derivatives in eq. 23 in the case of regularization) were used to compute the initial gradients  $\frac{\partial C}{\partial \mathbf{a}^{(L)}}$  and  $\delta^{(L)}$  in the backpropagation algorithm, respectively.

The values of the elements of the model weights were initialized with the standard normal distribution. In order to ensure that backpropagation would work in the first iteration, we initialized the biases with the tiny value 0.01 [3].

We performed a regression analysis with neural networks on data taken from the one-dimensional Runge function

$$f(x) = \frac{1}{1 + 25x^2},$$

with the addition of normally distributed noise with amplitude 0.05. We used a dataset of 300 data-points in total, drawn from a uniform distribution in the interval  $[-1, 1]$ . The dataset was split into training and testing data with a test size of 20% by using the `train_test_split` function from the `sklearn.model_selection` module in Python, with a random state of 44. Moreover, we performed a standard scaling of the data. In our previous project [4], we found that a polynomial fit with degree 12 gave minimum bias and variance for this dataset. Thus, we compared the mean squared error of the neural network on the testing data with the predictions of Ordinary Least Squares regression.

Furthermore, we fetched the full MNIST dataset of hand-drawn digits for the classification task. This was done using `sklearn.datasets.fetch_openml('mnist_784')` in Python. The input data are 70,000 greyscale 28x28 images that we flattened to input samples with 784 values in each, that all were scaled from the range  $[0, 255]$  to  $[0, 1]$  by dividing by 255. The network output layer consists of 10 neurons, corresponding to the network's predicted probabilities for each of the 10 digits. Thus, we one-hot encode the labels to match this output format.

Our implementation of neural networks were compared with the autograd and PyTorch libraries [7, 8]. We used the `autograd.grad` function to calculate the gradient of the cost function with respect to the weights and biases, and compared with the values found from our manual backpropagation algorithm. Moreover, we used PyTorch to train corresponding neural network models to compare with our results.

Finally, we also used the `linear_model` class from scikit-learn [9] to calculate the mean squared error on the testing data for Ordinary Least Squares (OLS), Ridge (for  $L2$  regularization) and Lasso (for  $L1$  regularization) regression. These values were compared with the neural network results with optimal hyperparameters.

## III. RESULTS AND DISCUSSION

### A. Regression

#### 1. Regression analysis without regularization

We started by doing a regression analysis without any regularization terms, in order to get a sense of the difference in performance between the optimal OLS regression fit and a FFNN. We trained two neural networks with 50 neurons in each layer, with both one and two hidden layers. We used the Sigmoid function as activation function in the hidden layers. Since we wanted to test the full capabilities of neural networks, we used a single neuron in the input layer, corresponding to the single  $x$  value (instead of using

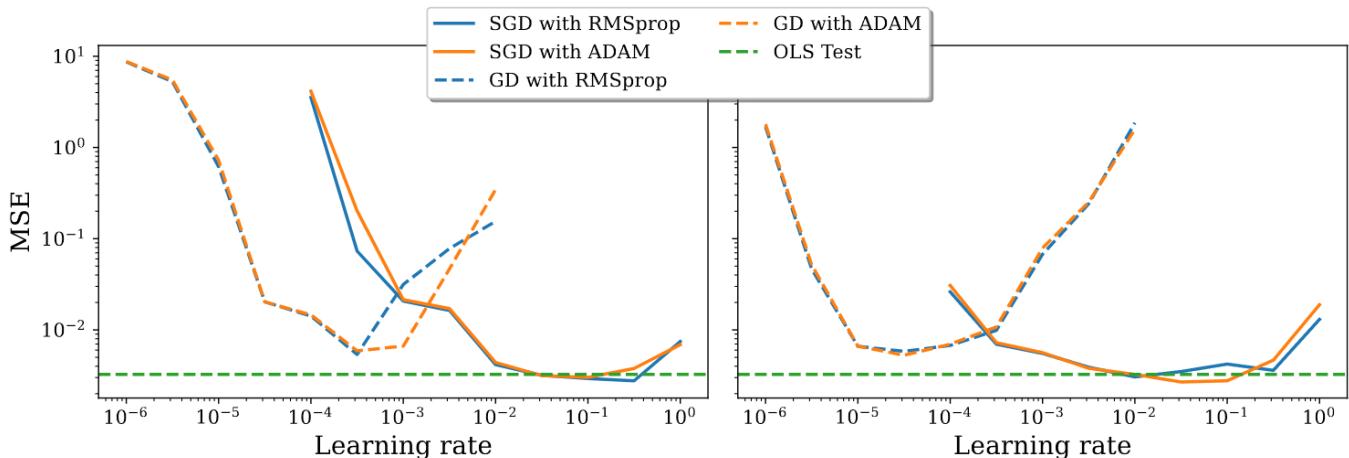


Figure 1: *MSE on the testing data as a function of learning rate, for neural network regression analysis on Runge function.* The FFNNs are trained with 3000 iterations and different gradient descent algorithms. "OLS Test" indicates scikit-learn's OLS regression result for the MSE, for a polynomial fit of degree 12. Left: FFNN with one hidden layer with 50 neurons. Right: FFNN with two hidden layers with 50 neurons in each.

several neurons for different powers of  $x$ ). The output layer also had a single neuron corresponding to the predicted output value for the Runge function. In Figure 1, we show the results as plots of the testing data MSE cost function as a function of the value of the learning rate, for neural networks trained with 3000 iterations or epochs. We have compared with the testing MSE for the optimal polynomial fit for OLS regression. Note that in our implementation of stochastic gradient descent, we divided the initial learning rate by the number of batches in order to be able to reliably compare both stochastic and non-stochastic gradient descent with the same number of iterations. We also chose to use 5 minibatches for our regression analysis, as well as the same learning schedule as in eq. 18 in our previous work [4], except that we used  $t_0 = t_1 = 30 \cdot n_{batches}$  in this case.

We notice that for all gradient descent methods, the MSE reaches a minimum for a specific choice of learning rate. It may be expected that the reason the MSE rises for smaller learning rates is that the model does not reach a minimum in the cost function after the 3000 iterations, or that it gets stuck in a local minimum. In the supplementary material on GitHub, we find that the MSE oscillates wildly as a function of the number of iterations for large learning rates, which indicates that the magnitude of the gradient of the MSE is significantly larger. This suggests that the increase in MSE for large learning rates is caused by the exploding gradient problem [3].

In Table I, we show the learning rate and MSE at the minima for each of the four methods and neural networks in Figure 1. Both from the table and the figure, it is clear that SGD prefers a larger (initial) learning rate. This may be due to its learning schedule,

which reduces the learning rate over time. Moreover, we see in Table I that we achieve about a factor 2 lower MSE with SGD compared to plain GD. This is also clear in Figure 1. The reason why this happens may be seen in Figure 2. Here, we show the MSE as a function of the number of iterations for the optimal learning rate for each of the gradient descent methods. We notice that for ordinary gradient descent, the MSE has still not stabilized around a minimum after 3000 iterations, even with the optimal learning rate. In order to ensure a reasonable training time, we settled on using 3000 iterations. It would not be of much help to use a significantly larger value anyway, since the MSE for ordinary GD decreases rather slowly at the end, especially for the model with two hidden layers. Thus, a clear advantage of stochastic gradient descent is that it stabilizes after fewer iterations. Another advantage is that we see in Figure 2 that for the first few iterations, the MSE drops faster for SGD than for ordinary GD. Thus, it appears that the stochastic nature of SGD is effective for finding and pushing the model down steep descents in the cost function.

Table I: *Optimal learning rate (LR) and MSE (both in powers of  $10^{-3}$ ) for each gradient descent method and network architecture in Figure 1.*

# of hidden layers	Stochastic GD				Plain GD			
	RMSProp		Adam		RMSProp		Adam	
	LR	MSE	LR	MSE	LR	MSE	LR	MSE
1	316	2.8	100	3.0	0.32	5.4	0.32	5.9
2	10	3.1	32	2.7	0.032	5.8	0.032	5.3

A disadvantage with SGD, however, is that we see in

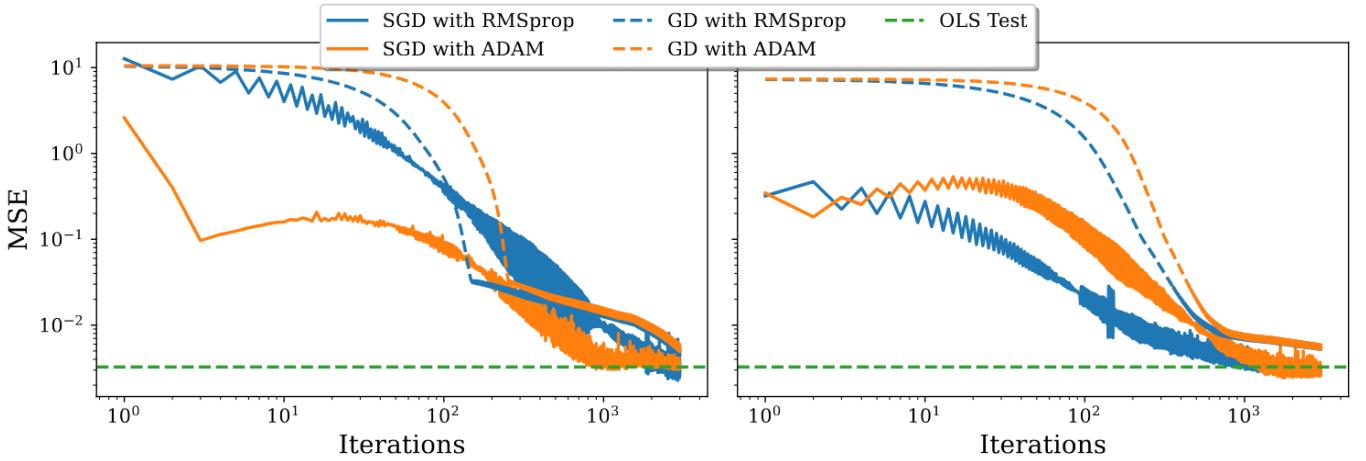


Figure 2: *MSE as a function of the number of iterations or epochs for the optimal learning rates in Table I, for each combination of gradient descent method and feed forward neural network.* Left: FFNN with one hidden layer with 50 neurons. Right: FFNN with two hidden layers with 50 neurons in each.

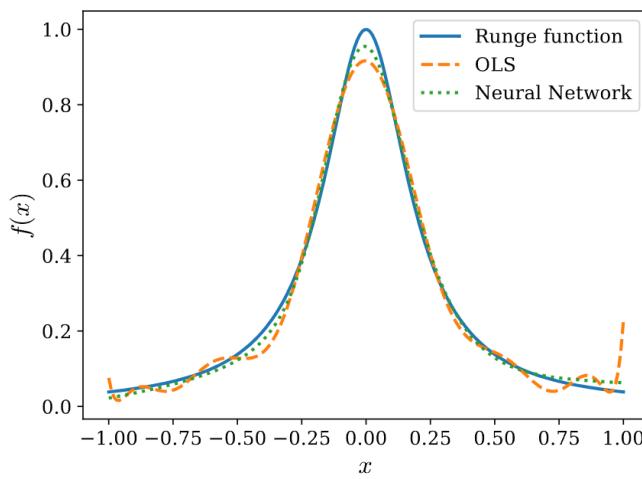


Figure 3: *Comparison of the predictions of OLS regression and neural network with the values of the Runge function. The FFNN has two hidden layers with 50 neurons in each.*

Table II: *Testing MSE (in powers of  $10^{-6}$ ) for each gradient descent method and network architecture in Figure 1, with the use of PyTorch.*

# of hidden layers	Stochastic GD		Plain GD	
	RMSProp	Adam	RMSProp	Adam
1	20.0	2.81	0.00442	5.74
2	6.77	2.63	0.0323	4.10

Figure 2 that there are significant fluctuations in the MSE as a function of the number of iterations. That may contribute some uncertainty to the final MSE that we find. However, this uncertainty is small enough that SGD is seen to perform better than GD.

It is harder to distinguish the performance of Adam and RMSProp. From Figure 1 and Table I, it is clear that they both end up at approximately the same MSE after 3000 iterations. In Figure 2, one also sees that the MSE converges faster for the Adam algorithm with only one hidden layer, but that it is slower with two hidden layers. However, we settled on using the Adam algorithm for the rest of the analysis. Partly, the reason is that it appears in Figure 2 that the oscillations of the MSE are slightly weaker for the Adam method. We also used it due to its popularity, and because it should theoretically converge faster due to the additional use of momentum [4].

Finally, we may compare our results with that of OLS regression with the polynomial fit of degree 12. We found that the MSE of OLS regression was  $3.25 \cdot 10^{-3}$ . From Table I, we see that the SGD MSE is slightly smaller than this value for all gradient descent algorithms and models. Thus, neural networks have somewhat better performance compared to the optimal OLS model. In the supplementary material, we show that this conclusion persists, even when we do a more thorough cross validation analysis to estimate the average testing MSE. Thus, this result is not merely contingent on the fact that we compute the testing MSE on a single testing dataset.

The reason that FFNNs perform better may be seen in Figure 3. Here, we show an example of the predictions of the FFNN with two hidden layers (and 50 neurons in each), and compare them with the results of OLS regression and the original Runge function. The models are trained on a single fold of the original dataset, which consists of 20 % of the original data, and we utilize the Adam SGD with a learning rate of 0.01 and 3000 epochs. We notice that OLS regression performs poorly on the edges

of the interval, due to large oscillations in the predicted output. This is characteristic of the *Runge phenomenon*. However, we see that the neural network is able to avoid this problem, and that it better approximates the Runge function on these edges. Thus, we see that neural networks are better at adapting to our desired dataset, even when we only pass it a single input value. This shows an example of how neural networks may be both simpler and more accurate than more conventional methods.

## 2. Comparison with Autograd and PyTorch

In order to test our implementation of neural networks, we compared with both Autograd and PyTorch. We calculated the gradient of the cost function with three hidden layers with 3 nodes each, using the ReLU, Sigmoid and LeakyReLU activation functions in sequence. We also added an  $L1$  regularization with  $\lambda = 10^{-2}$ . In the supplementary material, it is shown that we get exactly the same gradient with our manual backpropagation algorithm as with autograd. That is the desired consistency which shows that our implementation is most likely correct.

Using PyTorch, we also made neural networks in order to find the testing data MSE for the same model architectures, gradient descent methods, learning rates and number of epochs as in Table I. The corresponding MSE values that were found are given in Table II. We see that the PyTorch MSE is significantly smaller - more precisely, it is about three orders of magnitude smaller (or even more for some of the GD results). The fact that PyTorch performs much better is most likely not because our implementation is wrong - rather, it may be due to the fact that PyTorch is more optimized, including the way that it initializes the model parameters or the learning schedule for stochastic gradient descent. However, our results do show that using a standard library like PyTorch may be preferable in the long run.

## 3. Optimal model architecture

Next, we tried to find the optimal neural network architecture for the regression task. We trained FFNNs on the Adam SGD algorithm with 3000 epochs, with both 1, 3 and 5 hidden layers and the Sigmoid, ReLU and LeakyReLU activation functions. In order to reduce the scope of the search, we always used the same activation function in all hidden layers. Furthermore, we tested with different learning rates and numbers of nodes. The final MSE on the testing data for all of the combinations is shown in Figure 4. Each heatmap shows the MSE as a function of learning rate and the number of nodes. The heatmaps are displayed on a grid, with the type of activation function along the horizontal axis and the number of hidden layers along the vertical axis.

We notice a few key features. First, we recognize the pattern found in Figure 1, namely that the MSE is large for small learning rates, reaches a minimum for intermediate values, and then rises again for larger learning rates. This pattern is especially clear on all of the heatmaps with one hidden layer, as well as for the other results with the Sigmoid activation function.

The ReLU and LeakyReLU functions display some interesting behavior, however. One thing we notice is that, unlike the LeakyReLU function, the logarithm of the testing MSE for the ReLU function saturates at approximately  $-1.20$  for a large enough value of the learning rate. Moreover, this saturation begins to occur for smaller learning rates with a larger number of hidden layers. Not only that, but for learning rates smaller than the value at which this transition occurs, the results for the ReLU and LeakyReLU functions are almost identical. These observations strongly suggest that the cause of the saturation is exactly that the output from the ReLU function of several neurons have vanished, so that the gradient of the cost function drops to zero and the training halts. This happens earlier with more hidden layers, since there are now a larger number of neurons and ReLU functions available for the vanishing gradient problem to occur.

Furthermore, we distinctly notice that for the LeakyReLU function, the MSE rises sharply with the number of neurons for a learning rate of  $10^{1.0} = 10$ . We may recall that the reason the MSE is so large, is likely due to the exploding gradient problem. However, the fact that this effect is magnified for larger numbers of nodes may be due to *overfitting*. That is, as the model complexity increases with larger numbers of neurons, the model may overfit on the training and underperform on the testing data, leading to larger MSE values.

Surprisingly, we thus find that the Sigmoid function performs best. For all three numbers of hidden layers, it has a relatively large range of learning rates and numbers of nodes with a small MSE between approximately  $10^{-2.50}$  to  $10^{-2.60}$ . Unlike ReLU and LeakyReLU, it is also quite robust, with few sudden drops or jumps in the MSE. The fact that the values from the Sigmoid function do not saturate, as one may expect, is likely due to the relatively small number of hidden layers and neurons.

We find that no particular number of hidden layers performs best. Even with a single layer, the MSE tends to drop to its minimum value around  $10^{-2.50}$  to  $10^{-2.60}$ . There may be a weak tendency for these minimum values to occur for larger ranges of the hyperparameters with a larger number of hidden layers, but that is the only weakly noticeable effect. In fact, the absolute smallest learning rate occurs with only a single hidden layer with 40 neurons and a logarithm of the learning rate of about  $-1.1$ . We see that with this choice, the testing MSE is about  $10^{-2.60} \approx 2.51 \cdot 10^{-3}$ . In the supplementary ma-

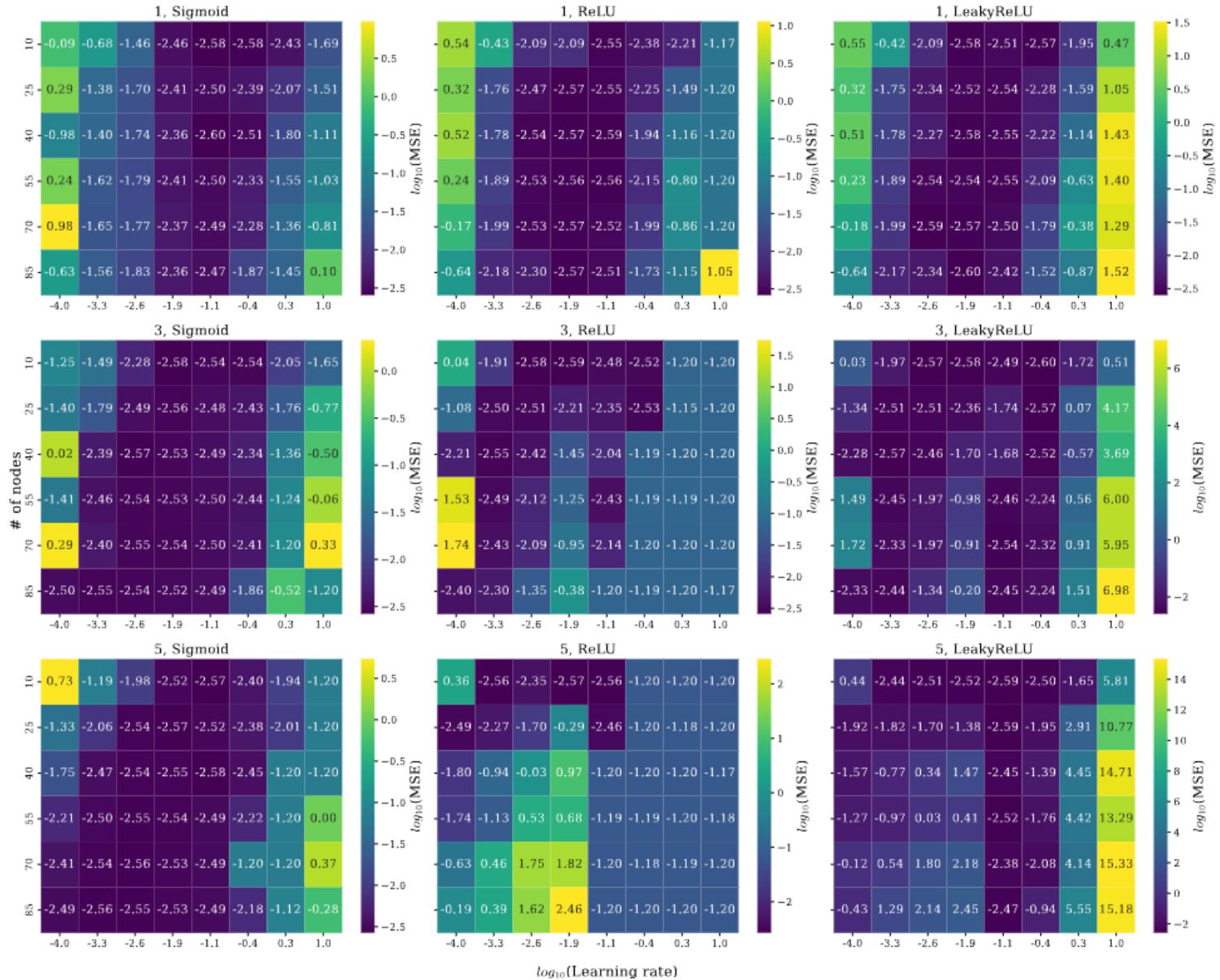


Figure 4: Testing MSE for regression analysis on Runge function, with different model architecture and learning rates. On each heatmap, the logarithm (of base 10) of the testing MSE is plotted as a function of the logarithm (of base 10) of the learning rate and the number of nodes in each of the hidden layers. The title of the heatmaps shows the the number of hidden layers and the activation function used in each of them.

terial, we do a cross validation analysis with five folds on the scaled data set with this choice of learning rate and model architecture, and we find that the neural network ends up with a  $R^2$  score (explained in our previous work [4]) of 96.4 %, compared to a score of 96.0 % for OLS regression. Thus, the neural network explains only slightly more of the data variability compared to OLS regression.

#### 4. Regression analysis with regularization

Finally, we performed a regression analysis with  $L1$  and  $L2$  regularization terms. In order to optimize the values of the hyperparameters, we first calculated the testing MSE as a function of the learning rate and regularization parameter, in order to find the optimal pair of these values. The results are shown in Figure 5. Here, we chose to use the same model architecture as the optimal one that we found with no regularization.

We notice that for both  $L1$  and  $L2$  regularization, the smallest value of the MSE is achieved with  $\lambda = 0.001$  and

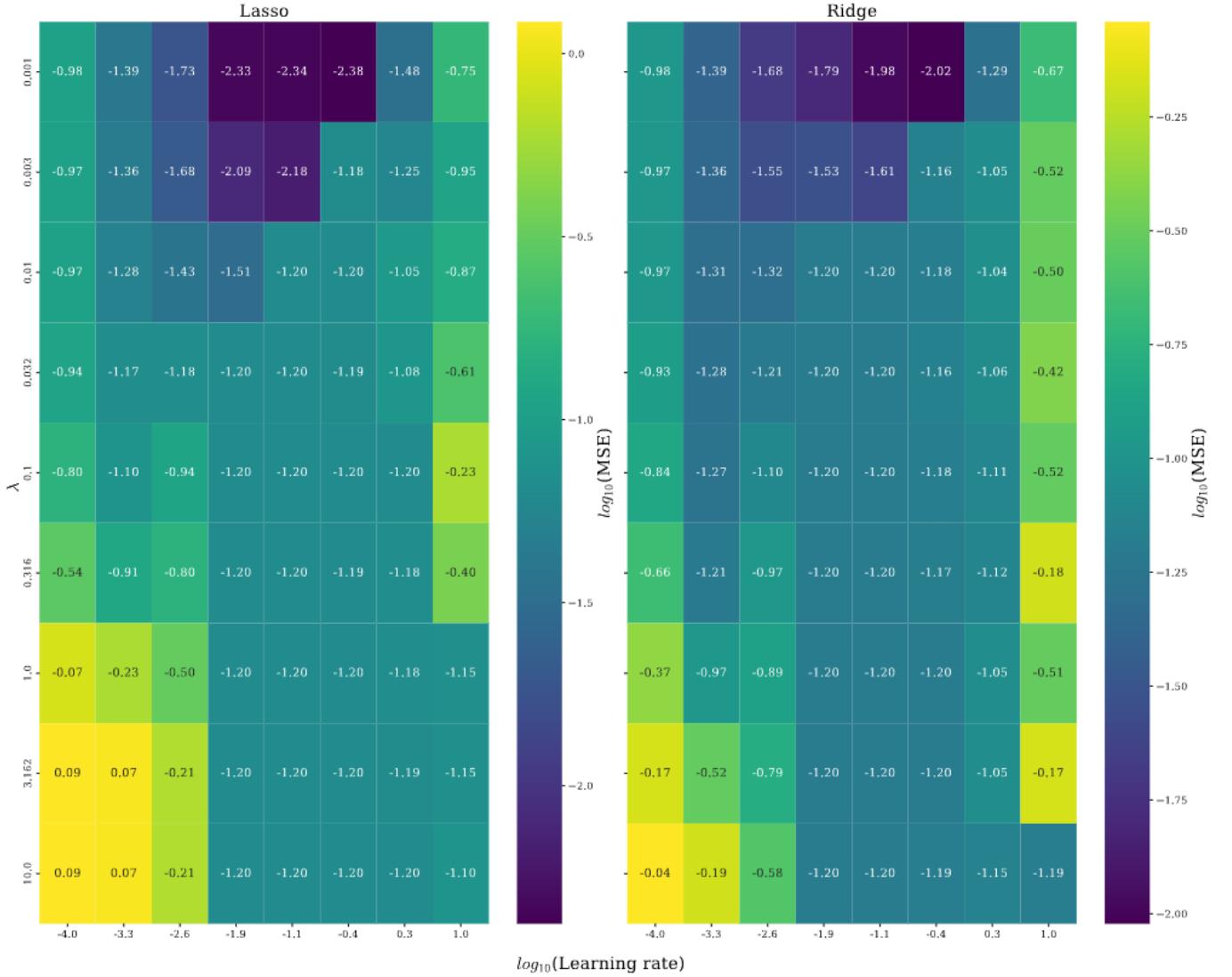


Figure 5: Testing MSE for regression analysis on Runge function with L1 (for Lasso) and L2 (for Ridge) regularization. On each heatmap, the logarithm (of base 10) of the testing MSE is shown as a function of the regularization parameter  $\lambda$  and the learning rate.

a learning rate  $\eta \approx 10^{-0.4}$ . Once again, we also notice that the logarithm of the MSE saturates to a value of  $-1.20$  for large enough values of the learning rate and regularization parameter. This time, however, the reason may be that the large regularization parameter causes the model to settle around a minimum in the MSE with weights and biases that are approximately equal to zero, instead of it being caused by a vanishing output of the activation function.

With our optimal  $\lambda$  and learning rate, we may now optimize the model architecture. In Figure 6, we show the results for the (logarithm of the) MSE as a function of the number of nodes, hidden layers and activation functions. We see that there is no clearly best model architecture, since there are many architectures with an MSE of about  $10^{-2.30}$  or less. Furthermore, we once

again find that with a sufficiently large model complexity (here given in terms of the number of hidden layers), the MSE saturates at  $10^{-1.20}$ .

With that said, we do see that the smallest MSE of  $10^{-2.54}$  is achieved for L1/Lasso regularization with one hidden layer and 25 neurons. Also, a possible model architecture for L2/Ridge regularization with an optimal MSE of  $10^{-2.54}$  is one with three hidden layers and 10 neurons.

As before, we may compare the  $R^2$  score for the optimal model architecture and regularization parameter with the results of scikit-learn after a cross validation with five folds on the scaled dataset. For L1 regression, our neural network gives a score of 93.6 %, whereas L2 regression gives 95.9 %. For a polynomial fit of degree 12,

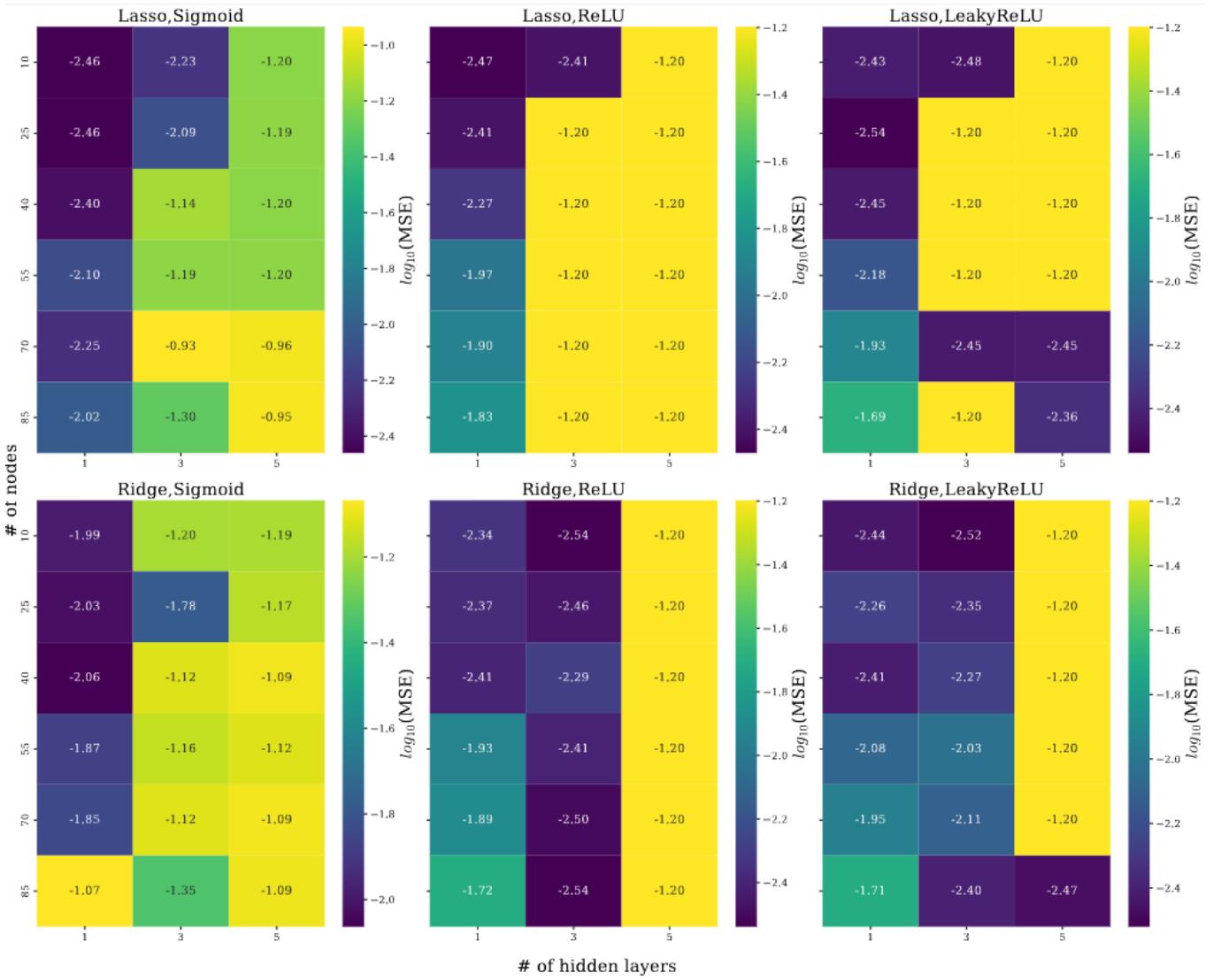


Figure 6: The logarithm of the testing MSE for regression analysis with L1 (Lasso) and L2 (Ridge) regression as a function of model architecture. Each heatmap displays the number of hidden layers on the horizontal axis, and the number of nodes on the vertical axis. The type of regularization and activation function is shown in the title.

scikit-learn, on the other hand, give scores of respectively  $-0.76.2\%$  and  $-0.89.4\%$ . This poor estimate may raise suspicion about numerical errors. However, we found similar values with a much smaller polynomial degree, and scikit-learn’s values do end up being much closer to our results when the regularization parameter is closer to 0. Thus, it appears, indeed, that the neural network performs much better than the polynomial fit with Ridge and Lasso regression.

## B. Classification

### 1. Optimizing a multiclass classification neural network

For the last problem, we analyze the performance of FFNNs on a multiclass classification task. We trained on the MNIST dataset (70,000 digits) with an 80/20 train/test split. The model used Softmax output with multiclass cross-entropy, optimized with the Adam algorithm with a batch size of 128, as well as parameters  $\beta_1 = 0.9$  and  $\beta_2 = 0.999$  [4].

We first trained a network with two hidden layers with 128 nodes for 2 epochs while sweeping a logarithmic grid of 5 learning rates  $\eta \in [10^{-1}, 10^1]$  and 6  $L_2$  penalties  $\lambda \in [10^{-10}, 10^{-5}]$ . As shown in Figure 7, the test accuracy varied little across  $\lambda$  values in this range. However, we see

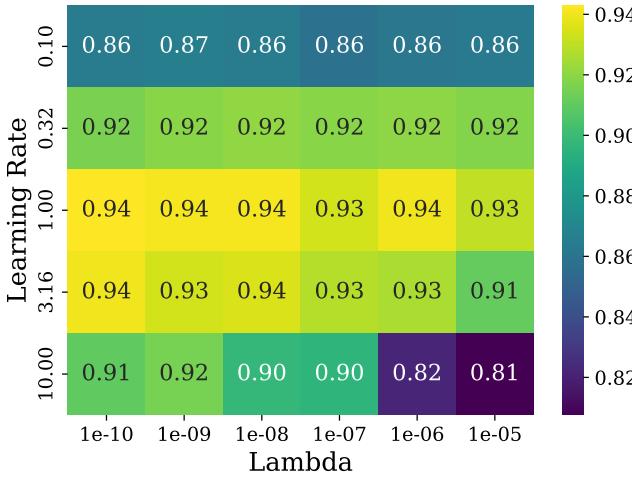


Figure 7: Heatmap plot of accuracies on the test split of the MNIST dataset for neural networks trained with a grid of learning rates and regularization parameters  $\lambda$ .

a clear difference in accuracy across the learning rate/ $\eta$  axis. The accuracy rises from  $\approx 0.86$  at  $\eta = 10^{-1}$  to around  $0.94$  for  $\eta = 10^0$  and  $\eta = 10^{0.5}$ , with the best accuracy of  $0.943$  at  $\eta = 10^0 = 1.0$  and  $\lambda = 10^{-10}$ . Only at the largest learning rate,  $\eta = 10$ , do we see a clear dependence on  $\lambda$ . The accuracy is still  $\approx 0.91$  for small  $\lambda$ , but drops to  $0.82$  and  $0.81$  at  $\lambda = 10^{-6}$  and  $\lambda = 10^{-5}$ . This indicates that the regularization shrinks the parameters so much that the network cannot fit the data with only two epochs. Overall, the grid reveals a clear learning rate sweet spot similar to the results in Figure 1 in the regression case, but the  $\lambda$  values in the chosen range do not significantly affect the accuracy.

Setting  $\eta = 10^0$  and  $\lambda = 10^{-10}$ , and training with 3 epochs this time, we scanned for the best combination of the number of hidden layers  $n_{\text{layers}} \in \{1, 2, 3\}$  and nodes per layer  $n_{\text{nodes}}$  equal to powers of 2 from  $2^5$  to  $2^9$ . The results were broadly similar across much of the grid, so we performed a new search with 6 epochs using the best range of values, namely  $n_{\text{layers}} \in \{1, 2\}$  and  $n_{\text{nodes}} \in \{128, 256\}$ . As seen in Figure 8, there were few differences in performance between these models. A single hidden layer with enough neurons appears sufficient to learn the dataset effectively.

We continue with the best performing values of  $n_{\text{layers}} = 1$  and  $n_{\text{nodes}} = 256$ , which gave an accuracy of  $0.9611$ , in order to analyze the effects of changing the activation function from Sigmoid to ReLU and LeakyReLU. Figure 9 shows the model accuracy on both the training and test set after each of the 5 epochs. With the Sigmoid function, a higher accuracy was reached faster and with a slightly higher value at the end, compared to ReLU and LeakyReLU for this model architecture, as seen in Table III. Given that the neural network only consists of 1 hidden layer, it appears that a smoother activation function, like Sigmoid, is more favorable here. Deeper neural

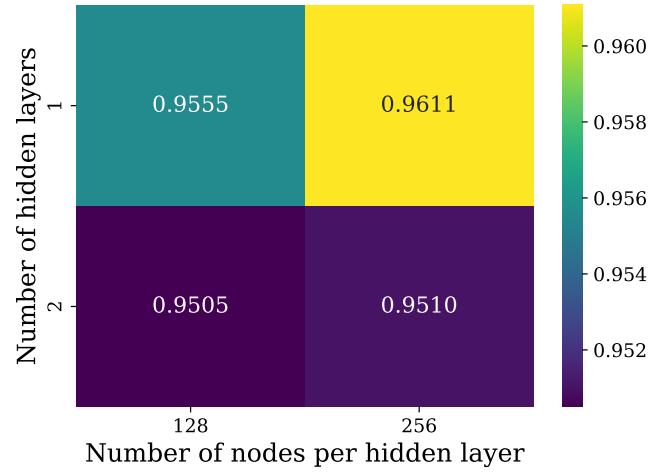


Figure 8: Heatmap plot of accuracies on the test split of the MNIST dataset for neural networks trained with a grid of the number of hidden layers and the number of nodes per hidden layer.

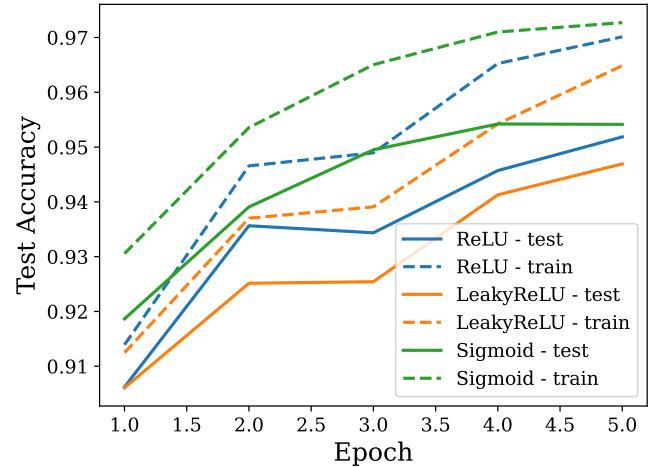


Figure 9: Plot of neural network train- and test-accuracy on the MNIST digits dataset as a function of the number of epochs.

networks or different model architectures could alter this result.

Table III: Neural Network train-set and test-set accuracy on the MNIST digits dataset after 5 epochs, using different activation functions for the hidden layers.

Activation function	Train accuracy	Test accuracy
Sigmoid	0.9727	0.9541
ReLU	0.9701	0.9519
LeakyReLU	0.9649	0.9469

We then trained the best found model architecture for a final session of 30 epochs to achieve the highest pos-

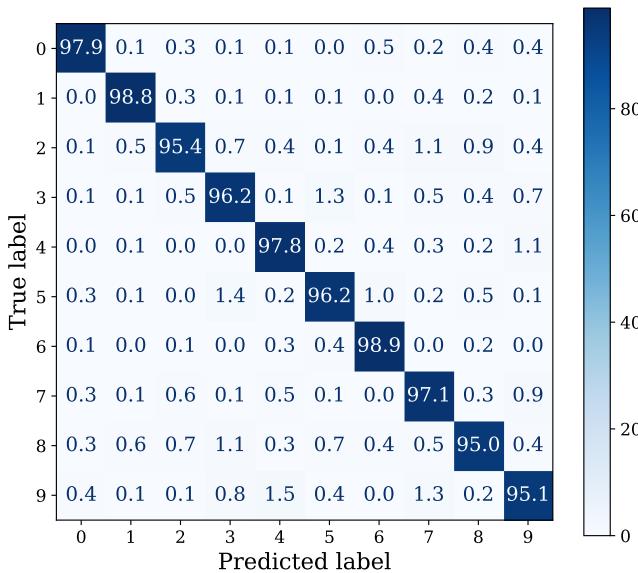


Figure 10: Confusion matrix for the model with the best found hyperparameters on the MNIST digits test-set. Values are given in percentages.

sible final accuracy. We found that the optimal model architecture was the following:

- Adam with a learning rate  $\eta = 1.0$
- Regularization parameter  $\lambda = 10^{-10}$
- 1 hidden layer with 256 nodes
- Sigmoid activation function

This gave a final test-set accuracy of 0.969. The confusion matrix for this model, shown in Figure 10, reveals which digits are most frequently incorrectly predicted. The largest misclassifications, from a true label to a false prediction, are  $9 \rightarrow 4$  (with a false classification rate of 1.55%),  $5 \rightarrow 3$  (with a rate of 1.41%), and  $9 \rightarrow 7$  (with a rate of 1.34%). These digits share similar curves and loops, and could therefore be hard for the model to distinguish in some cases.

## 2. Comparison with PyTorch

To verify that our handwritten implementation of neural networks was solid, we repeated the same process using a PyTorch neural network. After finding the best architecture and hyperparameters in the same way as before, we finally trained it with 30 epochs. The final test-set accuracy ended up at 0.974, which is slightly higher than what we achieved with our own implementation. Even though the difference in accuracy is small and could partly be explained by random variation in initialization, the training time improvement was substantial. Our manual implementation required roughly

860 seconds to complete 30 training epochs, while the same training run with PyTorch finished in about 30 seconds. The PyTorch framework is highly optimized from years of development and thousands of contributors. This makes PyTorch far more practical for experimentation and iterative development. The reduced runtime allows for faster testing of hyperparameters, longer training, and repeated runs for statistical consistency. Overall, these results confirm that our own implementation is correct. Both approaches converge to nearly identical performance, but they also highlight how important optimized libraries are for achieving high performance in practical machine learning workflows.

## 3. Logistic regression classification

For comparison, we also performed a logistic regression analysis on the same MNIST dataset, which achieved a test-set accuracy of 0.921. Our own neural network reached a significantly higher accuracy of 0.969, demonstrating its ability to capture more complex, nonlinear relationships in the data. Unlike logistic regression, a neural network can approximate arbitrary continuous functions, as stated by the universal approximation theorem.

## 4. Hyperparameter search discussion

Our hyperparameter search was unfair. We began with a Sigmoid model, identified the learning rate  $\eta$  and regularization parameter  $\lambda$  (and later also the number of neurons/layers) that worked best for that activation function. Only afterwards did we substitute in the ReLU and LeakyReLU activation functions while keeping these settings fixed. This procedure is efficient, but does not guarantee a globally optimal model across activation functions and model hyperparameters. Consequently, the ranking observed in Table III should be interpreted as "best under Sigmoid-tuned hyperparameters". A more thorough design would be to re-evaluate  $\eta$  and  $\lambda$  for each activation function, and double-check the optimal number of neurons and layers for the optimal pair. In addition to that, we only trained the model once with a specific random seed to initialize the weights and biases, but it would be much more accurate to try several seeds and calculate the average performance over these values. However, it was not feasible to do this because the manual implementation required substantial computation time, making it hard to iterate through several training and testing sessions.

## 5. Alternative model architectures

Although the fully connected FFNN performs well on the MNIST dataset, other architectures and methods are generally more effective for image classification tasks.

In particular, convolutional neural networks (CNNs) are designed to recognize local patterns such as edges and shapes. This makes them more parameter-efficient and robust to shifts or distortions [1, 3].

#### IV. CONCLUSION

In this project, we have implemented feed forward neural networks (FFNNs), and analyzed how well they perform on both regression and classification tasks, with data taken from the one-dimensional Runge function and the MNIST dataset, respectively. We also compared our implementation with the corresponding results from PyTorch. We found that FFNNs perform slightly better than OLS regression on the Runge function, with an  $R^2$  score of 96.4 % with the optimal hyperparameters, compared to 96.0 % for OLS regression. Moreover, it performs much better than linear regression with  $L_1$  and  $L_2$  regularization, having optimal  $R^2$  scores of 93.6 % and 95.9 % respectively, compared to -76.2 % and -89.4 % with linear regression. Finally, we achieved an optimal accuracy of 96.9 % on the MNIST dataset. Our manual neural networks underperformed compared to PyTorch, as it gave a slightly higher accuracy of 97.4 % on the MNIST dataset, and about three orders of magnitude smaller MSE on the Runge dataset.

We found that neural networks did have an advantage on our datasets by giving slightly more accurate results compared to simpler methods, such as linear and logistic

regression, and by automatically finding an appropriate fit to the Runge function (including the edges). However, they required extensive analysis to fine-tune the hyperparameters. There was also often a lot of noise in the performance of different model architectures, meaning that even the simplest networks (with one hidden layer and a few neurons) sometimes gave the best results.

Furthermore, it was also found that stochastic gradient descent performed better than plain gradient descent, having both a faster rate of convergence and a smaller final MSE in the case of regression. Moreover, for the classification task, it was indispensable due to the large number of minibatches used. However, the algorithm often suffered from significant stochastic fluctuations, giving some uncertainty in the final performance after training. To mitigate this, one could also optimize the parameters of the learning schedule.

Our analysis was limited by the fact that we used relatively simple datasets, which meant that implementing neural networks gave little reward for the effort required. Moreover, our grid search was limited by the range of values tested, meaning that we did not find the globally best hyperparameters. In addition to using more complex datasets and broader ranges, it would also be an advantage to use more accurate resampling techniques (such as cross validation) to evaluate the performance on the testing dataset for different hyperparameters. That could reduce some of the noise, and hence give more robust estimates for the optimal values of these parameters.

- 
- [1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016), <http://www.deeplearningbook.org>.
  - [2] F. Rosenblatt, Psychological Review **65**, 386 (1958), URL <https://doi.org/10.1037/h0042519>.
  - [3] M. Hjorth-Jensen, *Applied data analysis and machine learning* (2025), URL [https://compphysics.github.io/MachineLearning/doc/LectureNotes/\\_build/html/intro.html](https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html).
  - [4] H. E. H. Frederik Callin Østern, Viktor Bilstad (2025), URL [https://github.com/HeineEH/FYS-STK3155-4155\\_Project1/blob/main/Project\\_1.pdf](https://github.com/HeineEH/FYS-STK3155-4155_Project1/blob/main/Project_1.pdf).
  - [5] A. Jentzen, B. Kuckuck, and P. von Wurstemberger, *Mathematical introduction to deep learning: Methods, implementations, and theory* (2025), 2310.20360, URL <https://arxiv.org/abs/2310.20360>.
  - [6] T. Kurbiel, *Derivative of the softmax function and the categorical cross-entropy loss* (2021), medium (TDS Archive) – 6 min read, URL <https://medium.com/data-science/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-2a2a2a2a2a2a>.
  - [7] J. Duchi, E. Hazan, and Y. Singer, Journal of Machine Learning Research **12**, 2121 (2011).
  - [8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., *Pytorch: An imperative style, high-performance deep learning library* (2019), 1912.01703, URL <https://arxiv.org/abs/1912.01703>.
  - [9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., Journal of Machine Learning Research **12**, 2825 (2011), URL <http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html>.