Developing a Feed-Forward Neural Network from Scratch for Regression and Classification Analysis

FYS-STK4155 - Project 2

Jenny Guldvog, Ingvild Olden Bjerkelund, Sverre Johansen & Kjersti Stangeland
University of Oslo

(Dated: November 10, 2025)

Machine learning has become an essential tool for modeling complex relationships in datasets. However, it is often unclear when the increased model complexity of neural networks is justified compared to traditional regression methods. In this project, we develop a feed-forward neural network from scratch to investigate both regression and multiclass classification problems. The model is first applied to the one-dimensional Runge function and compared to Ordinary Least Squares, Ridge, and Lasso regression from our previous project. We also compare our own implementation with conventional libraries such as Scikit-learn, TensorFlow-Keras, and PyTorch. We then extend the implementation to the MNIST dataset to study the classification performance of our network. We evaluate different activation functions (Sigmoid, ReLU, and Leaky ReLU), regularization techniques (L1 and L2 norms), and network architectures. Our results show that neural networks outperform linear regression for nonlinear problems, especially with ReLU and Leaky ReLU activations, while regularization offers limited benefits for smooth functions. The custom implementation performs comparably to established libraries, and the MNIST classifier achieves over 96 % accuracy. These results demonstrate the flexibility of neural networks but also highlight that their added complexity is not always necessary for simple regression tasks.

I. INTRODUCTION

As technology continues to evolve, the influence of computational methods expands to nearly every aspect of modern science. Among these methods, machine learning (ML) has become a central tool due to its ability to recognize patterns and solve complex problems. Its applications range from simple classification tasks to real-world domains such as speech and image recognition [1].

In ML, a neural network is a computational model designed to learn patterns in data and make predictions based on them. Rather than being explicitly programmed to follow a set of rules, neural networks infer underlying relationships through exposure to examples, allowing them to generalize to unseen data, thus the machine autonomously learns how to solve the problem. This raises fundamental questions about how computers can represent and learn structure — a topic closely related to our understanding of intelligence itself. Artificial neural networks (ANNs) draw inspiration from the structure and information processing of biological systems such as the human brain [1].

Modern neural networks are built on well-established mathematical frameworks, most notably the principles of gradient descent and the chain rule of differentiation, which date back to the 17th and 19th centuries [2]. As the field developed, new architectures and optimization methods enabled networks to handle increasingly complex data, emphasizing the importance of regularization and efficient learning algorithms [3].

An illustration of a neural network is shown in Figure 1. Panel (a) displays a simple single-layer perceptron, while panel (b) shows a deeper network with multiple hidden layers. Each neuron applies nonlinear activation functions to weighted inputs, allowing the network to capture complex, non-linear relationships in data. Training is achieved by adjusting weights and biases to minimize a chosen cost function, using algorithms such as backpropagation [1].

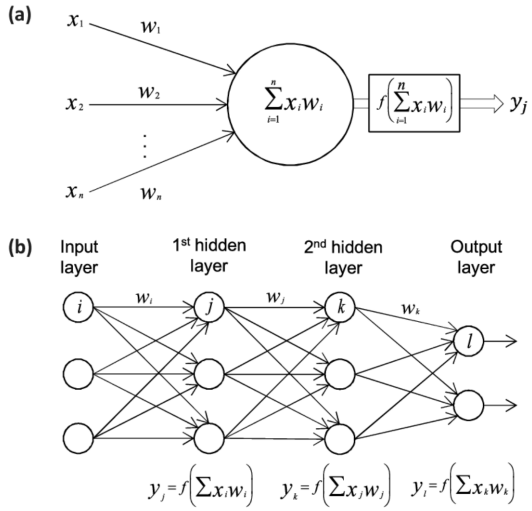

FIG. 1. Neural network architectures. a) A single perceptron model, and b) a multilayer perceptron. A network with two hidden layers, an input layer, and an output layer. Retrieved from [4].

An important theoretical result explaining the expressive power of neural networks is the universal approximation theorem. It states that a neural network with a single hidden layer and a finite number of neurons can approximate any continuous function to an arbitrary degree of accuracy [1, § 4]. By systematically combining simple nonlinear building blocks, complex functions can

be represented step by step. This property illustrates why neural networks are capable of modeling highly non-linear relationships and why they have become so powerful across a wide range of applications.

While libraries such as PyTorch and Scikit-learn provide efficient and well-tested implementations of neural networks, we also develop our own feed-forward neural network (FFNN) from scratch to better understand the underlying principles of training and generalization. By comparing the performance of this implementation to that of standard library models, we gain insight into both the theoretical foundation and the practical behavior of neural networks.

In this project, we study both regression and classification tasks to explore the versatility of neural networks. We begin by applying the network to a one-dimensional Runge function [5] and compare with our results with previous methods from an earlier project [6], and continue by optimizing the network with sensitivity analyzes of the learning rate before introducing stochastic gradient descent and further optimization methods. This comparison highlights how non-linear models can capture complex patterns beyond the capabilities of linear methods. In the second part, we extend the FFNN to handle multiclass classification using the MNIST dataset [7], where the goal is to classify handwritten digits. By analyzing how activation functions and regularization techniques influence performance across these two problem types, we aim to better understand when neural networks provide clear advantages over simpler approaches such as polynomial regression. A central practical challenge we revisit from Project 1 is overfitting: models with high capacity can fit noise and develop unstable edge behaviour on the Runge function. Neural networks may mitigate such artefacts, but they also risk overfitting without appropriate regularization and model selection.

In Section II, we describe the theoretical framework of neural networks and outline the mathematical foundation of the project. Section III presents the methods and implementation details, while Section IV reports our main findings and discusses their implications. We provide a brief conclusion and outlook in Section V. Additional results are provided in Appendix V. All code developed for this report is available on our GitHub repository.[1]

II. THEORY

A. Theoretical Background

An ANN is a computational model inspired by biological neural systems. A typical network consists of an input layer, one or more hidden layers, and an output layer. A basic illustration of such a network is shown in Figure 1, where each column of circles represents a layer, and each circle corresponds to a node or neuron. The arrows indicate the weighted connections between nodes and the direction of information flow through the network. Each node in one layer is connected to every node in the next, forming a fully connected architecture. Neural networks extend traditional supervised learning methods such as linear and logistic regression by introducing nonlinear transformations that enable them to capture more complex relationships in data [1, 3, 4, 8].

The FFNN, a specific type of ANN, represents the earliest and simplest form and is the one used in this project. In an FFNN, the information flows in a single direction—from the input layer, through any hidden layers, to the output layer [4]. Other network architectures also exist, such as convolutional neural networks (CNNs), which employ specific connection patterns but still propagate information forward, and recurrent neural networks (RNNs), where information can circulate in loops, allowing feedback connections [4].

The FFNN operates by first assigning each input variable to its corresponding input node. The user specifies the number of layers and the number of nodes in each layer. Each connection between nodes is initialized with a random weight and bias, meaning that each layer l has associated parameters $\phi^{(l)} = (W^{(l)}, b^{(l)})$. The forward propagation through the network is then computed layer by layer using the equations

$$z^{(l)} = a^{(l-1)}W^{(l)} + b^{(l)}, \tag{1}$$

$$a^{(l)} = \sigma(z^{(l)}), \tag{2}$$

where $z^{(l)}$ is the linear combination of inputs and weights for layer l, $\sigma(\cdot)$ is the activation function, and $a^{(l)}$ is the resulting output (or activation) from that layer.

B. Basic Concepts

There are many different ways to apply ML, but two of the most common approaches, and the ones we will focus on in this project, are regression and classification.

Regression refers to a type of ML where the task is to predict a continuous numerical value $y \in \mathbb{R}$ given some input $\mathbf{x}$. In this case, the objective is to approximate a function

$$f : \mathbb{R}^n \to \mathbb{R},$$

where n denotes the number of input features [3, § 5.1]. Regression tasks are therefore used when the output variable is continuous, such as predicting temperature.

Classification, on the other hand, concerns assigning an input to one of several discrete categories. Here, the goal is to learn a function

$$f : \mathbb{R}^n \to \{1, \ldots, k\},$$

[1] https://github.com/kjes4pres/Project_2_FYSSTK

where $\mathbb{R}^n$ represents the feature space of the input data. The function $y = f(\mathbf{x})$ maps each input $\mathbf{x}$ to a specific class label. In some cases, instead of predicting a single class, the model outputs a probability distribution over all possible classes [3, § 5.1].

We will compare our results to previous approaches using traditional linear regression methods, namely Ordinary Least Squares (OLS), Ridge, and Lasso regression, as seen in our first project [6].

The OLS method provides an analytical solution for the optimal parameter vector:

$$\hat{\boldsymbol{\theta}}_{\text{OLS}} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}, \tag{3}$$

where $\boldsymbol{X}$ denotes the design matrix and $\boldsymbol{y}$ the target vector. The final predictive model is then given by

$$\tilde{\boldsymbol{y}} = \boldsymbol{X}\hat{\boldsymbol{\theta}}. \tag{4}$$

To address issues such as overfitting and improve model generalization, regularization methods such as Ridge and Lasso regression have been introduced. The analytical solution for Ridge regression is

$$\hat{\boldsymbol{\theta}}_{\text{Ridge}} = \left(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}, \tag{5}$$

where $\lambda > 0$ is the regularization parameter that controls the strength of the penalty on the model coefficients.

Unlike OLS and Ridge, Lasso regression does not have an analytical solution due to the use of the ℓ_1-norm penalty term, and must therefore be solved using numerical optimization techniques such as gradient descent.

For additional background on OLS, Ridge, and Lasso regression, see our previous report [6]. These methods form the linear baseline models against which we will compare our neural network implementations in the following sections.

C. Cost Functions and the Chain Rule

In both linear models and neural networks, the cost function quantifies how well the model's predictions match the target values. The model parameters are then updated in the direction that minimizes this cost.

For regression tasks, a common choice is the mean squared error (MSE), defined as

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2, \tag{6}$$

where y_i are the target values, $\tilde{y}_i$ are the model predictions, and n is the number of data points. Minimizing the MSE corresponds to minimizing the average squared distance between predictions and observations.

1. The chain rule and gradient computation

In a simple linear regression model, the gradient of the MSE with respect to the weights can be derived directly as

$$\frac{\partial C}{\partial \mathbf{w}} = -\frac{2}{n}\boldsymbol{X}^T(\boldsymbol{y} - \tilde{\boldsymbol{y}}). \tag{7}$$

However, in a neural network, each weight affects the output indirectly through several layers and activation functions. Therefore, we must use the chain rule to compute the gradients efficiently.

For a single neuron, the output is given by

$$z = \mathbf{w}^T\mathbf{x} + b, \qquad a = f(z),$$

where f is the activation function. The cost C depends on the final output a, which in turn depends on both $\mathbf{w}$ and b. Using the chain rule, the gradient of the cost with respect to the weights can be expressed as

$$\frac{\partial C}{\partial W} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial W}. \tag{8}$$

Here,

$$\frac{\partial C}{\partial a} = -2(y - a), \qquad \frac{\partial a}{\partial z} = f'(z), \qquad \frac{\partial z}{\partial W} = \mathbf{x}.$$

Combining these terms gives

$$\frac{\partial C}{\partial W} = -2(y - a)f'(z)\mathbf{x}. \tag{9}$$

This expression illustrates how the chain rule allows the gradient to be written as a product of local derivatives, which forms the basis of the backpropagation algorithm used in training neural networks.

2. Extension to multiple layers

For a network with multiple layers, this principle is applied recursively. Each layer passes its error backward through the network, allowing the model to compute $\frac{\partial C}{\partial \mathbf{w}^{(l)}}$ for all layers l. This process enables efficient training of deep neural networks by gradient descent.

3. Regularization

A central goal in machine learning is to ensure that models generalize well beyond the training data. Regularization achieves this by penalizing model complexity and thereby reducing overfitting. Various regularization strategies exist, but in this project we focus on L_2 (Ridge) and L_1 (Lasso) regularization [3, § 7]. We denote the regularization strength by λ (called α in [3]).

With L_2 regularization, the total cost function becomes

$$\tilde{C}(W) = \frac{\lambda}{2}\|W\|_2^2 + C(W), \tag{10}$$

where $C(W)$ denotes the unregularized loss function. The gradient is

$$\nabla\tilde{C}(W) = \lambda W + \nabla C(W). \tag{11}$$

A single gradient descent update with learning rate η is then

$$W \leftarrow W - \eta\big(\lambda W + \nabla C(W)\big) = (1-\eta\lambda)\, W - \eta\, \nabla C(W),$$

which shows that L_2 regularization acts as *weight decay*: the weights are scaled by $(1-\eta\lambda)$ at each step, multiplicatively shrinking them, in addition to the usual gradient update [3, § 7.1.1].

L_1 regularization, on the other hand, adds a penalty proportional to the absolute value of the weights, which penalizes large weights and tends to drive many parameters toward zero, which promotes sparsity in the model. The total cost function becomes

$$\tilde{C}(W) = C(W) + \lambda \sum_j |w_j|, \tag{12}$$

where $C(W)$ is the standard loss function and $\lambda > 0$ controls the strength of the regularization.

Because the absolute value function is not differentiable at $w_j = 0$, we use a subgradient defined as

$$\frac{\partial |w_j|}{\partial w_j} = \begin{cases} \text{sign}(w_j), & w_j \neq 0, \\ \text{any value in } [-1,1], & w_j = 0. \end{cases}$$

The gradient descent update therefore becomes

$$w_j \leftarrow w_j - \eta\Big(\frac{\partial C}{\partial w_j} + \lambda\ \text{sign}(w_j)\Big),$$

which pulls each weight slightly towards zero at every step. Small weights are often driven all the way to zero, resulting in a sparse parameter set [3, § 7.1.2].

In the following section, we will consider cost functions for classification tasks, where the output represents discrete classes instead of continuous values. The most common choice in this case is the cross-entropy loss.

4. Binary Cross Entropy

Binary cross entropy, also called log loss, are the functions used when we have a classification case where we want to outputs; either yes/no, 0/1 etc. Although binary classification is not used in the main experiments of this project, we include it here for completeness, as it forms the conceptual foundation for multiclass classification.

For a binary classification (logistic regression) problem, the cost function is given by the negative log-likelihood (cross entropy):

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\Big[\log\big(1+e^{z_i}\big) - y_i z_i\Big], \tag{13}$$

where $z_i = \theta_0 + \boldsymbol{\theta}^\top \mathbf{x}_i$.
An equivalent form is:

$$C(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n}\Big[y_i \log(\sigma(z_i)) + (1-y_i)\log(1-\sigma(z_i))\Big], \tag{14}$$

where $\sigma(z_i) = \frac{1}{1+e^{-z_i}}$ is the sigmoid-function. For further background and derivations, see the lecture notes by Hjorth-Jensen [4].

To control model complexity, a regularization term can be added (again, we use the L_2 and L_1 regularization):

$$C_{\text{L2}}(\boldsymbol{\theta}) = C(\boldsymbol{\theta}) + \frac{\lambda}{2n}\sum_{j=1}^{p}\theta_j^2, \tag{15}$$

$$C_{\text{L1}}(\boldsymbol{\theta}) = C(\boldsymbol{\theta}) + \frac{\lambda}{n}\sum_{j=1}^{p}|\theta_j|. \tag{16}$$

5. Multiclass cross entropy

For a classification problem with K classes, the softmax function is applied to the network outputs (logits) z_i to obtain probabilities [4]:

$$s_i = \text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad i = 1, \ldots, K. \tag{17}$$

The multiclass cross-entropy cost (also called the Softmax loss) is then defined as

$$C = -\frac{1}{n}\sum_{p=1}^{n}\sum_{i=1}^{K} y_{pi}\log(s_{pi}), \tag{18}$$

where n is the number of samples, s_{pi} is the predicted probability for class i for sample p, and y_{pi} is the corresponding one-hot encoded target value.
The gradient of the cost function with respect to the logits z_i is given by

$$\frac{\partial C}{\partial z_i} = s_i - y_i.$$

This simple and elegant result arises because the derivative of the softmax function is

$$\frac{\partial s_i}{\partial z_j} = \begin{cases} s_i(1-s_i), & \text{if } i = j, \\ -s_i s_j, & \text{if } i \neq j, \end{cases}$$

and when combined with the cross-entropy loss, the terms simplify to $s_i - y_i$.

The softmax function is what we will use for our MNIST classification problem.

D. Activation Functions

An activation function determines how the neurons in a neural network transform their inputs, and is what enables the model to learn complex, nonlinear relationships in data. Without activation functions, the network would consist only of linear transformations, regardless of depth and layers, and would therefore behave as a linear regression model. Nonlinear activation functions are thus essential for allowing neural networks to represent complex and hierarchical patterns [8, § 12].

In this project, we use three common activation functions: the sigmoid (logistic) function, the rectified linear unit (ReLU), and its variant, the Leaky ReLU. These were chosen because they represent different approaches to handling nonlinearity and gradient propagation, while also being widely used in modern deep learning. The sigmoid function provides a probabilistic interpretation, whereas ReLU and Leaky ReLU mitigate the vanishing gradient problem that often occurs in deeper networks. For regression tasks, the output layer is left without an activation function (or the identity function) to preserve linearity in the final output.

a. Sigmoid (Logistic) Function The sigmoid activation function is often viewed as the one that most closely resemble how biological neurons behave [8]. Its output can be interpreted as the probability that a neuron becomes "activated" or "fires" — in this context meaning that the neuron strongly responds to the input rather than producing a physical spike as in the brain. When the output is close to zero, the neuron will barely "fire," and learning slows down. When many neurons enter this low-activation region, the gradients become very small, causing inefficient learning and a higher risk of getting stuck in local minima during training. This phenomenon is known as the *vanishing gradient problem* [8, § 12].

Inputs received by the Sigmoid function is then squeezed between 0 and 1, serving as a good choice for probabilistic problems.

The sigmoid function is defined as

$$\sigma(z) = \frac{1}{1+e^{-z}}, \tag{19}$$

with the derivative

$$\frac{d\sigma(z)}{dz} = \sigma(z)\big(1-\sigma(z)\big).$$

b. Rectified Linear Unit (ReLU) A popular alternative that mitigates the vanishing gradient problem is the ReLU function, defined as

$$a = \mathrm{ReLU}(z) = \max(0, z). \tag{20}$$

Its derivative is given by

$$\frac{da}{dz} = \begin{cases} 0, & \text{if } z < 0, \\ 1, & \text{if } z > 0\ . \end{cases}$$

For positive inputs, ReLU simply outputs the input, and the gradients do not vanish. This allows for faster and more stable training. Furthermore, ReLU is still nonlinear, enabling the network to model complex relationships. However, neurons that consistently fall into the region $z \leq 0$ receive $\frac{da}{dz} = 0$ and effectively stop learning, a problem known as *dead neurons* [4, §§ 12 & 17].

c. Leaky ReLU To prevent neurons from becoming inactive, the Leaky ReLU introduces a small, non-zero slope for negative inputs. It is defined as

$$a = \mathrm{LeakyReLU}(z) = \max(\alpha z, z), \tag{21}$$

where $\alpha > 0$ is a small constant (typically $\alpha = 0.01$) [4]. The derivative is then

$$\frac{da}{dz} = \begin{cases} \alpha, & \text{if } z < 0, \\ 1, & \text{if } z > 0\ . \end{cases}$$

This modification maintains a weak gradient even for negative activations, reducing the number of dead neurons and improving gradient flow, which leads to faster and more stable training across layers [8, § 17].

E. Optimization Methods

Computing the Hessian matrix of the loss function can be complex and time consuming, and it is therefore advantageous to use the gradient descent methods. Here, the second derivative is approximated through a constant (or adaptive) learning rate. For further details about these methods we refer back to our previous project, [9].

1. Simple gradient descent

In basic gradient descent, the parameter update for each iteration is expressed as:

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t). \tag{22}$$

Here $\nabla f(\theta)$ represents the gradient of the objective function $f(\theta)$, and η is the learning rate. The gradient $\nabla f(\theta)$ will differ depending on the specific regression method used (such as OLS, Ridge, and Lasso), and the corresponding cost function.

2. RMSprop

RMSProp, or Root Mean Square Propagation, is a modification that makes the learning rate decay more

gracefully [10]. The parameter update for each iteration is given by:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla f(\theta_t). \tag{23}$$

Here ϵ is still a small constant added for numerical stability. G_t is defined as:

$$G_t = \beta G_{t-1} + (1-\beta)\nabla f(\theta_t)^2. \tag{24}$$

Here, the β-term controls the influence of previous gradients.

3. Adam

Adam, or Adaptive Moment Estimation, integrates the benefits of Momentum and RMSProp [11]. In each iteration, the parameter updates are calculated as follows:

1. The first moment update:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(\theta_t) \tag{25}$$

2. The second moment update:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla f(\theta_t)^2. \tag{26}$$

The bias-corrected estimates for these moments are given by:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}. \tag{27}$$

These corrections help reduce the bias towards zero, which can occur if the moments are initialized at zero. The final update is computed by:

$$\theta_{t+1} = \theta_t - \frac{\eta m_t}{\sqrt{v_t + \epsilon}}. \tag{28}$$

The first moment represents the mean (m_t), and the second moment represent the variance (v_t) of the gradients. As before, the β term governs the influence of past gradients on the learning rate.

4. Stochastic gradient descent

The theory behind stochastic gradient descent (SGD) lies in the assumption that the true data set can be understood as a sum of subsets, and that the cost and its derivative follows that same assumption, as shown in equation 29. This means that model only processes batches of the dataset at each iteration, introducing randomness with randomizing the batch chosen. The number of subsets are defined as the number of samples in the data n, divided by the batch size M.

$$\nabla_\theta C(\theta) = \sum_{i=0}^{n-1} \nabla_\theta c_i(\boldsymbol{x_i}, \theta) \tag{29}$$

This makes training with SGD computationally more efficient and can prevent memory errors that occur when using the whole dataset each update. It can also prevent getting stuck in local minima as the derivative will be evaluated at different localities in the cost landscape [4].

5. Hyperparameters

The learning rate and the number of epochs (iterations) are key hyperparameters that control how the network learns. Choosing these appropriately is crucial for achieving convergence and avoiding overfitting or underfitting. As discussed in [8, §§ 2, 3, 6], several techniques exist for tuning hyperparameters. In this project, the optimal learning rate is selected based on MSE, (equation 6), while the number of epochs is randomly chosen.

Finding the correct hyperparameters is crucial, as this affect how well the model generalizes, and the overall learning. Choosing a learning rate that is too large can cause the algorithm to overshoot a global minimum (further exaplanation on this is given in [8, § 2, 3]). On the other side, choosing a too small learning rate can make training inefficient, as the algorithm then requires many epochs to reach convergence, especially for large datasets.

F. Theoretical Expectations and Hypotheses for Regression

In our previous project, OLS regression performed better than both Ridge and Lasso regression due to the simplicity of the model and the model's low variance [6]. In such cases, regularization is unnecessary and only adds bias, thereby reducing performance. This reflects the regularization–generalization tradeoff, often referred to as the bias–variance tradeoff.

When using neural networks with multiple hidden layers and nonlinear activation functions, we expect more complex underlying patterns. Complex functions generally have higher variance and an increased risk of overfitting, where regularization can help improve generalization. The model becomes more flexible through the use of nonlinear activation functions, which allow the network to represent curved and interacting relationships that linear models cannot capture. Consequently, we expect L_2 and L_1 regression to handle overfitting better, as both penalize large weights and stabilize the solution. Specifically, L_2 regularization is expected to reduce variance while retaining all features, whereas L_1 regularization may perform feature selection, potentially leading to sparser models. We therefore expect, for the neural network, L_2 to outperform OLS due to reduced overfitting, and L_1 to perform slightly worse if the underlying relationships are not sparse. As opposed to the polynomial models in Project 1, the neural network does not rely on explicit polynomial terms, and is therefore expected to avoid the boundary oscillations observed previously.

However, for simpler functions, neural networks are not necessarily superior, as they contain more parameters and therefore exhibit higher variance and computational cost (in terms of floating point operations, FLOPs). We expect comparable performance between neural networks and linear models on simple data, but lower computational efficiency for the neural networks, making linear models more practical in such cases.

Regarding activation functions, we expect the ReLU function to outperform the sigmoid function, as ReLU typically provides faster and more stable learning. The sigmoid function saturates for large $|z|$, leading to small gradients and slow learning, whereas ReLU maintains a constant gradient for positive inputs, allowing faster convergence. Furthermore, we expect Leaky ReLU to behave similarly to ReLU, but with slightly improved robustness by avoiding "dead neurons," thus supporting more stable gradient flow and convergence.

Overall, we expect regularization to improve generalization in more complex models, and networks using ReLU or Leaky ReLU activations to achieve faster and more stable learning compared to those using the sigmoid activation.

Finally, we expect the learning rate and number of epochs to affect both the speed of convergence and the final model performance, since non-optimal values can lead to either slow learning or unstable optimization.

G. Extension to multiclass classification

Neural networks have become the leading approach for complex pattern recognition problems such as image and speech recognition [1]. Even though their theoretical foundation is complex, they can be implemented in practice with relatively few lines of code, demonstrating the accessibility and efficiency of modern machine learning frameworks [1]. In this project, we apply such methods to image recognition using the MNIST dataset for number recognition[7].

Recognizing numbers and interpreting visual input are fundamental aspects of human perception—tasks and so ingrained in our daily lives that we rarely reflect on them. Nielsen [1, § 1] describes this ability as carrying around "supercomputers in our heads," refined through hundreds of millions of years of evolution. This naturally raises the question: how can we train a computer to perform the same task? What seems effortless for humans becomes a non-trivial computational challenge, and it becomes apparent why such models are referred to as *neural networks*, inspired by the structure and function of the human brain. A neural network approaches this problem differently from many traditional algorithms. By being "fed" a large number of training examples, in this case, thousands of handwritten digits,the model learns to extract underlying patterns and formulate its own internal rules for recognizing digits. Moreover, increasing the amount of training data generally improves the model's ability to generalize to new, unseen examples [1, § 1].

The MNIST dataset provides a standardized benchmark for this task, containing preprocessed images divided into separate training and test sets [7]. This allows us to focus on model design and optimization rather than data preparation.

To solve the classification task, we use the multiclass cross-entropy loss function as described in Section II C. The output layer applies the softmax function (Equation 17), which produces a probability distribution over the ten possible digit classes, representing the model's confidence that a given image corresponds to each of the ten digits.

1. *Confusion Matrix*

The confusion matrix visualizes a classification models performance by tabulating the predicted data and the true data together. This gathers together how many of the prediction are right and wrong, and splits it into subsets of true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN), for each class. The true positives tells us how many of the predictions are right per class. False positives tells us how many times a predicted class is labeled as something else in the true data. False negatives tells how many times a true labels, are predicted as something else, and the true negatives are counted as the sum of all samples minus TP, FP, and FN [4].

The true positive rate (TPR) gives us the rate at which the model is able to predict a class:

$$TPR_i = \frac{TP_i}{TP_i + FN_i},$$

where i is the class iteration. The false positive rate (FPR) indicates how often a model predict a class, labeled in the true data as something else:

$$FPR_i = \frac{FP_i}{FP_i + TN_i}.$$

III. METHODS

A. Building the neural network code

Our neural network is built as a Python class containing methods for forward propagation and backpropagation, updating weights, and training. The class is initialized with a number of input features, layer volume and size, activation functions with their respective derivatives, and the cost/loss function and its derivative. During class initialization, the model creates weight and bias matrices containing floating-point values sampled from a standard normal distribution. Their dimensions

are set according to the input size and layer structure. A supplementary initialization input for the network is the regularization terms L1 and L2 explained in section II. Moreover, additional methods for optimization through RMSProp and Adam are also included.

B. Implementation

We run our neural network (NN) with a variety of configurations to systematically evaluate its performance. For consistency, we initialize the dataset in the same manner as in Project 1, applying the same random seed for all computations and using 1500 samples, where the input variable x is sampled uniformly on the interval, $x \in [0, 1]$. We added Gaussian noise with a mean value of 0 and a standard deviation of 0.1. The data is split into test and training sets, with a 1:5 ratio, using Scikit-learn [12]. We reuse our scaling from project 1, where the input variable x is normalized by its mean [6].

$$f(x) = \frac{1}{1 + 25x^2}, \tag{30}$$

Next, we investigate how different optimization algorithms for stochastic gradient descent (SGD) used in backpropagation affect model performance. To validate our results, we compare them against outcomes from established software libraries. We then explore the influence of various activation functions within the hidden layers, along with the impact of network architecture-specifically, the number of hidden layers and the number of nodes per layer on predictive performance. Model performance is evaluated using the MSE (equation 6. Based on the best-performing architecture, we incorporate regularization terms into the network and assess their effects. The results are then compared with the Ridge and Lasso regression outcomes reported in Project 1 [6]. After thoroughly evaluating the network on the Runge function regression problem, we extend our analysis to a classification task. Specifically, we train the network on the MNIST dataset [7] and evaluate its performance using logistic activation.

1. *Prediction skill of NN on 1-D Runge*

Two neural networks were trained for the regression task using plain gradient descent: one consisting of a single hidden layer with 50 nodes, and another with two hidden layers containing 50 and 100 nodes, respectively. Both models were trained for 100 000 epochs using a learning rate of 0.01. The hidden layers employed the sigmoid activation function, while the output layer was linear. The MSE was used as the cost function. Using our best parameters from project 1 to approximate with OLS, we train a network with polynomial degree 8 and 1500 samples.

2. *Optimization methods*

To test different optimization methods, we keep the two model architectures the same and train them with seven learning rates between 0.001 and 2, making 14 models in total. All models iterate through 10000 epochs. When adding stochastic gradient descent and other optimizers, we train models for each architecture–optimizer pair, giving six models with stochastic gradient descent.

3. *Testing against other software libraries*

As a sanity check to ensure that our implementation of the derivatives, (equation 8) is implemented correct we compare the results to the Autograd library [13], which implements automatic differentiation. This is done by creating a random input and target for testing, and calculating the derivatives used in backpropagation.

To compare with other software libraries, the architecture of the neural networks needs to be as similar as possible. This includes selecting the same activation functions, number of layers and number neurons per layer, and a consistent choice of hyperparameters. For this reason, a short summary of the different libraries used to compare is given, for a more comprehensive description, the reader is directed to the literature and source code; Scikit-learn [12], TensorFlow Keras [14], and PyTorch [15].

For simplicity the stochastic gradient descent method is implemented with ADAM as optimizer for all networks (here $\beta_1 = 0.9$ and $\beta_2 = 0.99$). Moreover, the sigmoid activation function is used on all hidden layers, and there are no regularization terms added. As there is no option to define if the batches are selected with or without replacement in the libraries, we have chosen to not use replacement in our for a better comparison. However, without replacement in sampling, the samples are not independent, and the probability of selecting one sample does not stay the same. Scikit recommends shuffling to the training data before every iteration, and according to [8, p. 46], presenting the training data in random order, and implementing shuffling is important. For simplicity, all models have shuffling of training data implemented.

For a fair comparison, we set the number of epochs first to 300 then to 1000, to view the possibilities of underfitting and overfitting, respectively. Batch size is 20, and learning rate is 0.01.There is no implementation of early stopping criteria, as we want the models to be as similar as possible. However, it is worth noting that for future implementation, the stopping criteria could improve the model performance. To view the models capability of handling complexity, we compare one simple model with 1 hidden layer and 10 neurons to a more complex model with 5 hidden layers with 20 neurons each.

4. Architecture and Activation Functions

To evaluate network architecture and activation functions using the one-dimensional Runge dataset, we test how changing the number of hidden layers and nodes per layer affects predictive performance. Specifically, the network is trained with 49 different architectural configurations, with the number of hidden layers ranging from 1 to 7 and the number of nodes per layer varying from 2 to 128. Training is performed using stochastic gradient descent with the Adam optimizer for backpropagation, employing a batch size of 20 samples, 300 epochs, and a learning rate of 0.01. This process is repeated for three different activation functions: the sigmoid function (equation 19), the ReLU function (equation 20), and the leaky ReLU function (equation 21). In total, 147 neural networks are trained and evaluated. Model performance is assessed using the MSE calculated on the test set.

5. L_1 and L_2 Norms

Building on the best-performing architectures identified previously (in terms of number of layers and nodes), we construct new networks that incorporate L_1 and L_2 regularization, as defined in equations 12 and 10. For each architecture, we train the network using both regularization types across a range of hyperparameter values and learning rates. Specifically, we consider regularization strengths $\lambda \in (10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1)$ and learning rates $\eta \in (0.001, 0.01, 0.1, 1.0, 1.01, 1.1, 1.5, 2.0)$. This procedure is repeated for all three activation functions (sigmoid, ReLU, and leaky ReLU). In each case, the corresponding activation functions and their derivatives are applied to the hidden layers, with the identity function used for the output layer. Training is performed using stochastic gradient descent with the Adam optimizer, and both training and test errors are evaluated using MSE. Overall, this process results in a total of 336 trained networks. Model performance is assessed using the MSE calculated on the test set and compared to the findings of Ridge and Lasso regression on the same dataset in project 1 [6].

6. Classification analysis

Following the same structure of analysis, we initialize new models and train them through stochastic gradient descent with Adam. Introducing a new dataset (MNIST, [7]) we change the input size to the amount of features needed, and the output layer size to the amount of classes there are in the target dataset. The cost function used is cross-entropy, and our output layer activation function is changed to softmax. We look at how the three activation functions, in cooperation with a Softmax function for the output layer, perform with a varying number of layer, nodes, and learning rates. Optimizing these parameters, we analyze how regularization terms L_1 and L_2 affect the train to test scores, observing if over-/underfitting occurs. After quick analysis, we decided on running all models for 30 epochs, which gives the models enough time to converge, which happens at around 20 epochs.

The final model has one layer, 128 nodes, sigmoid activation function for hidden layer, and softmax for output layer. Trained for 30 epochs with batch size 1000, learning rate 0.01, and Adam optimization. The cost function has L1 penalty with lambda at 3e-2. We employ the scikitplot python package [12], a part of sklearn, to produce a confusion matrix.

C. Software

Our implementation relies heavily on several open-source Python packages. The NN is primarily built using NumPy for efficient vectorized operations [16]. To evaluate and compare our methods, we also employ Autograd [13], PyTorch [15], Keras [14], and TensorFlow [17]. For data handling and visualization, we use Pandas [18], Matplotlib [19], and Seaborn [20] to generate plots and heat maps. The code is available in our GitHub repository: https://github.com/kjes4pres/Project_2_FYSSTK.

D. Use of AI tools

Artificial Intelligence (AI) tools, namely ChatGPT [21] and GPT UiO [22], has been used for parts of this project. The usage has been limited and primarily used for debugging code, formatting of figures, making descriptive documentation strings for functions, and grammatical help when writing the report. All usage have been quality checked by the authors.

IV. RESULTS AND DISCUSSION

A. Linear regression

1. Comparing the neural network to linear regression results

We compare the OLS results from earlier work to predictions produced by two trained models, the first with one layer and 50 nodes, the second with two layers with 50 and 100 nodes respectively. The cost history of the two models are shown in Figure 2. The performance of the model improves in both cases indicating successful training.

Both models converge, with the two-layer network converging faster and reaching a lower final cost. Given that a single hidden layer is theoretically sufficient to approximate any continuous function (universal approximation

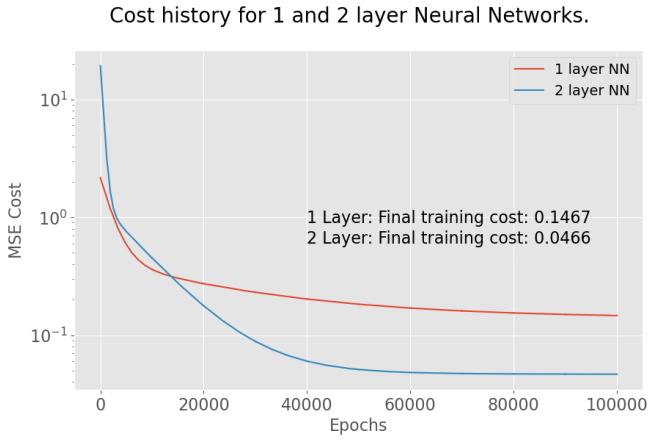

FIG. 2. Cost histories of training with gradient descent. Two models, a one-layer model with 50 nodes, and a two-layer model with 50 and 100 nodes respectively. Learning rate is set to 0.01 for both models. Final train mse is annotated in the center.

	Train	Test
OLS		0.0127
One-layer Model	0.1467	0.1501
Two-layer Model	0.0466	0.042

TABLE I. Comparison of MSE scores between OLS (Project 1) and neural networks with one (50 nodes) and two (50, 100 nodes) hidden layers, both trained with a learning rate of 0.01 using sigmoid activations and an identity output.

theorem), the difference in performance likely reflects an optimization challenge rather than a limitation in model capacity — suggesting that the one-layer model may be trapped in a local minimum.

The prediction cost for both model approximations and the OLS approximation is tabulated in table IV A 1. We then tried to optimize by adjusting the learning rate for both model architectures. Figure 3 indicates that increasing the learning rate will only give better results as the cost decreases. We can also see that the one-layer model performs better for higher learning rates, even with fewer epochs, indicating that it escapes local minima and giving strength to the argument that it could be an optimization problem. For both models, learning rate at 2.0 gave the best test scores, which is also our highest value tested. This could indicate that even higher learning rates could yield even better results, and could be telling about the shape of the cost landscape. Further learning rates are not tested as we will optimize further with adaptive learning rate methods.

Further optimizing the model we analyze how switching to a batch update system through stochastic gradient descent will affect the training. We also include Adam, and RMSProp as optimization methods. We keep the model architectures and run them with and without the optimization methods. Table II shows the resulting

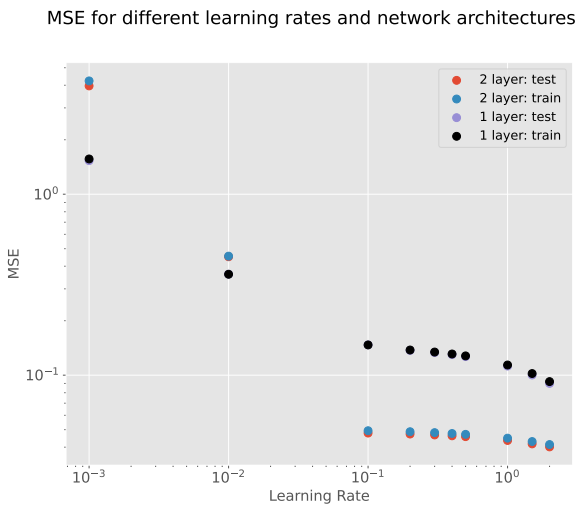

FIG. 3. MSE scores from training and evaluation on the test set for two models, a one-layer model with 50 nodes and learning rate 0.01, and a two-layer model with 50 and 100 respectively using the same learning rate. Hidden layers pass through a sigmoid activation function and output layer is the identity function.

	η	Train	Test
Plain SGD	0.001	0.1439	0.1468
Plain SGD	0.01	0.1079	0.1071
Plain SGD	0.1	0.0397	0.0361
Plain SGD	0.2	0.0367	0.0337
RMSProp	0.001	0.0111	0.0111
RMSProp	0.01	0.0153	0.0155
RMSProp	0.1	0.0269	0.0280
RMSProp	0.2	0.1507	0.1581
Adam	0.001	0.0117	0.0111
Adam	0.01	0.0233	0.0250
Adam	0.1	0.0171	0.0187
Adam	0.2	0.0141	0.0139

TABLE II. MSE for varying one-layer model architectures when changing the learning rate, η. Both train and test scores are shown to observe signs of overfitting. The model has 50 nodes and a sigmoid activation function and output layer is the identity function.

MSE for the one-layer model using different methods and learning rates. We see that different optimizers react differently to the learning rates. Plain stochastic gradient descent does better with increased learning rate. This tells us that increasing the learning rate even more might increase the model efficiency in this problem, without causing instable training. We saw from training models with regular gradient descent that increasing the learning rate further could be an option as it only increased in performance. We will not investigate further as adaptive gradient methods will optimize the model.

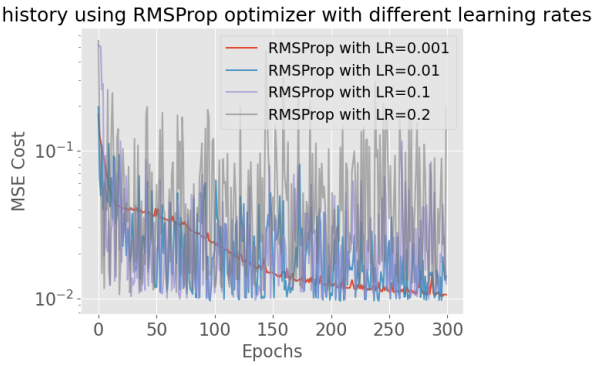

FIG. 4. Cost histories of models using RMSProp with beta = 0.9, with increasing learning rate. Instable cost values due to low velocities.

	η	Train	Test
Plain SGD	0.001	0.0482	0.0458
Plain SGD	0.01	0.0432	0.0407
Plain SGD	0.1	0.0279	0.0265
Plain SGD	0.2	0.0193	0.0187
RMSProp	0.001	0.0113	0.0111
RMSProp	0.01	0.0659	0.0641
RMSProp	0.1	0.3048	0.2960
RMSProp	0.2	13.7189	13.7282
Adam	0.001	0.0105	0.0103
Adam	0.01	0.0115	0.0111
Adam	0.1	0.0972	0.0959
Adam	0.2	0.0252	0.0278

TABLE III. MSE for varying two-layer model architectures when changing the learning rate, η. Both train and test scores are shown to observe signs of overfitting.

RMSProp performs worse when increasing the learning rate. Looking at the cost histories of training in Figure 4, we see that it is highly unstable for all the higher learning rates. When inspecting the velocity matrices, denoted as G_t in the theory, we see that for models with high learning rates, these values become small. The velocities are not dependent on the learning rate, but on the velocity from the last iteration and the gradients from the current iteration, indicating that the gradients themselves are quite small. The learning rate is divided by these velocities, causing big weight updates per iteration. The model with RMSProp and low learning rate have some of the highest score all together, beating the OLS approximation. With Adam, the model performance fluctuate, being the highest performer at best. There are some models that perform worse, telling us that its still important to do an analyzes if optimal scores are to be obtained. The two-layer model performs even better, but with the same issues arising. Training with RMSProp is instable with higher learning rate, something also the Adam optimizer is showing in the two-layer models. For

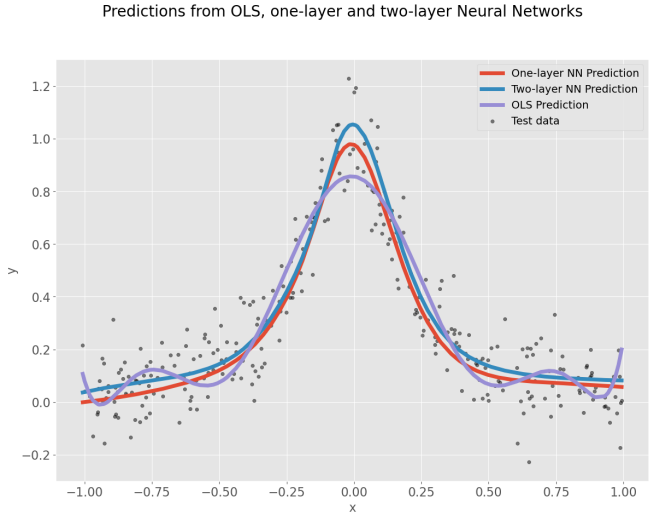

FIG. 5. Approximations made with a one-layer network (50 nodes), a two-layer network (50 and 100 nodes), and OLS, on the one dimensional Runge function. True test data points are scattered in the background.

the higher performance models, the gap between the train and the test sets are larger for the two-layer model, signifying overfitting with the increased complexity of the model.

Looking at the predictions in Figure 5 we can see that the neural regression models are able to mitigate the fluctuations often connected with the approximation of the Runge function. Hidden layers in the neural networks process the input before finally outputting the linear combination, mitigating the fluctuations by not relying on polynomial approximation. This is one of the strengths of neural regression with a weakness being the time it takes to setup.

2. *Comparing the neural network to results from other libraries*

Our initial comparison, detailed in Appendix Table VIII, evaluated the derivatives in our backpropagation against those calculated by the Autograd library ([13]) on a simple one-layer neural network. The results demonstrated a negligible difference of less than 1×10^{-17}. Despite this close match, integrating the Autograd library can be beneficial for more intricate experiments involving different cost functions, where derivatives become more complex.

Next, we compared our implementation of the FFNN against various libraries: Scikit-learn, Tensorflow-Keras, and PyTorch. For consistency and simplicity, we employed stochastic gradient descent with the Adam optimizer across all models, maintaining identical hyperparameters for a fair comparison.

In our simple neural network architecture featuring one hidden layer with 10 nodes, we conducted tests with 300

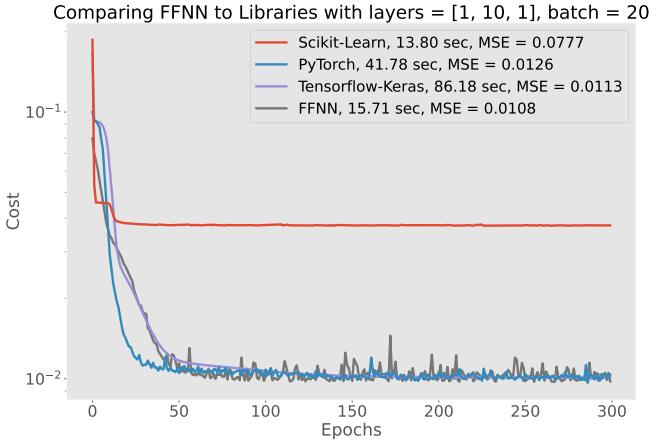

FIG. 6. The cost history for four different implementations (FFNN which only uses NumPy, Scikit-learn, PyTorch and Tensorflow-Keras) with one hidden layer, and ten nodes. All implemented using stochastic gradient decent, with Adam optimizer. Cost function is MSE for all networks, and the sigmoid activation function is used for the hidden layers. No stopping criteria added, running with 300 epochs, and a batch size of 20. The beta-parameters used for Adam is $\beta_1 = 0.9$ and $\beta_2 = 0.99$.

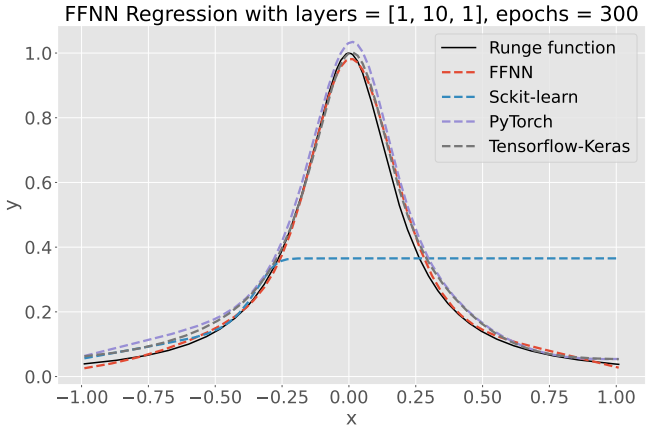

FIG. 7. The predictions made from the four different implementations (FFNN which only uses NumPy, Scikit-learn, PyTorch and Tensorflow-Keras) of a FFNN with one hidden layer, with ten nodes. In black, the real Runge function without noise is shown. All implemented using stochastic gradient decent, with Adam as optimizer. Cost function is MSE for all networks, and the sigmoid activation function is used for the hidden layers. No stopping criteria added, running with 300 epochs, and a batch size of 20. The beta-parameters used for Adam is $\beta_1 = 0.9$ and $\beta_2 = 0.99$.

and 1000 epochs, as we have not implemented stopping criteria. In Figure 6, we see that after 300 epochs, our FFNN, PyTorch and Tensorflow-Keras models achieved a test MSE of approximately 0.01, generating predictions closely aligned with the original Runge function without noise. In contrast, Scikit-learn yielded a test MSE of around 0.08, which is nearly an order of magnitude larger than the other models, suggesting it may have been trapped in a local minimum.

Our analysis indicate that the cost history for the FFNN is notably noisier than that of the other three models. This noise, due to the stochastic nature and randomness of the method, may help mitigate the risk of becoming stuck in a local minima. In contrast, the smoother cost history exhibited by Scikit-learn could explain the reason it seems to get stuck in a local minima. Further examining the predictions made by Scikit-learn, shown in Figure 7, reveals a significant difference in performance compared to the other models.

Time efficiency was highest for Scikit-learn, but it had the lowest performance of the four models. The FFNN emerged as the best-performing model, achieving the lowest test MSE while also being the quickest. Both PyTorch and Tensorflow-Keras yielded comparable test MSE values, but each required over twice the time to complete 300 epochs.

Moving over to the 1000-epoch simple experiment, shown in 8, Scikit-learn displayed unexpected behavior. In this case, it quickly reached the lowest training cost and significantly converged to a low MSE. However, the test MSE did not show this difference between the models, suggesting potential overfitting. Again, FFNN

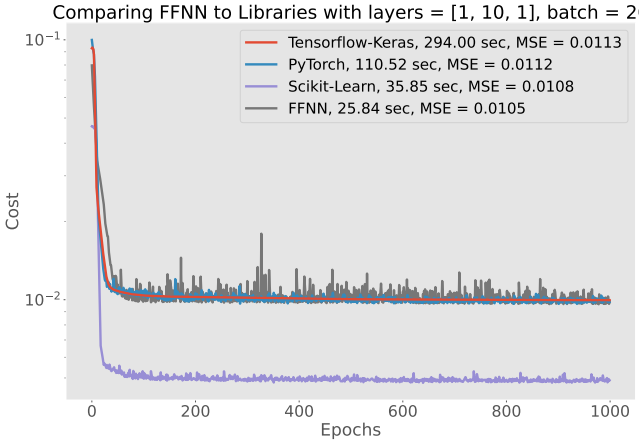

FIG. 8. Corresponding to figure 6, running with 1000 epochs, and a batch size of 20.

demonstrated the lowest test MSE, as well as being the fastest. The rest of the models have similar test MSE, however, again PyTorch and Tensorflow-Keras taking noticeably longer to complete training.

Once more, the cost history for the FFNN proved significantly more nosier compared to the other three models. This time, Scikit-learn exhibited greater noise levels that the last run, while achieving lower training MSE. PyTorch maintained a comparable level of noise, whereas Tensorflow-Keras continued to exhibit a smoother cost history.

In terms of prediction, shown in Figure 9, all four models were relatively consistent, closely aligning with

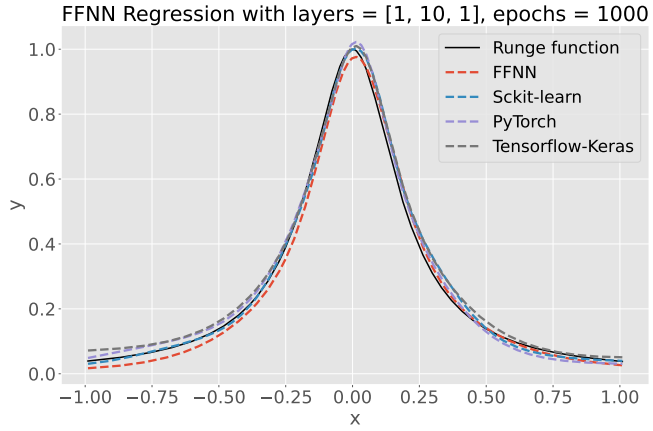

FIG. 9. Corresponding to figure 7, running with 1000 epochs, and a batch size of 20.

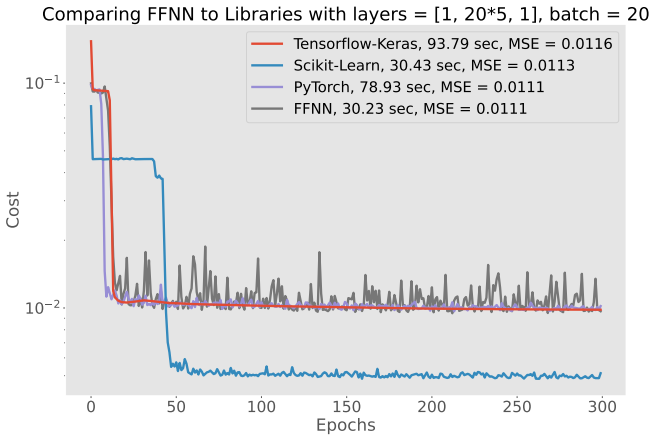

FIG. 10. Corresponding to figure 6, running with 300 epochs, and a batch size of 20.

the true Runge function. While Scikit-learn did exhibit signs of overfitting, it proved effective at replicating the Runge function. Interestingly, despite the FFNN achieving the lowest test MSE, it the performance in recreating the Runge function without noise is not significantly better. It is important to note that the test MSE is found by using the Runge function with noise. In terms of training time, the FFNN was the fastest, with Scikit-learn following closely. PyTorch was approximately four times slower than the FFNN, while Tensorflow-Keras took nearly twelve times as long.

It is also worth noting that for this simple model architecture with just one layer of 10 nodes, we observe no oscillations on the edges of the predictions. This was a problem with the linear regression models from Project 1 [9]. This suggests that even a simple one-layer FFNN can outperform linear regression models in recreating the Runge function without noise.

To evaluate the performance of more complex architectures, we extended our analysis to models featuring 5 hidden layers, with 20 nodes each, again conducting tests

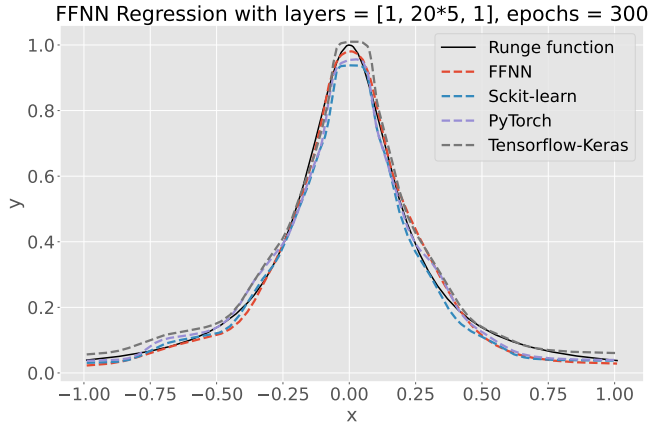

FIG. 11. Corresponding to figure 7, running with 300 epochs, and a batch size of 20.

for 300 epoch and 1000 epochs. In the 300 epoch experiment, shown in Figure 10, we observed similar patterns, with Scikit-learn achieving the lowest training MSE, but this time it seemed that it managed to escape what appeared to be a local minimum. The test MSE is similar for all four models, and lies around 0.01. Notably, the FFNN had the lowest test MSE, while also having the most noisy cost history.

In terms of execution time, the FFNN remained the fastest, though Scikit-learn was only slightly slower. They both approximately doubled the time compared to the simple model experiments. Tensorflow-Keras and PyTorch also increased their time, though Tensorflow-Keras increase was less pronounced, potentially indicating that it may keep pace with the fastest implementations as model complexity rises.

Analyzing the prediction from the complex models, shown in Figure 11, we noted the emergence of oscillations near the edges across all models, reminiscent of the linear regression results from Project 1 [9]. This suggest that while complex models can enhance fitting, they may also risk overfitting.

The 1000 epochs complex experiment cost history, shown in Figure 12, looks similar to the 1000 epochs simple experiment cost history. Scikit-learn again reached the lowest training MSE, while other models stopped before. Again the FFNN cost history was more noisy compared to the other models. This time, however, the FFNN had the highest test MSE, with Scikit-learn displaying the lowest test MSE.

In terms of processing time, Scikit-learn emerged now as the fastest model, with FFNN slightly behind, only seconds slower. Notably, Tensorflow-Keras exhibited a smaller increase in training time relative to the other models, indicating that it may approach or surpass them as model complexity and epoch increase.

The prediction from the complex models, shown in Figure 13, displayed a marked tendency towards overfitting. While not as distinct as the oscillations observed in the

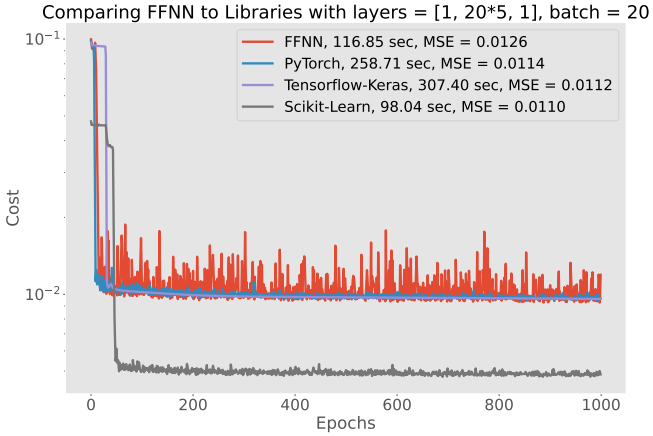

FIG. 12. Corresponding to figure 6, running with 1000 epochs, and a batch size of 20.

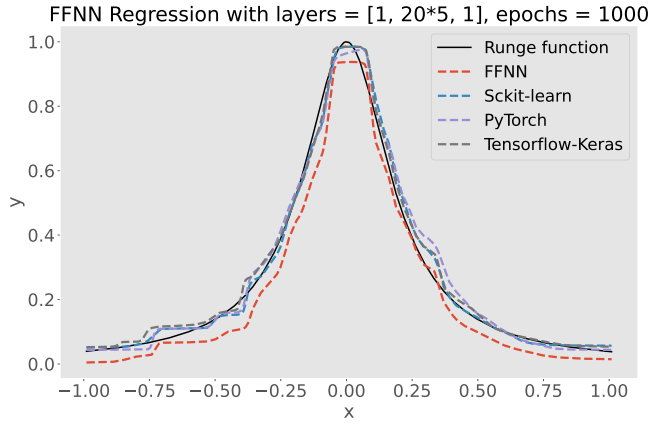

FIG. 13. Corresponding to figure 7, running with 1000 epochs, and a batch size of 20.

previous project, there were still indications of overfitting at high x-values. Interestingly, the FFNN appeared to perform worse in replicating the Runge function without noise, although this discrepancy was not significant.

Despite observing competitive performances from other libraries, we see that there is no clear 'best' model both in terms of time and performance. They all are comparable, and have different advantages. For the experiments used in this project, FFNN shows to be one of the fastest models, while also having among the lowest test MSE. Given these observations, we find it reasonable to continue using our own implementation of the FFNN, as it performs competitively while offering full control over the architecture and training procedure.

Given that all experiments utilized only the sigmoid activation function in the hidden layers, these results primarily illustrate the difference between simple and complex network architectures using this specific activation function. Therefore, it is crucial to further investigate the performance of various activation functions in both simple and complex architectures.

Activation	# Layers	# Nodes	Train MSE	Test MSE
Sigmoid	7	8	0.009542	0.010844
ReLU	6	16	0.009662	0.011041
LReLU	6	128	0.009575	0.010839

TABLE IV. Best neural network architectures for different activation functions, evaluated on train and test MSE. The networks were trained using stochastic gradient descent with Adam optimization, for 300 epochs with a batch size of 20 samples and a learning rate of 0.01 on the 1-D Runge dataset (1500 samples, additive Gaussian noise with mean 0 and standard deviation 0.1, with 80% of the data used for training).

B. Testing different activation functions

Building on the architectures explored in previous analyses, we trained networks with varying numbers of layers and nodes per layer, using three activation functions: Sigmoid, ReLU, and LReLU. Table IV shows the architectures that achieved the lowest MSE on the test set for each activation function. Overall, the training errors are approximately 11% smaller than the testing errors, indicating that while the networks capture the underlying structure of the Runge function effectively, some overfitting is present. The similar gap between training and test errors across activation functions suggests that the optimization scheme and training setup limit excessive overfitting and support reasonable generalization.

Figures 14–16 present heat maps showing how test MSE varies with network architecture. For shallow networks (few hidden layers), increasing the number of nodes per layer generally improves accuracy up to a point. Sigmoid networks perform relatively well in these shallow configurations because the gradient saturation is less pronounced, allowing the network to learn effectively with fewer layers. However, as the networks become deeper with few nodes per layer, Sigmoid networks exhibit large test errors likely due to the vanishing gradients, which hinder learning in deeper architectures. That is, in the case of inputs to the nodes being either very large or very small, the outputs are close to zero or one, resulting in very small gradients and slow learning. In contrast, ReLU networks achieve optimal performance at intermediate numbers of layers and nodes. With too few nodes per layer, the network seem to lack sufficient representational capacity, leading to consistently higher test errors. On the other hand, very wide layers can increase the risk of overfitting, especially in deeper networks, which can also degrade test performance. This is possibly what we see to the far right in Figure 15. LReLU networks, in Figure 16, show similar trends to ReLU. It seems to do slightly better in deeper networks due to the non-zero gradient for negative inputs. Overall, ReLU and LReLU seem to "prefer" a balance between depth and width, while Sigmoid is more suited for shallow, moderately wide architectures where gradient saturation is less limiting. Another visualization of these results can be

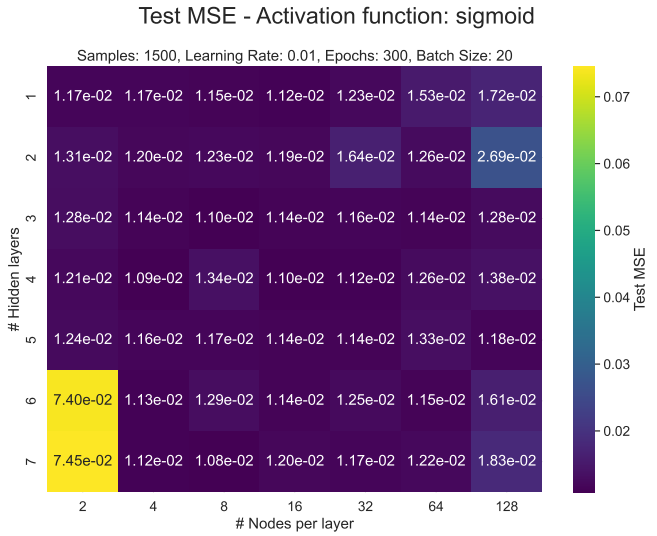

FIG. 14. Heat map showing test MSE for networks with Sigmoid activations. Shallow networks benefit from increasing nodes, while deeper networks with few nodes show higher errors due to vanishing gradients.

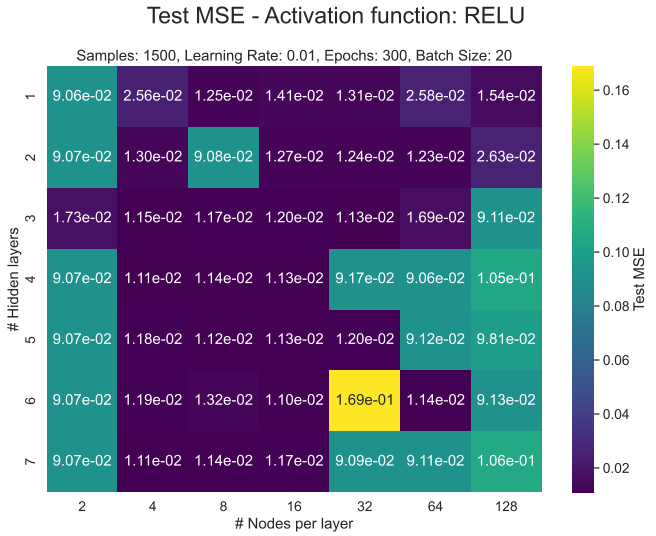

FIG. 15. Heat map showing test MSE for networks with ReLU activations. Performance improves with moderate depth and width, and ReLU avoids vanishing gradient issues.

found in the appendix V, Figure 25.

This behavior reflects the practical limitations of the *universal approximation theorem*, which guarantees that a single hidden layer with enough neurons can approximate any continuous to a given error [1]. However, it does not ensure efficient training or generalization on finite datasets. Deep architectures may introduce optimization challenges such as vanishing gradients, especially for Sigmoid, which can reduce effective learning and increase overfitting risk. Though, looking back at table IV, we see that for the optimal architectures, the three activation functions do not differ significantly in error, and thus all work equally well on predicting the Runge function.

C. Adding L_1 and L_2 norms

Using the optimal architectures identified in Table IV, we trained neural networks for each activation function on the training data using stochastic gradient descent and Adam optimization, keeping all other settings, such as the number of epochs and batch size, unchanged. In this set of experiments, we incorporated L_1 and L_2 regularization and evaluated a range of values for the regularization strength λ and the learning rate η. Regularization introduces a penalty on the weights to reduce overfitting, with L_1 adding the sum of absolute weight values to the loss, promoting sparsity by driving some weights to zero, and L_2 adding the sum of squared weights, discouraging large weights without setting them exactly to zero. The hyperparameter λ controls the strength of this penalization, with higher values producing stronger shrinkage, while the learning rate η determines the size of the updates during training, with small values ensuring stable

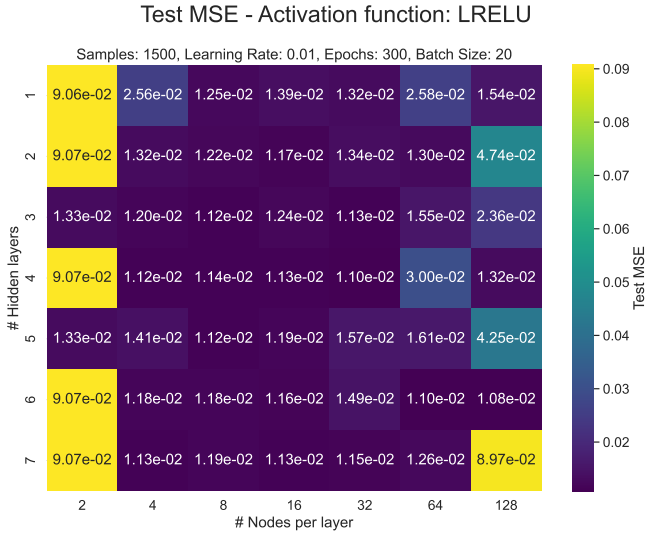

FIG. 16. Heat map showing test MSE for networks with LReLU activations. LReLU maintains gradient flow even for deep networks, resulting in stable performance across a wide range of architectures.

convergence and large values risking exploding gradients.

Figures 17 - 19 display heat maps of test MSE as a function of λ and η for each activation function. For networks using the sigmoid activation (Figure 17), L_1 regularization slightly outperforms L_2, with the lowest test MSE achieved for small λ and small learning rates. The sparsity induced by L_1 likely helps by removing unimportant weights, which can prevent saturation in the sigmoid activations and allow the network to generalize. In contrast, L_2 regularization shows minimal sensitivity to λ, though smaller learning rates are preferred to stabi-

lize training. Overall, the optimal performance occurs with almost negligible regularization, indicating that the network can generalize well even without strong penalization.

For ReLU networks (Figure 18), the results are largely insensitive to the choice of norm, with the lowest L_1 and L_2 MSE differing by only 0.0001. The best performance is observed for low λ and low learning rates, suggesting that regularization provides little benefit in this case. This behavior is consistent with the properties of the ReLU activation, which is less sensitive to large weight magnitudes and, in combination with the relatively low-noise dataset, does not require additional penalization to generalize effectively.

The Leaky ReLU networks (Figure 19) show a different pattern. High learning rates dominate the heat map, causing the MSE to explode, while optimal learning rates are moderate, in the range of 0.01 to 0.1, with small λ values providing the best results. Here, L_1 regularization again slightly outperforms L_2, though the overall effect is minor. These observations indicate that while penalization can influence training stability for Leaky ReLU, the benefits of regularization remain limited, and excessive penalization can harm performance.

Table V summarizes the best combinations of norm type, regularization parameter and learning rates for the best architectures per activation function. Comparing the added regularization to the unregularized networks in table IV shows that adding regularization generally worsened both training and test MSE. This outcome reflects the bias-variance tradeoff: regularization increases bias while reducing variance, but in this dataset, the variance is already low due to minimal noise, and any additional penalization only adds unnecessary bias.

Activation	Norm	λ	Train MSE	Test MSE	Learning Rate
Sigmoid	L1	1×10^{-6}	0.009981	0.011023	0.001
ReLU	L2	1×10^{-6}	0.010305	0.011842	0.01
LReLU	L1	1×10^{-5}	0.010047	0.011228	0.01

TABLE V. Best networks for different activation functions with L_1 and L_2 regularization, including the optimal λ and learning rate for each model. The number of layers and nodes used correspond to the best architectures reported in Table IV. All models were trained using stochastic gradient descent with 300 epochs and a batch size of 20 samples on the 1-D Runge dataset.

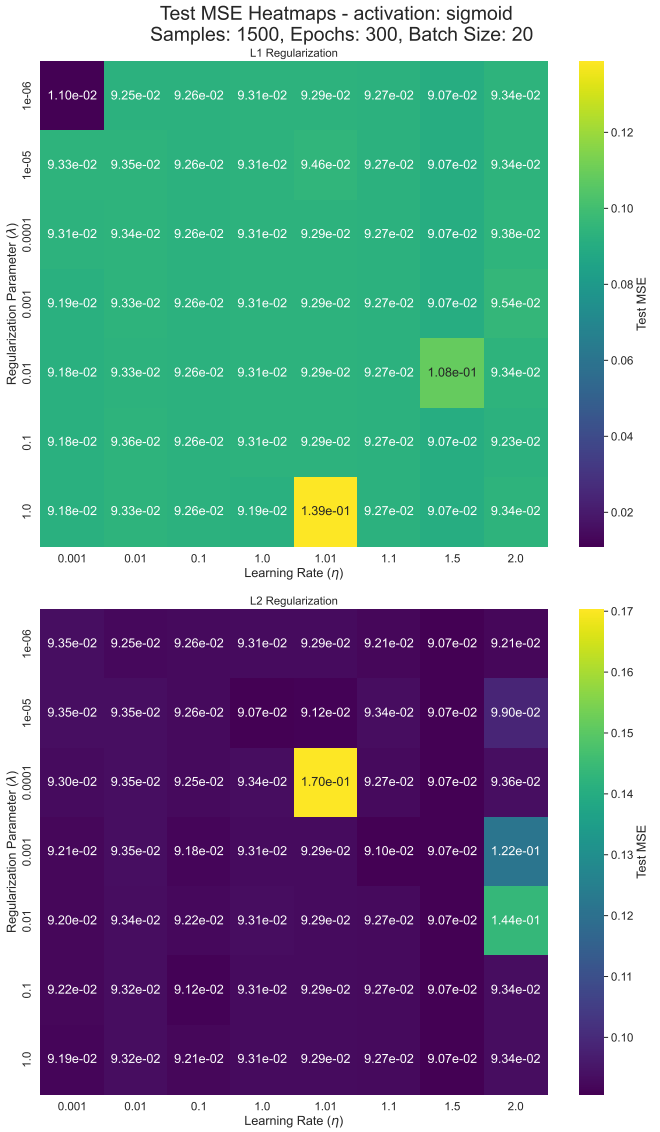

FIG. 17. Upper panel: heat map of test MSE for Sigmoid activation networks with an added L_1 norm. Columns represent different values of the learning rateemployed in stochatic gradient decent with Adam optimizer in the backpropagation of the network. Rows indicate different values of the regularization parameter. Lower panel: Same as above, but for L_2 norm.

1. *Comparison to Ridge and Lasso of Project 1*

In project 1, we assessed Ridge and Lasso regression on the Runge dataset. We have summarized our findings in table VI. The analytical Ridge and Lasso approaches achieved very low test MSEs, on the order of 10^{-5} to 10^{-4} for smaller datasets, while the K-fold–validated models trained on the larger sample (n = 10,000) yielded test errors around 0.0109 and 0.0143, respectively. These values are directly comparable in scale to the neural-network results reported in table V, which lie in the range 0.011–0.012. Interestingly, our NNs have been trained on the same amount of data as all methods in project 1, except K-fold. The close correspondence of the neural and linear models on the K-fold dataset suggests that the additional flexibility of the neural networks does not substantially improve predictive accuracy for this particular problem, where the underlying function is smooth and low-dimensional. It could happen that the machinery we built is too complex for this problem, as the MSEs

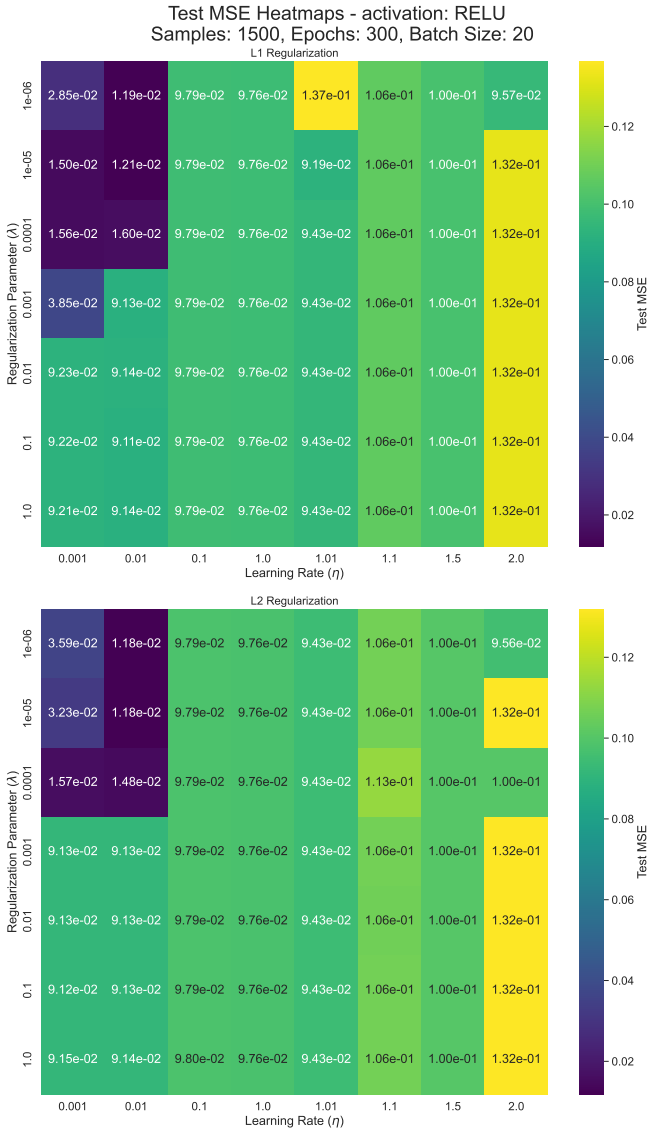

FIG. 18. Similarily to figure 17, but for ReLU activation networks.

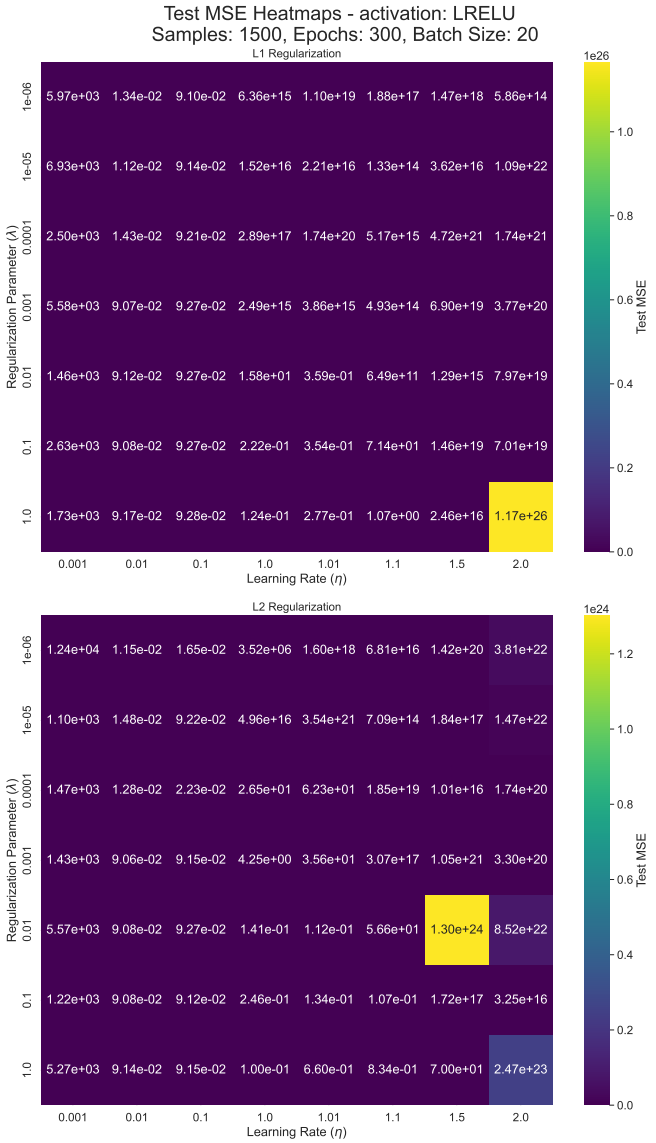

FIG. 19. Analogous to figure 17 and 18 but for LReLU activation networks.

for linear regression methods are magnitudes of orders smaller.

Conceptually, both the regression and neural-network models implement regularization in the same spirit: Ridge and Lasso explicitly penalize the magnitude of the coefficients to prevent overfitting, whereas the L_2 and L_1 norms used in the networks impose analogous constraints on the connection weights. However, the effects observed differ in magnitude. The network architectures possess implicit regularization through stochastic optimization and early stopping, which limit overfitting even without explicit penalties. In addition, the nonlinear models operate in a higher-dimensional parameter space, so applying the same level of shrinkage (λ) can excessively constrain learning, increasing bias. In both frameworks, the L_1 norm yielded slightly lower test errors, indicating that sparsity may help isolate the most relevant model components even in a nonlinear setting. But, for simple, well-behaved data, linear models with analytical or gradient-based Ridge/Lasso remain highly competitive to the NNs.

D. Multi-class classification

Training on the new dataset, we observe that shallow and wide networks perform the best, for all three activation functions as shown in Figures 20-22.

This is partially consistent with how the network responds to our linear problem, except the accuracy landscape for the classification problem shows a more gradual variation. Moreover, we found that models with learn-

Method	Hyperparameters	Test MSE
Analytical Ridge	$n = 1500$, $p = 8$, $\lambda = 10^{-3}$	5.0e-5
Analytical Lasso	$n = 1500$, $p = 8$, $\lambda = 10^{-3}$	7.5e-5
GD Ridge (Adam)	$n = 1500$, $p = 14$, $\lambda = 0.001$	9.0e-5
GD Lasso (Adam)	$n = 1500$, $p = 14$, $\lambda = 0.001$	7.5e-5
SGD Ridge	300 epochs, batch 20	9.0e-5–2.0e-4
SGD Lasso	300 epochs, batch 20	8.0e-5–2.0e-4
K-fold Ridge	$n = 10000$, $p = 12$, $\lambda = 10^{-4}$	0.0109
K-fold Lasso	$n = 10000$, $p = 8$, $\lambda = 10^{-4}$	0.0143

TABLE VI. Summary of Project 1 results for Ridge and Lasso regression on the 1D Runge dataset. Test MSEs are reported for different approaches.

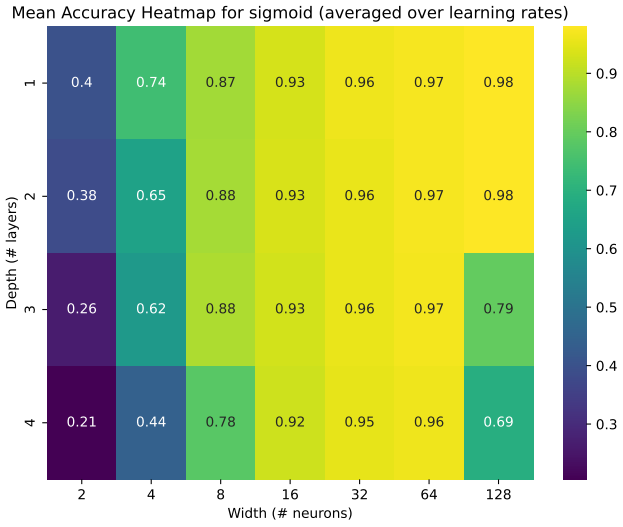

FIG. 20. Analysis of models with different architectures, with sigmoid activation functions at all hidden layers and different learning rates. The z-values are averages over all learning rates.

ing rate 0.01 had the overall best performance, highest individual accuracy and lowest spread, indicating that the optimal value, within the tested range, lies close to 0.01. The highest individual accuracy score measured for the training set is 0.999875, on a model with one hidden layer, 128 nodes, a sigmoid activation function, and a learning rate of 0.01. These parameters were therefore selected for our analyses on regularization. Evaluation on the test set with this parameter setup gives us a score of 0.96486, suggesting a small overfitting and room to test if penalization can mitigate it.

After training multiple penalized models, both the L1 and L2 norms induce some penalization especially with higher lambda values. This leads to a lower training accuracy, while increasing the accuracy on the test set, indicating that the models generalize better. We found that the best performing model utilizes L2 regularization with a lambda value at 0.01, as pictured in Figure 23.

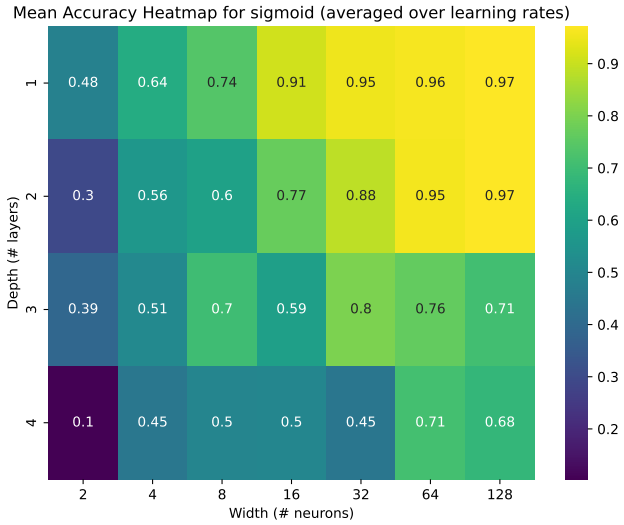

FIG. 21. Analysis of models with different architectures, with ReLU activation functions at all hidden layers and different learning rates. The z-values are averages over all learning rates.

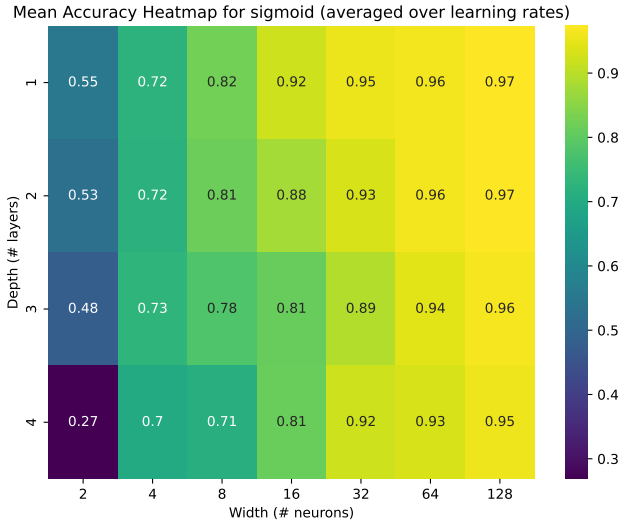

FIG. 22. Analysis of models with different architectures, with Leaky ReLU activation functions at all hidden layers and different learning rates. The z-values are averages over all learning rates.

Through a more thorough analysis we found the optimal lambda value to be 3e-2, which will be used for the final model.

Finally we look at the confusion matrix of the model prediction in Figure 24. The test sample sizes of classes are heterogeneous, having the most samples of handwritten versions of the number 1. The number 1 class also has

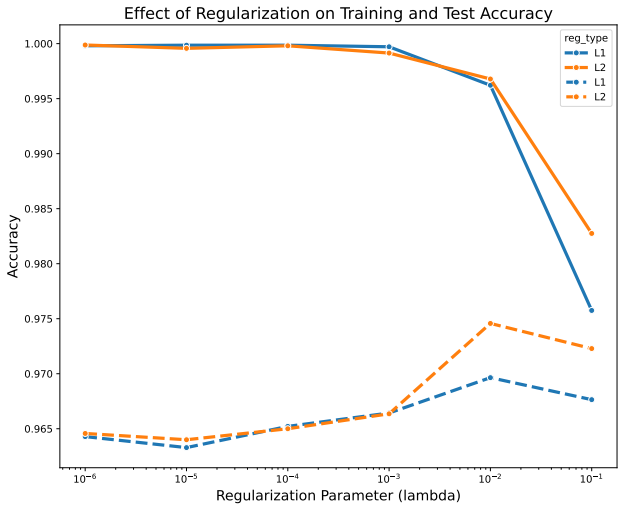

FIG. 23. Analysis of the type and strength of regularization using optimal model parameters from earlier analysis. One layer, 128 nodes, sigmoid activation function for hidden layer, and softmax for output layer. Trained for 30 epochs each with batch size 1000, learning rate 0.01, and Adam optimization. Dashed line represents the two-layer model.

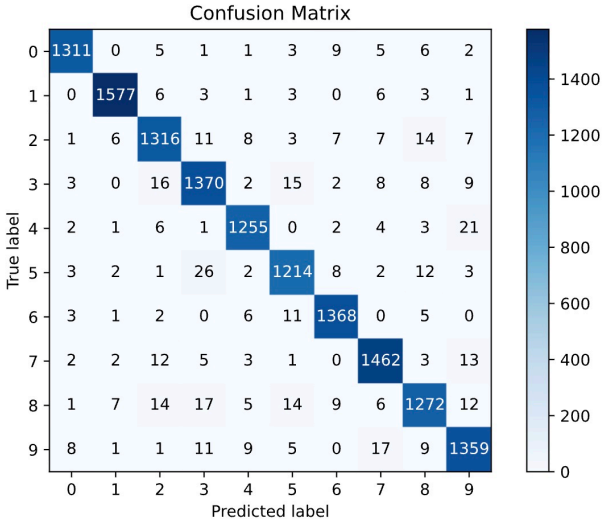

FIG. 24. Optimized model performance visualized with a confusion matrix. One layer, 128 nodes, sigmoid activation function for hidden layer, and softmax for output layer. Trained for 30 epochs each with batch size 1000, learning rate 0.01, and Adam optimization. L2 norms are applied to cost function with lambda equaling 3e-2.

the lowest amount of false predictions, giving it the highest TPR and lowest FPR. We see that model is weaker on some of the classes. Class 3 has the highest TPR. The model often predicts this class, when the true label is 2, 5, and 9, as seen in Figure 24. The same is true for fours predicted as nines. These numbers are similarly shaped, suggesting the errors could be due to ambiguities in the dataset, rather than a systematic bias in the model. Otherwise the model generalizes well for all classes as the true positive rates are generally high. The true positive and false positive rates are tabulated in table VII.

	TPR	FPR
Class 0	0.9762	0.00182
Class 1	0.9856	0.00161
Class 2	0.9536	0.005
Class 3	0.956	0.006
Class 4	0.97	0.003
Class 5	0.9536	0.00432
Class 6	0.98	0.003
Class 7	0.9727	0.0044
Class 8	0.9374	0.005
Class 9	0.957	0.0054

TABLE VII. Resulting true positive rate (TPR) and false positive rate (FPR) for each class. Classification model has one layer, 128 nodes, sigmoid activation function for hidden layer, and softmax for output layer. Trained for 30 epochs each with batch size 1000, learning rate 0.01, and Adam optimization. L2 norms are applied to cost function with $\lambda = 3 \cdot 10^{-2}$.

V. CONCLUSION

The comparison between the neural network and traditional linear regression methods (OLS, Ridge, and Lasso) shows that the neural network can achieve comparable or better predictive accuracy on the Runge function, particularly when using multiple layers and optimized learning rates. While OLS performs well for simple, low-dimensional problems, the neural network demonstrates its strength in capturing nonlinear behavior and mitigating the edge oscillations observed in polynomial regression models from Project 1. This improvement comes at the cost of increased computational time and sensitivity to hyperparameters such as the learning rate, number of layers, and optimizer. The results indicate that the model performance is highly dependent on these parameters, especially the learning rate, which impacts whether the model converges efficiently or becomes trapped in local minima. The two-layer network with optimized hyperparameters and methods produced the lowest mean squared error, outperforming OLS.

We conducted a comparative analysis of our own FFNN implementation against those from established libraries. Our findings illustrated a similar behavior in all models , which validates the implementation of our own FFNN. The comparison also highlighted that in cases of simple model architectures and small training datasets, the pre-built libraries might not show it's full potential, and ended up being more time-consuming. However, one also need to take into account the time spent on developing our own model, compared to set up from pre-built

libraries. In more complex cases, with more testing of different cost functions, it might be more beneficial to use libraries instead, especially Autograd for automatic differentiation. The comparison between a simple and a complex model architecture indicated that even a basic one-layer FFNN can outperform linear regression models in recreating the true Runge function. It also showed that while complex models can enhance fitting, they may also risk overfitting.

As for our analysis of different activation functions and network architecture, our results align somewhat with our hypotheses. ReLU and Leaky ReLU networks learned quickly and stably, while Sigmoid struggled in deeper architectures due to vanishing gradients, just as expected. However, adding L_1 or L_2 regularization didn't help performance, and sometimes slightly worsened it, reflecting that the Runge function is smooth and low-variance, so extra penalization just adds bias. With 300 epochs, batch size 20, and moderate learning rates (0.001–0.01), training was stable, and overall, linear Ridge and Lasso models remain very competitive, showing that neural networks offer little advantage for simple, well-behaved functions.

Applying the neural network to the MNIST dataset demonstrates its ability to generalize from regression tasks to complex, high-dimensional classification problems. With a single hidden layer of 128 neurons and a sigmoid activation combined with a softmax output layer, the network achieved stable convergence within 30 epochs and produced a well-balanced confusion matrix with minor deviations. The inclusion of L1 regularization with $\lambda = 3 \cdot 10^{-2}$ further improved generalization by reducing overfitting, resulting in consistent performance between training and test data.

Overall, linear models are the best choice for simple, smooth regression problems, while neural networks are more powerful for complex and nonlinear tasks where flexibility and pattern recognition are essential.

[1] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[2] Juergen Schmidhuber. Annotated History of Modern AI and Deep Learning, 2022. Version Number: 2.

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

[4] Morten Hjorth-Jensen. *Computational Physics Lecture Notes 2015*. Department of Physics, University of Oslo, Norway, 2015.

[5] Wikipedia contributors. Runge's phenomenon, August 2025. Page Version ID: 1303928185.

[6] Kjersti Stangeland, Jenny Guldvog, Ingvild Olden Bjerkelund, and Sverre Manu Johansen. Regression analysis and resampling methods, FYS-STK4155 - Project 1. Technical report, University of Oslo, Norway, June 2025.

[7] Hojjat K. MNIST Dataset.

[8] Sebastian Raschka. *Machine Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python*. Packt Publishing Limited, Birmingham, 1 edition, 2022.

[9] Kjersti Stangeland, Jenny Guldvog, Ingvild Olden Bjerkelund, and Sverre Johansen. kjes4pres/Project_1_fysstk. original-date: 2025-09-02T10:34:05Z.

[10] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. *Neural Networks for Machine Learning - Lecture 6a Overview of mini-batch gradient descent*. University of Toronto.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2014. Version Number: 9.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[13] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, page 5, 2015.

[14] François Chollet and others. Keras, 2015.

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019. Version Number: 1.

[16] Charles R. Harris, K. Jarrod Millman, Stéfan J. Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. Van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández Del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[17] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015.

[18] Wes McKinney. Data Structures for Statistical Computing in Python. pages 56–61, Austin, Texas, 2010.

[19] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[20] Michael L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021. Publisher: The Open Journal.
[21] OpenAI. ChatGPT, 2025.
[22] University of Oslo. GPT UiO, 2025.

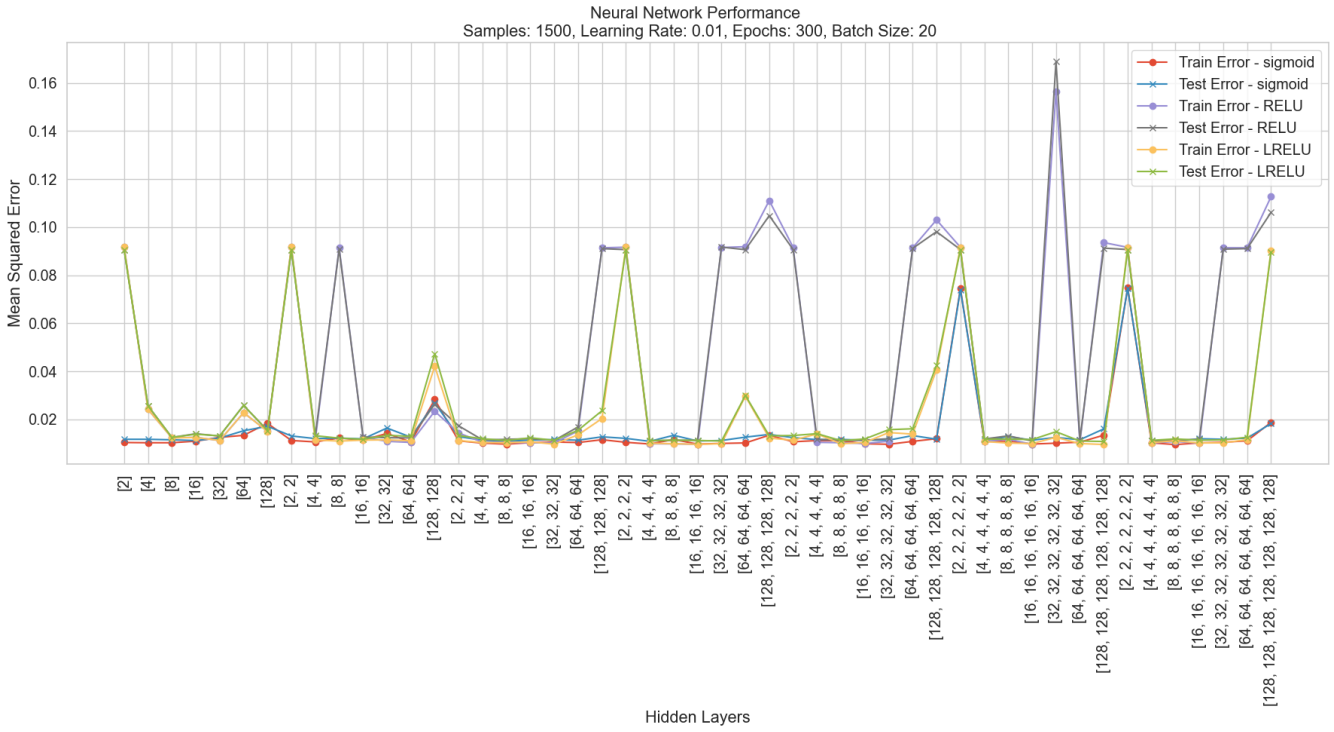

FIG. 25. Test MSE scores for different neural networks trained with stochastic gradient descent using Adam optimization for the 1-D Runge function. The networks were trained for 300 epochs with a batch size of 20 samples and a learning rate of 0.01 on the 1-D Runge dataset (1500 samples, additive Gaussian noise with mean 0 and standard deviation 0.1, with 80% of the data used for training). The different lines show how the different activation functions scores for different architectures, in terms of number of layers and number of nodes.

APPENDIX A

Layer	Weight Gradient (dC/dW) Difference 1	Weight Gradient (dC/dW) Difference 2	Bias Gradient (dC/db) Difference
1	-4.34e-19	-8.67e-19	-8.67e-19
2	-1.04e-17	-1.39e-17	-1.39e-17
3	2.77e-17	4.85e-17	4.85e-17

TABLE VIII. Differences in Gradient Values Between Manual Backpropagation and Autograd Calculations (Manual minus Autograd).