Neural Networks

Stan Daniels, Francesco Minisini, Teresa Ghirlandi, and Carolina Ceccacci
University of Oslo
Data Analysis and Machine Learning (FYS-STK3155/FYS4155)
(Dated: November 10, 2025)

Feed-Forward Neural Networks (FFNNs) are powerful tools for modeling complex, non-linear relationships, but their performance depends critically on architectural choices and optimization hyper-parameters. To explore these dependencies, this project develops a custom FFNN implementation and applies it to both regression and classification tasks. The study further examines the influence of hyper-parameters, including network size, regularization, activation functions, optimization methods, and learning rate, on performance, highlighting how these choices affect predictive accuracy and generalization. For regression, the network approximates the Runge function and its performance is compared with the OLS, Ridge, and Lasso models studied in our previous work [1]. The analysis highlights that the FFNN outperformed the traditional methods, achieving a lower error on specific non-linear tasks. Regarding classification, the FFNN is applied to the MNIST dataset: the results show that deep and wide architectures with Leaky ReLU activation reach higher accuracy. Validation is performed through benchmarking against PyTorch and through analytical-vs-Autograd gradient consistency checks, ensuring the correctness of the approach while revealing the inherent constraints of the custom FFNN.

I. INTRODUCTION

Artificial Neural Networks (ANNs) are among the most powerful tools in modern machine learning, applied to protein structure prediction [2], image classification [3], medical imaging [4], and more. Inspired by the neural architecture of the brain [5], ANNs can model highly non-linear relationships. Feed-Forward Neural Networks (FFNNs) are a fundamental deep learning model [6], capable of handling high-dimensional data and forming the basis for architectures such as convolutional networks.

The primary aim of this project is to design, implement, and evaluate a Feed-Forward Neural Network. FFNNs are powerful function approximators [7] and effective for classification tasks [8]. This study investigates these capabilities by applying the network to both regression and classification problems, analyzing the effects of hyper-parameters including network size, regularization, activation functions, optimization methods, and learning rate.

The FFNN initially approximates the Runge function, with performance measured using Mean Squared Error (MSE) and compared to OLS, Ridge, and Lasso regression from our previous work [1]. Subsequently, the study focuses on multi-class classification using the MNIST dataset, with performance evaluated in terms of accuracy.
Finally, the implemented FFNN has been benchmarked against established frameworks such as PyTorch, with the hardcoded analytical gradient computations validated through comparison with Autograd.

The report is structured as follows.
Section II "Theory", introduces the theoretical background and implementations of FFNNs, including their architecture, the datasets used, the performance metrics and the activation functions. Following this, Section III "Results and Discussion" presents and analyzes the numerical results obtained for both the regression and classification tasks. Finally, Section IV "Conclusion", summarizes the main results and insights of the study.

II. THEORY

In the *Feed-Forward Neural Network* (FFNN), information propagates strictly in one direction, from input to output, without recurrent feedback connections; an example is shown in Figure 1.

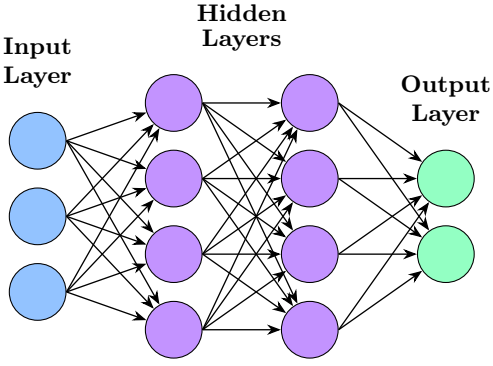

FIG. 1: Feedforward neural network with two hidden layers. Arrows indicate the weighted connections between neurons.

A. Data Generation

To examine the performance of an FFNN on regression tasks, data sets have been generated by the *Runge function*[9]:

$$f(x) = \frac{1}{1+25x^2}, \quad x \in [-1,1] \tag{1}$$

which is a well-known test function in numerical analysis. The Runge function, when fitted using linear methods, exhibits oscillatory behavior, especially near the boundaries of the interval, making it suitable for evaluating regression methods and overfitting tendencies [1]. To simulate more realistic conditions, Gaussian noise can optionally be added to the output:

$$y = f(x) + \mathcal{N}(0, \sigma^2), \tag{2}$$

where σ is the noise standard deviation.

B. Scaling of the data

All input and target data were standardized using `scikit-learn`'s `StandardScaler`, ensuring zero mean and unit variance. This preprocessing step was essential to improve the convergence and numerical stability of the FFNN training process.

C. Use of AI tools

The development process made use of AI tools such as ChatGPT and Grok, to optimize efficiency and ensure high-quality outcomes. Their functionality proved useful for generating code fragments, resolving implementation issues, and improving the clarity and structure of the report. Every AI contribution was carefully reviewed and refined prior to inclusion in the final work.

D. Feed-Forward Neural Network (FFNN)

A Feed-Forward Neural Network (FFNN) consists of multiple layers: an *input layer* $\mathbf{x}$ that receives the input features, one or more *hidden layers* where each neuron computes a weighted sum of its inputs plus a bias term, followed by a non-linear activation function $\mathbf{f}$, and an *output layer* $\tilde{\mathbf{y}}$ that produces the network's final predictions. The activation function regulates the flow of information through the network, introducing the non-linearity necessary for approximating complex, non-linear mappings between inputs and outputs. The bias term allows the activation function to be shifted, improving the flexibility of the model and enabling it to better fit the data.
The training of a Feed-Forward Neural Network consists of two main stages that are repeated iteratively during the learning process:

1. Feed-forward stage: the input data are propagated through the network to produce a final output, which is then compared with the target values using a chosen cost function.

2. Backpropagation stage: the network's parameters $\boldsymbol{\theta}$ are updated by minimizing the cost function through gradient-based optimization. The expressions for the gradients are obtained efficiently via the chain rule. [5]

1. Feed Forwarding

The propagation of input data through the network to generate an output is known as *feed forwarding*. In this process, information flows strictly in one direction, from the input layer, through the hidden layers, to the output layer, without any feedback connections. For a single neuron, the output can be written as:

$$\tilde{y} = f\left(\sum_i \omega_i x_i + b\right) \tag{3}$$

where ω_i are the weights, x_i are the input features, and b is the bias.

Extending this formulation to multiple layers, the feed-forward computation can be expressed in matrix form as:

$$\mathbf{a}^{(l)} = f\left(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right), \quad l = 1, \ldots, L \tag{4}$$

where $\mathbf{a}^{(l)}$ denotes the activation vector of the l-th layer, $\mathbf{W}^{(l)}$ is the weight matrix, and $\mathbf{b}^{(l)}$ is the bias vector.

Given an input $\mathbf{x}$, the feed-forward computation proceeds as follows:

$$\mathbf{a}^{(0)} = \mathbf{x}, \tag{5}$$

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \tag{6}$$

$$\mathbf{a}^{(l)} = f^{(l)}(\mathbf{z}^{(l)}), \quad l = 1, \ldots, L. \tag{7}$$

Here, $\mathbf{z}^{(l)}$ represents the weighted inputs to layer l, $\mathbf{a}^{(l)}$ the corresponding activations, and $f^{(l)}$ the activation function applied at that layer. The final network output is given by $\hat{\mathbf{y}} = \mathbf{a}^{(L)}$. The sequence of operations described above is implemented as outlined in Algorithm 1.

Algorithm 1 Feed-Forward Pass of the FFNN

Require: Input $\mathbf{X}$, weights $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}$, hidden activation f_h, output activation f_o
Ensure: Output $\mathbf{A}^{(L)}$
Initialize activations $\mathbf{A}^{(0)} \leftarrow \mathbf{X}$ (add bias if needed)
for $l = 1$ to L **do**
  $\mathbf{Z}^{(l)} \leftarrow \mathbf{A}^{(l-1)}\mathbf{W}^{(l)}$
  if $l < L$ **then**
    $\mathbf{A}^{(l)} \leftarrow f_h(\mathbf{Z}^{(l)})$
  else
    $\mathbf{A}^{(L)} \leftarrow f_o(\mathbf{Z}^{(L)})$
  end if
end for
return $\mathbf{A}^{(L)}$

2. Backpropagation

Backpropagation is the key algorithm that enables the training of FFNN [10]. It minimizes the difference between the predicted and actual outputs by propagating the errors backwards through the network. Using the chain rule of calculus, it computes the gradients of the cost function with respect to all weights and biases, which are then used to iteratively update the parameters. The algorithm proceeds in a *layer-wise reverse order*, starting from the output layer and propagating the error backward through the hidden layers. Combined with gradient-based optimization methods such as stochastic gradient descent or adaptive schedulers, backpropagation allows the model to reduce the loss across epochs and effectively learn complex, non-linear patterns from data.

Mathematically, backpropagation computes the gradient of the cost function C with respect to each weight and bias by recursively applying the chain rule. Denoting by $\delta^{(l)}$ the local error at layer l, we have:

$$\delta^{(L)} = \frac{\partial C}{\partial \mathbf{z}^{(L)}} = f^{(L)'}(\mathbf{z}^{(L)}) \odot \frac{\partial C}{\partial \mathbf{a}^{(L)}}, \tag{8}$$

$$\delta^{(l)} = \frac{\partial C}{\partial \mathbf{z}^{(l)}} = \left(\mathbf{W}^{(l+1)T}\delta^{(l+1)}\right) \odot f^{(l)'}(\mathbf{z}^{(l)}), \quad l = L-1, \ldots, 1 \tag{9}$$

where $\odot$ denotes element-wise multiplication.

The gradients of the weights and biases are then given by:

$$\frac{\partial C}{\partial \mathbf{W}^{(l)}} = \delta^{(l)}(\mathbf{a}^{(l-1)})^T + \lambda \mathbf{W}^{(l)}, \tag{10}$$

$$\frac{\partial C}{\partial \mathbf{b}^{(l)}} = \sum_i \delta_i^{(l)}, \tag{11}$$

where λ is the regularization coefficient. These gradients are subsequently used to update the parameters according to a chosen optimization algorithm, such as stochastic gradient descent, RMSProp, or Adam. For a detailed explanation of these optimization methods, the reader is referred to our previous work [1]:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial C}{\partial \mathbf{W}^{(l)}}, \tag{12}$$

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta \frac{\partial C}{\partial \mathbf{b}^{(l)}}. \tag{13}$$

The implementation of this procedure is summarized in Algorithm 2.

Algorithm 2 Backpropagation for FFNN

Require: Input batch $\mathbf{X}$, targets $\mathbf{t}$, weights $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}$, activations $\mathbf{a}^{(l)}$, pre-activations $\mathbf{z}^{(l)}$, hidden activation f_h, output activation f_o, regularization λ
Ensure: Updated weights $\mathbf{W}^{(l)}$
Compute derivatives for output and hidden activations
for $l = L$ down to 1 **do**
  if $l = L$ **then** ▷ Output layer
    if multi-class classification **then**
      $\delta^{(L)} \leftarrow \mathbf{a}^{(L)} - \mathbf{t}$
    else
      $\delta^{(L)} \leftarrow f_o'(\mathbf{z}^{(L)}) \odot \frac{\partial \mathcal{C}}{\partial \mathbf{a}^{(L)}}$
    end if
  else ▷ Hidden layers
    $\delta^{(l)} \leftarrow (\mathbf{W}_{1:}^{(l+1)}\delta^{(l+1)}) \odot f_h'(\mathbf{z}^{(l)})$
  end if
  $\nabla \mathbf{W}^{(l)} \leftarrow (\mathbf{a}_{1:}^{(l-1)})^T\delta^{(l)} + \lambda \mathbf{W}_{1:}^{(l)}$
  $\nabla \mathbf{b}^{(l)} \leftarrow \sum \delta^{(l)}$
  Update weights and biases using learning scheduler
end for

E. Activation Functions

The activation function is a mathematical function applied to the output of a neuron. It introduces non-linearity, enabling the model to learn and represent complex data patterns. Without it, even a deep neural network would behave like a simple linear regression model. Each neuron applies an activation function to its weighted input, which determines the signal passed to the next layer. The network supports several commonly used activation functions. First, the *Sigmoid* function is defined as:

$$f(x) = \frac{1}{1 + e^{-x}}$$

It maps input values into the interval $(0, 1)$, making it suitable for probabilistic outputs and binary classification. Its derivative is $f'(x) = f(x)(1 - f(x))$, but it can suffer from vanishing gradients for very large or small inputs.
A solution to this problem is to replace the Sigmoid with the *ReLU (Rectified Linear Unit)* function:

$$f(x) = \max(0, x)$$

It outputs zero for negative inputs and the identity for positive inputs. This simple form makes ReLU computationally efficient, preserves non-linearity without complex transformations, and helps mitigate the vanishing

gradient problem. Its derivative is given by

$$f'(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Another variant of the ReLU function is the *Leaky ReLU*:

$$f(x) = \max(\alpha x, x)$$

which allows a small, non-zero gradient α for negative inputs. This addresses the "dying ReLU" problem which occurs when standard ReLU neurons output zero for all inputs. In such cases, the gradient becomes zero during backpropagation, preventing weight updates and making the neuron inactive. Leaky ReLU ensures that neurons continue to learn even when receiving negative signals. Furthermore, there is a *Linear* function:

$$f(x) = x$$

Applies no non-linearity and is typically used in output layers for regression tasks. Its derivative is constant, $f'(x) = 1$. These activation functions are shown in Figure 2.

Lastly, the *softmax* activation function is generally used in multi-classification problem:

$$f(z)_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \quad k = 1, \ldots, K$$

where K is the total number of classes, z_k is the raw score produced by the neural network for class k, f_k is the predicted probability of the input belonging to class k. It converts the raw output scores of a neural network into probabilities, making the predictions easier to interpret. The resulting probabilities are normalized so that their sum equals one, $0 \leq f_k \leq 1$ and $\sum_{k=1}^{K} \hat{y}_k = 1$, ensuring that all classes are considered all together. The softmax activation function is shown in Figure 3.

During backpropagation, the derivatives of the activation functions are required to compute gradients for updating weights and biases. In the case of the *softmax* function, the derivative is more complex. However, when combined with the *Categorical Cross-Entropy* loss 15, the gradient of the loss with respect to the pre-activation values (the inputs to the softmax) simplifies to the difference between the predicted probabilities and the true labels:

$$\frac{\partial C_{\text{CCE}}}{\partial z} = \hat{y} - y, \tag{14}$$

where z represents the input vector to the softmax function. This simplification is one of the reasons why the softmax activation combined with CCE is a standard choice for multi-class classification tasks.

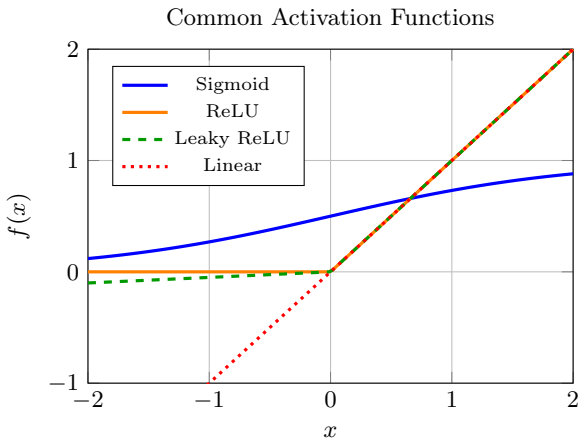

FIG. 2: Comparison of common activation functions: Sigmoid (blue) saturates for large $|x|$, ReLU (red) is zero for $x < 0$, Leaky ReLU (green, dashed) allows a small negative slope, and Linear (orange, dotted) passes input directly.

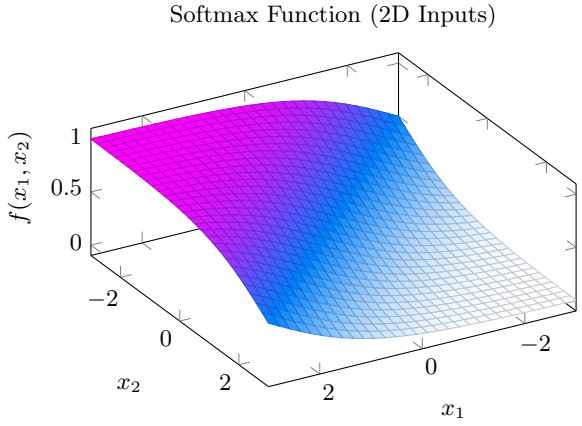

FIG. 3: Visualization of the Softmax function for a 2-dimensional input vector $[x_1, x_2]$. This shows the multi-dimensional nature of Softmax.

F. Performance metrics

The performance of a FFNN is quantified using a *cost function*, which measures the discrepancy between the predicted outputs $\hat{\mathbf{y}}$ and the true targets $\mathbf{y}$. The choice of cost function depends on the type of learning task. For regression problems, the *Mean Squared Error (MSE)* is commonly used:

$$C_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

where N is the number of samples, $\mathbf{w}$ are the network weights, λ_1 and λ_2 are the optional regularization coefficients for L_1 and L_2 penalties, respectively. These regularization terms help prevent overfitting by encouraging

sparsity and reducing the magnitude of weights.
For classification tasks, *Binary Cross-Entropy (BCE)* is used in binary settings:

$$C_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\Big],$$

while *Categorical Cross-Entropy (CCE)* is adopted for multi-class problems:

$$C_{\text{CCE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik}\log(\hat{y}_{ik}), \tag{15}$$

where K is the number of classes and $y_{ik} = 1$ if sample i belongs to class k, and 0 otherwise.

Regularization terms such as L_1 and L_2 penalties can optionally be added to any cost function:

$$C_{\text{reg}} = C + \lambda_1\|\mathbf{w}\|_1 + \lambda_2\|\mathbf{w}\|_2^2,$$

where λ_1 and λ_2 are the regularization coefficients that help prevent overfitting by promoting sparsity and constraining the magnitude of the network weights.

To evaluate the performance of the multi-class classification task, the *accuracy score* is adopted, defined as:

$$\text{Accuracy} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}(\hat{y}_i = y_i) \tag{16}$$

where N is the total number of targets, y_i represents the true label of the i-th sample, and $\hat{y}_i$ is the corresponding predicted label produced by the feed-forward neural network (FFNN).

Here, $\mathbb{I}(\cdot)$ denotes the *indicator function*, defined as

$$\mathbb{I}(\hat{y}_i = y_i) = \begin{cases} 1, & \text{if } \hat{y}_i = y_i \\ 0, & \text{otherwise.} \end{cases} \tag{17}$$

In other words, the accuracy represents the fraction of correctly classified samples over the total number of samples.

III. RESULTS AND DISCUSSION

In this section, the results of the application of the FFNN to both regression and classification problems are presented. The effects of network architecture, activation functions, regularization, and optimization strategies have been systematically investigated. The objective is to highlight the advantages and limitations of FFNNs in comparison with traditional methods and established machine learning libraries.

A. Regression: Runge Function

The analysis begins with a regression task in which the FFNN is applied to data generated from the Runge function. The network architecture is designed to be flexible, allowing for different numbers of hidden layers, amount neurons and activation function used. The output layer of regression tasks uses the linear acivation function. Biases were initialized to a small constant values close to zero, and weights were initialized with small random Gaussian values to break symmetry. The network was trained using the Adam optimizer with a learning rate of 0.001 to ensure efficient convergence. During all experiments, the validation dataset was maintained as the clean, noise-free values of the Runge function, even when the training data included noisy samples. This approach ensures that the performance evaluation reflects the network's ability to generalize to the true underlying function, rather than merely fitting the noisy training data. The use of a clean, noise-free validation set is only feasible for a subset of regression problems where the true underlying function is known analytically, such as the Runge function. This approach cannot be generally applied to real-world datasets, where the true function is unknown and noise may be inherent in both training and validation data.

1. *Effect of Network complexity and Activation function*

The influence of different activation functions, ReLU, Leaky ReLU and Sigmoid, on the performance of the network was analyzed as a function of number of hidden layer and neurons per layer. The hidden layers acts as an intermediary between the input and the output data, capturing the underlying patterns. The network was configured with one to five hidden layers and between one and 40 neurons per layer. The number of neurons is a critical architecture choice that affects the network's performance. Too few hidden layer neurons may result in underfitting, where the network struggles to capture complex patterns and yields low accuracy. In contrast, too many hidden layer neurons may lead to overfitting, when the network becomes overly focused on the training set, resulting in poor generalization and decreased accuracy on brand-new, untried data. [11].

Figure 4 shows the MSE across varying networks architecture. The result reveal that all the three activation functions have a slight preference to an intermediate level of complexity. Very small architectures perform poorly and fail to capture the underlying structure, while overly deep networks tend to overfit the data.

ReLU tends to favor models with fewer layers and this may be due to ReLU's tendency to "kill" neurons. If a neuron consistently receives negative inputs during training, it stops updating and effectively becomes

inactive. This effect is more pronounced in deeper networks, where the chance of neurons receiving mostly negative inputs increases. Nevertheless, ReLU does not favor overly simple models either, indicating that a certain level of complexity is still necessary.
Leaky ReLU, on the other hand, does not show a strong correlation with the number of layers, though there is a slight preference for shallower networks. Since Leaky ReLU assigns a small non-zero slope to negative inputs, it avoids neuron death and maintains gradient flow even when activations are negative. As with ReLU, however, the poorest performance is observed in the smallest architectures, suggesting that minimal model capacity is still required.
Sigmoid activation appears to favor simpler models, with fewer nodes per layer and a slight preference for fewer layers overall. This behavior can be explained by the saturation property of the sigmoid: in deeper or wider networks, many neurons tend to produce outputs near 0 or 1, where gradients vanish, making training inefficient and inhibiting convergence.

Overall, the results confirm the importance of balancing model complexity and activation choice: selecting an appropriate level of complexity ensures sufficient representational power and robust generalization.

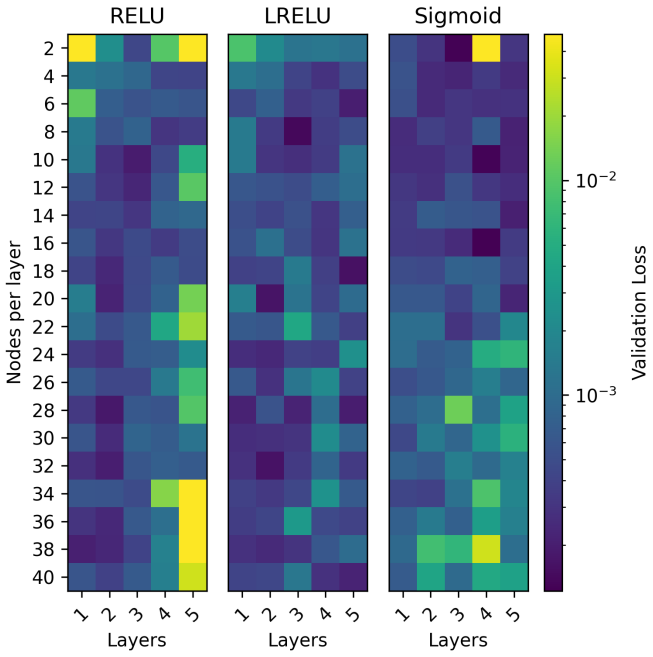

FIG. 4: MSE, per number of nodes per layer, per number of layer, for the ReLU, LeakyReLU and the Sigmoid activation functions. Results are displayed in heatmap format with logaritmic scale for easy visualization.

2. *Effect of activation function on FFNN predictions*

Based on the previous analysis, which showed that intermediate-complexity architectures perform best across activation functions, a FFNN with 3 hidden layers and 30 neurons per layer was trained. The predicted outputs were then compared to the true Runge function, and the mean squared error (MSE) was calculated for each activation function. As shown in Figure 5, ReLU produces sharper transitions and flat regions due to neurons becoming inactive when their pre-activation is negative. This allows the network to fit point-wise variations, but also increases the tendency to overfit and generates irregular, piecewise-linear predictions. Leaky-ReLU represents a compromise: it preserves the modeling flexibility of ReLU while reducing abrupt changes. Since the gradient remains non-zero in the negative region, the network avoids "dead neurons" and yields smoother and more stable predictions. Sigmoid inherently smooths the output because of its saturating behavior. Neurons tend to saturate in high-activation regions, pushing predictions toward a maximum value and reducing the model's ability to represent rapid local variations. As a result, predictions appear smoother but less expressive.

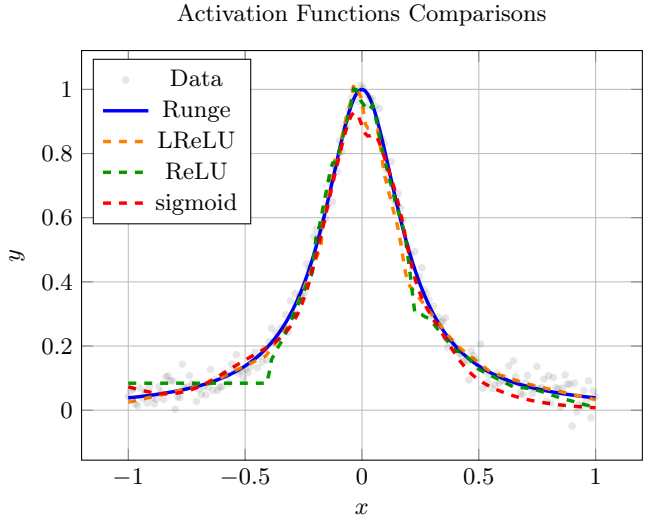

FIG. 5: Comparison of the runge function and the activation functions.

3. *Effect of L1 and L2 Regularization Parameters*

The impact of L_1 and L_2 regularization was studied by introducing both penalties simultaneously into the cost function, allowing the model to benefit from sparsity (L_1) and weight stabilization (L_2). Regularization affects models differently depending on their complexity. Indeed, two network architectures were tested: a simple one with 4 nodes for one hidden layer and a complex one with 40 nodes for 5 hidden layers. All architectures were

trained using the ReLU activation function.
The best λ_1 and λ_2 values found are reported in table I.

Model	λ_1	λ_2	Validation Loss
Simple	$1e{-}6$	$1e{-}6$	$3.28e{-}3$
Intermediate	$1.67e{-}4$	$1.67e{-}4$	$2.31e{-}3$
Complex	$1e{-}6$	$3.59e{-}6$	$1.94e{-}3$

TABLE I: Optimal regularization parameters (λ_1 and λ_2) and corresponding validation loss for the three FFNN models.

For simple architectures, as shown in Figure 6, the optimal values are extremely small, indicating that almost no regularization is needed to achieve good performance. Excessive regularization increases validation loss, indicating underfitting.

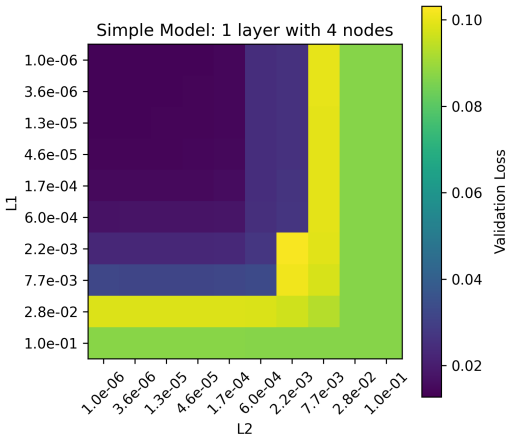

FIG. 6: Heatmap showing the effect of L_1 and L_2 regularization on a simple model. The model error gradually increases with stronger regularization, as expected for low-complexity architectures.

The complex model reaches its lowest validation loss with small but asymmetric regularization values, suggesting that even in highly parameterized networks, light regularization can help stabilize training and control overfitting, as shown in Figure 7. The complex model maintains low MSE for moderate regularization, but exhibits a sudden drop in performance at high λ values. Beyond this optimal region, further increasing λ_1 or λ_2 rapidly degrades performance, as the network weights become overly constrained and the model underfits the data.

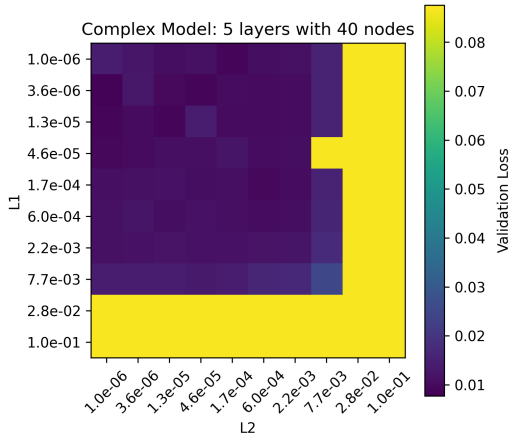

FIG. 7: Heatmap showing the effect of L_1 and L_2 regularization on a more complex model. The model achieves low MSE for moderate regularization, but a sudden drop in performance occurs at high regularization factors.

4. *Effect of Gradient Methods and Learning Rate*

Training neural networks requires careful selection of optimization algorithms and learning rates, which are crucial for ensuring convergence and stability. If the learning rate is too small, convergence becomes slow; if it is too large, parameter updates may overshoot, causing instability during training.

In this experiment, different optimization algorithms and a wide range of learning rates were tested for both full-batch gradient descent and mini-batch stochastic gradient descent (SGD). Specifically, the following methods are evaluated: Vanilla (constant), Momentum, Adragrad, Adagrad with momentum, RMSProp and Adam.

The results of the full-batch setting are shown on the left panel of Figure 8, where good performance occurs only in a relatively narrow range of learning rates, approximately $10^{-5} - 10^{-3}$. Outside this range, the validation loss becomes higher. On the other hand, full-batch training computes weight updates using the entire training dataset at each epoch, producing accurate gradient estimates but fewer updates per epoch. Consequently, training is highly sensitive to the learning rate: too high, and the training can quickly diverge; too low, and convergence is slow.

In contrast, stochastic gradient descent (SGD) uses smaller subsets of the data to compute each update, allowing more frequent updates. This not only accelerates convergence but also helps the optimizer escape shallow local minima and explore the loss surface more effectively. As a result, the range of learning rates that lead to successful training is broader, as shown on

the right panel of Figure 8. Adaptive optimizers like Adam, RMSProp, and Adagrad amplify this robustness, allowing even larger learning rates to remain stable. Indeed, they show low validation loss over a much wider range of learning rates, as visible from the large dark region in the lower rows of the SGD heatmap. There is no upper bound in this case, so if the learning rate range were extended, the validation loss would eventually increase again. Overall, adaptive methods tend to offer more robust performance across different hyperparameter settings, often leading to more reliable training compared to standard gradient descent.

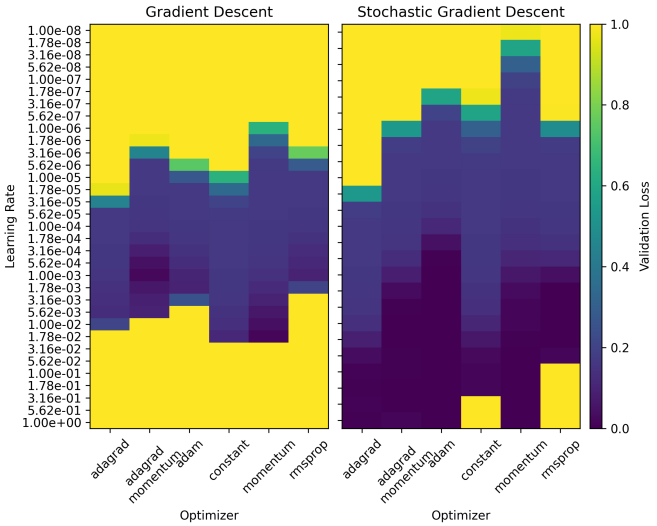

FIG. 8: Validation loss as a function of the learning rate for different optimization algorithms. The left panel shows the results for full-batch gradient descent, while the right panel displays the results for mini-batch stochastic gradient descent (SGD).

5. *Comparison to Traditional Methods*

Traditional regression methods, such as OLS, Ridge and Lasso, are classical linear modeling techniques used to estimate model parameters. OLS provides a simple closed-form solution, but assumes a strictly linear relationship between inputs and targets, limiting its ability to capture complex patterns. Ridge regression adds an L_2 penalty to reduce overfitting and handle multicollinearity, while Lasso introduces an L_1 penalty to shrink some coefficients to zero, improving interpretability and reducing model complexity. However, all three methods struggle with non-liner relationships and complex interactions between variables. FFNNs overcome this limitation through hidden layers and non-linear activation functions.

To demonstrate this advantage, a comparison was carried out between the best-performing OLS, Ridge and Lasso configuration from our previous work [1] and the Feed-Forward Neural Network (FFNN) implemented in this work. All models were trained on the same datasets generated from the one dimensional Runge function. Ridge and Lasso included a model selection step using an inner validation split to choose the optimal regularization parameter (λ), while the FFNN had two hidden layers with 30 nodes each, used the Leaky ReLU activation function in the hidden layers and the linear activation function in the output layer, and was trained using the Adam optimizer.

The results, measured in terms of MSE, confirmed that the FFNN outperformed all linear methods in approximating the behavior of the Runge function. OLS and Ridge achieved nearly identical performance, indicating that the L_2 regularization in Ridge had little effect in this context. Lasso, on the other hand, showed higher errors due to the stronger coefficient shrinkage introduced by the L_1 penalty, which led to underfitting. Figure 9 shows the comparison between the true Runge function, the polynomial fits from OLS, Ridge, Lasso and the FFNN predicition, while Table II summarizes the averaged performance metrics over 30 runs. The FFNN produced smoother, less oscillatory predictions, especially near the boundaries, whereas the polynomial models tended to follow the oscillations of the Runge function more closely.

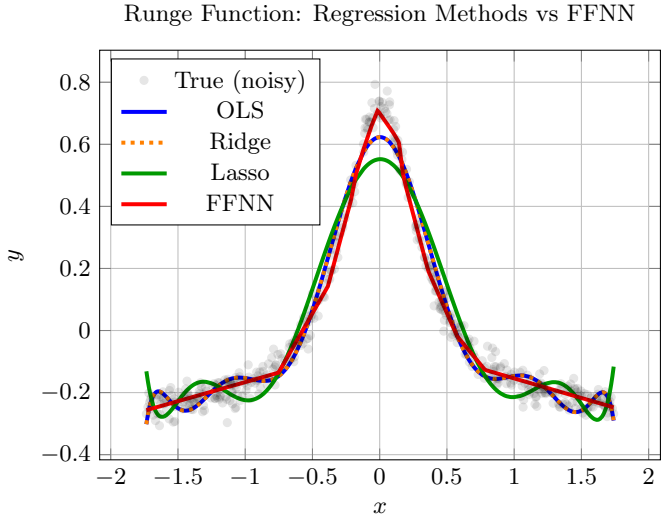

FIG. 9: Comparison of true Runge function data points (with noise) and predictions from OLS, Ridge, Lasso and FFNN on the test set.

Even thus, the best-performing linear models were used as a benchmark, yet the neural network's ability to approximate non-linear functions yielded superior results, showing the lower MSE values, even without any prior optimization of hyper-parameters and, therefore, without any guarantees on the validity of the chosen architecture. These values are particularly significant, as they confirms that the FFNN generalizes better than the

traditional models, capturing the non-linear structure of the Runge function.

TABLE II: Comparison of averaged performance metrics over 30 runs for OLS, Ridge, Lasso, and FFNN models on the Runge function.

Model	Train MSE	Test MSE
OLS	0.00245 ± 0.00009	0.00249 ± 0.00017
Ridge	0.00245 ± 0.00009	0.00249 ± 0.00017
Lasso	0.00524 ± 0.00018	0.00529 ± 0.00041
FFNN	0.00132 ± 0.00037	0.00135 ± 0.00033

B. Validation and testing

In order to evaluate the implemented Feed-Forward Neural Network and its backpropagation algorithm, two different tests were performed.

First, the results are compared with those obtained using a standard machine learning framework, PyTorch. Across six network layouts with different hidden layers and widths, the comparison between the custom FFNN and the PyTorch MLP highlights clear and consistent differences in both accuracy and stability. The custom FFNN performs well for smaller architectures, achieving validation MSE values on clean data around 5×10^{-4} to 3×10^{-3} for networks with up to two hidden layers, but its performance deteriorates as model depth and width increase, reaching approximately 2.2×10^{-3} for three hidden layers of width 20 and 6.9×10^{-3} for three hidden layers of width 50. In contrast, the PyTorch MLP consistently maintains lower validation MSE values between 2×10^{-4} and 8×10^{-4} across all layouts (except the linear baseline at around 6.8×10^{-2}), demonstrating higher robustness and more stable convergence, as evident in the bar chart of final clean validation losses in Figure 10.

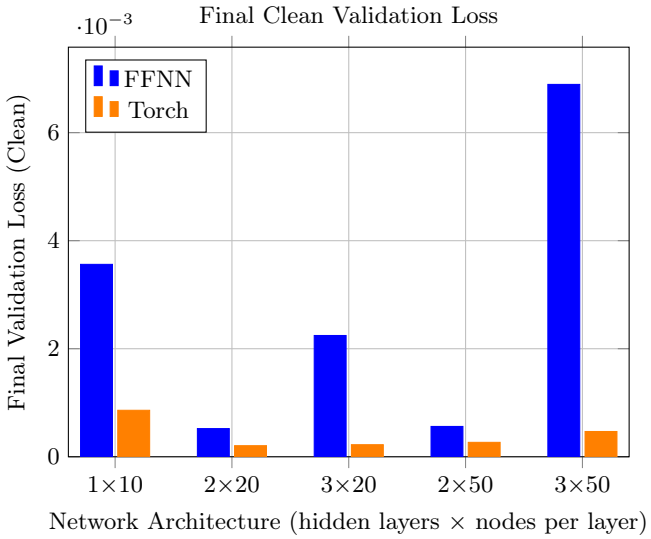

FIG. 10: Final clean validation loss for FFNN vs Torch across different network architectures.

Training time for both implementations scales roughly as a power law with the number of parameters, as shown in the computation time benchmark in Figure 11. PyTorch, however, is systematically slower by a factor of roughly 1.8-2.3 across the tested layouts, likely due to the computational overhead of its dynamic computation graph and general-purpose tensor management.

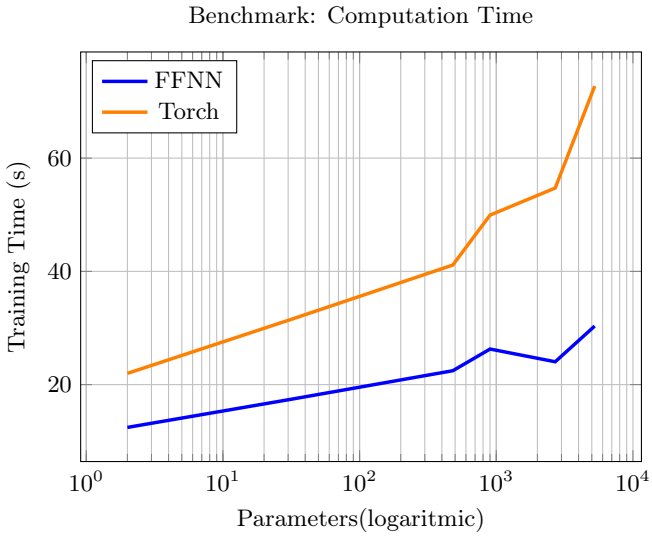

FIG. 11: Comparison of computation time between FFNN and PyTorch implementations

Overall, these results demonstrate that while the custom FFNN captures the correct learning dynamics in low-capacity regimes, it becomes unstable as complexity increases, particularly for deeper networks. The superior performance and consistency of the PyTorch implementation are primarily attributable to its opti-

mized initialization schemes, numerically stable Adam updates with bias correction, and mini-batch shuffling, which together ensure smoother optimization and better generalization.

Furthermore, the gradients computed analytically by the implemented backpropagation algorithm were tested against Autograd, an automatic differentiation tools. The relative error was $\sim 3 \times 10^{-17}$, indicating numerical agreement up to machine precision. This confirms that the analytical derivatives computed in the FFNN code are mathematically correct and numerically stable.

C. Classification: MNIST Dataset

The Feed-Foward Neural Network is then adapted to a multiclass classification task using the MNIST dataset, which is a widely used collection of handwritten digits, serving as a benchmark for training and evaluating image classification algorithms in machine learning. For classification, the output layer uses the Softmax activation function and the cross-entropy loss is employed. Performance is measured using accuracy.

1. *Effect of Network complexity and Activation function*

The experiment analyzed the performance of feedforward neural networks (FFNNs) on the MNIST dataset by varying the network architecture and activation functions. Networks with one to five hidden layers and between 32 and 512 neurons per layer were tested, using either ReLU or Leaky ReLU as activation functions. The learning rate of 0.001, combined with the Adam optimizer, allowed stable and smooth convergence. The results are summarized through heatmaps of accuracy and loss: the accuracy heatmap highlights how the classification accuracy improves with increasing network capacity, while the loss heatmap shows a corresponding decrease in training and validation loss. Indeed, as shown in Figures 12 and 13, networks with a small number of hidden layers or neurons tend to underfit the data, resulting in high validation loss and low accuracy. As the number of layers and neurons increases, the networks are able to capture more complex patterns, leading to improved accuracy and reduced loss. The choice of the activation functions has also influence on the performance: ReLU can lead to dead neurons during training, especially in deeper networks, while Leaky ReLU achieves slightly lower validation loss and higher accuracy, especially in larger or deeper networks.

Networks with four and five hidden layers and between 256 and 512 neurons per layer, using Leaky ReLU, achieve the best trade-off between complexity and accuracy. The network achieving the best results has five hidden layers with 512 neurons per layer, which is the largest architecture that was tested. Since this represents the upper bound of the explored architectures, it's likely that even deeper or wider networks could further improve performance.

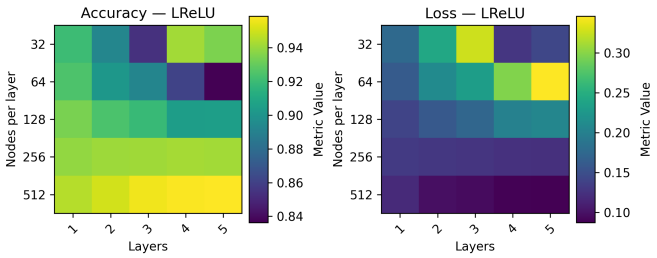

FIG. 12: Accuracy and Loss for the Leaky ReLU activation function as a function of model complexity.

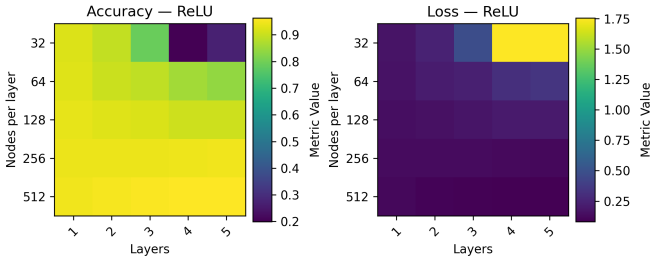

FIG. 13: Accuracy and Loss for the ReLU activation function as a function of model complexity

Then, a confusion matrix was computed on this validation set. The confusion matrix provides a detailed view of the model's predictions for each digit class, showing both correct classifications along the diagonal and misclassifications in the off-diagonal elements. As shown in Figure 14, most digits are predicted accurately, with high values along the diagonal, confirming the high overall accuracy of this network and its ability to generalize effectively to unseen data. Misclassifications are relatively rare but tend to occur between visually similar digits, such as 4 and 9, or 3 and 5. These errors are consistent with the ambiguity in handwritten digits. Overall, the confusion matrix confirms that the selected architecture achieves high accuracy across all classes and demonstrates the network's ability to generalize effectively to unseen data.

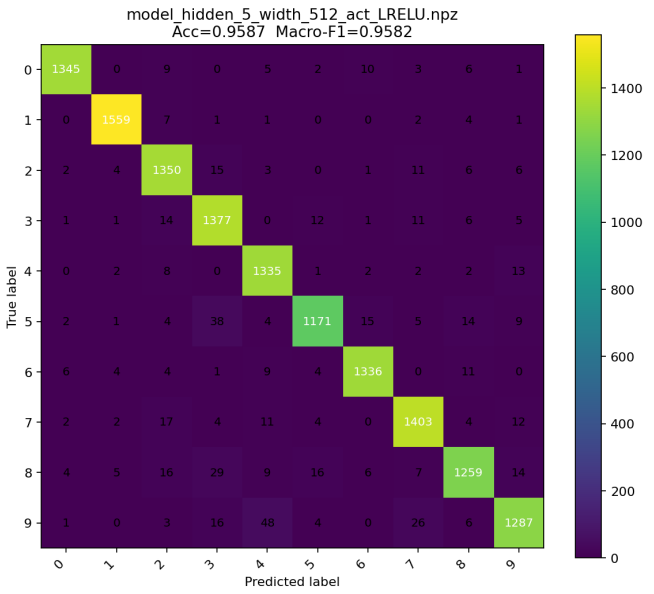

FIG. 14: Confusion matrix

D. Advantages and disadvantages of NN

The application of feed-forward neural networks to both the Runge function and the MNIST Dataset demonstrates their strengths and limitations. For the regression task, neural networks effectively capture the non-linear and oscillatory nature of the data, outperforming traditional regression methods such as OLS, Ridge, and Lasso. Their flexible and adaptable architecture and non-linear activations allow them to approximate complex relationships that linear models cannot represent. Similarly, the experiments on MNIST show that neural networks are well suited for classification tasks, as their non-linear modeling capabilities enable them to capture patterns in handwritten digits, achieving high accuracy when the architecture and activation functions are properly chosen. Unlike regression problems, classification of MNIST cannot be addressed with traditional linear methods like OLS, making neural networks a natural choice for achieving high accuracy.

However, neural networks require considerable processing power and memory, making them computationally intensive compared to traditional regression methods, particularly for deep or wide architectures such as those used in our MNIST experiments. Neural networks are also less interpretable, as their learned parameters provide little insight into the underlying relationship between input and output. Furthermore, overfitting can occur if the network is too complex or the dataset is too small. Overall, neural networks are advantageous when modeling highly non-linear systems and achieve high accuracy in classification tasks such as MNIST, but for simpler or low-dimensional problems, classical regression methods remain more efficient and interpretable alternatives.

IV. CONCLUSION

The main goal of this project was to develop a malleable custom Feed-Forward Neural Network framework capable of addressing both regression and classification problems, using different gradient descent–based optimization algorithms.

For regression, the FFNN was applied to the one-dimensional Runge function and was analyzed by varying both the network complexity and the activation function. The results show that, Within the explored search space (1–5 hidden layers; 1–40 neurons/layer), moderate-capacity networks achieved the lowest validation error, achieve the best trade-off between underfitting and overfitting. This general trend is observed across all three activation functions used (ReLU, Leaky ReLU, and Sigmoid). Similar patterns emerge when analyzing other hyperparameters, such as optimization methods and L_1 and L_2 regularization, highlighting the importance of balanced model complexity for optimal performance.

Furthermore, FFNN significantly outperforms traditional linear regression methods, such as OLS, Ridge and Lasso, in the regression of the Runge function reaching the lowest MSE justified by its non-linear nature.

For the classification task, the FFNN was applied to the MNIST dataset. Within the range of architectures we tested (1–5 hidden layers, 32–512 neurons per layer), the highest accuracy was obtained by deeper and wider networks using Leaky ReLU activation. The confusion matrix indicates that most digits were predicted accurately, while misclassifications mostly occurred between visually similar digits, suggesting that the networks effectively learned the main features of each class. Compared to regression tasks, classification cannot be addressed with simple linear models, reinforcing the very general strength of FFNNs in handling complex, high-dimensional data.

Despite their strong performance, FFNNs present several limitations. They are computationally expensive, particularly for deeper or wider architectures, and require careful tuning of hyperparameters such as the number of layers, neurons per layer, activation functions, and learning rates. Due to the long runtimes required by the implemented code, the experiments were restricted to moderately sized networks. Increasing the depth or width of the architecture would likely have achieved more accurate results; however, the training time grew significantly with complexity.

An additional limitation emerged repeatedly during both the learning–rate experiments and the complexity analysis (for both regression and classification): in several grids, the best-performing configuration was located at the boundary of the tested domain. For example, in the learning–rate sweep, the smallest validation loss for adaptive optimizers was obtained at the upper edge of the explored learning–rate interval, while in the architecture search for MNIST, the highest accuracy occurred at the maximum tested capacity.

These observations suggest that better-performing models may lie outside the explored domain, and that our search space was likely too conservative. Future work could therefore focus on extending both the architectural and hyperparameter search beyond the current limits, while compensating for the increased runtime by using a sparser sampling of candidate models. Additionally, automated model–selection strategies (e.g., learning–rate schedules or neural architecture search) could be employed. Such approaches would not only help identify superior configurations outside the tested boundaries, but also reduce the overall computational cost while improving generalization performance.

Moreover, extending these methods to other complex regression and classification tasks, such as time-series forecasting or image recognition in real-world datasets, could further highlight the versatility and practical applicability of FFNNs.

[1] T. G. C. C. Stan Daniels, Francesco Minisini, Analyzing different regression and resampling methods, urlhttps://github.com/STK3155-25H/Project-1.git (2025), accessed: 2025-29-10.

[2] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, *et al.*, nature **596**, 583 (2021).

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, Advances in neural information processing systems **25** (2012).

[4] P. Rajpurkar, J. Irvin, K. Zhu, B. Yang, H. Mehta, T. Duan, D. Ding, A. Bagul, C. Langlotz, K. Shpanskaya, *et al.*, arXiv preprint arXiv:1711.05225 (2017).

[5] M. Hjorth-Jensen, Machine learning lecture notes, week 41, `https://github.com/CompPhysics/MachineLearning/blob/master/doc/LectureNotes/week41.ipynb` (2025), accessed: 2025-10-09.

[6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) `http://www.deeplearningbook.org`.

[7] K. Hornik, M. Stinchcombe, and H. White, Neural networks **2**, 359 (1989).

[8] C. M. Bishop, *Neural networks for pattern recognition* (Oxford university press, 1995).

[9] C. Runge, Zeitschrift für Mathematik und Physik **46**, 224 (1901), english translation: "On empirical functions and interpolation between equidistant ordinates".

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, nature **323**, 533 (1986).

[11] T. Deng, Highlights in Science, Engineering and Technology **74**, 462 (2023).