Comparison of Feed Forward Neural Networks with Classical Polynomial Regression Methods

Live L. Storborg and Henrik Haug
Department of Mathematics, University of Oslo,
PO Box 1053, Blindern 0316, Oslo, Norway

AND

Simon S. Thommesen and Adam Falchenberg
Institute of Theoretical Astrophysics, University of Oslo,
PO Box 1029, Blindern 0315, Oslo, Norway

(Dated: November 11, 2025)

Neural networks are widely used in modern machine learning, yet it remains non–trivial to quantify when they offer a practical advantage over classical statistical models. In this work, we therefore compare feedforward neural networks to linear regression methods (OLS, Ridge, Lasso) on both a controlled regression task (Runge function) and a classification task (MNIST dataset of handwritten digits). We systematically investigate how activation functions, optimizers, network depth, and L_1/L_2 regularization influence training dynamics and generalization. Learning curve analysis shows that adaptive optimizers (Adam, RMSprop) converge rapidly and consistently outperform full–batch gradient descent, particularly in deeper networks. Batch-normalization proves to add more stability to different architectures. Regularization further stabilizes optimization and improves robustness in noisy settings. For the low–noise regression task, both L_1 and L_2 regularization produce lower validation MSE than unregularized networks and the corresponding linear baselines. Finally, we apply the same methodology to MNIST classification. A hyperparameter and architecture search reveals that deeper architectures benefit from the larger dataset, achieving a test accuracy of 97.96% on the full training set, which is close to estimated human performance ($\approx 98\%$). Overall, our neural network proves to be very good at the MNIST classification problem, while the best regression methods highly depend on problem complexity and characteristics of the data.

I. INTRODUCTION

Neural networks have become an increasingly essential tool within the field of machine learning. While they are more complex than traditional regression methods, neural networks offer flexible and efficient ways to approximate complex, nonlinear relationships between variables. This makes it particularly interesting to investigate how their performance compares to more conventional regression models. This comparison can guide the field of machine learning into making models more efficiently, especially when considering the contest between simplicity and accuracy.

In this project, we extend our previous analysis of the Runge function

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

by applying feedforward neural networks (FFNNs) to the same regression problem previously studied using polynomial fits [1] (the reason for choosing the Runge function is explained in the previous article [1], and is in essence because of Runge's phenomenon [2]). The main objective is to compare the performance of FFNNs with traditional regression approaches such as Ordinary Least Squares (OLS), Ridge, and Lasso regression, and to examine how neural networks can overcome the limitations encountered in polynomial regression. We will explore the impact of different activation functions such as the sigmoid, ReLU, and Leaky ReLU, while comparing the effects of L_1 and L_2 regularization and testing various optimization algorithms, including Gradient Descent, RMSprop, and Adam.

Through this comparison, we aim to investigate how neural networks learn nonlinear mappings, how the choice of activation and optimizer influences convergence and generalization, and how regularization affects model performance. This provides a deeper understanding of how neural networks extend the principles of linear regression to more flexible, data-driven learning frameworks.

To further demonstrate the neural network's effectiveness and ability of solving complex problems, we address a classification problem in addition to approximating the Runge function. A classification problem aims to assign inputs to discrete categories rather than predicting continuous values, as in regression. We used neural networks to identify and group handwritten digits from MNIST. The MNIST dataset of handwritten digits [3] was obtained from Kaggle's mirrored version [4].

Section II introduces the theoretical background with the implementation details in Section III, followed by the presentation of our results in section IV and a discussion in section V. Finally, we will conclude our findings in Section VI.

II. THEORY

A. Neural Network

A neural network is a computational tool designed to mimic a biological neuron system. The human brain is a good example of a biological system. The brain consist of neurons communicating with each other through electric signals. A neuron receives a signal before sending a signal to other neurons. A neural network also consists of neurons communicating with each other, although here, through mathematical functions. Each neuron receives an input, which is operated on by an activation function (see below). This new value is the neurons output. The output is either sent to another layer of neurons, or sent to the "output layer", i.e. the result of the neural networks operation on the inputs. The layers of neurons are called hidden layers, where each layer has a set number of neurons.

In addition to the activation functions, the output from a neuron also depends on weights and biases. Each connection between neurons has an associated weight, which determines how strongly the output from the first neuron influences the next. The bias can shift the weighted activation function, allowing the neuron to represent data not passing through the origin. The weights and biases are the parameters the neural network model learns when trying to approximate target data.

B. Feed Forward

A FFNN is a type of neural network where information moves in only one direction, from the input layer, through one or more hidden layers, to the output layer. Each neuron computes a weighted sum of its inputs and adds a bias term

$$z_j^{(l)} = \sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)}, \tag{2}$$

for each neuron j in layer l, where $a_i^{(l-1)}$ are the activations (explained in Section II C) from the previous layer (with $a_i^{(0)} = x_i$ being the input features). This relation can also be expressed in vectorized form:

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}, \tag{3}$$

where $\boldsymbol{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$, $\boldsymbol{b}^{(l)} \in \mathbb{R}^{n_l}$, and $\boldsymbol{a}^{(l)} \in \mathbb{R}^{n_l}$. This weighted sum is then passed through an activation function $f^{(l)}$

$$\boldsymbol{a}^{(l)} = \boldsymbol{f}^{(l)}(\boldsymbol{z}^{(l)}), \tag{4}$$

which introduces nonlinearity to the model. For the output layer L, the same relation holds. The final vector $\mathbf{a}^{(L)}$ represents the network's output, given the input $\mathbf{x}$.

C. Activation Functions

An activation function determines the output of a node. Each input and hidden node first computes a weighted sum of its inputs plus a bias term, and the activation function then transforms this value. This introduces nonlinearity, allowing the neural network to learn complex relationships rather than just simple linear ones. Different choices of activation functions can significantly affect speed, learning rate, stability and performance.

1. Sigmoid

The logistic sigmoid function

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{5}$$

with its derivative being

$$\sigma'(x) = \sigma(x)\,(1-\sigma(x)), \tag{6}$$

is a common choice for an activation function as it forces the input values to be between 0 and 1. For simplicity, we will refer to Equation (5) as just the sigmoid function. As the sigmoid function has a smooth derivative, it is perfect for gradient-based optimization and it is useful in neural networks for assigning weights on a relative scale. The sigmoid function does however struggle with large inputs causing gradients to vanish.

2. ReLU

The Rectified Linear Unit (ReLU) function is defined as

$$f(x) = \begin{cases} x \text{ if } x > 0 \\ 0 \text{ if } x \leq 0 \end{cases}, \tag{7}$$

with its derivative being

$$f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}. \tag{8}$$

It passes positive input values directly while setting negative inputs to zero, making it simple and computationally efficient to implement. By zeroing out negative activations, ReLU introduces sparsity in the network, which can help reduce overfitting and improve generalization. Additionally, it fixes the vanishing gradient problem of the sigmoid. A big drawback with the ReLU, however, is that if all inputs are negative, then all outputs become 0, effectively stopping the learning. This is known as the "dying ReLU problem".

3. Leaky ReLU

The leaky ReLU activation function is defined as

$$f(x) = \begin{cases} x \text{ if } x > 0 \\ \alpha x \text{ if } x \leq 0 \end{cases}, \tag{9}$$

with its derivative being

$$f'(x) = \begin{cases} 1 & x > 0 \\ \alpha & x \leq 0 \end{cases}, \tag{10}$$

where α is a small constant that determines the slope for the negative inputs. α helps solve the dying ReLU problem, as negative inputs will be small, but not 0. While fixing the dying ReLU problem, the leaky ReLU is still computationally efficient and even introduces better learning stability. It is, however, still unbounded from above, and large, positive activations can explode if not properly normalized. According to [5, p. 601], "the leaky ReLU activation function permits non-zero gradients for negative inputs as well, and as a result, it makes the networks more expressive overall."

4. Linear activation function

A linear activation function would simply return the same as the input layer, and we have

$$a = z. \tag{11}$$

The linear activation function is a common choice for the output layer for regression models because the network should be free to output any real number.

5. Softmax

"Softmax activation in the last layer is used to support multiclass classification" [5, p. 397]. The reason it is used within multiclass classification is that it turns raw input data into a probability distribution

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}, \tag{12}$$

where each output value represents the estimated probability of the model assigning the input to class i. Since all probabilities lie in $(0, 1)$ and sum to 1, the softmax function is well-suited as the output activation for multiclass classification tasks.

6. Gradient instability

The sigmoid activation in Equation (5) maps inputs to the interval $[0, 1]$. When the absolute magnitude of z becomes large, its derivative (Equation 6) approaches zero. From the recursive backpropagation relation in Equation (26), we see that the error signal $\boldsymbol{\delta}^{(l)}$ at layer l depends both on the derivative of the activation at that layer and the propagated $\boldsymbol{\delta}$-values from layers above. If many consecutive layers have small $\sigma'(z)$, the resulting $\boldsymbol{\delta}^{(l)}$ becomes extremely small. Consequently, the weight and bias updates at that layer also become tiny, slowing or effectively stopping learning. This is the vanishing gradient problem.

The ReLU activation function (7) does not have the problem of vanishing gradients, but rather the problem of dying neurons. Instead of multiplying many small gradients, the gradients for the ReLU function is zero for negative inputs, see Equation (8). Similarly to the vanishing gradient problem, this also leads to stalled learning, since here, the gradient becomes permanently zero instead of diminishing. We then say that the neuron has died. When the neurons die, it effectively stops contributing to the model. The leaky ReLU activation function fixes this problem by introducing the constant α, making the gradient non-zero for negative inputs. Although these activation functions do not really have the problem of vanishing gradients, they can still experience exploding gradients. In these cases, the gradients become too large, causing large updates to the weights and biases, which prevents the model from converging to a stable minimum of the loss function. This happens when weights and biases are initialized poorly in accordance to the input values (for weight and bias initialization see Section II F, or alternatively [6]).

D. Cost/Loss Functions

A cost/loss function is a measure of how well a model's predictions match the true data. Different cost functions will be more or less suitable depending on the problem type. During training, the main goal is to minimize the cost function, as we want our model to predict the true values as accurately as possible. We will employ three different cost/loss functions in this project.

1. Mean Squared Error

The mean squared error (MSE) is a common loss function, as it directly calculates the average difference between each point, penalizing larger errors more heavily. In neural networks, we will calculate the MSE as

$$\mathcal{L}_{MSE} = \frac{1}{N} \sum_{i=1}^{N} (a_i^{(L)} - y_i)^2, \tag{13}$$

which allows us to write the gradient of the loss w.r.t. the output activations as

$$\frac{\partial \mathcal{L}}{\partial a_i^{(L)}} = \frac{2}{N}(a_i^{(L)} - y_i). \tag{14}$$

where $a_i^{(L)}$ is the activation function for the last layer L, for neuron i, and y_i are the true values. By minimizing the MSE, we encourage our model to produce results that are closer to the true values on average. To minimize this we need to find the derivative. The minimization process is done in Section II G. The MSE with regularization can be seen in Section II J 2 (or more specific form in Section II H 2 for L_2-norm and Section II H 3 for discussion around derivative with L_1-norm).

2. Binary Cross Entropy

Binary cross entropy (BCE) is used for binary classification problems, where the target values $y_i \in \{0, 1\}$, and is calculated as

$$\mathcal{L}_{BCE} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log\left(a_i^{(L)}\right) + (1-y_i)\log\left(1-a_i^{(L)}\right), \tag{15}$$

with its derivative w.r.t the output being

$$\frac{\partial \mathcal{L}_{BCE}}{\partial a_i^{(L)}} = -\frac{1}{N}\left(\frac{y_i}{a_i^{(L)}} - \frac{1-y_i}{1-a_i^{(L)}}\right). \tag{16}$$

It measures the dissimilarity between the predicted probabilities $a_i^{(L)}$ and the true values y_i. Minimizing the BCE encourages the model to output probabilities close to 1 for the correct class and close to 0 for the incorrect one.

3. Multiclass Cross-Entropy

The Multiclass Cross-Entropy (MCE) loss extends the concept of BCE to multi-class classification problems. It measures the dissimilarity between the predicted probability distribution $a_{ij}^{(L)}$ and the true class labels y_{ij}.

$$\mathcal{L}_{MCE} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log\left(a_{ij}^{(L)}\right), \tag{17}$$

with its derivative w.r.t the output being

$$\frac{\partial \mathcal{L}_{MCE}}{\partial a_{ij}^{(L)}} = -\frac{1}{N}\frac{y_{ij}}{a_{ij}^{(L)}}, \tag{18}$$

where M is the number of classes, y_{ij} is 1 if sample i belongs to class j (otherwise $y_{ij} = 0$), and $a_{ij}^{(L)}$ is the predicted probability for that class. Minimizing the MCE encourages the model to assign high probability to the correct class for each sample.

E. Metrics

To evaluate the performance of our models, we employ different metrics depending on the learning task. For regression problems, we use the mean squared error (MSE), while for classification problems, we evaluate performance in terms of accuracy.

1. MSE

When using MSE as a performance metric, we evaluate it using Equation (13). The MSE quantifies the average squared difference between the predicted values and the true targets, and therefore provides a direct measure of how far, on average, the model's predictions deviate from the true values.

2. Accuracy

For classification problems, we evaluate model performance using the accuracy metric. Let t_i denote the true class label for sample i, and let $\tilde{y}_i$ denote the predicted class label. The accuracy is then defined as

$$\text{Accuracy} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}(t_i = \tilde{y}_i), \tag{19}$$

where $\mathbb{I}$ is the indicator function which returns 1 if its argument is true and 0 otherwise. The accuracy score therefore lies in the interval $[0, 1]$, where a score of 1 means that all predictions are correct.

3. Confusion Matrix

For classification problems, we additionally evaluate model performance using a confusion matrix. A confusion matrix summarizes the predicted versus true class assignments and reports the counts of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). In the multi–class setting that we consider, the confusion matrix generalizes this concept by placing all correct predictions along the main diagonal, while off–diagonal entries indicate misclassifications and specify which incorrect class the model predicted.

F. Weight and Bias Initialization

Proper weight initialization is crucial for training deep neural networks, as poor initialization can lead to vanishing or exploding gradients. We employ activation-aware initialization strategies tailored to different activation functions.

For bias initialization, we use small random values drawn from a normal distribution with standard deviation 0.01, rather than the common practice of initializing to zero. This choice is motivated by the Runge function's output range of [0, 1]. Initializing biases to zero is standard and works well in most cases, but small random

values can provide a slight advantage by giving neurons different starting points in their activation ranges, potentially leading to more diverse learned features in the early stages of training.

For layers with ReLU or Leaky ReLU activations, we use He initialization [6], which accounts for the asymmetric nature of rectifier nonlinearities. Weights are drawn from a normal distribution with a standard deviation

$$\sigma = \sqrt{\frac{2}{n_{\text{in}}}}, \tag{20}$$

where n_{in} is the number of input units.

For sigmoid activations, we apply Xavier initialization [5, Ch. 11], with

$$\sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}, \tag{21}$$

where n_{out} is the number of output units. This initialization is designed for symmetric activation functions and maintains variance across layers.

G. Back Propagation

During training, the network's weights and biases are updated by propagating the error backwards using gradient descent methods. This is done by computing the gradients of the loss function with respect to all trainable parameters. By applying the chain rule of differentiation, we can propagate the error backwards, layer by layer.

Let $\mathcal{L}(\boldsymbol{a}^{(L)}, \boldsymbol{y})$ denote the loss function, where $\boldsymbol{a}^{(L)}$ is the network output and $\boldsymbol{y}$ is the target. In this derivation, we consider the MSE loss function, following the standard formulation presented in [5]

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(a_i^{(L)} - y_i)^2, \tag{22}$$

which allows us to write the gradient of the loss w.r.t. the output activations as

$$\nabla_{a^{(L)}}\mathcal{L} = \frac{2}{N}(a^{(L)} - y). \tag{23}$$

The gradient is then given by

$$\boldsymbol{\delta}^{(l)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}}. \tag{24}$$

Computing this error term at the output layer will be the starting point. This is done by

$$\boldsymbol{\delta}^{(L)} = \nabla_{\boldsymbol{a}^{(L)}}\mathcal{L} \odot f'(\boldsymbol{z}^{(L)}), \tag{25}$$

where $f'(\boldsymbol{z}^{(L)})$ is the derivative of the last activation function and $\odot$ denotes element-wise multiplication.

Now, moving backwards through the hidden layers, we will calculate the previous gradient by

$$\boldsymbol{\delta}^{(l)} = \left(\boldsymbol{W}^{(l+1)T}\boldsymbol{\delta}^{(l+1)}\right)\odot f'(\boldsymbol{z}^{(l)}), \quad l = L-1, L-2, \ldots, 1, \tag{26}$$

where we clearly notice that the gradients at layer $l+1$ affect the gradients at layer l.

Once all $\boldsymbol{\delta}^{(l)}$ are known, we can start computing the parameter gradients. This is done by computing the derivative of the loss function $\mathcal{L}$ w.r.t the parameters:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(l)}} = \boldsymbol{\delta}^{(l)}(\boldsymbol{a}^{(l-1)T}), \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(l)}} = \boldsymbol{\delta}^{(l)}. \tag{27}$$

The step made in Equation (27) is the step that ties the backward pass to the actual parameter updates. Once the gradients in (27) are computed, the network parameters can be updated in order to minimize the loss function.

H. Linear Regression

We start our analysis by focusing on linear regression. We consider three variants: OLS, Ridge regression, and Lasso regression (for a more in depth explanation and discussion around these regression models, see our previous work [1]).

1. Ordinary Least Squares

In OLS regression, the parameter vector is obtained by minimizing

$$C_{\text{OLS}}(\boldsymbol{\theta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2. \tag{28}$$

By computing the gradient and setting it to zero, we obtain the closed-form expression

$$\hat{\boldsymbol{\theta}}_{\text{OLS}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}. \tag{29}$$

2. Ridge Regression

Ridge regression augments the OLS cost with an L_2 penalty on the parameters:

$$C_{\text{Ridge}}(\boldsymbol{\theta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2, \tag{30}$$

where $\lambda > 0$ controls the strength of the regularization. The minimizer is given by

$$\hat{\boldsymbol{\theta}}_{\text{Ridge}} = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{y}. \tag{31}$$

3. Lasso

Lasso regression, however, adds an L_1 penalty to the OLS parameters

$$C_{\text{Lasso}}(\boldsymbol{\theta}) = \frac{1}{N}\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1, \tag{32}$$

where λ again controls the strength of the regularization. While Ridge has a closed–form solution, Lasso does not, since the absolute value penalty is not differentiable at zero. As a result, Lasso regression is typically solved with iterative numerical methods. The L_1 penalty encourages sparsity in the solution vector, meaning that Lasso tends to set many coefficients exactly to zero and can therefore perform variable selection.

I. Optimization Methods

We employ three optimization methods to train our neural network: plain gradient descent, and two adaptive stochastic gradient descent variants (RMSprop and Adam). All methods aim to minimize the loss function $\mathcal{L}(\boldsymbol{\theta})$ by iteratively updating the network parameters $\boldsymbol{\theta} = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$ in the direction of steepest descent.

1. Gradient Descent (GD)

Let $F : \mathbb{R}^p \to \mathbb{R}$ be the objective we want to minimize. Gradient descent is an iterative optimization scheme that updates the parameter vector by stepping in the direction opposite to the gradient of F, meaning the direction of maximal decrease. From an initial parameter value $\boldsymbol{\theta}_0$, the update rule becomes

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \nabla F(\boldsymbol{\theta}_k), \qquad k = 0, 1, \ldots \tag{33}$$

where $\eta_k > 0$ denotes the step size (often referred to as the learning rate). This generates a sequence of iterates which, under appropriate assumptions, will approach a minimum of F. The update in Equation (33) corresponds to the standard, or "plain", gradient descent method.

2. Stochastic Gradient Descent

When working with large datasets, computational efficiency becomes crucial. Evaluating the full gradient at each update step can be expensive, since the computational cost scales with the size of the dataset. Stochastic Gradient Descent (SGD) alleviates this by approximating the gradient using a single randomly selected sample, or a small mini–batch, at each iteration. This drastically lowers the cost per update and makes training feasible even on very large datasets. The injected randomness can also help the optimizer escape shallow local minima or saddle points.

There are many methods that can be viewed as adaptive variants of SGD, of which we will consider RMSprop and Adam.

RMSprop maintains an exponentially decaying average of the squared gradients, which enables an element-wise adaptive rescaling of the learning rate.

Let $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t)$ denote the gradient at iteration t, and let

$$\mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2.$$

denote the second-moment accumulator.

The parameter update rule then becomes

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}}, \tag{34}$$

where β controls the decay rate of the second-moment estimate, η_t is the learning rate, and ϵ is a small constant added for numerical stability.

By down-weighting updates associated with consistently large gradients and up-weighting those with smaller gradients, RMSprop tends to produce more stable progress and can speed up convergence in settings with noisy, stochastic gradients.

Adam is another momentum based optimizer, which keeps track of both the first and second moment, denoted by

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t, \quad \mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2,$$

respectively, where β_1 and β_2 are hyperparameters that control the rate of decay. Since these accumulators are initialized at zero, they are biased toward zero early in training. Adam corrects this by

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}, \quad \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2^t}.$$

We then obtain the update rule for Adam as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}, \tag{35}$$

where η_t is the learning rate.

3. Gradient Clipping

To solve the exploding gradient problem which is often encountered when using activation functions such as the sigmoid, we utilize a gradient clipping technique. After computing the gradient, but before applying the parameter update, we enforce a maximum norm by rescaling the gradient whenever it exceeds a specified threshold. Formally, for a threshold τ, if $\|\mathbf{g}_t\|_2 > \tau$, we replace the gradient by

$$\mathbf{g}_t \leftarrow \mathbf{g}_t \frac{\tau}{\|\mathbf{g}_t\|_2}, \tag{36}$$

which will stabilize the training.

4. Batch Normalization

To make the training faster, and more stable, we use a technique called batch normalization. The method aims to address the vanishing/exploding gradient problem described in Section II C 6. The method entails normalizing each mini-batch, making sure the batch's mean $\mu = 0$ and variance $\sigma^2 = 1$ before sending this activation to the next layer. Given a batch of activations $a_1, \ldots, a_m$, you first calculate the mean and variance

$$\mu_B = \frac{1}{M}\sum_i^M a_i, \quad \sigma_B^2 = \frac{1}{M}\sum_i^M (a_i - \mu_B)^2, \tag{37}$$

and then normalize the activations

$$\tilde{a}_i = \frac{a_i - \mu_B}{\sqrt{\sigma^2 + \epsilon}}. \tag{38}$$

Finally, a learnable affine transformation is applied

$$y_i = \gamma \tilde{a}_i + \beta, \tag{39}$$

where γ and β are trainable parameters, so the network can still learn any scale/offset if needed.

J. FFNN and Regularization

We will be utilizing fully connected FFNNs, which is previously defined in Section II B. The network parameters are optimized by gradient based methods (GD, RMSprop, Adam) as described in Section II I and trained to minimize the MSE loss. We will analyze their performance with different types of regularization parameters.

1. FFNN without regularization

In the FFNN without regularization scenario, the loss function is simply the MSE from Equation (13). The optimization updates therefore only depend on the gradients propagated through the network weights and biases.

2. FFNN with regularization

To study the effect of regularization, we also train networks where the loss is augmented with an L_1 or L_2 penalty on the weight matrices:

$$\mathcal{L}_{\text{reg}} = \mathcal{L}_{\text{MSE}} + \lambda \sum_{l=1}^{L} \|\boldsymbol{W}^{(l)}\|_p, \quad p = 1 \text{ or } p = 2.$$

This corresponds to Lasso–type (L_1) and Ridge–type (L_2) regularization applied directly to the neural network parameters.

K. Use of AI Tools

To enhance the quality of this scientific report, we utilized several AI tools throughout the writing and development process. Claude and ChatGPT were used to assist with formulating precise scientific language and improving sentence structure to meet academic writing standards. For code development, we used Claude, ChatGPT, and GitHub Copilot to support debugging processes, optimize code structure, and ensure consistency across our implementations. Additionally, these tools helped generate figures of good quality that maintain visual consistency throughout the document.

L. Tools

For code collaboration, we have used GitHub, and our code can be found by following this link [7]. Our code is written in Python, with libraries: matplotlib [8], numpy [9], scikit-learn [10], seaborn [11], pandas [12] and pytorch [13].

III. METHOD

A. FFNN

For all neural network experiments, we use a standard train–validation–test split as provided by Scikit-learn [10]. If not specified otherwise, the data is split into 60% training data, 20% validation data and 20% test data. In addition, we apply feature scaling using our own implementation, ensuring that training, validation and test sets are scaled consistently using statistics computed only from the training data. We scaled our data using z-score normalization. The importance of this is discussed in our previous work [1]. The design matrix is scaled as (for each column)

$$\mathbf{X}_s = \frac{\boldsymbol{X} - \boldsymbol{\mu}_X}{\boldsymbol{\sigma}_X}, \tag{40}$$

where $\mathbf{X}$ is the design matrix, $\boldsymbol{\mu}_X$ is the column-wise mean of $\mathbf{X}$ and $\boldsymbol{\sigma}_X$ the column-wise standard deviation (if the standard deviation $\sigma_X < 10^{-8}$, we set $\sigma_X = 1$). Considering the structure of the design matrix (see below) scaling may seem somewhat unimportant, however, after splitting the data, the scaling helps with generality of the model. The target values $\mathbf{y}$ are centered using mean centering

$$\mathbf{y}_s = \mathbf{y} - \bar{\mathbf{y}}, \tag{41}$$

where $\bar{\mathbf{y}}$ is a vector of length N, with every entry equal to the mean of $\mathbf{y}$, μ_y.

In our work with polynomial regression we (manually) constructed a design matrix with features representing polynomial degree [1]. In this way, the regression methods would find the optimal coefficients for the polynomials to approximate the target (through $\tilde{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$, again see [1]). The models nonlinearity comes from the features in the design matrix (the polynomials). A neural network introduces non-linearity in its hidden layers (primarily through the activation functions), thereby learning its own internal features. That is, the hidden layers act as learned basis functions (doing the same as the polynomial features do in the classical design matrix). This means that a neural network does not need a design matrix in the same sense as a regression problem. The design matrix in a neural network is an input matrix. This input matrix consists of the raw input values, organized as an Nx1 column vector

$$\mathbf{X} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}, \tag{42}$$

where N is the number of data points. The input matrix can have more features if necessary, however, in our Runge function approximation network, we only need one.

We have implemented a flexible feed-forward neural network framework that supports both regression and classification tasks. The design allows network depth, number of neurons, activation functions, regularization type, and optimizer choice to be changed through simple parameter updates. This modular approach makes hyperparameter sweeps efficient and enables reproducibility of experiments across different architectures.

We employ two complementary early stopping mechanisms. First, patience-based early stopping monitors validation loss: if it fails to improve by at least $\delta_{\min} = 10^{-5}$ for consecutive epochs, training terminates and restores the best weights. Second, overfitting trend detection analyzes loss trajectories over a 10-epoch sliding window. If validation loss increases while training loss decreases, and this persists for a significant period, training stops immediately. This dual mechanism ensures we capture the best model before overfitting and terminate early when clear overfitting patterns emerge.

B. Regression on the Runge Function with Optimization and Activation Analysis

We investigated the performance of different optimization algorithms and activation functions in training feed-forward neural networks for a one-dimensional regression task based on the Runge function. The dataset consisted of $N \in 100, 300$ uniformly sampled input points from the interval $[-1, 1]$, with additive Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$ where $\sigma = 0.1$.

Two network architectures were considered: a shallow model with one hidden layer of 50 nodes, and a deeper model with two hidden layers of 100 nodes each. All networks used a linear output layer to predict the scalar function value. We evaluated three activation functions; sigmoid, ReLU, and LeakyReLU, and compared their convergence behavior using full-batch Gradient Descent.

In addition, the adaptive optimizers Adam and RMSprop were evaluated using the sigmoid activation. For each optimizer and network configuration, a learning-rate sweep was carried out to identify the step size η that produced the lowest validation MSE. Training employed an early-stopping criterion that terminated runs upon convergence, or when signs of overfitting or divergence were detected. Each model was trained for a fixed maximum number of epochs, and the evolution of training and validation MSE was monitored throughout. The optimal learning rates corresponding to the lowest validation error are indicated in the respective learning curves.

To ensure robust and comparable results, all experiments were repeated using three different random seeds. This averaging procedure reduces the influence of favorable or unfavorable train–validation splits and random weight initializations. The final reported MSE values were computed as the mean across these three independent runs.

All neural networks were initialized using Xavier initialization [5, p.414], where the weights were sampled from a normal distribution and scaled by $\sqrt{1/(n_{\text{in}} + n_{\text{out}})}$ for each layer. Biases were initialized to zero, which is standard practice as discussed in Section II F. For the output layer, a linear activation was used with weights scaled by $\sqrt{1/n_{\text{in}}}$.

Model performance was compared against an OLS baseline and a PyTorch reference implementation (Figure 1) to ensure consistency and correctness of the custom implementation.

C. Network Complexity

In this part we expand on the analysis above. Now we want to investigate how the three activation functions performance changes if we extend the network. We consider the same data-set as before with $N \in 300$ uniformly sample points. Here, however, the train-validation-test split is set to 70%, 15% and 15% respectively. We extend the neuron grid to $[50, 100, 150, 200, 250]$, and the hidden layer grid to $[1, 2, 3, 4]$. We then chose to estimate the appropriate learning rates for the more complex models, using the same anchor points as in the previous section: $[50, 100]$ neurons and $[1, 2]$ hidden layers. The reason for this is that we wanted to see how the different activation functions respond to a suboptimal initialization of learning rate - inducing stress on the gradients in higher model complexities. We therefore ran the same search for the learning rates as before, with 500 epochs of training picking the best learning rate in the range

$\eta \in [10^{-5}, 10^{0}]$. Using the best learning rate, we then trained all the models on a fixed 500 number of epochs, keeping track of the layer-wise gradients during learning, for each of the configurations. We ran the experiment ten times over different seeds to reduce the random effects of the initial training-validation-test split, and then an additional five trials over the training loops, to reduce the randomness involved in training - like the randomness involved in batch-shuffling.

Further, we wanted to test whether batch normalization can improve stability in the high-complexity regime. In Section (II C 6) we discussed how the three activation functions have different properties when it comes to gradient stability. Batch normalization can help addressing the vanishing/exploding gradient problem. We therefore implemented batch normalization as described in Section (III I 4), and reran the full experiment.

To get an idea of how the gradients behave in the different architectures, both with and without batch-normalization, we need to see the absolute value of how the parameters change the cost function. Without batch-normalization the parameters are the weights and biases. With batch-normalization we have the additional parameters γ and β (see Section III I 4). This gradient quantity is given denoted by $\Delta^{(l)}$ as

$$\Delta_i^{(l)} = \left(\sum_j \left(\frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)}} \right)^2 + \left(\frac{\partial \mathcal{L}}{\partial b_i^{(l)}} \right)^2 + \mathbf{1}_{\mathrm{BN}} \left[\left(\frac{\partial \mathcal{L}}{\partial \gamma_i^{(l)}} \right)^2 + \left(\frac{\partial \mathcal{L}}{\partial \beta_i^{(l)}} \right)^2 \right] \right)^{1/2}, \tag{43}$$

where W_{ij} is the weights connecting this layers neurons i with last layers neurons j, $\mathbf{1}_{BN}$ is a vector of zeros when we consider architecture without batch-normalization, and a vector of ones if we do consider batch-normalization.

D. L_1 and L_2 regularization

In this section, we evaluate the performance of L_1 and L_2 regularization on the regression task using learning curves. To select optimal hyperparameters, we performed a grid search over learning rates (η) and regularization strengths (λ), both sampled logarithmically from the range $[10^{-5}, 10^{-1}]$ with 5 values each. For each combination of hyperparameters, we trained a neural network with two hidden layers of 100 nodes each using sigmoid activation and RMSprop optimization for 300 epochs with early stopping (patience of 150 epochs). The model was trained three times with different random data splits (train-test-split at 60%, 20% and 20%), and the configuration that yielded the lowest average validation MSE was selected. Using these optimal hyperparameters, we trained final models and presented their learning curves showing both training and validation MSE over 10,000 epochs. The neural network results with L_1 and L_2 regularization were compared against Lasso and Ridge regression baselines, respectively.

E. Neural Network Classification on MNIST

We applied fully connected feedforward neural networks to the MNIST digit classification task using a two-stage methodology. The MNIST dataset (70,000 grayscale 28×28 images) was split into training (60%), validation (20%), and test (20%) sets. Pixel values were normalized to [0, 1] and standardized; labels were one-hot encoded.

To efficiently identify optimal hyperparameters, we first conducted an extensive grid search on a randomly selected subset of 10,000 images, evaluating 6 network architectures (1-2 hidden layers with 100, 150, or 200 nodes per layer) with sigmoid, ReLU and LeakyReLU activations, with softmax activation for the last layer (since this is a multiclass classification problem, see Section II C). For each architecture, we systematically varied regularization parameters (λ) and learning rates (η), each spanning $[10^{-5}, 10^{-1}]$ in 5 logarithmically-spaced values, using both Adam and RMSprop optimizers. Models were trained with mini-batch gradient descent (batch size 100) for up to 100 epochs with early stopping (patience of 10 epochs).

Using the optimal hyperparameters identified for each architecture, we then retrained all configurations on the complete dataset, employing a larger batch size (128) to better exploit the increased data volume while maintaining identical early stopping criteria. Performance was evaluated using classification accuracy on the validation set, with the best model further analyzed via confusion matrix. We compared our results against multinomial logistic regression from scikit-learn.

IV. RESULTS

A. Optimizers

This section compares the performance of optimization algorithms with sigmoid activation and activation functions with full batch Gradient Descent for the Runge function regression task (the methods here coincide with those explained in Section III B). Each figure use its optimal η (values noted in the figures), the results are presented through learning curves of the Mean Squared Error (MSE).

In Figure 1 (N=300) and Figure 2 (N=100), we show the averaged learning curves over three different seeds, for two different architecture and for two different optimizers, namely, Adam and RMSprop. The networks use the sigmoid activation function. Each plot includes the mean training MSE and validation MSE from our implementation, together with the corresponding OLS validation

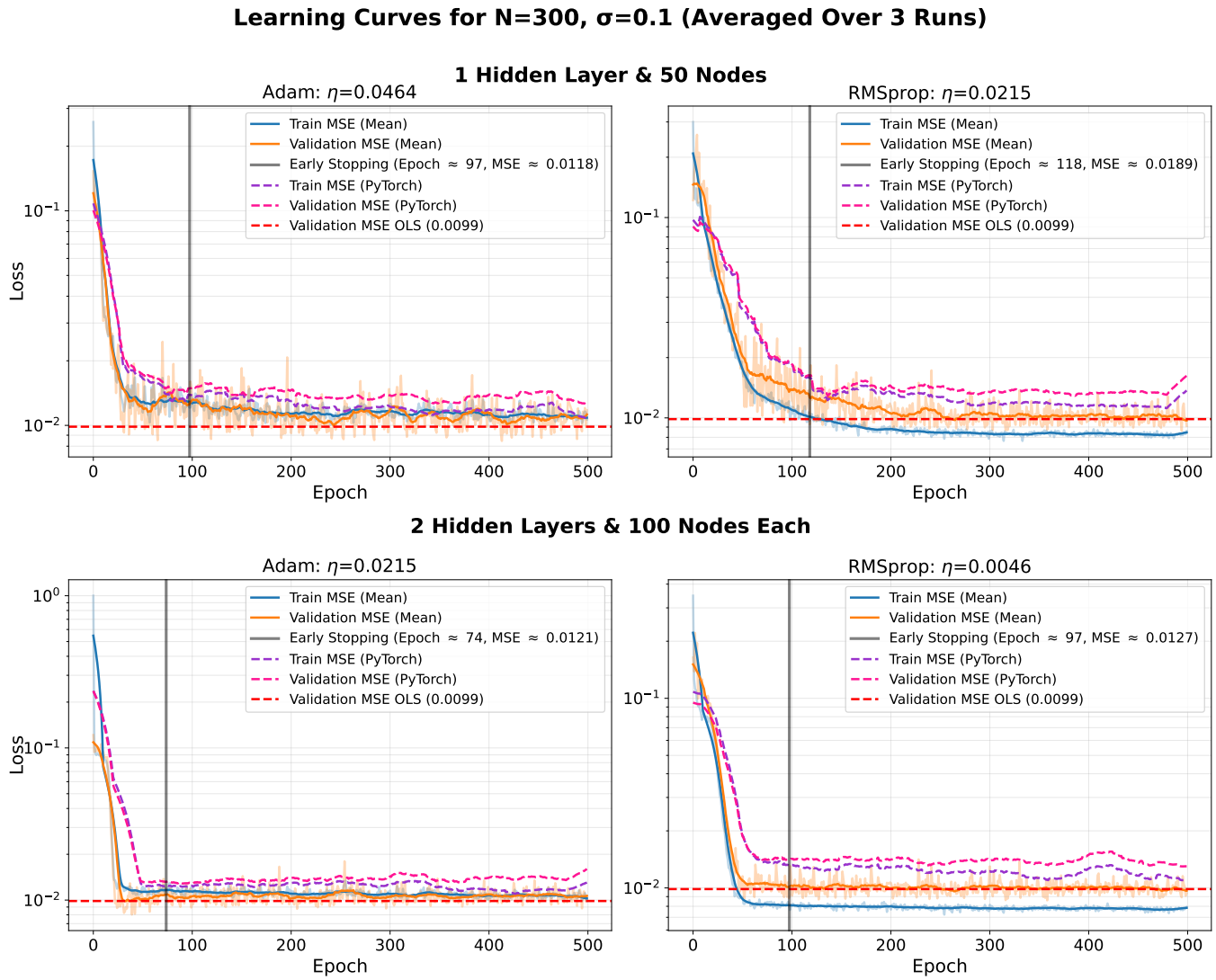

Figure 1. MSE learning curves for feedforward neural networks trained on the one-dimensional Runge function regression task using the sigmoid activation function. The dataset contains $N = 300$ uniformly sampled points with Gaussian noise $\sigma = 0.1$. Two architectures are shown: one hidden layer with 50 nodes and two hidden layers with 100 nodes. Results are compared for the Adam and RMSprop optimizers, both trained without regularization. The MSE's are averaged over 3 runs. The MSE of the neural network is compared to MSE calculated by PyTorch, as well as the analytical OLS solution.

MSE (as in our previous work [1]) baseline. We additionally mark the average early–stopping point, illustrating the approximate epoch where the model begins to overfit. For verification, we also include PyTorch–computed learning curves, which serve as an external reference confirming that our custom neural network implementation produces consistent results.

In the Gradient Descent comparison in Figures 3 (1×50) and 4 (2×100), we plot the train MSE and validation MSE of a neural network using full batch gradient descent, and using different activation functions, namely, sigmoid, ReLU, and LeakyReLU. This is all plotted together with a validation MSE for OLS as a baseline. Additionally in Figure 3, we add an early stopping marker.

We performed a learning–rate analysis to determine the learning rate that yields the lowest validation MSE. The optimal value of η for each configuration is indicated above the corresponding learning curve in the figures.

B. Architecture

We analyzed how the network complexity affects the performance of the various activation functions. We performed the analysis described in III C. The results of the analysis without batch-normalization is presented in figures 5, 6 and 7.

Figure 5 shows number of hidden layers along the y-axis and number of neuron along the x-axis. Columns 1, 2 and 3 correspond to neural networks with activation

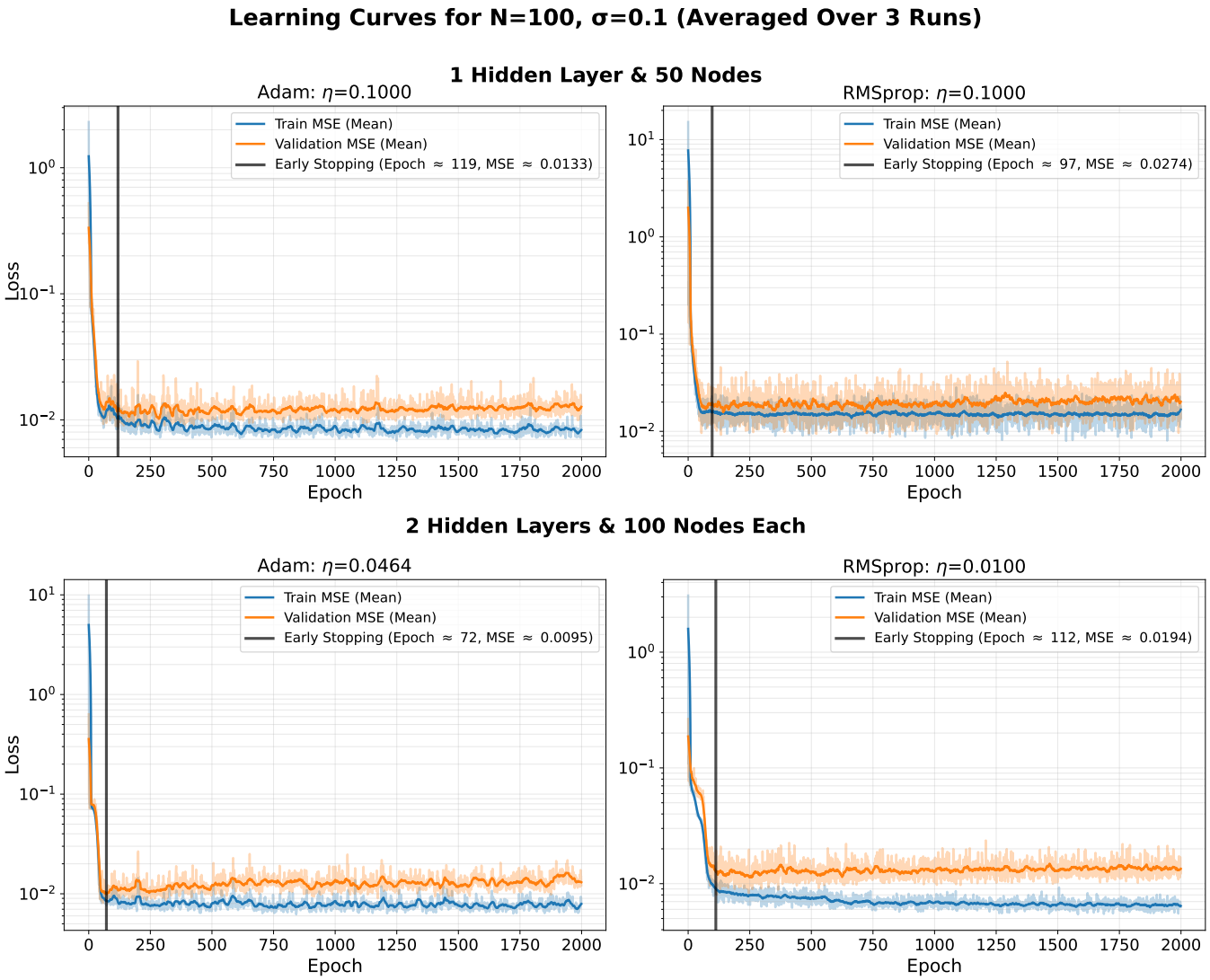

Figure 2. MSE learning curves for feedforward neural networks trained on the one-dimensional Runge function regression task using the sigmoid activation function. The dataset contains $N = 100$ uniformly sampled points with Gaussian noise $\sigma = 0.1$. Two architectures are shown: one hidden layer with 50 nodes and two hidden layers with 100 nodes. Results are compared for the Adam and RMSprop optimizers, both trained without regularization. The MSE's are averaged over 3 runs.

functions sigmoid, ReLU and Leaky ReLU respectively. Rows 1, 2 and 3 show heatmaps of MSE, ratio of the test and train MSE and the ratio between highest and lowest gradient absolute value of $\Delta^{(L)}$ (from Equation 43) at the final epoch of simulation (i.e. epoch 500). The fractions in the Leaky ReLU MSE heatmap (below the MSE value in the cells) denote the number of seed-dependent simulations that result in MSE's lower than 1. If no fraction is noted, all simulations had MSE result lower than 1. This limit is arbitrarily chosen to deem a model as failed. Failed models are excluded from the MSE, test-train ratio and gradient heatmap calculations. The reason for this is to not saturate the heatmaps for models that blow up, and therefore being able to see the underlying pattern. They are however, included in the next two figures.

Figure 6 shows number of epochs along the x-axis. Columns correspond to activation function, like in figure 5. Top row shows graphs of the MSE for the training set, validation set and average across seeds and trails against epochs. The irreducible error (at $\sigma^2 = 0.01$) is present as the red dashed line. The bottom row shows the fraction of alive neurons (the neuron is considered dead if the norm of the gradient of the corresponding neuron is $< 10^{-10}$). The fraction of living neurons is shown at all layers (excluding input layer).

Figure 7 shows again epochs along the x-axis. Columns are again activations functions like the previous two figures. The rows correspond to the depth (number of hidden layers) with one hidden layer at the top, increasing downwards. Each graph shows the absolute value of the gradient $\delta^{(l)}$. The graphs are shown for every hidden

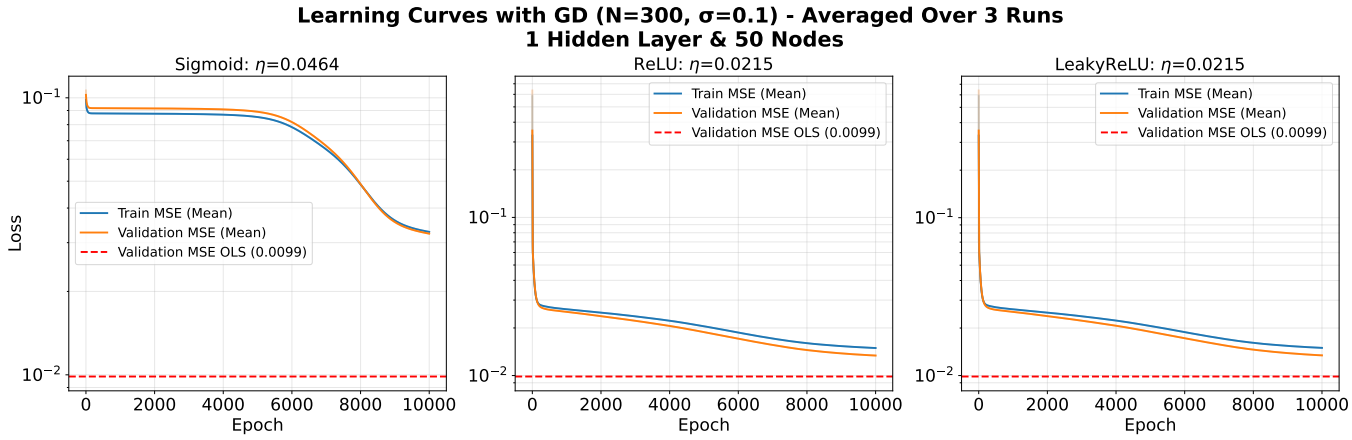

Figure 3. Comparison between activation functions sigmoid, ReLU and LeakyReLU with full batch Gradient Descent, N=300 datapoints and noise $\sigma = 0.1$ without regularization, for 1 hidden layer with 50 nodes.

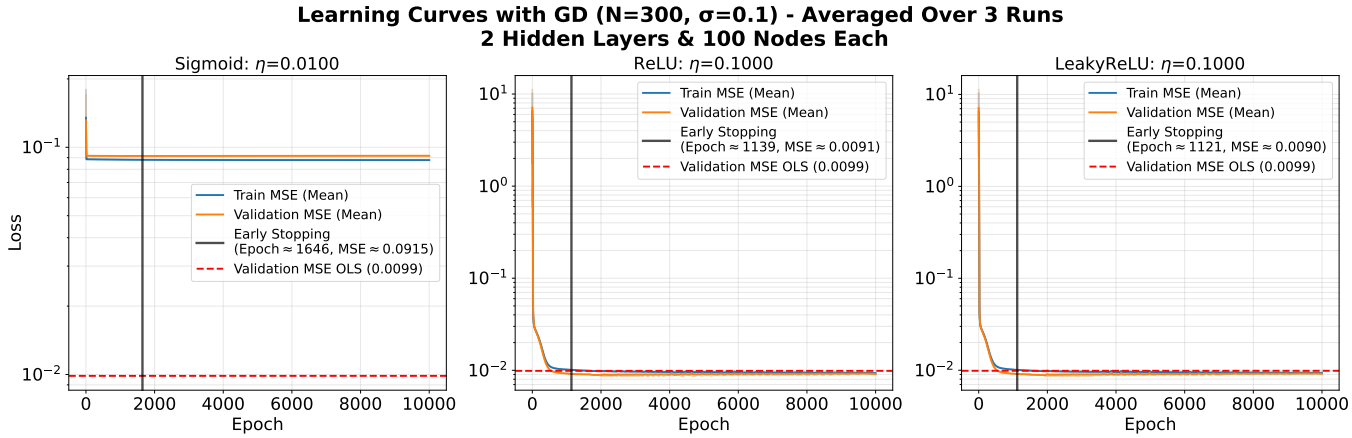

Figure 4. Comparison between activation function sigmoid, ReLU and LeakyReLU with full batch Gradient Descent, N=300 datapoints and noise $\sigma = 0.1$ without regularization, for 2 hidden layers with 100 nodes.

layer l as well as the output layer. Corresponding figures with batch-normalization are presented in figures 8, 9 and 10.

C. Regularization

Figure 11 and Figure 12 present the learning curves comparing neural network performance with L_1 and L_2 regularization against their classical regression counterparts, Lasso and Ridge regression, evaluated for noise levels $\sigma = 0.1$ and $\sigma = 0.3$, respectively (method follows Section III D). The results demonstrate how regularization affects network training dynamics and final performance across different noise conditions.

D. Classification

This section shows the results from using neural networks to solve the classification problem of the MNIST's handwritten digits, using the method described in Section I. To identify the optimal neural network configuration for MNIST digit classification, we conducted a comprehensive hyperparameter search across multiple architectures, activation functions, and optimizers. Figure 13 illustrates the validation accuracy landscape across the hyperparameter space for each architecture using ReLU activation with RMSprop optimization, revealing distinct optimal regions for different network depths and widths. Figure 14 compares training and validation accuracies across architectures, demonstrating that deeper networks with more nodes generally achieve higher performance while maintaining good generalization. The quantitative results in Table I confirm that ReLU and LeakyReLU activations consistently outperform sigmoid across both optimizers, with the best configuration which

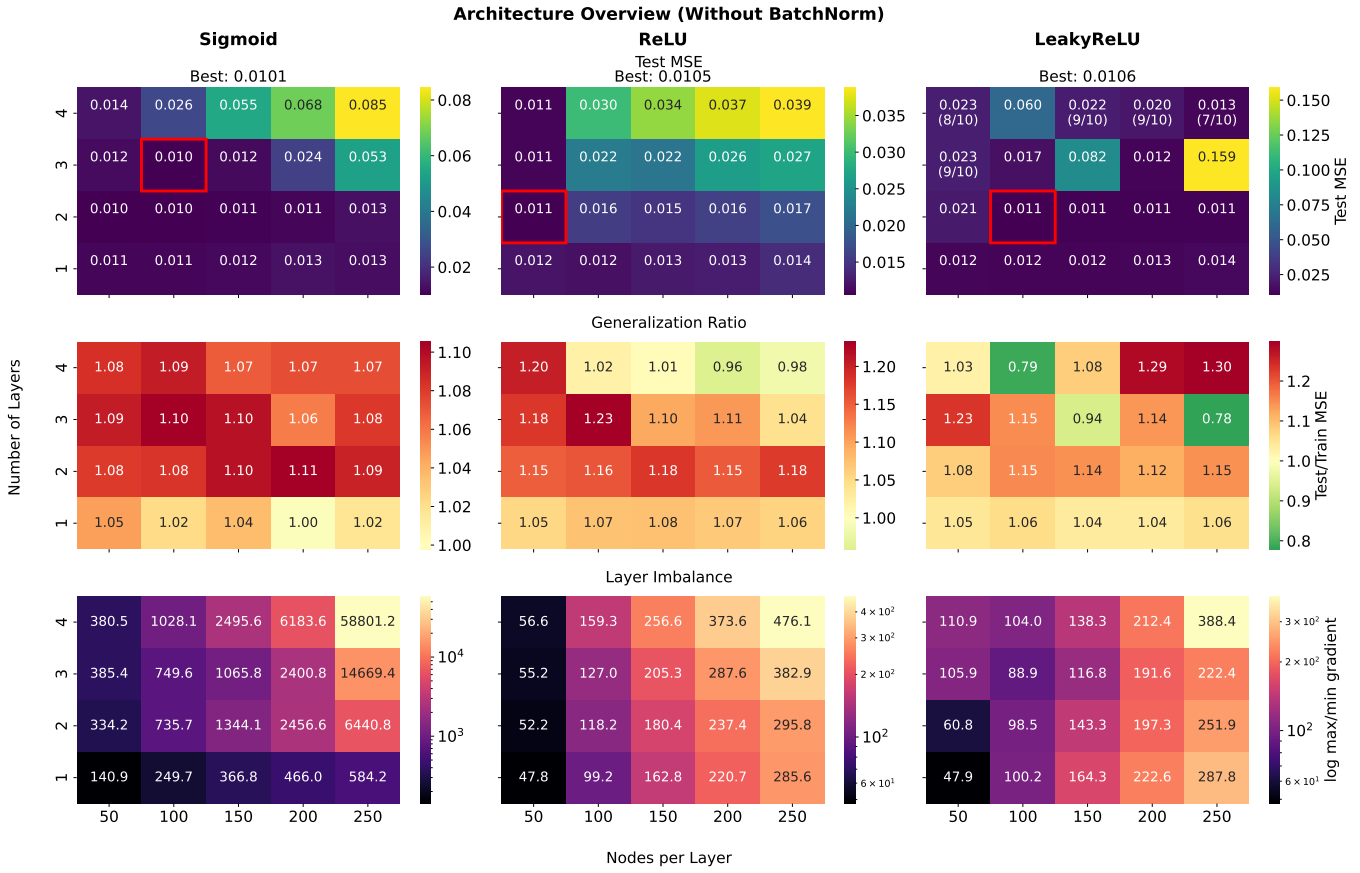

Figure 5. Overview of neural network performance across architectures and activation functions without batch-normalization. The heatmaps show MSE, test/train MSE ratio, and the ratios of the max/min ratios of the averaged gradient norms per layer for feedforward neural networks trained on the Runge function for 500 epochs. Columns correspond to activation functions (sigmoid, ReLU, Leaky ReLU) and rows to performance metrics. The x-axis shows the number of neurons per layer and the y-axis the number of hidden layers. Models with MSE > 1 are excluded from the visualizations.

is a 2-layer network with 200 nodes using ReLU activation and RMSprop, achieving 97.96% validation accuracy and 97.79% test accuracy. To assess the model's per-class performance, Figure 15 presents the confusion matrix for this optimal configuration, revealing strong classification across all digits with minimal confusion between visually similar classes.

V. DISCUSSION

A. Comparing overfitting for different optimizers without regularization

Figures 1 (N=300) and 2 (N=100) demonstrate that the adaptive optimizers, Adam and RMSprop, converge rapidly and achieve comparable, low validation MSE across both the 1×50 and 2×100 network architectures. These results generally surpass the OLS baseline, and the agreement with the PyTorch reference curves confirms that our implementation is correct.

The Gradient Descent activation comparison in figures 3 and 4 shows that ReLU and LeakyReLU activations reach convergence significantly faster than sigmoid, especially in the deeper network. The sigmoid networks clearly train slower.

In Figure 1, we compare how the models behave across the two optimizers (Adam and RMSprop) and the two architectures (1 hidden layer with 50 nodes, and 2 hidden layers with 100 nodes each). The larger model consistently achieves lower validation error, and with $N = 300$ datapoints the model complexity is still not high enough to trigger severe overfitting before early stopping.

We also observe that Adam and RMSprop behave differently with respect to learning rate sensitivity. For both architectures, Adam reaches a lower validation loss than RMSprop, and typically does so slightly earlier. Moreover, the optimal learning rate differs substantially: RMSprop obtains its lowest validation MSE at a surprisingly small $\eta \approx 0.0046$. A small optimal global η for RMSprop is consistent with the fact that its per-parameter rescaling increases the effective step size internally, requiring a reduced external step size for stability. Adam appears more robust in this setting and does not require such

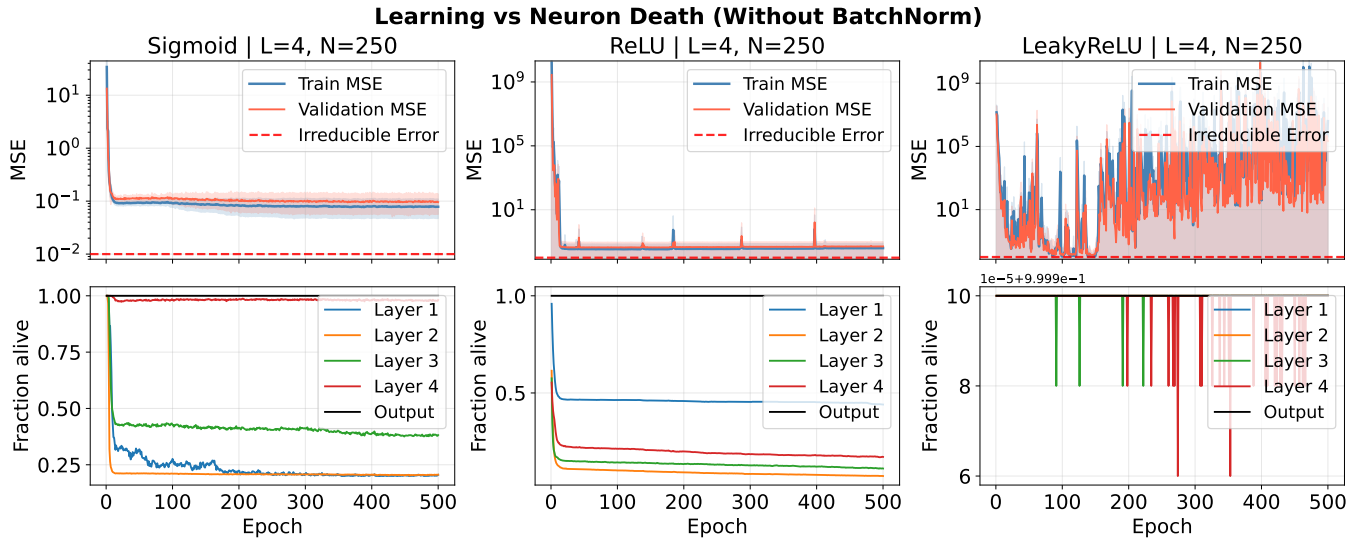

Figure 6. Evolution of MSE and neuron activity during learning for different activation functions without batch-normalization. The x-axis shows the number of epochs, while the columns correspond to activation functions (sigmoid, ReLU, and Leaky ReLU). The top row displays MSE curves for the training and validation sets, averaged over all seeds and trials. The red dashed line indicates the irreducible error corresponding to the data noise level ($\sigma^2 = 0.01$). The bottom row shows the fraction of active ("alive") neurons in each layer during training, where a neuron is considered dead if the norm of its gradient is below 10^{-10}. Results are shown for all hidden layers and output layer (input layer excluded).

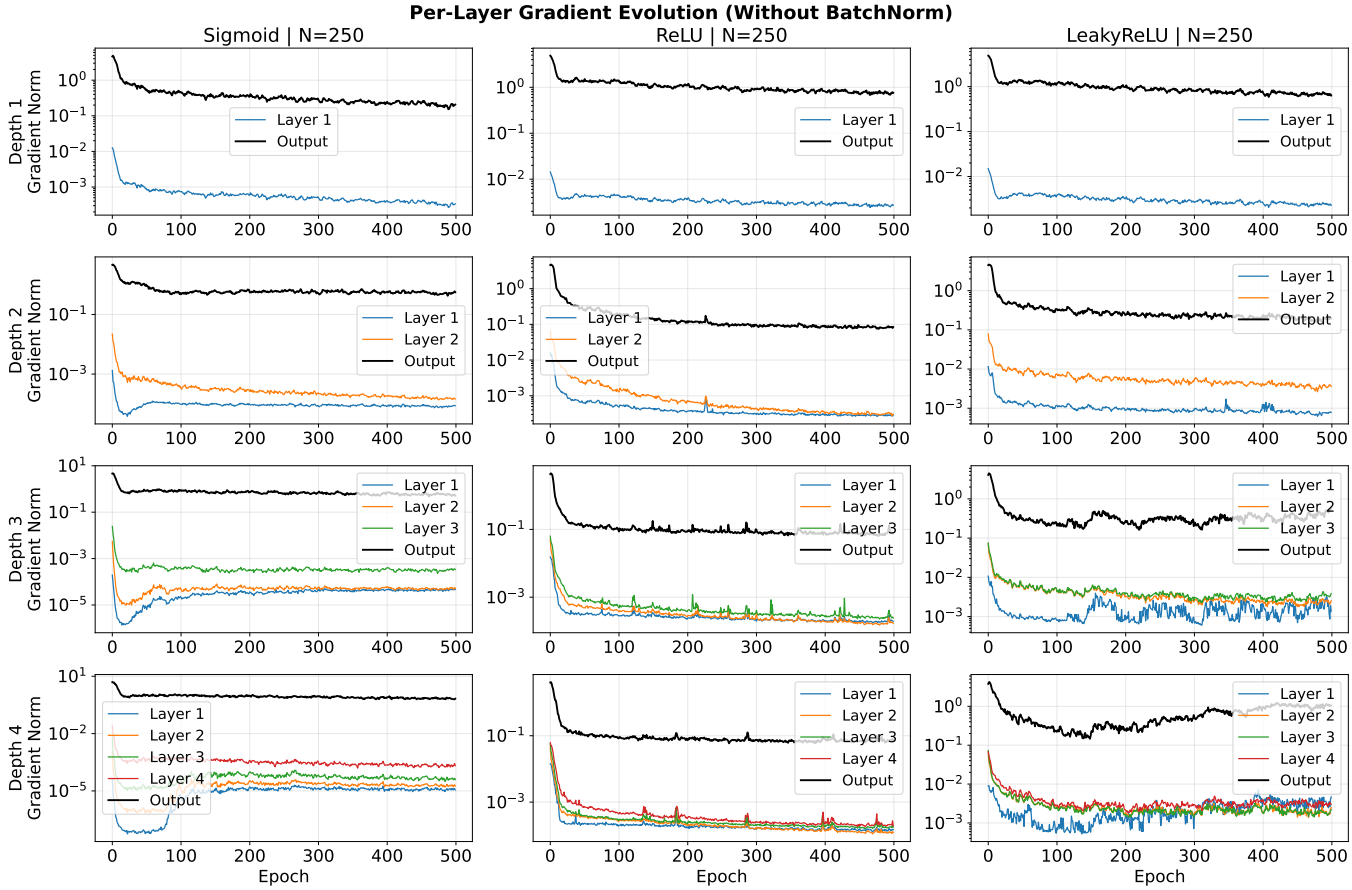

Figure 7. Epoch-wise gradient magnitude per layer (log scale); consistent colors highlight hidden layers and the output layer.

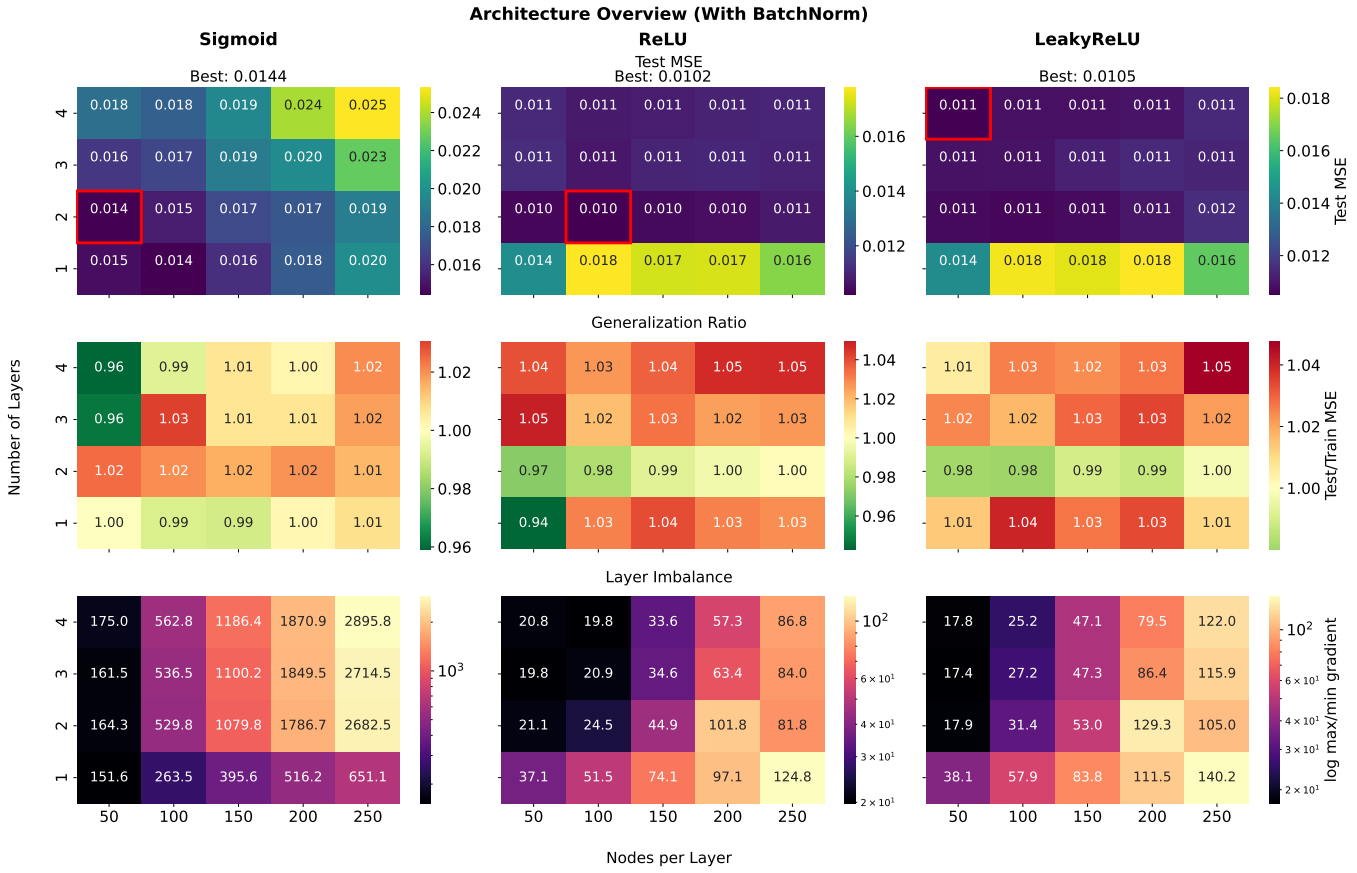

Figure 8. Overview of neural network performance across architectures and activation functions with batch-normalization. The heatmaps show MSE, test/train MSE ratio, and the ratios of the max/min ratios of the averaged gradient norms per layer for feedforward neural networks trained on the Runge function for 500 epochs. Columns correspond to activation functions (sigmoid, ReLU, Leaky ReLU) and rows to performance metrics. The x-axis shows the number of neurons per layer and the y-axis the number of hidden layers. Models with MSE > 1 are excluded from the visualizations.

Table I. Performance comparison of neural network architectures on the MNIST dataset using optimal hyperparameters (λ, η) identified through grid search. Results show the best-performing architecture for each activation function and optimizer combination, evaluated on the full dataset with a 60/20/20 train/validation/test split.

Architecture	Activation	Optimizer	λ	η	Train	Validation	Test
2L, 200N	sigmoid	Adam	10^{-5}	10^{-1}	0.9882	0.9596	0.9594
2L, 200N	ReLU	Adam	10^{-3}	10^{-3}	0.9971	0.9778	0.9737
2L, 200N	LeakyReLU	Adam	10^{-3}	10^{-3}	0.9992	0.9772	0.9751
2L, 150N	sigmoid	RMSprop	10^{-4}	10^{-3}	0.9997	0.9744	0.9733
2L, 200N	ReLU	RMSprop	10^{-3}	10^{-3}	0.9983	**0.9796**	**0.9779**
2L, 200N	LeakyReLU	RMSprop	10^{-3}	10^{-3}	0.9960	0.9781	0.9744

aggressive tuning of η.

We also note that the OLS baseline is quite strong. This can partly be due to a favorable train–validation split, but it is also consistent with the fact that, for moderate polynomial degrees, the Runge function is well approximated. Although Runge's phenomenon warns that high–degree global polynomials oscillate near the domain boundaries, our polynomial degree is not in that extreme regime, therefore classical polynomial regression performs reasonably well. The neural networks consequently have a limited margin to improve much beyond OLS.

Across all configurations, full batch gradient descent performs worse than Adam and RMSprop. It does not converge within 10,000 epochs, as seen in Figure 3 and Figure 4, demonstrating the practical benefit of adaptive optimizers for this task.

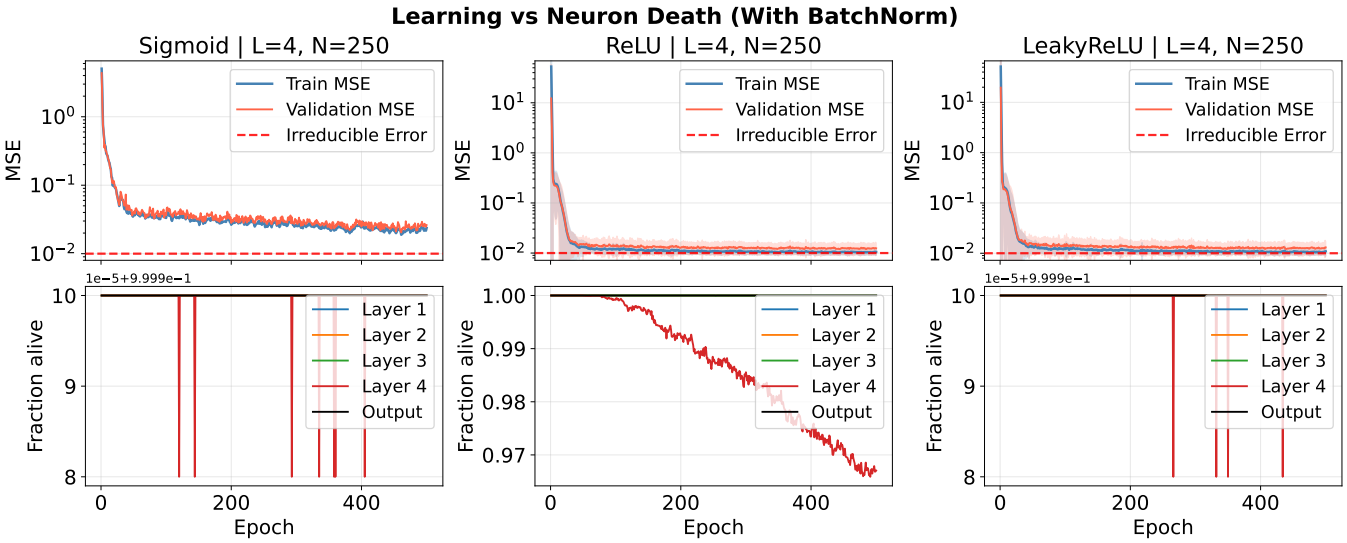

Figure 9. Evolution of MSE and neuron activity during learning for different activation functions with batch-normalization. The x-axis shows the number of epochs, while the columns correspond to activation functions (sigmoid, ReLU, and Leaky ReLU). The top row displays MSE curves for the training and validation sets, averaged over all seeds and trials. The red dashed line indicates the irreducible error corresponding to the data noise level ($\sigma^2 = 0.01$). The bottom row shows the fraction of active ("alive") neurons in each layer during training, where a neuron is considered dead if the norm of its gradient is below 10^{-10}. Results are shown for all hidden layers and output layer (input layer excluded).

B. Comparing neural network architecture

As previously mentioned we considered two architectures: one hidden layer with 50 nodes each, and two hidden layers with 100 nodes each. In general, networks with higher capacity (more layers and more nodes) have a higher tendency to overfit compared to simpler architectures. This behavior is visible in Figure 2.

However, we also observe in Figure 2 that the deeper model performs better, especially when using early stopping. This can occur because deeper networks may fit the underlying function more accurately when the data supports it, and because the optimization dynamics can vary significantly between runs. In some configurations, the simpler network may underfit (too low capacity), while the larger network can overfit, but when using early stopping, perform better overall.

C. Performance of Full Batch Gradient Descent

Full batch gradient descent exhibits severe performance limitations that depend critically on both activation function choice and network depth. Our experiments reveal fundamental differences in how sigmoid, ReLU, and Leaky ReLU activation functions interact with the gradient descent optimization algorithm.

The sigmoid activation function consistently fails with full batch gradient descent. For the shallow network (1 hidden layer, 50 nodes), sigmoid shows delayed convergence: the MSE remains nearly flat for approximately 6,000 epochs before beginning a very slow decrease (figure 3, left panel), never reaching the OLS baseline of 0.0099. For the deeper network (2 hidden layers, 100 nodes), the failure is complete: both training and validation MSE remain constant at approximately 0.09 throughout 10,000 epochs (figure 4, left panel), showing no improvement whatsoever.

This behavior stems from the vanishing gradient problem. The sigmoid's derivative $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ has a maximum value of 0.25, and approaches zero for large $|x|$. During backpropagation, gradients are multiplied through layers. In our 2-layer network, a gradient of 1.0 shrinks to approximately $0.25^2 = 0.0625$. These rapidly shrinking gradients make weight updates negligible, stopping learning. The optimal learning rate differs dramatically between architectures: $\eta = 0.0463$ for the shallow network versus $\eta = 0.001$ for the deep network. Even with this drastic reduction, the deep network fails completely.

In contrast, ReLU and Leaky ReLU enable rapid convergence. As shown in figures 3 and 4, both achieve rapid MSE decrease within a few hundred epochs, reaching the OLS baseline of 0.0099 and stabilizing. For the deep network, ReLU achieves early stopping at epoch 1139 with MSE = 0.0091, while Leaky ReLU stops at epoch 1121 with MSE = 0.0090, both vastly outperforming sigmoid.

Their success stems from gradient properties (Sections II C 2 and II C 3): for positive inputs, both have a constant derivative of 1, eliminating the gradient shrinking problem. Their nearly identical performance suggests the dying ReLU problem is not significant for this task. Both accommodate much larger learning rates ($\eta = 0.1$ for the

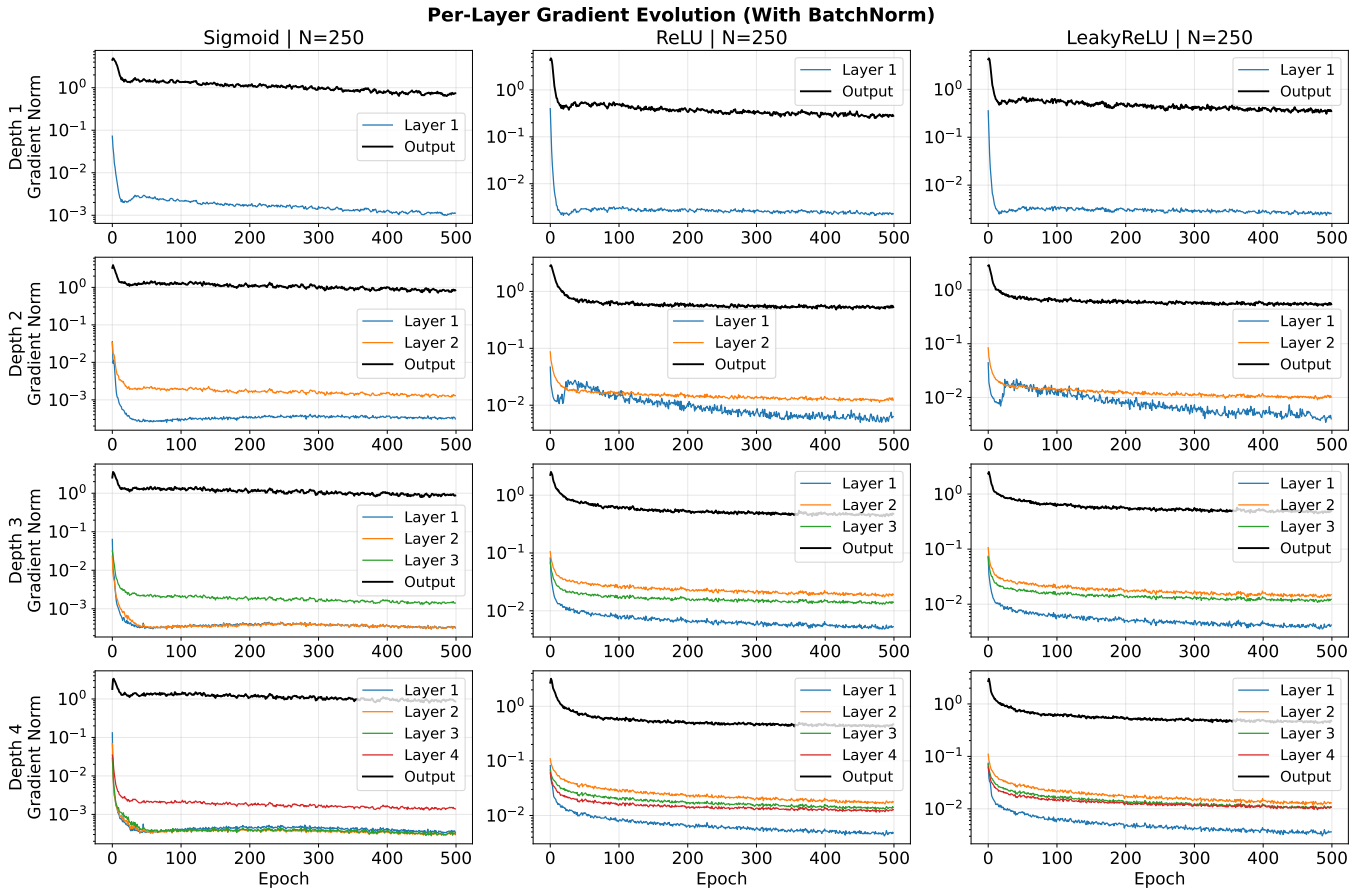

Figure 10. Evolution of the absolute gradients $\boldsymbol{\delta}^{(l)}$ across training epochs. Columns correspond to different activation functions, while rows indicate network depth, with one hidden layer at the top increasing downward. Each plot shows the gradient for every hidden and output layer.

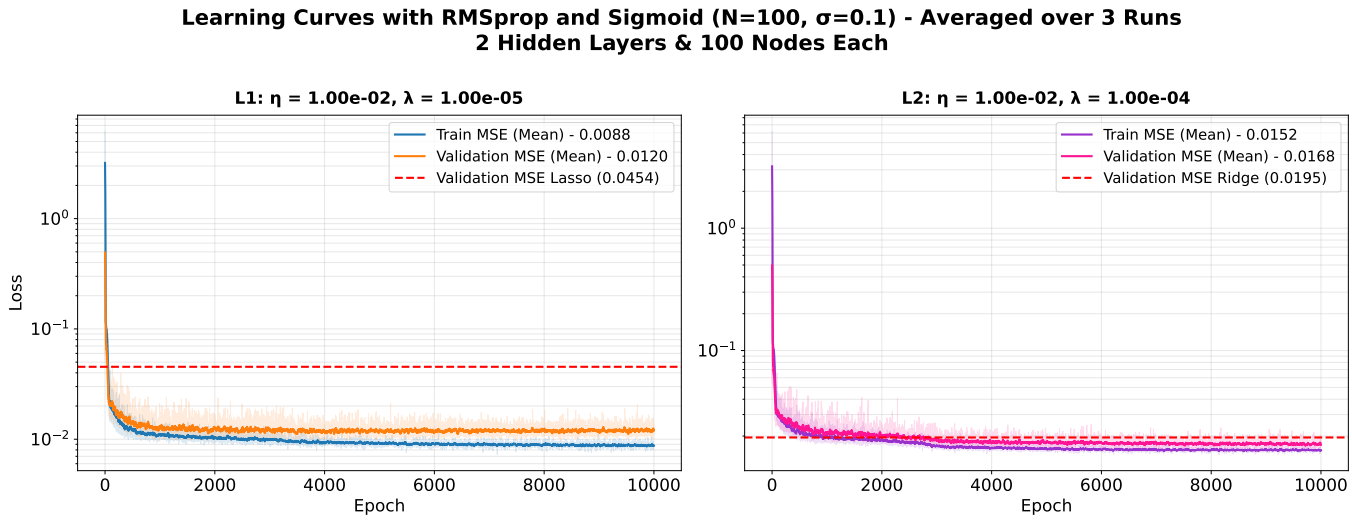

Figure 11. Comparison of neural network performance with L_1 and L_2 regularization against Lasso (left) and Ridge (right) regression respectively. Both networks use two hidden layers with 100 nodes each, trained with RMSprop and sigmoid activation on $N = 100$ data points with noise level $\sigma = 0.1$. The Lasso baseline uses $\lambda = 0.01$ and a polynomial degree of 15. The left panel shows L_1 regularization, while the right panel shows L_2 regularization, with validation MSE tracked over $10,000$ epochs.

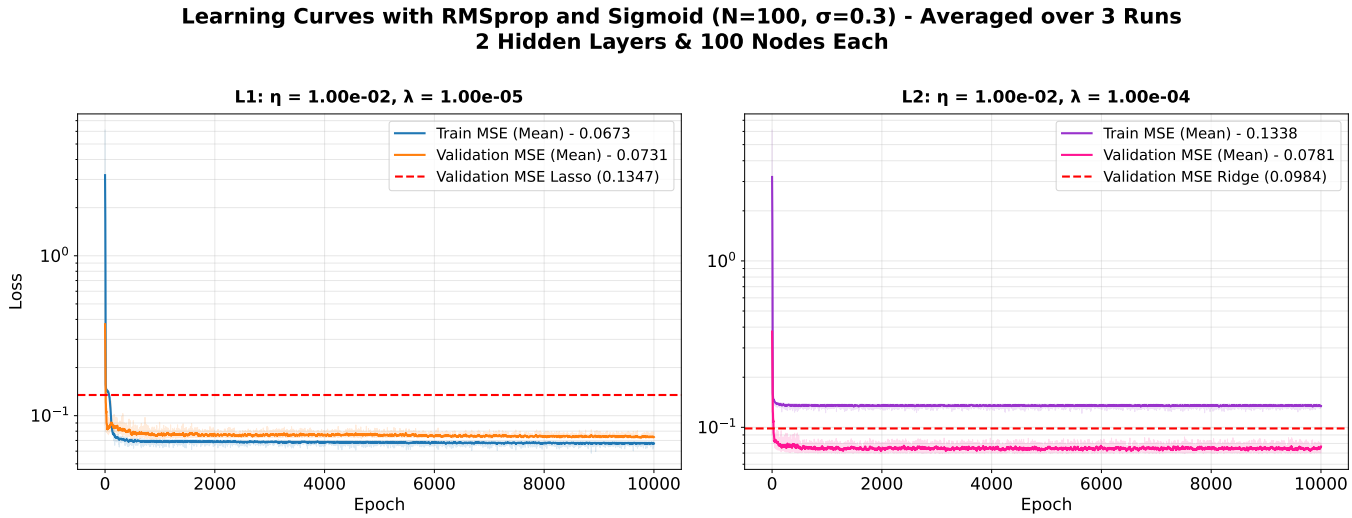

Figure 12. Comparison of neural network performance with L_1 and L_2 regularization against Lasso (left) and Ridge (right) regression respectively. Both networks use two hidden layers with 100 nodes each, trained with RMSprop and sigmoid activation on $N = 100$ data points with noise level $\sigma = 0.3$. The Lasso baseline uses $\lambda = 0.01$ and a polynomial degree of 15. The left panel shows L_1 regularization, while the right panel shows L_2 regularization, with validation MSE tracked over $10,000$ epochs.

deep network versus sigmoid's $\eta = 0.001$), reflecting the stability of non-saturating activations.

Network depth amplifies these differences. For sigmoid, increasing depth transforms slow convergence into complete failure. For ReLU and Leaky ReLU, depth has a positive impact. The deeper network converges fast and stable, while the shallow network fails. This confirms that gradients get multiplied through each layer: sigmoid's bounded derivative makes it incompatible with deep architectures under full batch gradient descent.

Despite ReLU variants avoiding vanishing gradients, full batch gradient descent remains inferior to stochastic methods. Stochastic gradient descent variants (RMSprop, Adam) should be strongly preferred, providing better generalization, faster convergence, and robust performance across activation functions and architectures.

D. Activation Functions and Network Depth

In section IV B, we present the results of our analysis of how network complexity affects the performance of the various activation functions. From the test-MSE in figure 5 we see that all models perform well on low model-complexities - typically scoring in the range of $MSE \in [0.010, 0.017]$. Both the Sigmoid and the ReLU activation functions show a decrease in performance as model-complexity increase. At first glance this might look like classic overfitting, but looking at the ratios between test and training MSE remain close to one accross the models. This is indicative of training failing, and stabilizing above the irreduible error of $\sigma^2 = 0.01$ - the noise floor. We also see that LeakyReLU displays a more unstable performance distribution. In the high-complexity regime above three hidden layers, LeakyReLU completely fails to train ($MSE > 1$) eight times all together. Whereas the models using Sigmoid and ReLU always managed to converge. This, a bit unexpectedly, seem to indicate that LeakyReLU is more unstable than the other activation functions at higher complexities. From II C 6, we know that LeakyReLU, normally resists vanishing gradients and dying neurons. However, LeakyReLU is not immune to exploding gradients. With a bad initialization of weights and biases, and with a too high learning rate, LeakyReLU can experience unstable gradients. If the gradients blows up erratically, then this might explain the scattered performance of LeakyReLU. Looking at the results 6 and 7, we definitely see this. The gradients of LeakyReLU oscillate wildly, and has an has a very unstable learning curve compared to Sigmoid and ReLU. Looking the gradient evolution of sigmoid in 7 and the fraction of neurons alive in ReLU in figure 6, it becomes clear that these models too have issues in the high-complexity regime. Sigmoid displays the characteristic issue of vanishing gradients at higher depths with 250 neurons. Looking at the heatmap of layer imbalance in 5, we see that the trend is similar across the model-complexities. The heatmap clearly shows that the contribution to learning from various layers in the network varies by orders of magnitude. ReLU on the other hand is displaying classic death of neurons in figure 6. All of these effects kick in at higher complexeties, explaining the degredation in performance.

In figures 8, 9 and 10 we can the some of the same trends as above. The differences are that the batch-normalization brings more stability to the heatmaps, as in lowering the max/min gradient values. We can also see that the Leaky ReLU MSE does not behave as wildly

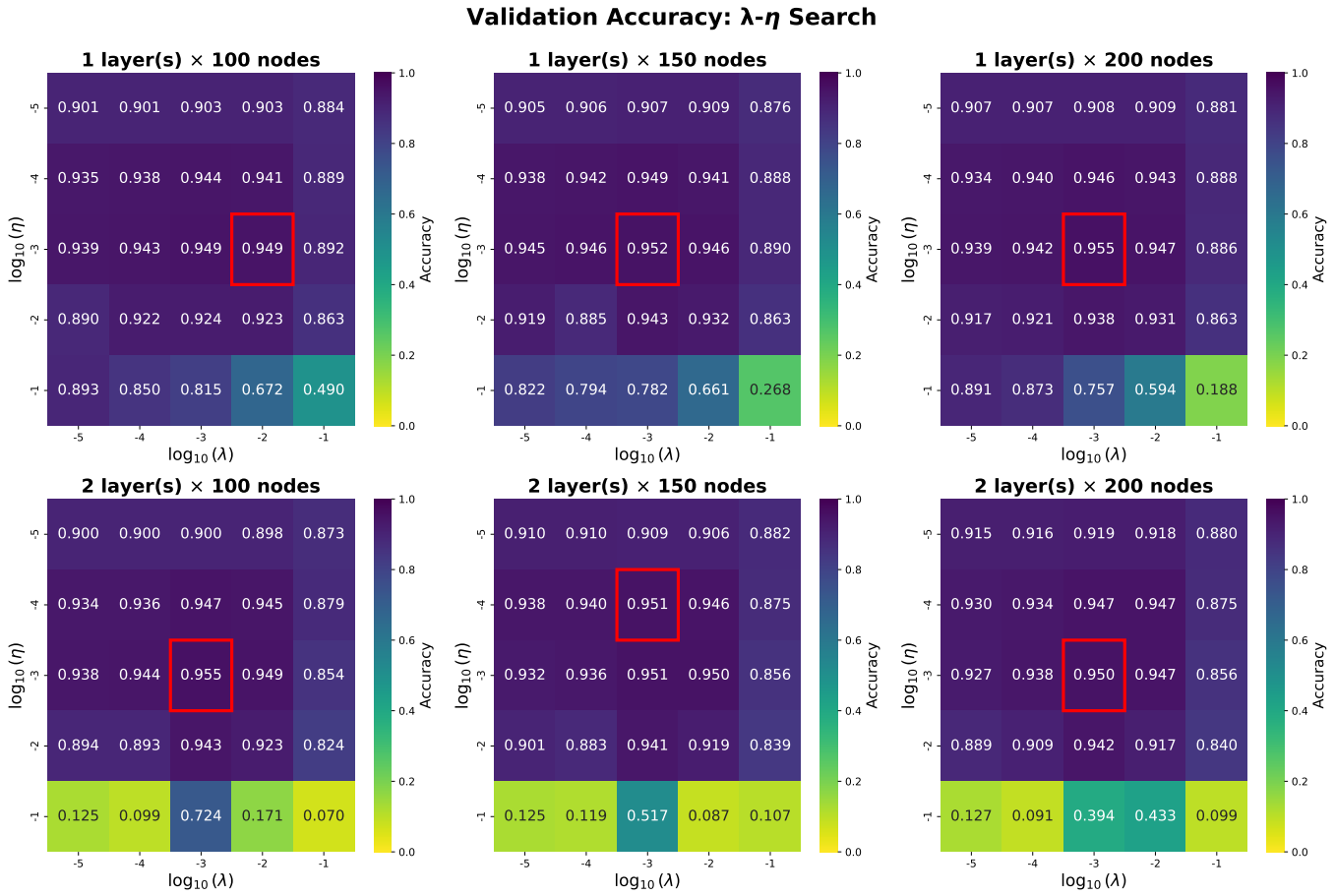

Figure 13. Validation accuracy heatmaps for classification on MNIST dataset across the λ-η hyperparameter space for six network architectures (1-2 hidden layers; 100, 150, 200 nodes per layer) using ReLU activation and RMSprop optimizer. Each subplot shows the grid search results for one architecture, with the optimal configuration marked by a red box. The color intensity indicates validation accuracy, revealing that moderate regularization ($\lambda \approx 10^{-3}$) and learning rates ($\eta \approx 10^{-3}$) yield the best performance across architectures.

as in the non batch-normalization case, again showing signs of improving stability. We can also see in figure 10 that the ReLU activation function comes with the dying neuron problem.

E. Regularization Effects

A fundamental difference between linear regression models and feedforward neural networks is their feature representation. Linear models rely on a fixed feature space, in our case, a Vandermonde matrix of polynomial basis functions [1], that is manually specified and remains unchanged during training. In contrast, FFNNs learn internal representations automatically through their hidden layers, enabling them to construct non-linear transformations of the input rather than relying on predefined polynomial terms. This automatic feature learning often allows FFNNs to model highly non-linear target functions more effectively, provided sufficient data and adequate regularization are available.

1. *Regularization Stabilizes Training*

As shown in figure 11, both L_1 and L_2 regularization produce substantially more stable learning curves compared to the unregularized network (figure 2, bottom-right panel). Without regularization, the model with two hidden layers of 100 nodes begins to overfit after approximately 250 epochs, achieving a validation MSE of 0.0194 with early stopping at epoch 122. With L_1 regularization, the network achieves a validation MSE of 0.012, and with L_2 regularization, 0.0168, with the L_1-regularized network showing superior performance in this low-noise scenario. Both represent substantial improvements over the unregularized model and significantly outperform their linear counterparts (Lasso: 0.0454, Ridge: 0.0195).

Notably, while the L_1-regularized network achieves the lowest validation error overall, the L_2-regularized network also demonstrates strong performance. The L_1 regularization appears particularly well-suited for this prob-

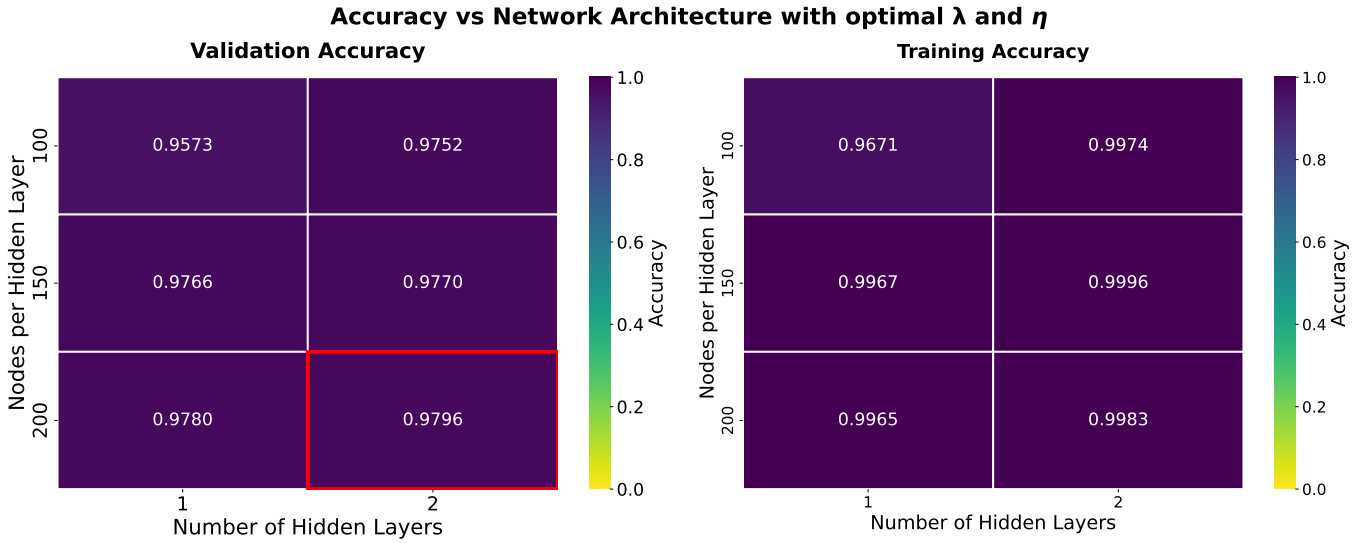

Figure 14. Training and validation accuracy comparison for different network architectures using their respective optimal hyperparameters. The left panel shows validation accuracy, while the right panel displays training accuracy. The red box highlights the best-performing architecture (2 layers, 200 nodes). Both metrics increase with network complexity, with the gap between training and validation accuracy indicating modest overfitting that remains well-controlled through regularization.

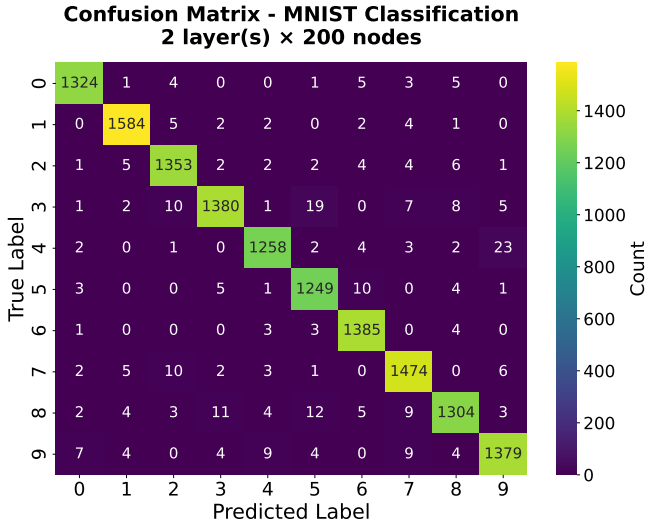

Figure 15. Confusion matrix for the best-performing model (2 layers, 200 nodes, ReLU activation, RMSprop optimizer) on the MNIST test set. The diagonal elements represent correct classifications, while off-diagonal elements indicate misclassifications.

lem configuration, achieving a training MSE of 0.0088 while maintaining excellent generalization. The L_2-regularized network, with a training MSE of 0.012 and validation MSE of 0.0168, shows slightly more conservative regularization but still substantially outperforms Ridge regression. Both regularization strategies enable fast convergence and excellent performance when combined with early stopping.

2. *Performance on Noisy Data*

The high-noise scenario ($\sigma = 0.3$, Figure 12) tests the limits of both approaches. Here, the combination of limited data ($N = 100$), high noise, and a relatively complex network architecture (two hidden layers, 100 nodes each) presents a challenging learning problem. The linear baselines perform worse than the neural networks: Lasso achieves a validation MSE of 0.1347, while Ridge reaches 0.0984.

Both the L_1 and L_2-regularized neural networks demonstrate strong performance under these challenging conditions, substantially outperforming their linear counterparts. The L_1-regularized network achieves a validation MSE of 0.0731 with a training MSE of 0.0673, representing a 46% improvement over Lasso's 0.1347. The L_2-regularized network achieves a validation MSE of 0.0781 with a training MSE of 0.1338, showing a 21% improvement over Ridge's 0.0984.

Interestingly, the L_1-regularized network achieves both the lowest training and validation errors in this high-noise scenario, with the training and validation MSEs being remarkably close (0.0673 vs 0.0731), suggesting excellent generalization without overfitting. The L_2-regularized network shows a larger gap between training and validation error (0.1338 vs 0.0781), indicating that the higher training MSE reflects the stronger regularization constraint rather than poor model capacity. This behavior demonstrates that neural networks, when properly regularized, can significantly outperform linear models even in challenging high-noise, limited-data scenarios.

These results demonstrate that both L_1 and L_2 regularization enable neural networks to extract more com-

plex patterns from noisy data than their linear counterparts can achieve. The choice between L_1 and L_2 regularization shows only modest differences in this problem, with L_1 achieving slightly better performance in both noise levels, though both approaches substantially outperform the corresponding linear methods.

3. *Optimal Regularization Strengths*

The optimal regularization strength λ differs markedly between the linear models and neural networks. In Project 1 [1], we found that Ridge regression required $\lambda = 10^{-2}$, while the L_2-regularized neural network achieves optimal performance with $\lambda = 10^{-4}$ for both the low-noise and high-noise problems (Figure 11 and Figure 12), two orders of magnitude smaller. Similarly, the L_1-regularized network uses $\lambda = 10^{-5}$ for both noise levels, three orders of magnitude smaller than typical Lasso values.

The consistency of optimal λ values across different noise levels suggests that these neural network regularization parameters are more robust to data characteristics than their linear counterparts. This difference in optimal λ values between linear models and neural networks arises from two factors. First, neural networks distribute their expressivity across multiple layers and many parameters, whereas Ridge regression applies the penalty to a single set of polynomial coefficients. Second, because neural networks have substantially more parameters, a weaker penalty (smaller λ) is needed to avoid over-regularization while still controlling model complexity.

4. *Interpretability Tradeoff*

While regularized neural networks consistently outperform their linear counterparts across all tested scenarios, they sacrifice interpretability. Lasso regression produces sparse polynomial coefficients that directly indicate which terms (e.g., x^3, x^7, x^{11}) contribute most to the fit. In contrast, L_1 regularization in neural networks distributes sparsity across hundreds of weights in multiple hidden layers, making it difficult to extract meaningful insights about which input features or learned transformations are important. For applications requiring model interpretability—such as medical diagnosis or scientific discovery—this represents a significant tradeoff against improved predictive performance.

F. Classification of the MNIST dataset

To classify the MNIST digits as accurately as possible, we first performed a grid search over the regularization parameter λ and the learning rate η on a subset of 10 000 images in order to identify suitable hyperparameter combinations (Figure 13). Both λ and η were sampled on a logarithmic scale in the range $[10^{-5}, 10^{-1}]$, and model performance was evaluated using the accuracy metric.

From this search, we found that moderate regularization in combination with a moderate learning rate, $\lambda = 10^{-3}$ and $\eta = 10^{-3}$, yielded the best performance. This behavior is reasonable. Too large learning rates can lead to unstable training and divergence, while too small learning rates slow down convergence and risk getting trapped in suboptimal regions. Similarly, overly strong regularization suppresses the model's capacity to represent complex patterns in the data, whereas too weak regularization allows the network to overfit. The intermediate scenario therefore provides a balance between expressive power and generalization, which explains why the best-performing region is located near the middle of the (λ, η) search space rather than at the boundaries of the ranges.

After finding the optimal hyperparameters in the subset, we employ an architectural search on the full dataset, evaluating which network architecture gives us the best result. When training on the full dataset (Figure 14), the model achieves its highest accuracy using an architecture with two hidden layers of 200 nodes. This differs from the pattern observed in Figure 13, which was based on the reduced 10 000–sample subset. The discrepancy can be explained by the amount of data available during the search: with substantially more training data, larger architectures benefit from the increased capacity without overfitting, and are therefore able to exploit the additional representational power more effectively.

In Figure 15, we analyze the model's performance using a confusion matrix. The diagonal entries represent correct classifications, while off–diagonal entries indicate misclassifications. The model performs generally well across all digit classes, with most errors occurring between visually similar digits such as 4 and 9, or 3 and 5.

We observe some interesting asymmetric misclassification patterns. For example, when the true label is 4, the model predicts 9 in 23 cases, whereas the reverse confusion (true label 9 predicted as 4) only occurs 9 times. A similar asymmetry appears between the digits 3 and 5. This suggests that certain digit classes are more visually ambiguous in one direction than the other, for example the representation learned by the network may capture features that make some digits (such as 4) appear more similar to the shape of another digit (such as 9) than the reverse. In other words, the class embedding learned by the network is not necessarily symmetric in how it perceives visual similarity.

Overall, these results are consistent with the findings shown in Figure 14, where the best architecture achieved a validation accuracy of 97.96%. Our model outperforms Scikit-Learn by 5.96% (see notebook under confusion matrix in part f in [7]).

We have seen here that the method that best ap-

proximated the classification problem is a feed forward neural network with sufficient layers and nodes (2 and 200 respectively) and ReLU activation function, using the RMSprop optimizer and moderate hyperparameters ($\lambda = \eta = 10^{-3}$). From the results in IV A we can see that the neural networks do not beat OLS, meaning that in these low noise, or smooth, scenarios, OLS performs better than neural networks. However, from Section IV C we can see that neural networks with regularization significantly outperform linear methods. At $\sigma = 0.3$, our L_1-regularized network achieved validation MSE of 0.0731 versus Lasso's 0.1347. We have also seen that the adaptive optimizers consistently outperform the full-batch gradient descent. We have seen that Adam is proven to achieve better results than RMSprop quicker. The sigmoid activation function is worse than both ReLU and Leaky ReLU, as the two latter converge faster and avoid the vanishing gradient problem. Batch normalization adds more stability in the networks. In essence, our results demonstrate that the choice of method for regression problems should be chosen considering problem complexity, data characteristics (like noise), and whether model interpretability is needed.

VI. CONCLUSION

We have investigated how classic linear regression methods like OLS, Ridge and lasso perform in comparison to fully connected feed forward neural networks. Across all experiments, the results consistently demonstrate that the performance of feedforward neural networks is strongly determined by a combination of (i) optimization method, (ii) activation function, (iii) network depth/capacity, and (iv) the chosen regularization strategy. For the regression task on the Runge function, adaptive optimizers (Adam and RMSprop) clearly outperform full batch Gradient Descent in both convergence speed and final validation MSE. These optimizers also exhibit more robust learning rate behavior, whereas Gradient Descent requires extremely careful tuning and, in several cases (notably sigmoid networks), completely fails to converge due to vanishing gradients. ReLU and Leaky ReLU activations therefore provide a substantial advantage over sigmoid, especially in deeper networks.

Regarding model capacity, the larger architecture (2×100) generally achieves lower validation MSE than the smaller (1×50), provided early stopping or adequate regularization is used. When capacity is too limited, the small model underfits. When capacity is high and left unregularized, the large model eventually overfits. Early stopping and explicit regularization effectively stabilize training in this intermediate region. In particular, both L_1 and L_2 regularization significantly suppress overfitting, with L_2 being slightly more reliable on the low–noise problem, while L_1 is more flexible in the high–noise setting. Importantly, the optimal λ for neural networks differs from linear regression by several orders of magnitude, reflecting the fundamentally different parameterization and representation learning mechanism.

From our analysis on the performance of the three activation functions across model architectures, we observed degradation in performance at higher complexities. This performance degradation could, however not be concluded to be the classic sign of overfitting, as Sigmoid, ReLU and LeakyReLU were suffering from vanishing gradients, dying neurons and exploding gradients respectively. We found that batch-normalization could help stabilize the gradients - thus leveling the playing field across the activation functions. This illustrated the importance of proper initialization of weights, biases and learning rates in deep networks.

Finally, in the MNIST classification task, the same trends persist. We encountered that moderate λ values combined with adaptive optimizers yields the best performance, and larger architectures benefit from increased data availability. The confusion matrix further reveals that misclassification is not symmetric across visually similar digits, indicating that the network's internal representation is not purely distance-based in pixel space, but shaped by learned hierarchical features.

Overall, these results show that neural networks can match or outperform classical polynomial regression models, but only when appropriate architectural, activation, optimization, and regularization choices are made. Linear models remain competitive when the target function is smooth and low-dimensional, but neural networks scale more effectively to high-capacity settings and complex datasets such as MNIST.

In the future one could extend our work by both looking at more complex regression problems, as well as classification problems. This is especially interesting considering the inability of choosing a sole "winner" among the methods for the regression problem.

[1] H. Haug, S. S. Thommesen, C. A. Falchenberg, and L. L. Storborg. Project 1: *Comparison of Regression Methods for Approximating a Noisy Runge Function*. https://github.com/livelstorborg/FYS-STK4155/tree/main/project1, 2025. GitHub repository.

[2] Wikipedia contributors. Runge's phenomenon. https://en.wikipedia.org/wiki/Runge%27s_phenomenon, 2025. Accessed: 2025-10-05.

[3] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 1998.

[4] Hojjat Kashani. Mnist dataset (kaggle mirror). https://www.kaggle.com/datasets/hojjatk/mnist-dataset. Accessed: November 2025.

[5] Sebastian Raschka, Yuxi Liu, Vahid Mirjalili, and

Dmytro Dzhulgakov. *Machine Learning with PyTorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python*. Packt Publishing, Birmingham, UK, 1 edition, 2022.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 1028–1031. IEEE, 2015. https://doi.org/10.1109/ICCV.2015.123.

[7] H. Haug, S. S. Thommesen, C. A. Falchenberg, and L. L. Storborg. Project 2. https://github.com/livelstorborg/FYS-STK4155/tree/main/project2, 2025. GitHub repository.

[8] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[9] 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. https://doi.org/10.1038/s41586-020-2649-2.

[10] 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.

[11] Michael L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021. https://doi.org/10.21105/joss.03021.

[12] The pandas development team. pandas-dev/pandas: Pandas, August 2025. https://doi.org/10.5281/zenodo.16918803.

[13] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.