NEURAL NETWORKING

FYS-STK3155: Project 2

Bror Johannes Tidemand Ruud (640394)
Egil Furnes (693784)
Ådne Rolstad (693783)

University of Oslo
Date: 10.11.2025
GitHub: `github.com/egil10/FYSSTK3155`

Abstract

Over the past decade, Neural Networks (NNs) have reshaped modern machine learning, driving breakthroughs in computer vision, natural language processing, medical diagnostics, and scientific computing. These advances have helped create some of the world's most valuable technology companies and contributed to Nobel Prize–winning research. The growing impact of NN makes it essential to understand not only how these models perform, but also how they are built and optimized. In this project, we implement a Feed-Forward Neural Network (FFNN) from scratch to study both regression and multiclass classification.

For regression, we approximate the analytical Runge function $f(x) = (1/(1+25x)^2)$ and compare FFNN performance against traditional regression methods. Even without hyperparameter tuning, our [20-20-1] sigmoid NN outperforms an optimized Ordinary Least Squares (OLS) model with a test Mean Squared Error (MSE) of 7.77×10^{-5} versus 1.06×10^{-2} respectively. By applying gradient-based optimization, optimizing learning rates η, mini-batch sizes b, and regularization strengths λ for both L_1 and L_2 penalties, we obtain models that also outperform Ridge and Least Absolute Shrinkage and Selection Operator (LASSO) regression. Using test MSE, our best NN model with L_2 penalty scores 2.94×10^{-6} versus 1.90×10^{-3} and 2.80×10^{-3} for Ridge and LASSO respectively. For classification, we apply our FFNN to the FashionMNIST dataset using softmax cross-entropy loss and adaptive Gradient Descent (GD). Our best NN model achieves an accuracy of 89.5% on the FashionMNIST dataset. Overall, this project demonstrates the flexibility and predictive power of neural networks for both regression and classification tasks, especially when trained with well-tuned hyperparameters and modern GD methods.

I. INTRODUCTION

Neural Networks (NNs) have become one of the most influential tools in modern machine learning, powering applications ranging from Large Language Models (LLMs) to medical diagnostics. Their success stems from the ability to learn complex, highly non-linear relationships directly from data, in contrast to traditional regression models.

Our *main objective* is to investigate how network architecture, activation functions, regularization, and optimization influence predictive performance on both *regression* and *classification* problems. For regression, we approximate the analytical Runge function $f(x) = (1/(1+25x)^2)$ and compare the Feed-Forward Neural Network (FFNN) to the Ordinary Least Squares (OLS), Ridge, and LASSO models developed in Project 1. For classification, we apply our network to the fashion MNIST dataset using softmax cross-entropy loss and mini-batch GD.

The report is organized as follows. In Section II we present the theoretical framework behind neural networks, including evaluation metrics, activation functions (sigmoid, Rectified Linear Unit (ReLU), and Leaky ReLU), cost functions, and the mathematics of backpropagation. Section II also introduces the Gradient Descent (GD) optimizers used in training (plain GD, Root Mean Square Propagation (RMSProp), and Adaptive Moment Estimation (Adam)). In Section III we present the results for both the Runge regression and MNIST classification tasks, analyzing how model architecture, activation choices, and regularization affect predictive accuracy and generalization. References are cited in Section V. Finally, Section IV summarizes the main findings and discusses possible directions for future work, while additional figures, tables, and heatmaps are provided in the appendix in Section VI.

II. METHODS

This section outlines the methodological framework used to design, train, and evaluate the NNs applied in this project. Note that the regression methods used as baselines (OLS, Ridge, LASSO) follow the mathematical framework developed and derived in Project 1. Therefore, we omit a full re-derivation here and only summarize their role as comparison models.

A. Method

1. *Evaluation Metrics*

Before presenting the models and results, we introduce the evaluation metrics used to assess model performance. For regression tasks, we employ the Mean Squared Error (MSE), which quantifies the average squared deviation between the predicted values $\tilde{\mathbf{y}}$ and the true target values $\mathbf{y}$. A lower MSE generally indicates a better predictive performance, as seen in Equation (1), where n is the number of data points [1].

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

For classification tasks, we use the *accuracy score*, which measures the proportion of correctly classified samples among all predictions. Accuracy is one of the most widely used metrics for evaluating classification models, including the NN trained on the FashionMNIST dataset, and is defined in Equation (2), where $\mathbb{I}$ is the indicator function, which equals 1 if the predicted label $\tilde{y}_i$ matches the true label y_i and 0 otherwise.

$$\text{Accuracy}(\mathbf{y}, \tilde{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}\left(y_i = \tilde{y}_i\right), \tag{2}$$

To verify our implementation, we compute a vector containing the predictions difference between our implementation and predictions made by the `MLPRegressor` method by scikit-learn [2]. We then compute the L_2 norm, L_1 norm, and mean absolute difference (MAD) of this vector, as presented in equations Equation (3), Equation (4), and Equation (5).

$$\|\tilde{\mathbf{y}}_{\text{ours}} - \tilde{\mathbf{y}}_{\text{skl}}\|_2 = \left[\sum_{i=1}^{n} \left(\tilde{y}_{i,\text{ours}} - \tilde{y}_{i,\text{skl}}\right)^2\right]^{1/2}, \tag{3}$$

$$\|\tilde{\mathbf{y}}_{\text{ours}} - \tilde{\mathbf{y}}_{\text{skl}}\|_1 = \sum_{i=1}^{n} \left|\tilde{y}_{i,\text{ours}} - \tilde{y}_{i,\text{skl}}\right|, \tag{4}$$

$$\text{MAD}(\tilde{\mathbf{y}}_{\text{ours}}, \tilde{\mathbf{y}}_{\text{skl}}) = \frac{1}{n}\sum_{i=1}^{n} \left|\tilde{y}_{i,\text{ours}} - \tilde{y}_{i,\text{skl}}\right|. \tag{5}$$

2. *Splitting and scaling the data*

For Runge regression we used 100 datapoints and a 80/20 train-test split. On the FashionMNIST dataset we have 70000 datapoints, and use an $\sim 85/15$ split. The ten discrete classes in FashionMNIST represents 10 different clothing articles. A fixed random seed (`6114`) was applied throughout the project to ensure reproducibility.

For the Runge regression, the input features were standardized using the `StandardScaler` from `scikit-learn` [2], which centers the data to zero mean and unit variance according to Equation (6).

$$z = \frac{x - \mu}{\sigma}, \tag{6}$$

where x is the raw feature, μ its sample mean, and σ its standard deviation. Although scaling does not alter the analytical Runge function, it has a significant impact on optimization. For linear models such as Ridge and LASSO it ensures that all coefficients are penalized equally under regularization, and for NN it stabilizes gradient magnitudes, reducing vanishing/exploding gradients and improving training speed.

3. *Neural Networks*

NN are flexible function approximators capable of learning highly non-linear relationships between inputs and outputs. Unlike some regression models, which assumes a fixed analytical model, a NN learns the mapping directly from data by iteratively updating its weights and biases during training [3].

By the *universal approximation theorem*, a FFNN with at least one hidden layer and non-linear activations can approximate any continuous function on a compact domain. While this expressiveness allows NN to model complex structure, it also increases the risk of overfitting, motivating regularization strategies such as L_2 and L_1 penalties.

Throughout this report, NN architectures are written in the form $[h_1 - h_2 - \cdots - h_L - 1]$, where h_ℓ denotes the number of neurons in hidden layer ℓ for $\ell = 1, \ldots, L$, and the final 1 indicates a single output neuron. For example, the architecture $[4 - 4 - 4 - 1]$ (illustrated in Figure 1) corresponds to $L = 3$ hidden layers with four neurons each, followed by a scalar output. Here, we omit the input layer, as in our Runge task there is 1 input and 1 output node (one x to one y), while in our FashionMNIST there is 784 input and 10 output nodes.

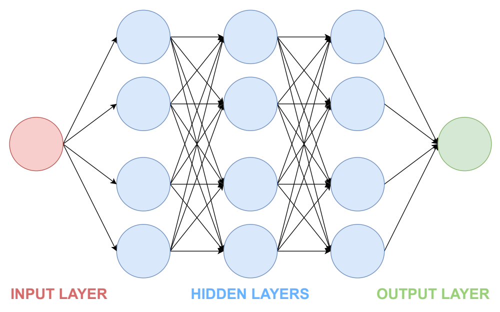


Figure 1: Schematic showing a general NN with input, hidden, and output layers. The circles represent individual nodes, and lines the connections and directions between them. Here is shown a [4-4-4-1] NN representing 3 hidden layers with 4 nodes each.

In practice, many different architectures can be defined depending on the task, ranging from uniform-width networks to those with varying numbers of neurons per layer. One systematic approach to designing such architectures is the *pyramid-scheme* formulation, where each successive hidden layer has fewer neurons than the previous one. This structure gradually compresses information toward the output layer, encouraging hierarchical feature abstraction and parameter efficiency.

We define the layer widths W_k for a pyramid-shaped architecture using the following general formula:

$$W_k = \max\Big(\lceil W_1\, r^{\,k-1} \rceil,\ W_{\min}\Big), \quad k = 1, 2, \ldots, D, \tag{7}$$

where W_1 is the width of the first hidden layer, D is the total number of hidden layers, $r \in (0, 1]$ is the geometric decay ratio, and $W_{\min}$ is the minimum allowed layer width.A pseudocode description of the corresponding algorithmic procedure is shown in Algorithm 1 in Section VI.

Now, looking at the mathematics behind the NN. A FFNN consists of sequential layers of neurons, where each neuron computes a weighted sum of inputs from the previous layer, adds a bias, and applies a non-linear activation function, as shown in Equations (8) and (9):

$$z^{(\ell)} = W^{(\ell)} a^{(\ell-1)} + b^{(\ell)}, \tag{8}$$

$$a^{(\ell)} = \sigma(z^{(\ell)}). \tag{9}$$

where $W^{(\ell)}$ and $b^{(\ell)}$ are the weights and biases of layer ℓ, and $\sigma(\cdot)$ is the activation function. The parameters are optimized by minimizing a cost (loss) function $\mathcal{C}$ that measures the discrepancy between the predicted output $\tilde{\mathbf{y}}$ and the true targets $\mathbf{y}$.

Cost functions and gradients

In this project we derive and use three standard cost functions: (i) the MSE for regression, (ii) the Binary Cross Entropy (BCE) for binary classification, and (iii) the Softmax Cross Entropy (Softmax CE) for multiclass classification.

The MSE loss, commonly used in regression, quantifies the average squared difference between predictions and true values:

$$\mathcal{C}_{\text{MSE}} = \frac{1}{2n} \sum_{i=1}^{n} (a_i^L - y_i)^2, \quad \frac{\partial \mathcal{C}_{\text{MSE}}}{\partial a_i^L} = \frac{1}{n}(a_i^L - y_i), \tag{10}$$

where n is the number of training samples, a_i^L is the output neuron value, and y_i the corresponding target. Adding L_2 (Ridge) or L_1 (LASSO) regularization penalizes large weights:

$$\mathcal{C}_{L_2} = \mathcal{C}_{\text{MSE}} + \lambda \sum_{\ell,j,k} (w_{jk}^{(\ell)})^2, \quad \mathcal{C}_{L_1} = \mathcal{C}_{\text{MSE}} + \lambda \sum_{\ell,j,k} |w_{jk}^{(\ell)}|. \tag{11}$$

Their gradients gain additional terms $2\lambda w_{jk}^{(\ell)}$ and $\lambda \operatorname{sgn}(w_{jk}^{(\ell)})$, respectively.

For binary classification, we use the Binary Cross Entropy (log loss) defined as

$$\mathcal{C}_{\text{BCE}} = -\frac{1}{n} \sum_{i=1}^{n} \Big[y_i \log p_i + (1 - y_i) \log(1 - p_i) \Big], \tag{12}$$

where $p_i = \sigma(z_i^L)$ is the predicted probability from the sigmoid activation. The key simplification in the gradient arises from $\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big)$, giving

$$\frac{\partial \mathcal{C}_{\text{BCE}}}{\partial z_i^L} = p_i - y_i. \tag{13}$$

For multiclass classification, the output layer uses the Softmax CE loss, with probabilities

$$p_{j,i} = \frac{e^{z_{j,i}}}{\sum_{\ell=1}^{K} e^{z_{\ell,i}}}, \tag{14}$$

and the corresponding loss and gradient:

$$\mathcal{C}_{\text{Softmax}} = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{K} y_{j,i} \log p_{j,i}, \quad \frac{\partial \mathcal{C}_{\text{Softmax}}}{\partial z_{j,i}} = p_{j,i} - y_{j,i}. \tag{15}$$

This expression generalizes BCE to multiple classes and provides a numerically stable gradient for backpropagation.

Backpropagation

To train the network efficiently, gradients are computed using the backpropagation algorithm, which applies the chain rule layer by layer in reverse order:

$$\delta^{(\ell)} = (W^{(\ell+1)})^\mathsf{T} \delta^{(\ell+1)} \odot \sigma'(z^{(\ell)}), \tag{16}$$

where $\delta^{(\ell)}$ denotes the error in layer ℓ, and the gradients of the cost function with respect to the parameters are

$$\frac{\partial \mathcal{C}}{\partial W^{(\ell)}} = \delta^{(\ell)} (a^{(\ell-1)})^\mathsf{T}, \quad \frac{\partial \mathcal{C}}{\partial b^{(\ell)}} = \delta^{(\ell)}. \tag{17}$$

These gradients are then used in an optimization algorithm such as Stochastic Gradient Descent (SDG), RMSProp, or Adam to update the parameters iteratively.

Activation functions

Activation functions introduce non-linearity into the network and allow FFNNs to approximate complex mappings that linear models cannot. In this project we test three commonly used choices, each with different optimization properties.

The *sigmoid* activation is smooth and bounded, mapping all inputs to $(0, 1)$. It is useful when outputs represent probabilities, but gradients become very small for large $|z|$, slowing training. Its function and derivative are shown in Equation (18) and Equation (19):

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \tag{18}$$

$$\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big). \tag{19}$$

The *ReLU* activation is computationally efficient and alleviates vanishing gradients for positive inputs, which typically leads to faster convergence in deep networks. However, neurons can "die" if they remain in the zero-gradient region. ReLU is defined in (20)–(21):

$$f(z) = \max(0, z), \tag{20}$$

$$f'(z) = \begin{cases} 0, & z < 0, \\ 1, & z > 0. \end{cases} \tag{21}$$

Leaky ReLU introduces a small negative slope to avoid dying neurons while retaining ReLU's computational benefits. It is given by (22):

$$f(z) = \begin{cases} \alpha z, & z < 0, \\ z, & z \geq 0, \end{cases} \qquad \alpha = 0.01. \tag{22}$$

For regression tasks, the final output layer uses a linear activation, whereas classification uses a softmax layer to produce class probabilities. In this project we pair the regression model with MSE loss and the classification model with softmax cross-entropy, using the analytical gradients derived above.

4. GD methods

The main idea behind GD is to start with an initial set of parameters, compute the gradient of the cost function with respect to these parameters, and then update them in the direction opposite to the gradient. This process is repeated until convergence or until a predefined number of epochs is reached. The general update rule for the parameters $\boldsymbol{\theta}$ is

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta\, \nabla_{\boldsymbol{\theta}} \mathcal{C}(\boldsymbol{\theta}^{(t)}), \tag{23}$$

where η denotes the learning rate that controls the step size of each update.

Full-batch GD

In *full-batch GD*, the gradient $\nabla_{\boldsymbol{\theta}}\mathcal{C}$ is computed using the entire training dataset at each iteration. This approach gives exact gradient directions and typically yields smooth convergence, but it can be computationally expensive for large datasets such as the FashionMNIST dataset. In this project, we use full-batch GD mainly for the regression task on the Runge function, where the dataset size is small enough that full-batch training is efficient.

Stochastic and mini-batch GD

To reduce computational cost, *stochastic* or *mini-batch GD* can be used. Instead of computing the gradient on the full dataset, we estimate it from a randomly selected subset (mini-batch) of size m:

$$\nabla_{\boldsymbol{\theta}} \mathcal{C}_b(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i \in \text{batch}} \nabla_{\boldsymbol{\theta}} \mathcal{C}_i(\boldsymbol{\theta}). \tag{24}$$

This introduces stochasticity into the training process, which can help the algorithm escape shallow local minima and improve generalization. The mini-batch approach is used in all NN training in this project, both for regression and classification.

RMSProp

While plain GD uses a fixed global learning rate, RMSProp adapts the learning rate for each parameter based on an exponentially decaying average of past squared gradients. This prevents the effective learning rate from decaying too quickly and stabilizes training:

$$v_t = \beta v_{t-1} + (1-\beta) g_t^2, \quad \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} g_t, \tag{25}$$

where g_t is the gradient at iteration t, β controls the decay rate (typically 0.9), and ϵ is a small constant for numerical stability. RMSProp performs well in deep learning tasks and was used to train some of the NN in this project.

ADAM

The Adam optimizer combines the advantages of RMSProp and momentum. It maintains exponentially decaying averages of both the gradients m_t and their squared values v_t, with bias corrections applied to both:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \tag{26}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \tag{27}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \tag{28}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t. \tag{29}$$

Typical parameter choices are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. Adam is widely used in NN training because it converges quickly and is robust to noisy gradient estimates.

B. Implementation

The code in this project was implemented in `Python` and written using the `VSCode` and `Cursor IDE`, with version control and collaboration handled through `GitHub Desktop`, uploaded to a public repository on `GitHub`: `github.com/egil10/FYSSTK3155`. The report was co-written in LaTeX using Overleaf. For reproducibility, a global random seed of `6114` was used throughout the project, corresponding to the sum of our day–month birthdays.

When developing the project, we used a range of open-source Python packages. Among these were `NumPy` [4] for efficient array operations and linear algebra, `Pandas` for data manipulation and analysis [5], `scikit-learn` for testing and comparison of results [2], `Autograd` for gradient checking and validation [6], and `matplotlib.pyplot` together with `seaborn` for visualization and plotting [7–9]. Regarding using the FashionMNIST dataset [10], we initialized `PyTorch` [11], and retrieved the dataset from there [12].

The core implementation of the NN, including activation and loss functions, optimizers, and all unit tests, was written as standalone `.py` modules to ensure clarity, reusability, and proper testing. The problem-solving tasks and numerical experiments described in the project assignment were conducted in separate `Jupyter Notebooks` [13], which imported these modules and were used for interactive analysis, visualization, and result presentation.

The implementation is modular and divided into several components. A function was implemented for preparing datasets for the Runge regression task, including a train/test split and optional Gaussian noise. The activation functions, loss functions, and their analytical derivatives were implemented as separate, reusable functions and imported when constructing the NN.

A fully connected FFNN was implemented as a Python class. This class supports an arbitrary number of layers and neurons, with configurable activation functions and cost functions. The forward propagation is vectorized using `NumPy` [4] operations for computational efficiency, while the backward propagation follows the analytical derivation of the chain rule to compute gradients for all parameters. Optional L_2 and L_1 regularization terms can be applied to penalize large weights and improve generalization.

The network parameters are initialized with weights drawn from a normal distribution with mean zero and standard deviation 0.01, and all biases are initialized to zero. This simple but stable initialization scheme ensures that all layers start close to linearity while avoiding symmetry problems during training. The activation function for the final output layer is linear for regression tasks and softmax for classification.

For the FashionMNIST classification experiments, the network does not apply an explicit softmax activation in the final layer. Instead, the model outputs raw logits, and the softmax operation is computed implicitly within the loss function `cross_entropy_with_logits`. This approach is mathematically equivalent to applying a softmax layer followed by cross-entropy loss but is more numerically stable, as it combines the exponential and logarithmic operations into a single step to avoid underflow or overflow when logits have large magnitudes. During inference, the predicted class probabilities are obtained by explicitly applying the softmax function to the final-layer logits. This indirect formulation is standard practice in deep learning libraries and was adopted for our implementation to ensure stable and consistent gradient computations.

Since the gradient descent methods from Project 1 were implemented as standalone functions, it was more suitable in this project to reimplement them as Python classes to better integrate with the NN framework and object-oriented design.

Functions and methods concerning the NN class were tested in separate test files, which can be found in the folder `Testing` in our `GitHub` repository. A `requirements.txt` file was also included, containing the necessary packages. The project structure was designed to promote code reusability, clarity, and extensibility. By organizing the implementation into modules for data preparation, activation functions, losses, optimization, and model architecture, we established a clean framework that supports both regression and classification analyses in a consistent manner.

C. Use of AI tools

In writing this project on NNs, we have actively used one of the most incredible innovations from this technology, like LLMs. Since the explosion from `ChatGPT` in late 2022, we have adopted various new AI models, including coding agents like `Cursor IDE` [14] and `GitHub Copilot` [15], or LLMs competing with OpenAI, like `Grok` from xAI [16] and `Claude` from Anthropic [17].

Using these AI tools have by any measure increased our productivity in writing Project 2, with everything from programming actual code to suggesting report paragraphs, and formatting in Overleaf. Referencing every single AI tool used is difficult, but below we have noted some very specific use-cases for AI in this project, including actual printouts and links to some of our chat conversations.

Moreso, in the `GitHub` repository, we added a folder named `Use_of_AI_Tools` which includes a `Markdown`-file with an exported reference chat from the `Cursor IDE` [18]. This interaction let the new native `Composer 1` model read through our repository and write a professional and comprehensive `README.md`-file for our Project 2.

Using `Grok` from xAI has helped immensely in formatting our report using Overleaf [16]. For example, we used `Grok` to format tables using a specific layout [19], formatting Table II [20], and brainstorming title and introduction [21].

We have of course also used the standard `GPT` models using `ChatGPT 5` from OpenAI [22]. This LLM has helped on suggesting figure captions [23], formatting Overleaf [24], and helping with programming in `Python` [25].

III. RESULTS AND DISCUSSION

A. Regression on Runge's Function

In this section, we study how NNs predict Runge's function $f(x) = (1/(1 + 25x)^2)$. We start by constructing a NN with an arbitrary, simple architecture, to illustrate that NN are capable of making good predictions without finetuning parameters such as learning rate η, batch size b, and architecture. Then, we compare two NNs with architectures given by the project description with the best OLS model from Project 1. To verify that our NN implementation is correct, we compare our models to `scikit-learn` [2], and our derivatives with `Autograd` [6]. Also, as mentioned in Section II, the Runge regression is done on 100 datapoints.

Then, we test different NN architectures and activation functions, seeking to find the optimal combination of hidden layers, hidden nodes and activation functions. We then add L_2 and L_1 regularization to our NN and compare our results with the best Ridge and LASSO model from Project 1, respectively.

When adding regularization, we experiment by tuning the hyper-parameters λ, the learning rate η, and the batch size b. We chose a high-performing, but simple architecture for this part, simplifying training while yielding small test errors. Table I summarizes the main findings, where we see that our NN models outperforms the standard regression models OLS, Ridge, and LASSO.

The simple [20-10-1] sigmoid NN model the standard OLS, Ridge, and LASSO regression models Project 1, without any specific hyperparameter optimization, yielding a test MSE of 7.77×10^{-5}. Finetuning the NN, we find our best model to predict Runge's function to be a [32-32-32-1] Leaky ReLU NN+L_2 penalty model with $\lambda = 10^{-5}$, $\eta = 10^{-2}$ and $b = 32$ with a test MSE of 2.96×10^{-6}, as seen in Table I.

Model	**Architecture / Parameters**	**Test MSE**
NN	[20-10-1], sigmoid	7.77×10^{-5}
OLS	$p = 8$	1.06×10^{-2}
FFNN1	[50-1], sigmoid	5.29×10^{-5}
FFNN2	[50-50-1], sigmoid	3.58×10^{-5}
OLS (w/ noise)	$p = 8$	2.24×10^{-2}
FFNN1 (noise)	[50-1], sigmoid	1.36×10^{-2}
FFNN2 (noise)	[50-50-1], sigmoid	1.26×10^{-2}
Ridge	$p = 13$, $\lambda = 10^{-5}$	1.90×10^{-3}
LASSO	$p = 10$, $\lambda = 10^{-5}$	2.80×10^{-3}
NN+L_2 penalty	[32-32-32-1], Leaky ReLU, $\lambda = 10^{-5}$, $\eta = 10^{-2}$, $b = 32$	2.94×10^{-6}
NN+L_1 penalty	[32-32-32-1], Leaky ReLU, $\lambda = 10^{-4}$, $\eta = 10^{-2}$, $b = 50$	5.27×10^{-6}

Table I: Comparison of Test MSE across linear models and feed-forward NN (NN), and some models with noise_scale = 0.01. While OLS deteriorates substantially under noise, deeper NN retain low error, and L_1/L_2-regularized NN achieve the best generalization overall, reducing test error by multiple orders of magnitude.

1. *Fitting NN to the Runge function*

Figure 2 shows a simple NN fitted to the Runge function, without any fine-tuning or optimization. As shown in the figure, and as summarized in Table I, we see that the NN already outperforms the optimized OLS model. This illustrates the great potential in NN models, but we still want to optimize the NN architecture and hyperparameters to find the optimal model for predicting the Runge function.

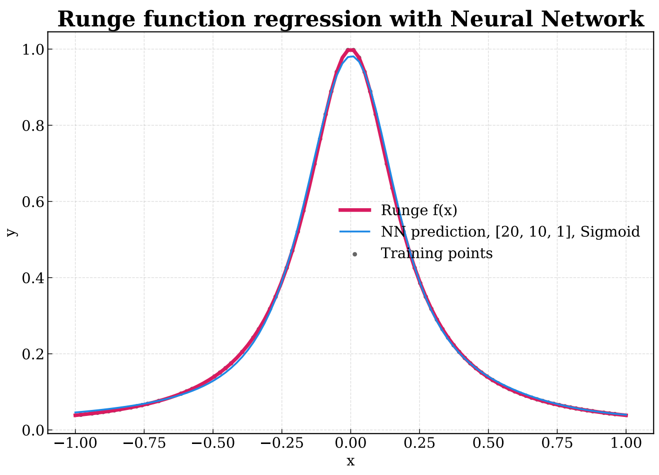


Figure 2: Runge function regression using a NN. True Runge with [20-10-1] NN using Sigmoid activation.

2. *Comparing NN to OLS*

We compare two FFNN models with 1 and 2 hidden layers, and 50 nodes in each hidden layer, with the optimal OLS model from Project 1. We call these FFNN1 and FFNN2. The architecture for the two models is [50-1] and [50-50-1], with sigmoid activation in the hidden layers and linear in the output layer. To find the optimal training scheme, we plot the learning curves for different optimizers and learning rates in Figure 3, with a batch size of $b = 32$ and epochs 500, 1500, and 2000 for ADAM, RMSprop, and plain GD respectively. Note that the figure shows learning curves for the NN with 2 hidden layers only. The figure shows that stochastic mini-batch GD with ADAM performs best, with a learning rate of $\eta = 0.05$ which yields a Test MSE of 3.6×10^{-5}.

The noisy learning curves observed for RMSprop with small batch sizes arise because highly variable gradient estimates cause large fluctuations in the adaptive learning rates, leading to unstable and oscillatory training behavior. The learning curves for plain, full-batch GD shows that this optimizer most likely reaches a local minima or saddle point yielding a Test MSE of ≈ 0.0725 for all learning rates η. Since the method is non-stochastic and has no momentum, the optimizer is unable to escape this point. Based on these learning curves, we chose to use the ADAM optimizer with batch size $b = 32$ and learning rate $\eta = 0.05$ to train the NN we will compare with OLS from Project 1.

By experimenting without plotting we found that the ADAM optimizer with $\eta = 0.01$, $b = 32$ and 1000 epochs was optimal for the NN with 1 hidden layer.

Using the training schemes described above, we trained the two NNs, and compared the results with the optimal OLS model from Project 1. As seen in Figure 4, the NN avoids the end-point oscillations that characterizes the OLS model. As presented in Table I, the Test MSE from these FFNNs outperform OLS significantly.

3. *Testing predictions against `scikit-learn`*

To verify the correctness of our implementation, we compared the predictions from our NN to `scikit-learn MLPRegressor` [2]. Several architectures were tested, varying the number of hidden layers, nodes, and activation functions. For each configuration, we evaluated test predictions differences using L_2-norm and MAD.

The results, summarized in Table II, show that the predictions from our own implementation are very close to those from `scikit-learn`, though not identical [26]. The small discrepancies are mainly due to differences in the initialization of weights and biases: our own NN initialize all weights from a normal distribution with mean zero and standard deviation 0.01, and all biases are set to zero. However, `scikit-learn` uses a different initialization scheme. Consequently, the optimization starts from different initial parameter values, leading to slightly dif-

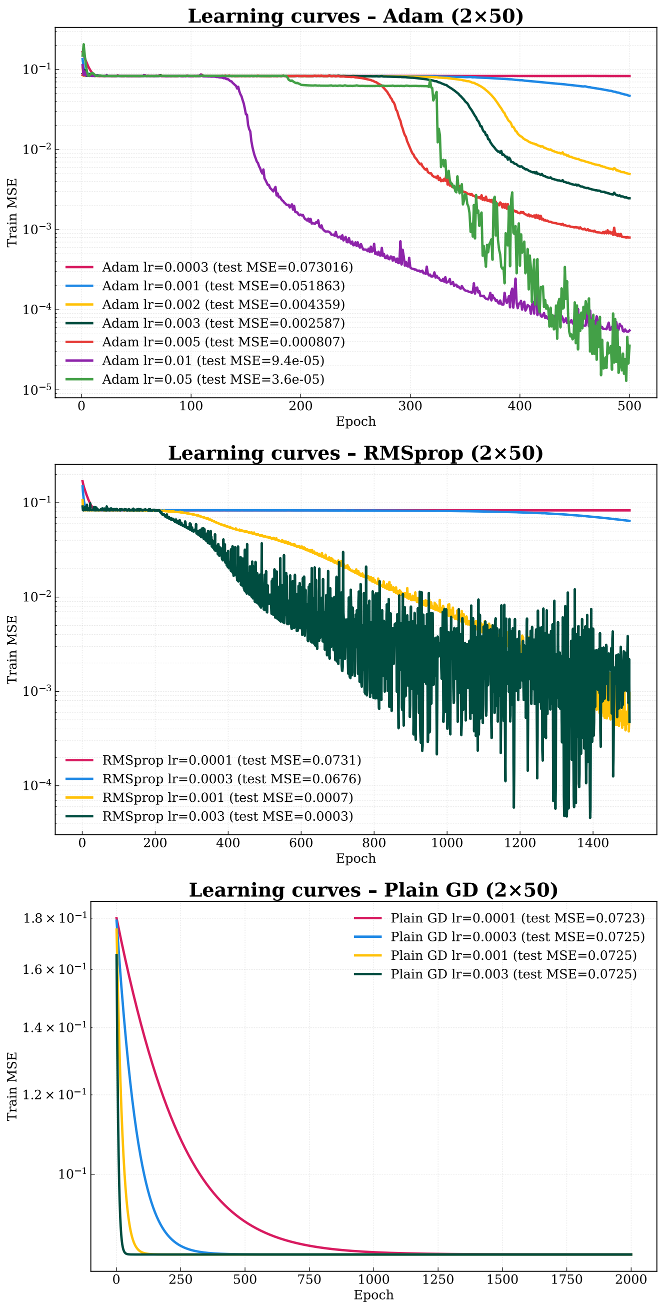


Figure 3: Learning curves for three optimization algorithms on a two–layer FFNN (50 nodes per hidden layer). ADAM converges rapidly and consistently reaches the lowest Test MSE across a broad range of learning rates. In particular, ADAM with learning rate 0.05 achieves the best performance overall, obtaining a Test MSE of 3.6×10^{-5}. RMSProp achieves comparable error but shows more oscillatory behavior at moderate–to–large learning rates η. Plain GD is failing to reach the same accuracy as adaptive optimizers. The curves highlight the practical benefits of adaptive step–size methods and stochastic mini-batch when training NN.

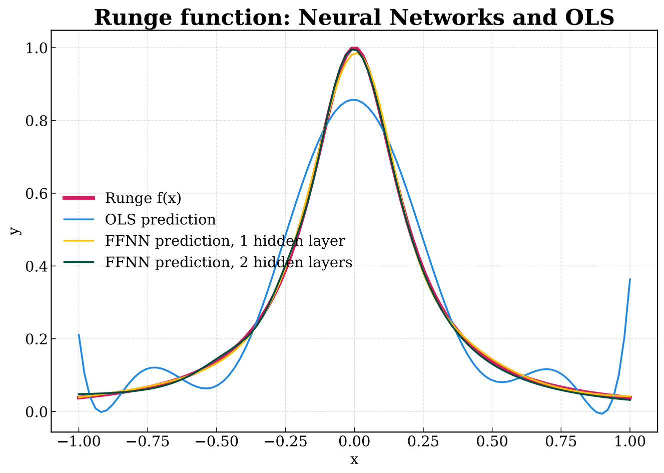


Figure 4: Comparison of OLS and NN on the Runge function. The OLS fit fails to capture the steep curvature near $x = 0$ and suffers from oscillations near the boundaries, while both NN architectures (1 and 2 hidden layers) produce smooth approximations that follow the true function closely. This demonstrates how NN outperform linear models on highly non-linear regression tasks. We use the learning rate $\eta = 0.05$, which yields the lowest Test MSE from using ADAM 2×5 in Figure 3.

ferent final weights and biases. Nevertheless, the resulting predictions are in strong agreement, as illustrated in the parity plot in Figure 5. A plot comparing the predictions from our own NN and `scikit-learn` can be found in Figure 19 in Section VI.

Architecture	L_2**-norm**	**MAD**
Arch 1	3.45×10^{-1}	6.72×10^{-2}
Arch 2	1.18×10^{0}	1.98×10^{-1}
Arch 3	8.82×10^{-2}	1.61×10^{-2}
Arch 4	4.82×10^{-2}	8.56×10^{-3}

Table II: Prediction differences (sklearn vs. own implementation) across architectures. Arch 1: [10-1] Logistic; Arch 2: [50-50-1] Logistic; Arch 3: [50-50-1] ReLU; Arch 4: [64-64-64-1] ReLU.

4. Testing derivatives against `Autograd`

To verify the correctness of our analytical backpropagation implementation discussed in Section II, we compared the computed gradients of our network with reference gradients obtained using the automatic differentiation library `Autograd` [6]for two different network architectures, namely (i) [50-50-01], sigmoid, and (ii) [20-20-20-1], ReLU, with both models using a linear activation function in the output layer.

The gradient comparison for (i) in Table III shows extremely small numerical differences in L_2-norm between our implementation and `Autograd`, in the order of $< 10^{-16}$, corresponding to double-precision rounding errors.

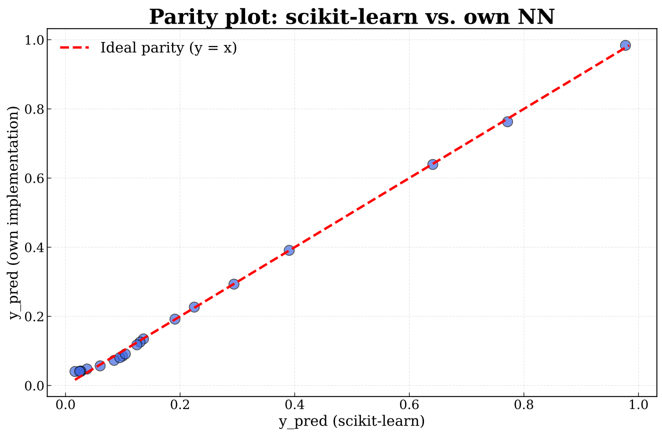


Figure 5: Parity plot comparing our NN implementation and `scikit-learn`. Predicted values from our own FFNN are plotted against predictions from the `scikit-learn` `MLPRegressor`. The dashed red line represents the ideal parity line $y = x$. The close alignment of the scatter points with the diagonal indicates numerical agreement between the two implementations.

For (ii), the comparison yields zero difference (0.00×10^0) across all layers. This confirms that our backpropagation implementation produces numerically accurate gradients for both shallow and deeper networks with both smooth and piecewise-linear activations.

Layer	[50-50-1] sigmoid	[20-20-20-1] ReLU
0	4.26×10^{-21}	0.00×10^0
1	4.98×10^{-18}	0.00×10^0
2	1.92×10^{-16}	0.00×10^0
3	—	0.00×10^0

Table III: `Autograd` gradient verification for two feed-forward NNs. Entries show the L_2-norm differences per layer between analytic backpropagation and automatic differentiation. The [50-50-1] sigmoid network shows discrepancies $< 10^{-16}$, while the deeper [20-20-20-1] ReLU network matches `Autograd` to machine precision across all layers.

5. *Testing different architectures of NNs*

We next investigate how the choice of activation function affects the network performance as the depth increases. The activation functions sigmoid, ReLU, and Leaky ReLU are discussed in detail in Section II.

Figure 6 shows the Test MSE as a function of the number of hidden layers for these activation functions, with 32 nodes in each hidden layer. For all activation functions, the test error decreases as the network depth increases up to four hidden layers.

Beyond this point, both ReLU and Leaky ReLU exhibit small oscillations in Test MSE, with Leaky ReLU showing slightly larger variability. In contrast, the sigmoid activation suffers a sharp increase in test error for deeper networks, caused by vanishing gradients that prevent effective learning. These results demonstrate that ReLU-type activations are significantly more stable and scalable than sigmoid for training deeper architectures.

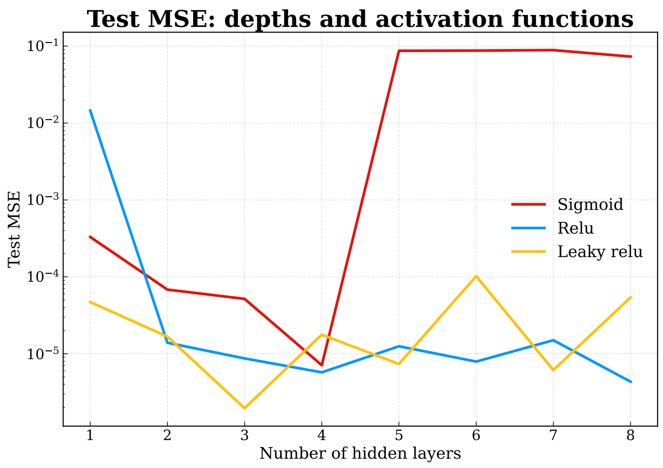


Figure 6: Test MSE as a function of network depth for different activation functions, using sigmoid, ReLU, and Leaky ReLU. We use 32 nodes in each hidden layer, so [32-1], [32-32-1] etc.

Since the network performance is best for depths up to four hidden layers, we restrict our analysis to this range. Learning curves for different activation functions and depths are provided in Figure 7, but these are not analyzed further here. Instead, we focus on how both network depth and the number of neurons per hidden layer affect the Test MSE for the different activation functions.

Figure 7 shows heatmaps of the Test MSE, with the number of neurons per hidden layer on the x-axis and the number of hidden layers on the y-axis, for the sigmoid, ReLU, and Leaky ReLU activations, respectively.

For the sigmoid activation, the lowest Test MSE is achieved for four layers with 32 hidden nodes per layer. Both very narrow and very wide networks (8–16 or 128 nodes) perform significantly worse, especially for deeper models, suggesting an optimal trade-off between network width and depth around 3–4 layers and 32–64 nodes per layer.

For the ReLU activation, shallow networks with only one hidden layer perform poorly, as do small architectures with few nodes per layer (8–16). Performance generally improves with increased network depth and width, indicating that ReLU benefits from larger and deeper architectures.

The Leaky ReLU activation outperforms both previous cases as model complexity increases, showing consistently lower Test MSE for wider and deeper networks. The best result is obtained with four hidden layers and 128 nodes per layer, yielding the lowest overall Test MSE of 1.54×10^{-6}. All Test MSEs can be found in Table VI in Section VI.

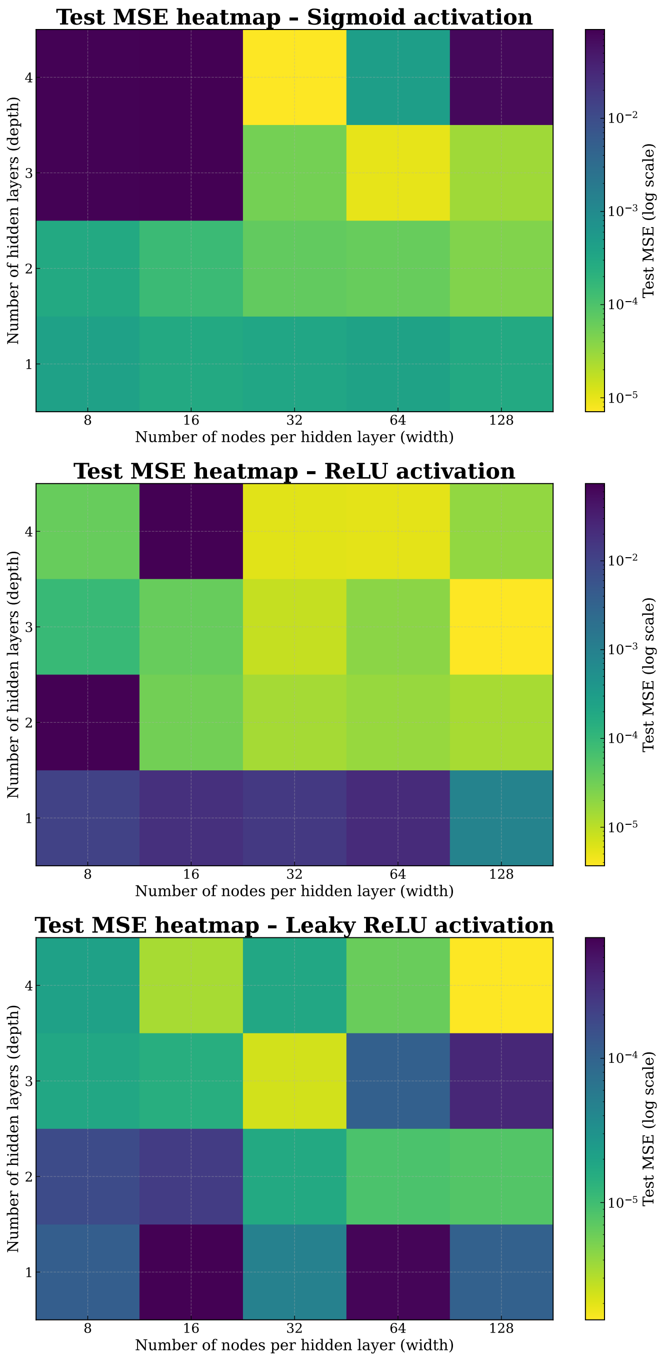


Figure 7: Test MSE heatmaps comparing network depth (vertical axis) and width (horizontal axis) for three activation functions: sigmoid, ReLU, and Leaky ReLU. Sigmoid networks improve with additional depth and width, but suffer from vanishing gradients when too shallow or extremely wide. ReLU achieves low error primarily through depth, but very wide shallow architectures generalize poorly. Leaky ReLU delivers the most robust performance across architectures, avoiding dead neurons and producing consistently low test error, with best results around 2–3 hidden layers and 32–64 nodes per layer.

6. *Testing L_2 and L_1 norms*

Now we add the hyperparameter λ with the L_2 and L_1 norm to the cost function of the NN. We seek to minimize Test MSE in our NN model finding the optimal hyperparameter λ, learning rate η and batch size b. Finally, we compare our NN models with our best Ridge and LASSO models from Project 1.

Although the architecture with four hidden layers and 128 neurons per layer with Leaky ReLU activations gave the lowest Test MSE in the most recent experiments, we chose to use a smaller network with three hidden layers and 32 neurons per hidden layer. We still use Leaky ReLU activations in the subsequent analysis of L_2 and L_1 regularization. This [32-32-32-1] architecture was previously found to give comparable performance and represents a more balanced model in terms of computational cost and training stability.

In Figure 8 we show the Test MSE for different values of the hyperparameter λ with L_2 regularization, and see that $\lambda = 10^{-5}$ yields the smallest Test MSE. Figure 22 in Section VI show the corresponding plot for the L_1 norm, yielding an optimal hyperparameter $\lambda = 10^{-4}$. We use these values of λ to find the optimal learning rate η with L_2 and L_1 norms.

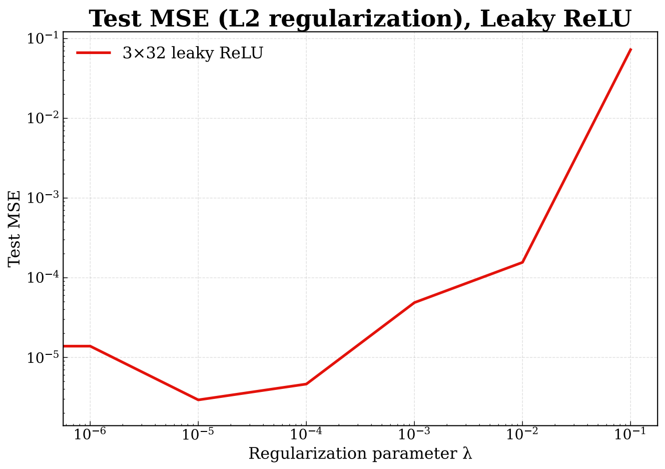


Figure 8: Test MSE as a function of L_2 regularization strength for a Leaky ReLU network. Without regularization or with very weak penalties, the model generalizes well and reaches low test error. As λ increases beyond 10^{-3}, the model becomes overly constrained, leading to worse test performance and clear underfitting.

Figure 9 shows the Test MSE as a function of the learning rate η for the NN with L_2 norm. From the figure we see that the optimal learning rate is $\eta = 0.01$. Figure 22 in Section VI show the corresponding results for L_1 norm, yielding an optimal learning rate of $\eta = 0.01$. We use these values of η to find the optimal batch size for NNs with L_2 and L_1 norms.

Figure 10 shows the Test MSE as a function of the batch size. From the figure we see that the optimal batch size for the NN with L_2 norm is $b = 32$. From the

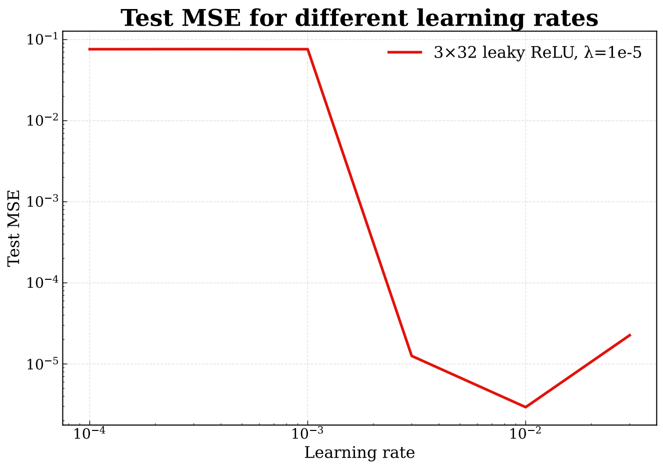


Figure 9: Test MSE for Leaky ReLU networks trained with different learning rates. Test error decreases sharply as learning rate increases up to $\eta = 10^{-3}$, after which too large rates degrade performance. The optimal learning rate lies in the range 10^{-3}–10^{-2}.

corresponding plot in Figure 22 in Section VI, we see that the optimal batch size for the NN with L_1 regularization is $b = 50$.

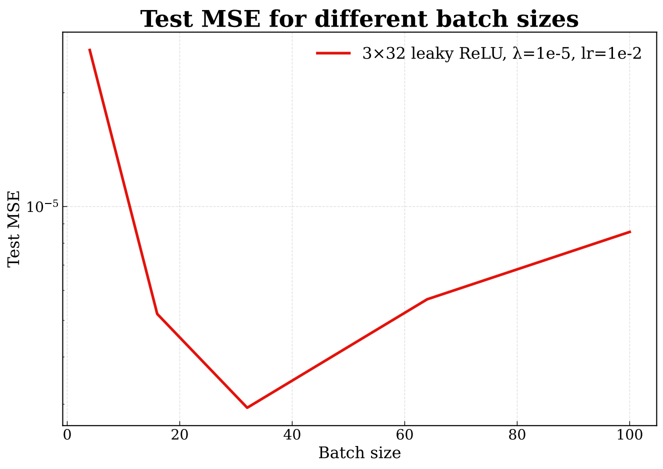


Figure 10: Test MSE as a function of batch size for Leaky ReLU training. Very small batches (e.g., 4) generalize poorly due to noisy gradient steps, while medium-sized batches achieve the lowest error. Extremely large batches reduce training noise but may lead to slower convergence and slightly worse generalization.

After this analysis, we have the following optimal parameters for the NNs with L_2 and L_1 norms, shown in Table IV:

Norm	Hyperparameter λ	Learning rate η	Batch size b
L_2	1.0×10^{-5}	0.01	32
L_1	1.0×10^{-4}	0.01	50

Table IV: Hyperparameters used for L_2 and L_1 regularization.

We use these parameters and compare the NNs with the optimal Ridge and LASSO models from Project 1. Figure 11 shows the predictions with L_2 norm compared with the Ridge model. Figure 20 in Section VI show the corresponding plot for the NN with L_1 norm and LASSO regression. The test MSEs for these models can be found in Table I. We see that the NN outperforms the methods from Project 1, and that the NN with L_2 norm performs best out of all the models.

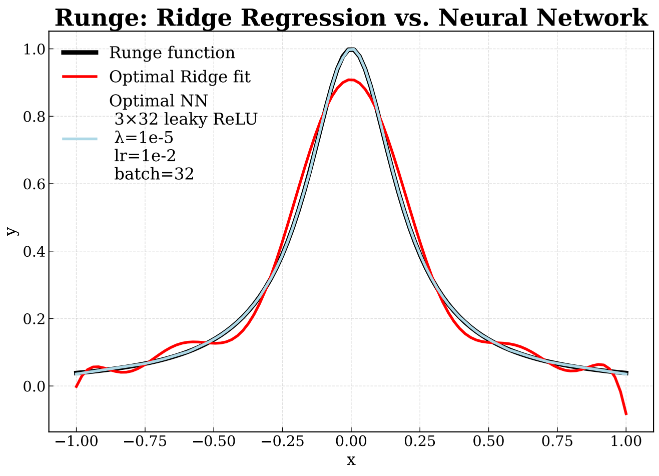


Figure 11: Comparison of the Runge function fitted with Ridge regression and a NN using Leaky ReLU and L2 regularization. Ridge captures the global shape but struggles near steep curvature, while the NN follows the function more accurately across the entire domain. This highlights the advantage of NN for non-linear regression tasks.

B. Classification on FashionMNIST

1. *Visualizing the dataset*

Before training the classification model, the Fashion-MNIST dataset was inspected to gain an intuitive understanding of the input distribution and the visual similarity between categories [10]. The dataset consists of grayscale images of size 28×28 pixels across ten clothing classes, each containing approximately 6,000 training examples and 1,000 test examples. Representative samples from all classes are shown in Figure 12. Several categories exhibit overlapping visual patterns, such as '7: Sneaker' and '9: Ankle boot', which can make class separation challenging for the network.

2. *Architecture analysis: Pyramid scheme*

To evaluate how architectural design influences classification performance, we employed the pyramid-scheme formulation described in Equation (7). The initial width W_1 and depth r were varied to generate networks with systematically decreasing hidden-layer widths. The goal was

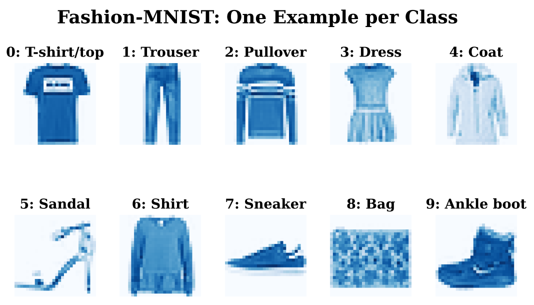


Figure 12: Representative images from the ten FashionMNIST classes. Each image corresponds to one of the dataset's labels (0–9). Visually similar categories is reflected in 27.

to identify an architecture that balances model capacity and generalization.

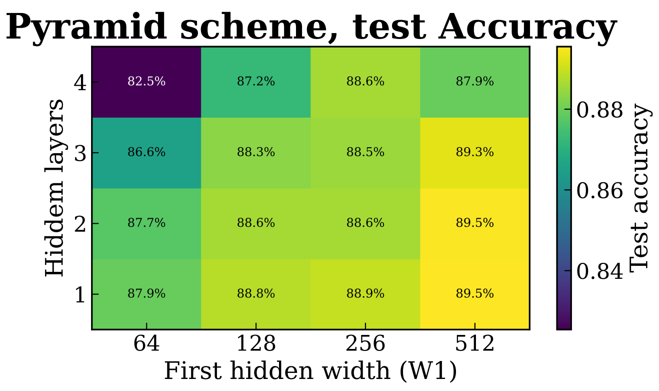


Figure 13: Test accuracy for pyramid-shaped architectures. Each point represents a network with an initial width W_1 and decay ratio $r = 0.5$ with hidden layers of depth from 1 to 5, where colors indicate the resulting test accuracy. The highest-performing region corresponds to [512,10], achieving a test accuracy of 89.5%

.

Test accuracy generally increases with larger W_1 and shallower network depth. Architectures that compress too aggressively or include many layers tend to underfit, as excessive depth limits the model's ability to capture simple spatial patterns in the data. Since FashionMNIST images are low-resolution and contain limited structural complexity, a single wide layer captures most discriminative features effectively. Deeper networks add unnecessary parameters without improving generalization. Comparing architectures with equal-sized layers to those using the pyramid scheme (Figure 24), we find that the pyramid structure slightly improves performance by encouraging compact feature representations and reducing overparameterization. However, the best overall performance is still achieved by a one-layer network with a high number of neurons, indicating that for this dataset, wider architectures are more beneficial than deeper ones. The final configuration chosen for subsequent experiments was [512-10], which achieved a favorable trade-off between accuracy, model complexity, and training efficiency.

3. Classification performance and confusion matrix

The chosen architecture was evaluated on the full FashionMNIST test set. Figure 14 shows the normalized confusion matrix, where each row sums to one and diagonal elements indicate correctly samples.

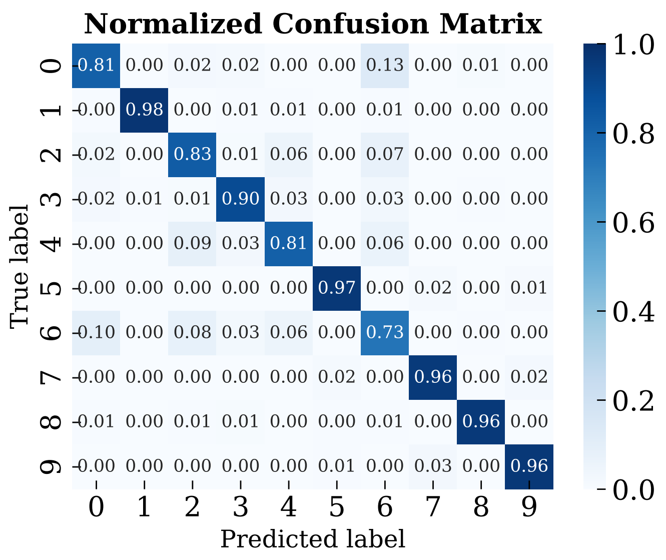


Figure 14: Normalized confusion matrix for the best-performing network [512-10], trained with the Adam optimizer $\eta = 5 \times 10^{-4}$ and L_2 $\lambda = 1 \times 10^{-5}$. Values on the diagonal are correct classifications, while off-diagonal entries highlight frequent confusions between classes.

The model demonstrates strong performance across most classes, with the majority of predictions concentrated along the diagonal. However, certain visually similar categories such as '9: Ankle boot'/'7: Sneaker', '2: Pullover'/'6: Shirt', and '3: Dress'/'6: Shirt' remain partially overlapping, indicating that feature extraction in these regions could benefit from deeper or more specialized architectures. The corresponding non-normalized confusion matrix shown in Figure 27 further illustrates the impact of class frequency on total misclassifications.

4. Implementation verification

To validate the correctness of our custom neural network implementation, we compared it against the `PyTorch` reference model trained with the same architecture and hyperparameters: [512-10], Adam optimizer $\eta = 5 \times 10^{-4}$, and L_2 regularization $\lambda = 1 \times 10^{-5}$. As shown in Figure 15, the two implementations produce nearly identical output logits and classification accuracy.

Both implementations achieve similar final performance, though our custom network show slightly greater instability during training—most noticeably around epoch 50, where the accuracy dips before recovering. This behavior is likely due to differences in weight initialization, as `PyTorch` employs a more advanced and adaptive initialization scheme that promotes smoother convergence.

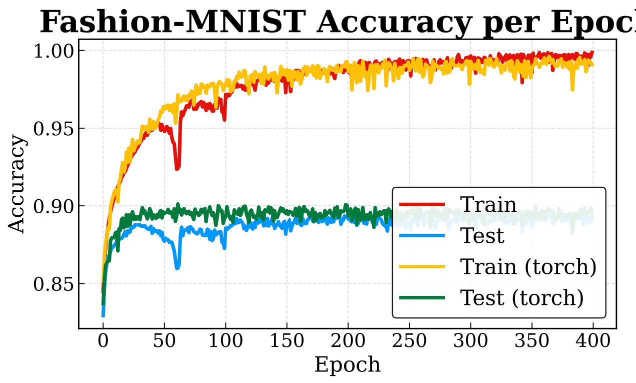


Figure 15: Parity plot comparing predictions from our custom implementation and the `PyTorch` baseline, both trained with the architecture [512, 10], Adam optimizer (`lr = 5×10`$^{-4}$), and L_2 regularization ($\lambda = 1 \times 10^{-5}$).

Despite these fluctuations, the overall training trajectory and final accuracy confirm that our implementation reproduces the expected behavior of a standard deep learning library.

5. Comparison with baseline models

Finally, we compared our best-performing neural network to a logistic regression classifier trained on the same dataset. The comparison metrics are summarized in Table V, including accuracy, precision, recall, F1-score, and runtime.

Table V: Comparison between Logistic Regression and our Neural Network on the FashionMNIST dataset.

Model	Accuracy	Precision	Recall	F1-score	Runtime (s)
Logistic Regression (sklearn)	**0.8442**	**0.8431**	**0.8442**	**0.8434**	**538.8**
Neural Network [512,10]	**0.8918**	**0.8933**	**0.8918**	**0.8921**	**175.2**

Compared to the logistic regression baseline, our neural network achieves a 5.6% relative improvement in accuracy(0.8918 vs. 0.8442) while reducing runtime by approximately 67% (175.2 s vs. 538.8 s). This improvement demonstrates the advantage of using a non-linear architecture capable of capturing higher-order feature interactions that logistic regression cannot model. Despite its greater flexibility, the network remains computationally efficient due to vectorized operations and mini-batch training with the Adam optimizer.

Further analyses of learning rate, batch size, regularization, and activation functions are provided in the appendix (Figures 24 to 28).

From the learning rate and batch size heatmap (Figure 25), we observe that larger batch sizes generally yield higher test accuracy, likely due to more stable gradient estimates during training. The optimal configuration was found at a learning rate of 5×10^{-4} with a batch size $b = 128$, balancing convergence speed and generalization performance.

The optimizer and regularization study (Figure 26) shows that Adam and Momentum optimizers achieve the highest accuracies, whereas plain SGD is more stable but converges more slowly. Across all optimizers, performance declines when the regularization parameter exceeds approximately 1×10^{-2}, suggesting over-penalization of weights. Both Adam and Momentum remain relatively stable in the range 1×10^{-5}–1×10^{-4}, and based on this observation, we selected a regularization strength of 1×10^{-5} for the final configuration.

The activation function comparison in Figure 28 reveals that ReLU and Leaky ReLU achieve high accuracy early in training, reflecting their efficient gradient propagation, but both exhibit slight declines toward the final epochs. In contrast, the Sigmoid activation converges more gradually yet maintains greater stability, eventually catching up and slightly outperforming the ReLU-based networks. This suggests that while ReLU-type activations accelerate early learning, sigmoid may provide more consistent performance when training over many epochs.

IV. CONCLUSION

Critical evaluation of the various algorithms

In writing this report we have learned to implement a NN from scratch, which has been immensely rewarding. It was especially interesting to see how a NN using straight mathematics and linear algebra with our own implementation compared to a major machine learning library like `scikit_learn`.

Across the experiments, several optimization algorithms and activation functions were tested to evaluate their stability, convergence properties, and predictive performance. Throughout the project, we found that some NN models outperformed others, and that they had different use cases.

For the Runge regression task, gradient descent with the Adam optimizer consistently produced the lowest test error and the most reliable convergence across a wide range of learning rates and batch sizes. However, this was not without limitations. As seen in the learning curves in Section VI, particularly Figure 17, the RMSProp optimizer showed highly erratic training trajectories, likely caused by fluctuations in the squared gradient estimates when using small mini-batches. This instability led to oscillatory learning curves and reduced reliability.

Even when using Adam, convergence was not always smooth. In Figure 18, deeper networks trained with ReLU and Leaky ReLU activations exhibit noisy or inconsistent learning curves. For ReLU, this behavior may result from "dying neurons" that stop updating once activations fall to zero. While Leaky ReLU is designed to mitigate this issue, the learning curves still show noticeable variance, suggesting that deep networks with aggressive activation functions require careful initialization and hyperparameter tuning to train stably. In contrast, sigmoid activation converged more slowly but produced smoother learning curves and more stable training for deeper architectures.

In the classification case, we used accuracy to predict on the FashionMNIST dataset. NNs with shallow architectures did not appear to suffer from the vanishing gradient problems we found in the regression task. In this case, the sigmoid activation function seemed to be more stable than the ReLU versions, although it exhibits slower convergence compared to the aforementioned variants.

In conclusion we have found that NN works great both on regression and classification problems. In the regression task we found that NNs were incredibly flexible, and that even a random unoptimized NN fitted almost perfectly to the Runge function, way better than the standard regression models from Project 1. The main takeaway from the classification problem was the NNs also did very well in predicting different classes, and an obvious improvement would be to expand on different NN methods.

Possible directions

Within our project we have gone about tuning our models several times and have found that there is quite a large number of tuning parameters and especially combinations of those tuning parameters. Possible directions for further study would be to spend even more time fully going down all of these rabbit holes. When testing our neural nets we first found the optimal L_2 and L_1-norm parameters λ, then the learning rate η and batch size b. Although given large amounts of compute one could possibly even try different approaches and expand the possibility space that one searches when finding these optimal combinations.

Future improvements

For further analysis we might expand the NN to include other kinds of NN. Following the past years extensive growth within AI, it is obvious that NN has already shown immense use-cases and might even show even bigger future opportunities. In our specific example we used FFNN but it would be interesting to figure out how to use other kinds of NNs as well, namely like Convolutional Neural Networks (CNN) for images. Moreso, evaluation a NN can always be difficult, and in this project we mainly used MSE and accuracy measures, although, a perfect accuracy is not necessarily the same as having a useful model.

A natural next step for improvement would be to implement a CNN, which is particularly well-suited for image-based datasets such as FashionMNIST. Unlike fully connected networks, CNNs exploit the two-dimensional spatial structure of images through local receptive fields and shared weights, allowing them to detect edges, textures, and higher-level patterns more effectively. This architecture can substantially reduce the number of trainable parameters while improving generalization and translation invariance. Introducing convolutional and pooling layers would therefore be expected to enhance both performance and robustness, particularly for visually similar classes.

V. REFERENCES

Link to `GitHub`: `github.com/egil10/FYSSTK3155`

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12: 2825–2830, 2011.

[3] Sebastian Raschka, Yuxi (Hayden) Liu, and Vahid Mirjalili. *Machine Learning with PyTorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python*. Expert Insight. Packt Publishing, Birmingham, UK, 2022. ISBN 9781801819312.

[4] 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, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi: 10.1038/s41586-020-2649-2.

[5] The pandas development team. pandas-dev/pandas: Pandas, 2020. URL `https://doi.org/10.5281/zenodo.3509134`.

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

[7] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[8] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[9] Michael L. Waskom. Seaborn: Statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021. doi: 10.21105/joss.03021.

[10] Zalando Research. Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms. `https://www.kaggle.com/datasets/zalando-research/fashionmnist`, 2017. Accessed: 2025-11-10.

[11] PyTorch Foundation. Pytorch: An optimized tensor library for deep learning. `https://docs.pytorch.org/docs/stable/index.html`, 2017. Accessed: 2025-11-10.

[12] PyTorch Foundation. torchvision.datasets.FashionMNIST documentation. `https://docs.pytorch.org/vision/stable/generated/torchvision.datasets.FashionMNIST.html`, 2017. Accessed: 2025-11-10.

[13] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks – a publishing format for reproducible computational workflows. In *Positioning*

and Power in Academic Publishing: Players, Agents and Agendas, pages 87–90. IOS Press, 2016.

[14] Cursor. Cursor. https://cursor.com/. AI coding assistant. Accessed: 2025-11-09.

[15] GitHub. Github copilot. https://github.com/features/copilot, 2025. AI pair programmer developed by GitHub in collaboration with OpenAI. Accessed October 2025.

[16] xAI. Grok. https://grok.com/. Accessed: 2025-11-09.

[17] Anthropic. Conversation with claude ai (sonnet 4.5). https://claude.ai/share/035710da-2469-4a38-928d-cfbc75897de9, 2025. Accessed via Claude AI, Sonnet 4.5 model at https://claude.ai/.

[18] Cursor AI. Cursor: The ai code editor. https://cursor.com/, 2025. AI-assisted development environment for programming. Accessed October 2025.

[19] Table caption formatting in . https://grok.com/share/bGVnYWN5_fcbf91a2-7700-437c-a107-cbb56ab3ce99, . Accessed: 2025-11-10.

[20] Grok chat share. https://grok.com/share/bGVnYWN5_87d3972f-6e19-4e13-97f7-879e6dd3e635, . Accessed: 2025-11-10.

[21] Grok chat share. https://grok.com/share/bGVnYWN5_ffd54079-40fd-4125-bf41-bc4429ec2554, . Accessed: 2025-11-10.

[22] OpenAI. Chatgpt. https://chat.openai.com/. Accessed: 2025-11-09.

[23] Chatgpt conversation share. https://chatgpt.com/share/6911e072-df30-8005-98d4-3be7c89b769f, . Accessed: 2025-11-10.

[24] Chatgpt conversation share. https://chatgpt.com/share/6911e0f6-480c-8005-9852-835e3c59b243, . Accessed: 2025-11-10.

[25] Chatgpt conversation share. https://chatgpt.com/share/6911e19e-5164-8005-853b-f9e82c6fced2, . Accessed: 2025-11-10.

[26] 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.

LIST OF ABBREVIATIONS

Adam	Adaptive Moment Estimation.
BCE	Binary Cross Entropy.
FFNN	Feed-Forward Neural Network.
GD	Gradient Descent.
LASSO	Least Absolute Shrinkage and Selection Operator.
LLMs	Large Language Models.
MSE	Mean Squared Error.
OLS	Ordinary Least Squares.
ReLU	Rectified Linear Unit.
RMSProp	Root Mean Square Propagation.
SDG	Stochastic Gradient Descent.
Softmax CE	Softmax Cross Entropy.

VI. APPENDIX

Sigmoid	**8**	**16**	**32**	**64**	**128**
Depth 1	4.071×10^{-4}	2.940×10^{-4}	3.303×10^{-4}	3.825×10^{-4}	2.870×10^{-4}
Depth 2	2.910×10^{-4}	1.419×10^{-4}	6.823×10^{-5}	6.279×10^{-5}	4.312×10^{-5}
Depth 3	8.401×10^{-2}	8.977×10^{-2}	5.171×10^{-5}	1.017×10^{-5}	2.733×10^{-5}
Depth 4	8.508×10^{-2}	8.877×10^{-2}	7.089×10^{-6}	4.459×10^{-4}	7.298×10^{-2}
ReLU	**8**	**16**	**32**	**64**	**128**
Depth 1	1.048×10^{-2}	1.888×10^{-2}	1.452×10^{-2}	2.223×10^{-2}	8.919×10^{-4}
Depth 2	7.429×10^{-2}	3.094×10^{-5}	1.390×10^{-5}	1.720×10^{-5}	1.345×10^{-5}
Depth 3	9.350×10^{-5}	3.628×10^{-5}	8.690×10^{-6}	2.052×10^{-5}	3.680×10^{-6}
Depth 4	3.753×10^{-5}	7.431×10^{-2}	5.727×10^{-6}	5.609×10^{-6}	1.850×10^{-5}
Leaky ReLU	**8**	**16**	**32**	**64**	**128**
Depth 1	1.108×10^{-4}	6.861×10^{-4}	4.701×10^{-5}	6.255×10^{-4}	1.037×10^{-4}
Depth 2	1.716×10^{-4}	2.320×10^{-4}	1.670×10^{-5}	8.627×10^{-6}	7.815×10^{-6}
Depth 3	1.771×10^{-5}	1.498×10^{-5}	2.340×10^{-6}	1.065×10^{-4}	3.445×10^{-4}
Depth 4	2.116×10^{-5}	3.437×10^{-6}	1.769×10^{-5}	6.201×10^{-6}	1.541×10^{-6}

Table VI: Test MSE across depth (rows) and width (columns) for neural networks using Leaky ReLU, ReLU, and Sigmoid activations.

Algorithm 1 Pyramid-scheme layer generation for feed-forward NN

```
Require: Initial width $W_1$, number of layers $D$, decay ratio $r$, minimum width $W_{\min}$
Ensure: Layer width list $\{W_1, W_2, \dots, W_D\}$
1: $W_{\text{prev}} \leftarrow W_1$
2: for $k = 1$ to $D$ do
3:     $W_k \leftarrow \max(\lceil W_1 \, r^{\,k-1} \rceil, W_{\min})$
4:     $W_k \leftarrow \min(W_k, W_{\text{prev}})$                ▷ Prevent increasing widths
5:     $W_{\text{prev}} \leftarrow W_k$
6: end for
7: return $\{W_1, W_2, \dots, W_D\}$
```

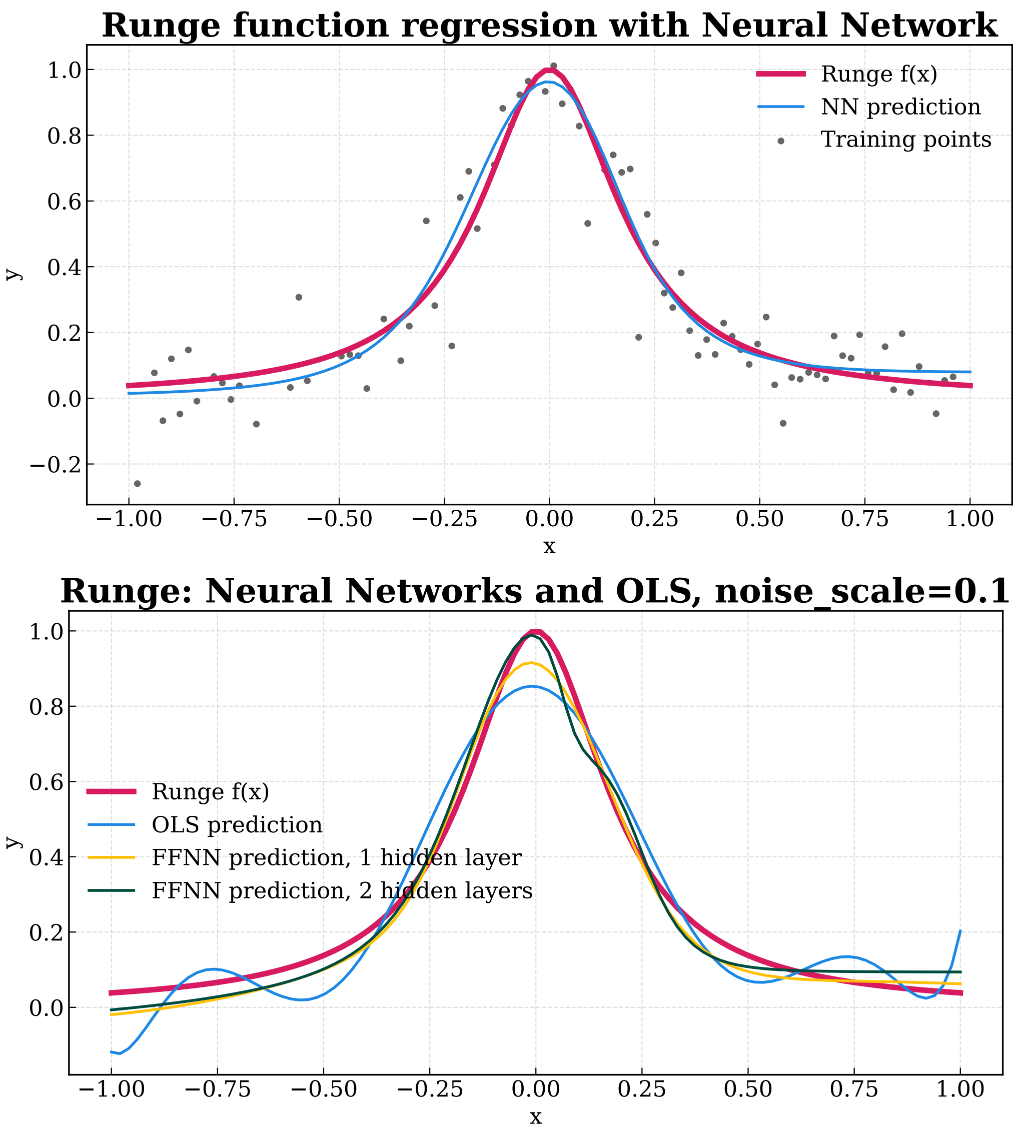


Figure 16: Approximation of the Runge function using neural networks and OLS. **Top:** The true Runge function and its characteristic steep curvature near the edges. **Bottom:** Comparison of neural network regression and OLS, illustrating how NN avoids oscillation artifacts associated with polynomial fits and better captures boundary behavior.

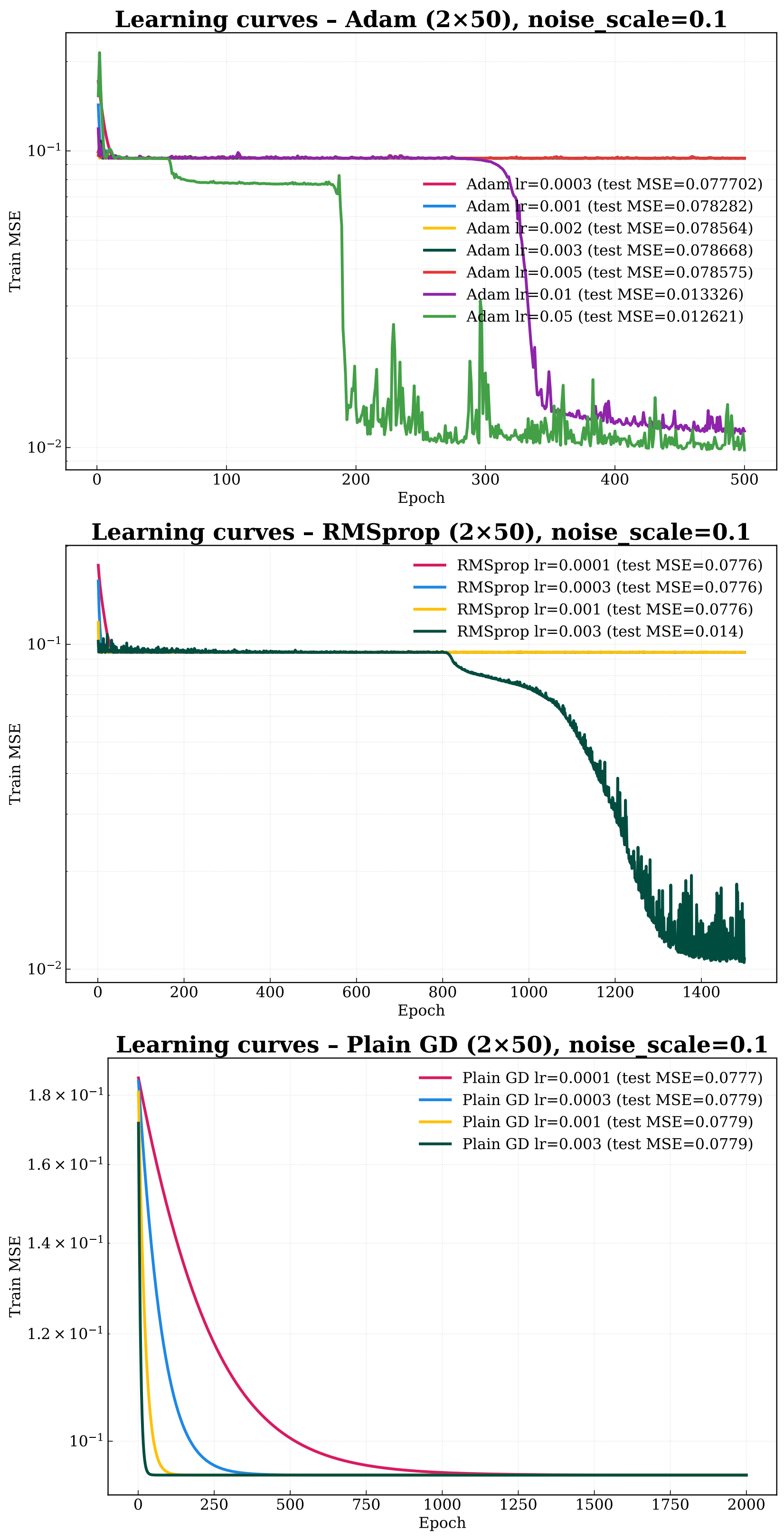


Figure 17: Learning curves for ADAM, RMSProp, and plain gradient descent across different learning rates. ADAM converges rapidly and consistently reaches the lowest error over a broad range of learning rates. RMSProp provides competitive performance but exhibits more oscillatory dynamics at moderate-to-large η. Plain gradient descent converges slowly and is highly sensitive to learning rate, failing to match the stability or accuracy of adaptive optimizers.

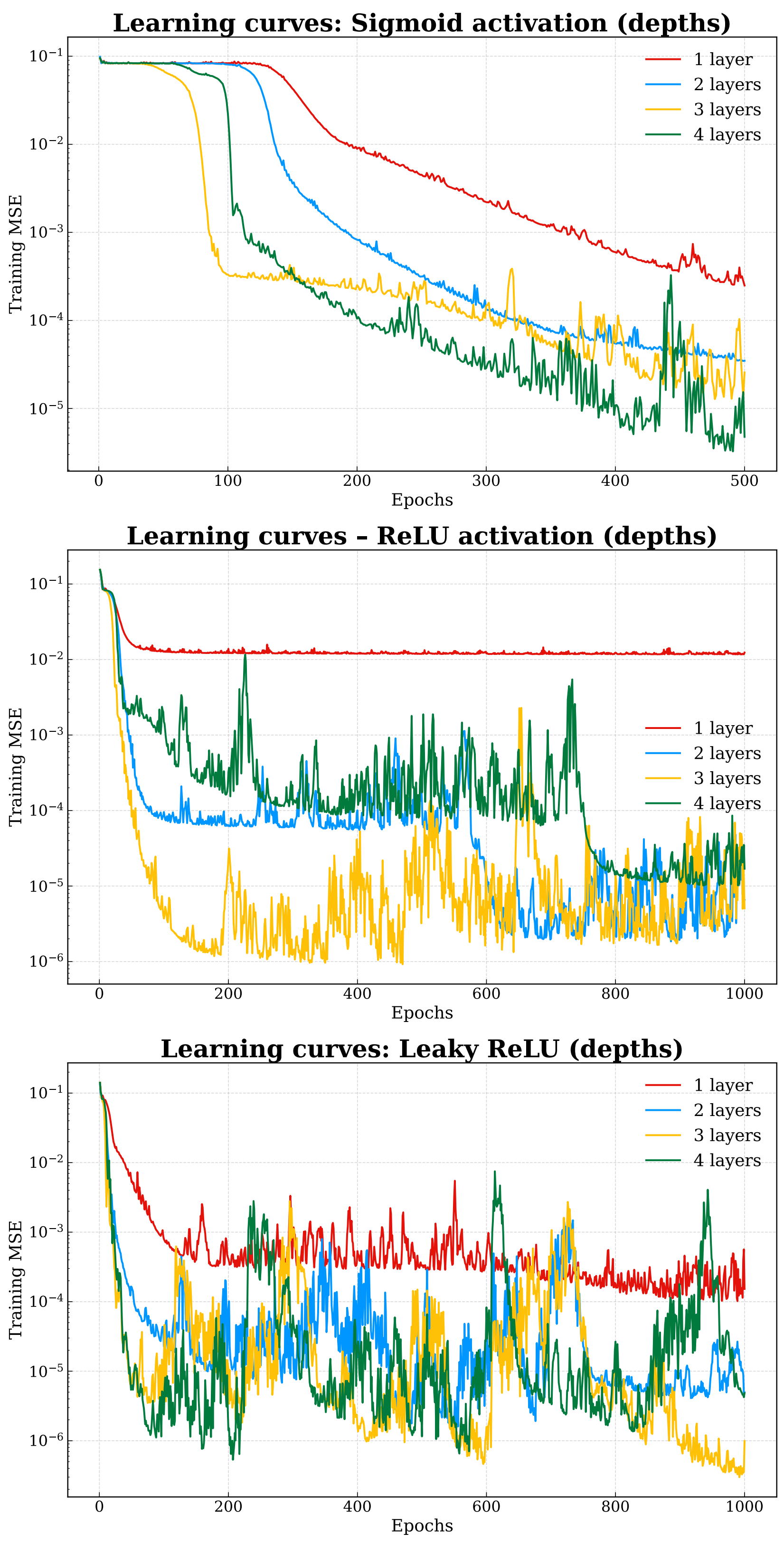


Figure 18: Training curves for sigmoid, ReLU, and Leaky ReLU activation functions with increasing network depth (1–4 hidden layers). Sigmoid networks converge slowly and suffer from vanishing gradients, leading to early plateaus in shallow architectures and delayed learning in deeper ones. ReLU provides faster initial convergence and stronger gradient signals, but training becomes slightly noisy for deeper models. Leaky ReLU avoids the dying-ReLU problem and achieves the lowest overall training MSE, with stable convergence even at greater depth. Across all activations, deeper networks ultimately achieve lower error, demonstrating the representational benefits of depth when gradients propagate effectively.

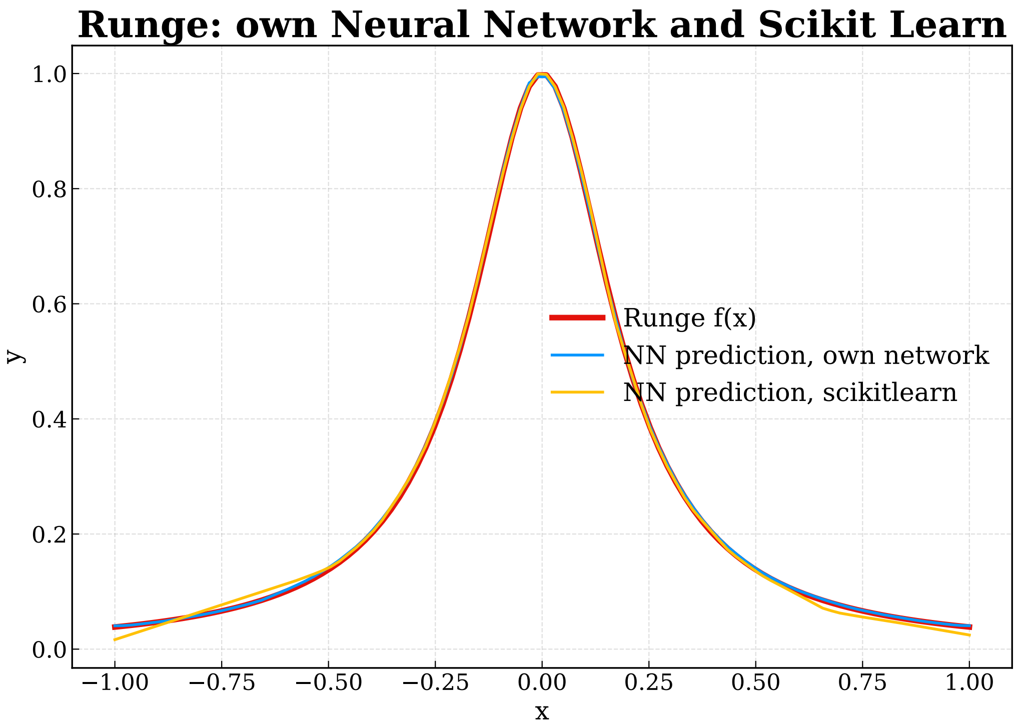


Figure 19: Runge function regression using our FFNN versus `scikit-learn`. We compare the fitted Runge function using our own neural network implementation and the `scikit-learn` `MLPRegressor`. Both neural networks capture the non-linear shape of the Runge function accurately, demonstrating that our implementation matches the behaviour of a widely-used machine learning library.

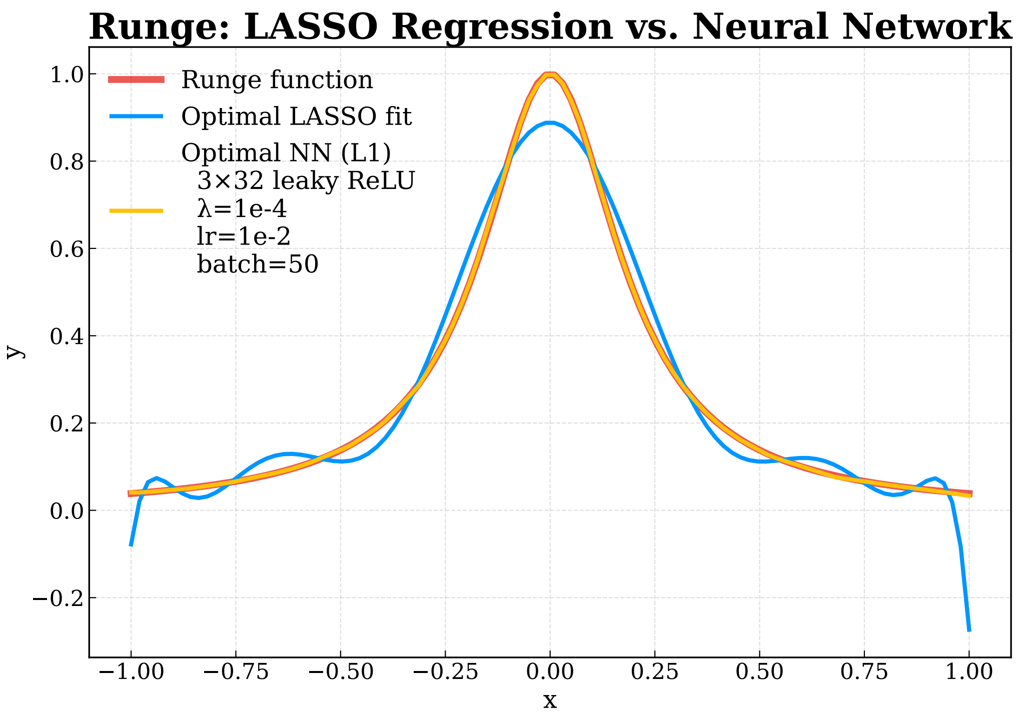


Figure 20: Comparison of the Runge function fitted with LASSO regression and a neural network using leaky ReLU. LASSO captures the main trend but struggles near regions of high curvature due to its linear basis. The neural network tracks the nonlinear behavior more accurately across the entire domain, demonstrating superior expressive power on this problem.

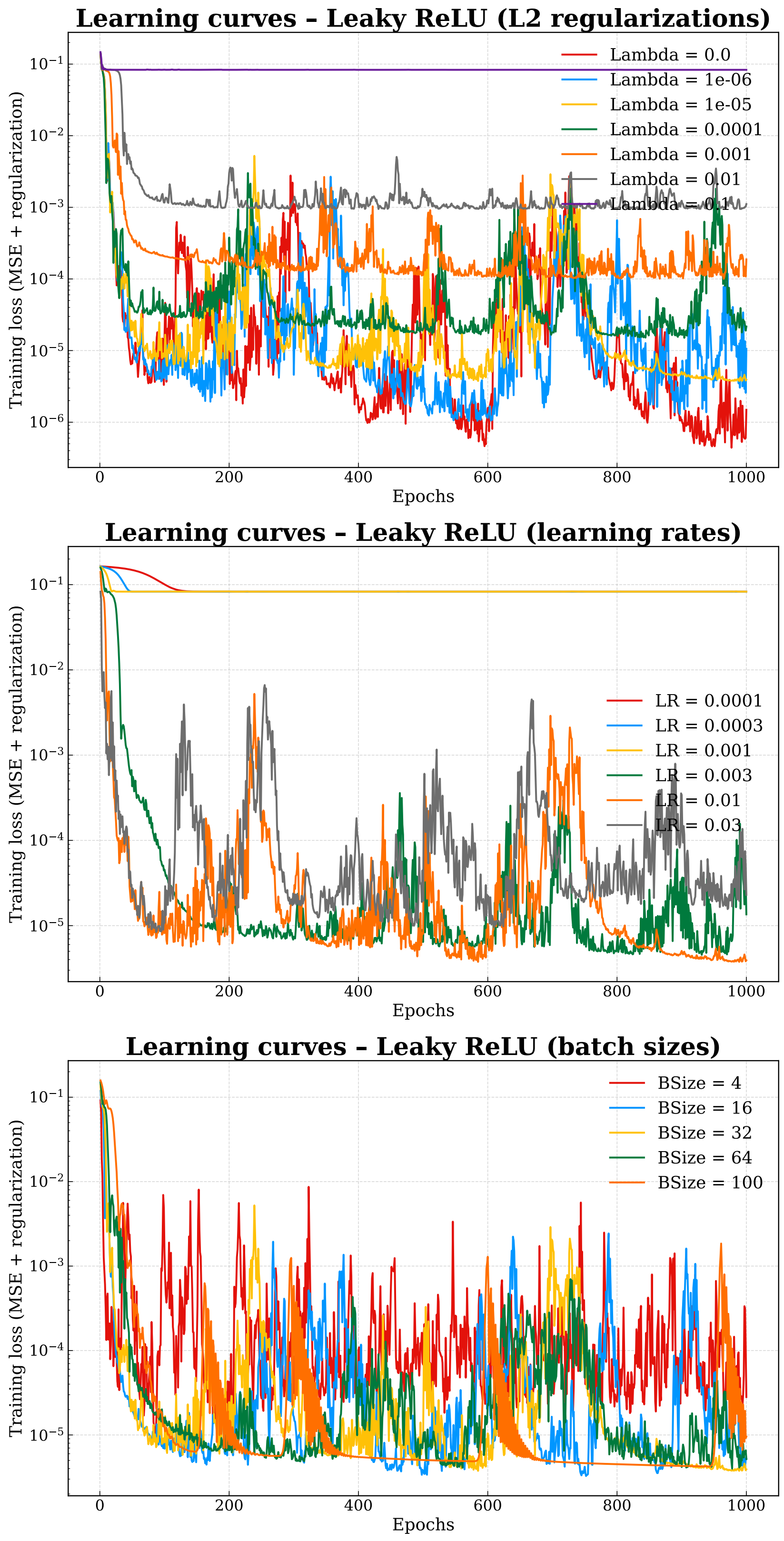
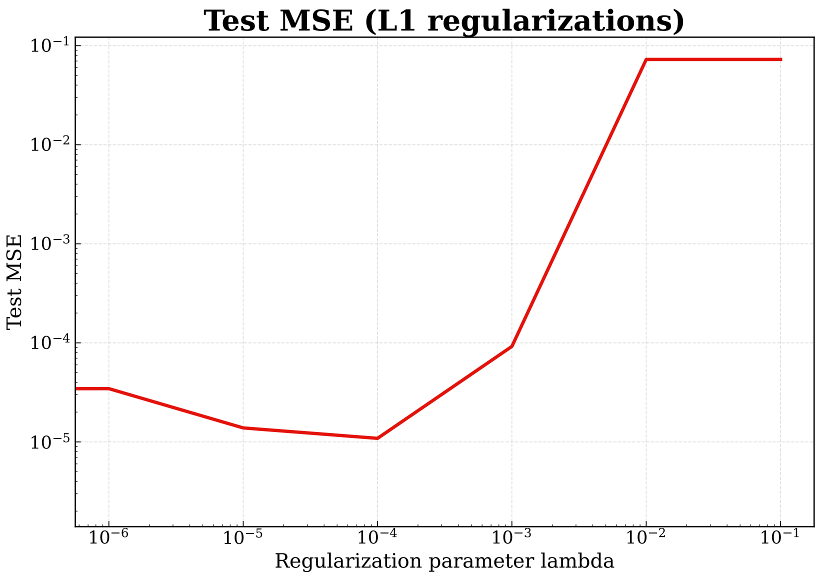
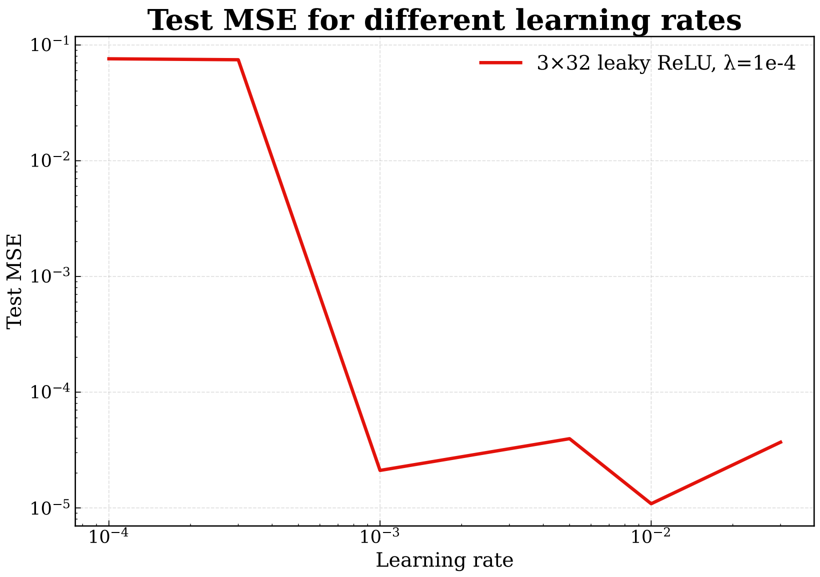


Figure 21: Training curves for Leaky ReLU networks under variations of three key hyperparameters. **Top:** Different L2 regularization strengths. Small regularization values reduce oscillations and improve stability, while overly strong regularization (e.g. $\lambda = 0.1$) prevents the model from fitting the data. **Middle:** Different learning rates. Extremely small η yields slow convergence; moderate η (10^{-3}–10^{-2}) converges quickly and stably; very large η causes oscillations and overshooting. **Bottom:** Different mini-batch sizes. Small batches produce noisy updates, while large batches converge smoothly but may reduce optimization speed. Batch sizes between 16 and 64 provide the best balance.





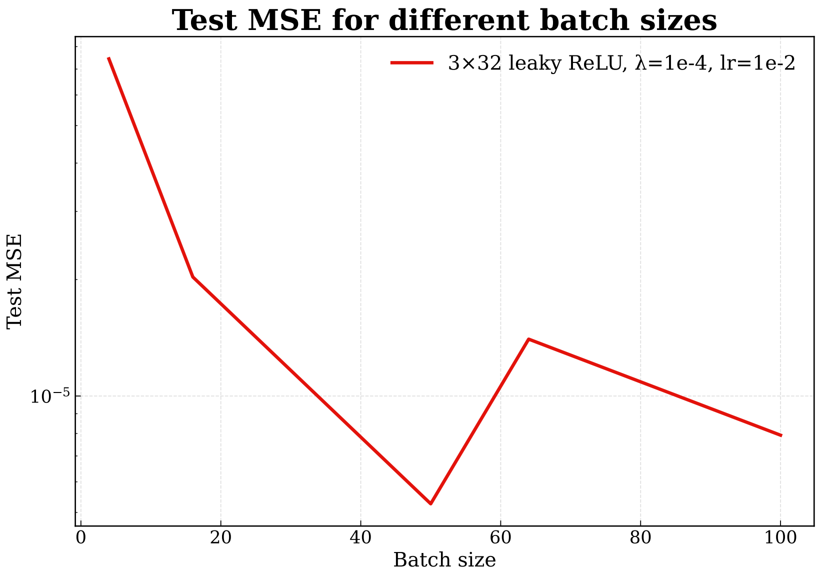


Figure 22: Test MSE for Leaky ReLU networks under variations of L1 regularization strength, learning rate, and mini-batch size. **Top:** L1 regularization. Mild regularization improves generalization, while large λ values drive weights toward zero, causing severe underfitting and sharp increases in test error. **Middle:** Learning rates. Learning rates near 10^{-3} achieve the lowest test MSE, whereas both very small and very large η degrade performance. **Bottom:** Batch size. Small batches generalize poorly due to noisy gradient estimates, and excessively large batches reduce gradient diversity. Medium batch sizes (roughly 32–64) provide the best generalization balance.

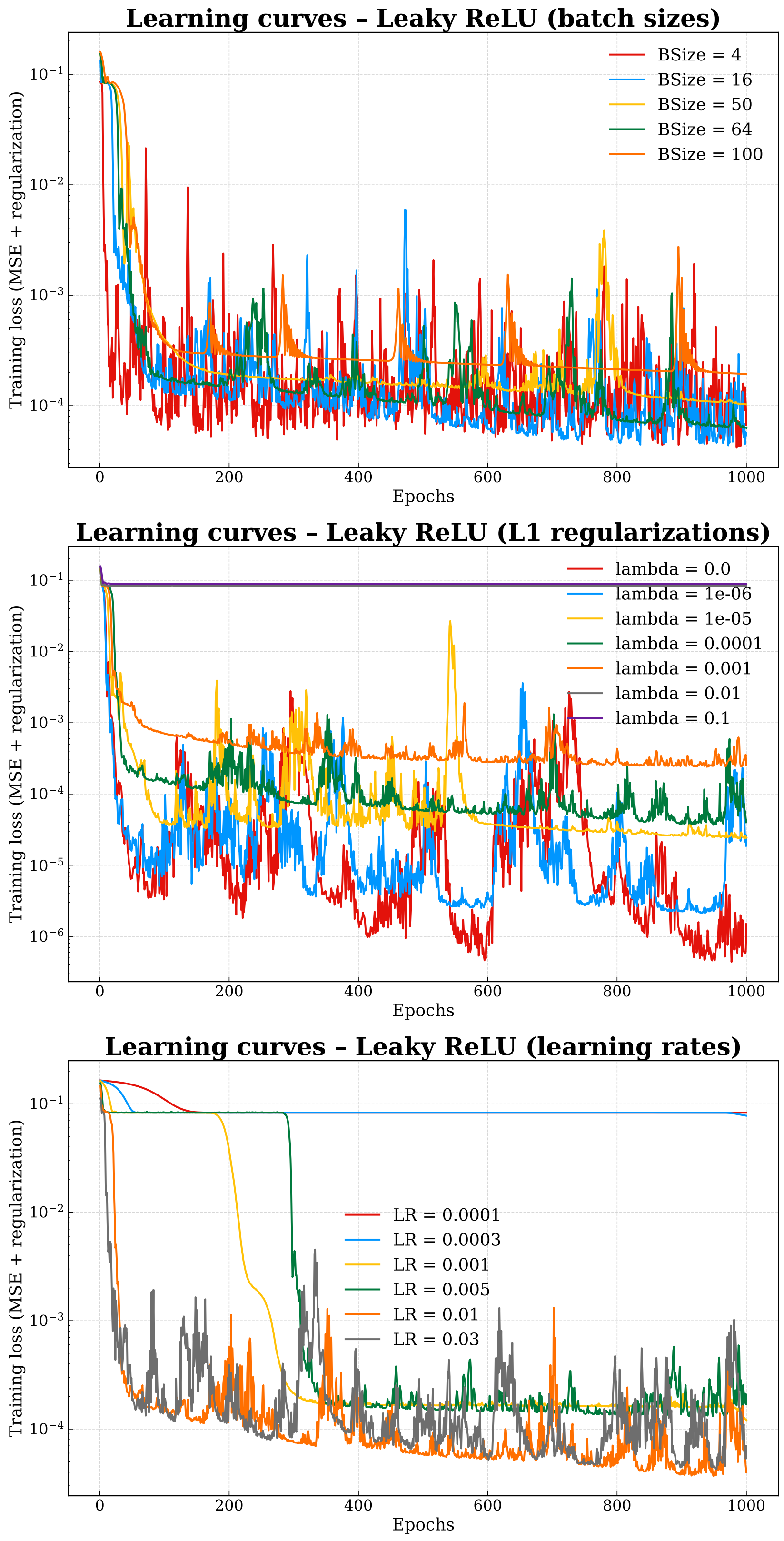


Figure 23: Training curves for Leaky ReLU networks under variations of mini-batch size, L1 regularization strength, and learning rate. **Top:** Batch size. Extremely small batches (e.g., 4) result in noisy optimization due to high variance in gradient estimates, while large batches converge smoothly but may slow down learning. Batch sizes between 16 and 64 offer a good balance of stability and speed. **Middle:** L1 regularization. Mild L1 regularization promotes generalization, whereas strong penalties drive weights toward zero, slowing convergence and degrading performance. **Bottom:** Learning rate. Very small rates converge slowly, moderately large rates (10^{-3}–10^{-2}) provide fast and stable descent, and extremely large rates (e.g., 3×10^{-2}) cause oscillations from overshooting.

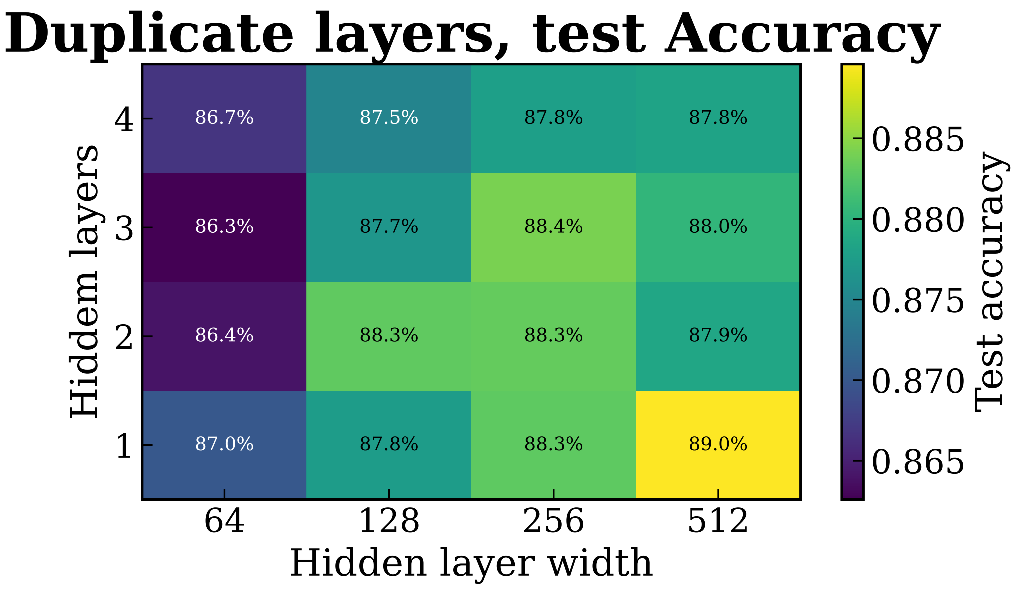


Figure 24: Test accuracy heatmap for architectures with same-sized hidden layers. Each point represents a network trained for 60 epochs with a batch size of 128 using the Adam optimizer (lr $= 5e-4$). The base cost function was categorical cross-entropy with logits and an L_2 penalty of $\lambda = 1 \times 10^{-5}$ (applied to weights only). All networks used ReLU activation in hidden layers. The results show that while deeper networks with uniform width can achieve strong performance, accuracy saturates beyond a certain width, and the single-layer 512-node model remains competitive.

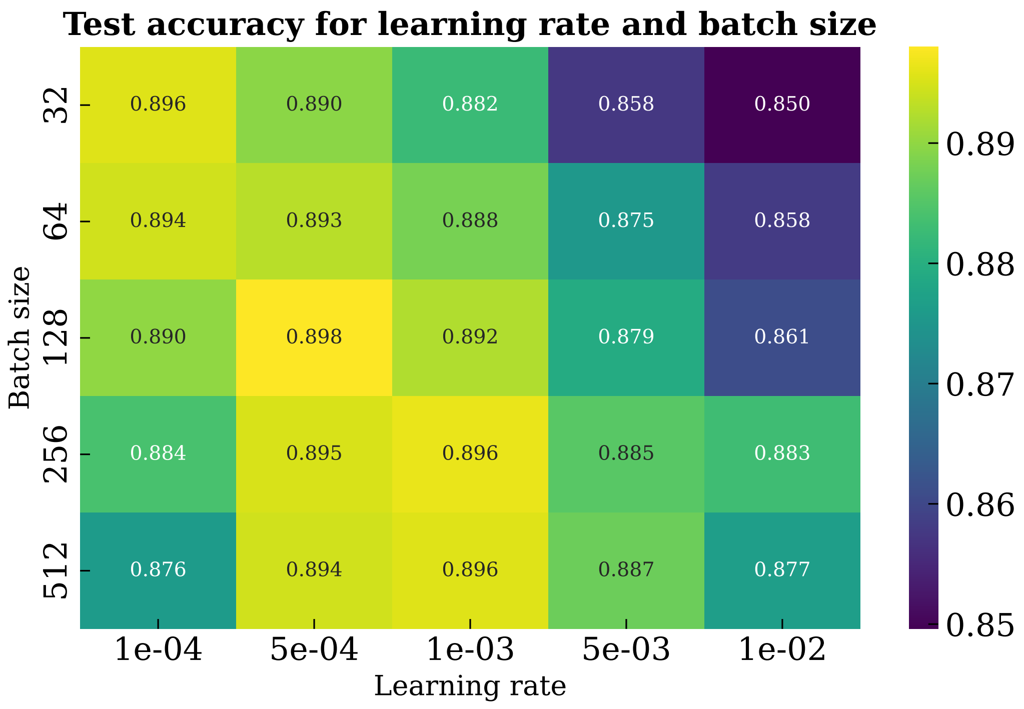


Figure 25: Test accuracy as a function of learning rate and batch size. Each cell corresponds to a network trained for 60 epochs with architecture [512, 10], ReLU activation, and L_2 regularization $\lambda = 1 \times 10^{-5}$. Optimization was performed using Adam. The highest accuracy was observed around a learning rate of 5e-4 and a batch size of 128. Larger batch sizes slightly reduced generalization, likely due to reduced gradient noise, while smaller batches introduced higher variance but slower convergence.

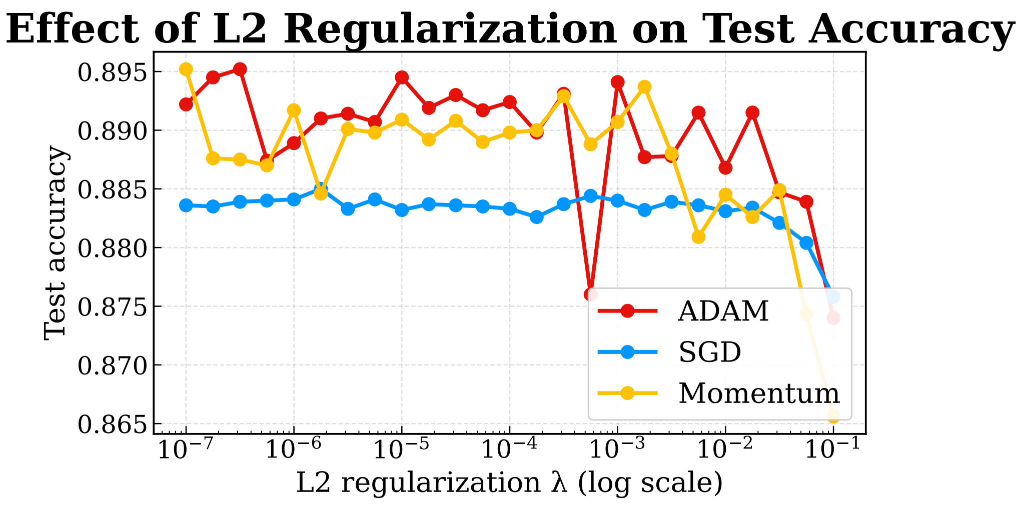
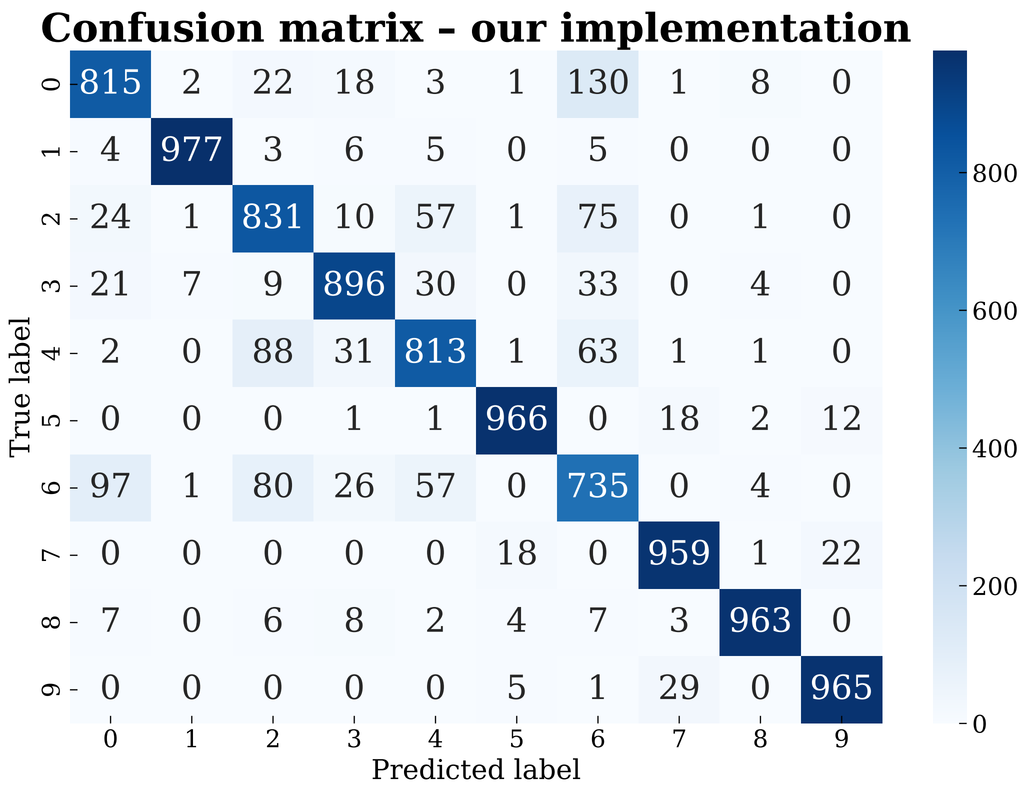


Figure 26: Effect of optimizer type and regularization strength on test accuracy. Each curve corresponds to a different optimizer (Adam, Momentum, or Stochastic Gradient Descent) for the same network architecture [512, 10] trained with 60 epochs and a batch size of 128. The L_2 regularization parameter λ was varied logarithmically. All runs used categorical cross-entropy with logits. The best result was achieved with the Adam optimizer at $\lambda = 3.2 \times 10^{-7}$, though $\lambda = 5 \times 10^{-5}$ gave comparable accuracy with more stable training.



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	815	2	22	18	3	1	130	1	8	0
1	4	977	3	6	5	0	5	0	0	0
2	24	1	831	10	57	1	75	0	1	0
3	21	7	9	896	30	0	33	0	4	0
4	2	0	88	31	813	1	63	1	1	0
5	0	0	0	1	1	966	0	18	2	12
6	97	1	80	26	57	0	735	0	4	0
7	0	0	0	0	0	18	0	959	1	22
8	7	0	6	8	2	4	7	3	963	0
9	0	0	0	0	0	5	1	29	0	965

Figure 27: Non-normalized confusion matrix for the FashionMNIST classification task. The network was trained for 60 epochs using the architecture [512, 10], ReLU activation, Adam optimizer (lr $= 5e-4$

), and L_2 penalty $\lambda = 1 \times 10^{-5}$ with a batch size of 128. The matrix shows the absolute number of test samples per predicted and true class. Compared with the normalized confusion matrix in Figure 14, this plot highlights how class imbalance affects total misclassification counts.

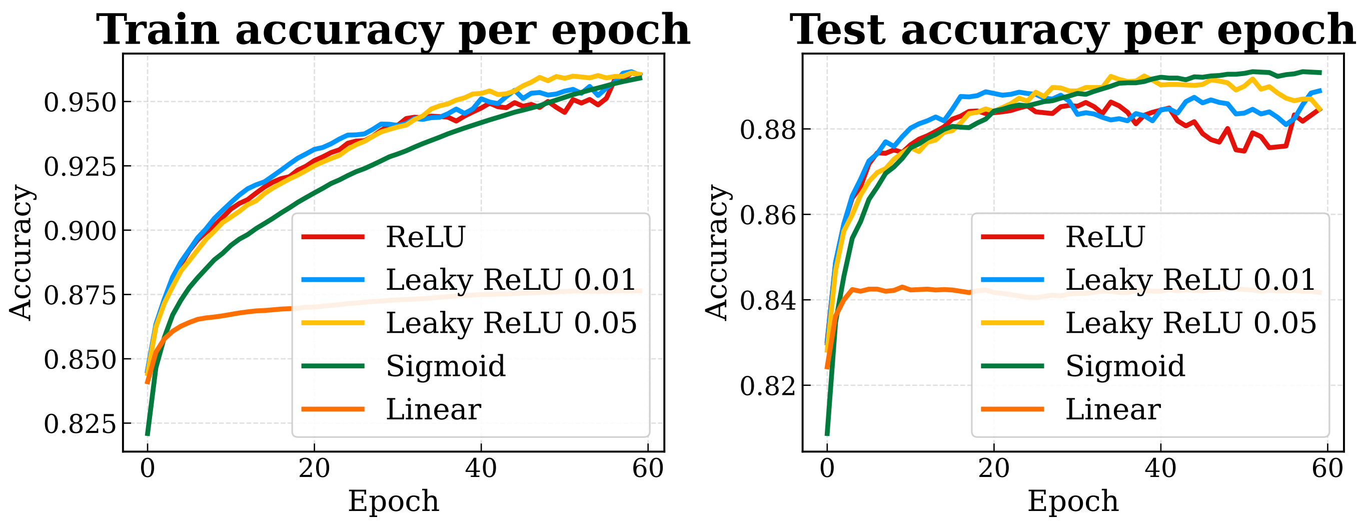


Figure 28: Test accuracy over 60 epochs for different activation functions, including ReLU, Leaky ReLU ($\alpha = 0.01$), Leaky ReLU ($\alpha = 0.05$), Sigmoid, and Linear. Each model was trained on the FashionMNIST dataset with the same configuration used in the main experiments: architecture [512-10], Adam optimizer ({lr $= 5\times10^{-4}$), and L_2 regularization ($\lambda = 1 \times 10^{-5}$). The results show that ReLU and Leaky ReLU activations reach higher accuracies early in training, reflecting faster convergence, but their performance fluctuates due to training instability. In contrast, the Sigmoid activation converges more gradually yet continues improving, eventually matching or slightly surpassing the ReLU-based models in final accuracy.