Applied Data Analysis and Machine Learning
FYS-STK3155 - Project 2

See Gek Cheryl Ong, Boon Kiat Khaw, Naden Jarel Anthony Koh
University of Oslo

(Dated: October 31, 2025)

The abstract gives the reader a quick overview of what has been done and the most important results. Try to be to the point and state your main findings. It could be structured as follows:

- Short introduction to topic and why its important
- Introduce a challenge or unresolved issue with the topic (that you will try to solve)
- What have you done to solve this
- Main Results
- The implications

I. INTRODUCTION

Neural networks are computational networks inspired by the human brain. In the fields of statistics and data science, it is often used in machine learning algorithms to recognise patterns in data and make decisions. Like a human brain, it consists of layers of interconnected nodes, or "neurons" which transmit information. Training the network enables it to perform complex tasks such as classification and regression analysis. These two tasks will be the main focus of this study. In particular, we will be zooming in on the implementation on the Feedforward Neural Network (FFNN) and analysing its performance in regression and classification tasks. In the process, we will be testing different algorithms, activation functions and optimisation methods. For regression, we will first begin with a one-dimensional Runge function and we will explore the classification task using the MNIST dataset later on. By the end of this study, we aim to gain a deeper understanding of how neural networks work, and identify the best performing algorithm for regression and classification.

II. METHODS

A. Method 1/X

In part a, we derive the analytical expressions for the activation functions and loss functions that will be used throughout the project. Understanding these mathematical foundations is crucial, as they form the basis for the backpropagation algorithm and optimization processes used in neural networks.

Three activation functions were analyzed: Sigmoid, ReLU, and Leaky ReLU.

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

The Sigmoid function maps inputs into the range $(0,1)$ and is particularly suitable for probabilistic outputs in binary classification. However, it can suffer from vanishing gradients for large $|z|$ values.

$$f(z) = \max(0,z), \qquad f'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

ReLU is computationally efficient and mitigates the vanishing gradient problem, but can lead to "dead neurons" when weights push activations permanently below zero.

$$f(z) = \begin{cases} z, & z > 0 \\ \alpha z, & z \leq 0 \end{cases} \qquad f'(z) = \begin{cases} 1, & z \geq 0 \\ \alpha, & z < 0 \end{cases}$$

where α is a small positive constant. This function alleviates the ReLU dead-neuron problem by maintaining a small gradient for negative z.

$$L_{MSE} = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2, \qquad \frac{\partial L_{MSE}}{\partial \hat{y}_i} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)$$

For regression problems, the MSE measures the average squared difference between predicted and true values.

Including L_1 and L_2 regularization terms, the full cost becomes:

$$L = L_{MSE} + \lambda_1\|W\|_1 + \frac{\lambda_2}{2}\|W\|_2^2,$$

where $\|W\|_1 = \sum|w|$ and $\|W\|_2^2 = \sum w^2$. The gradient with respect to W is given by:

$$\frac{\partial L}{\partial W} = \frac{\partial L_{MSE}}{\partial W} + \lambda_1 \operatorname{sign}(W) + \lambda_2 W.$$

The Binary Cross Entropy loss measures the discrepancy between predicted probabilities and actual binary labels. We define the likelihood function as:

$$L(\theta) = P(D|\theta) = \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{1-y_i}, \quad \text{where } p_i = \hat{y}_i.$$

Taking the negative log-likelihood gives the cost function:

$$C(\theta) = -\log L(\theta) = -\sum_{i=1}^{n} \left[y_i \log p_i + (1 - y_i) \log(1 - p_i) \right].$$

To compute the gradient, we apply the chain rule:

$$\frac{\partial C}{\partial \theta} = \frac{\partial C}{\partial p_i} \frac{\partial p_i}{\partial z_i} \frac{\partial z_i}{\partial \theta}.$$

Expanding each term gives:

$$\frac{\partial C}{\partial \theta} = \sum_{i=1}^{n} \left(-\frac{y_i}{p_i} + \frac{1 - y_i}{1 - p_i} \right) \sigma'(z_i) x_i,$$

where $\sigma'(z_i) = p_i(1 - p_i)$ is the derivative of the sigmoid activation. Simplifying this expression, we obtain:

$$\frac{\partial C}{\partial \theta} = \sum_{i=1}^{n} (p_i - y_i) x_i.$$

In compact vector form:

$$\boxed{\frac{\partial C}{\partial \theta} = (p - y)x.}$$

Including L_1 and L_2 regularization terms, the final gradient becomes:

$$\frac{\partial L_{\text{Regularization}}}{\partial w} = \frac{\partial L_{\text{BCE}}}{\partial w} + \lambda_1 \operatorname{sign}(w) + \lambda_2 w.$$

For multi-class classification, each input i produces a vector of logits $z^{(i)} = [z_1^{(i)}, z_2^{(i)}, \ldots, z_c^{(i)}]$, one for each class $k \in \{1, \ldots, c\}$. The predicted class probabilities are obtained by applying the Softmax function:

$$p_k^{(i)} = \text{Softmax}_k(z^{(i)}) = \frac{e^{z_k^{(i)}}}{\sum_{j=1}^{c} e^{z_j^{(i)}}}.$$

The likelihood of the dataset under the model parameters θ is

$$L(\theta) = \prod_{i=1}^{n} \prod_{k=1}^{c} \left(p_k^{(i)} \right)^{y_{ik}},$$

where y_{ik} is a one-hot encoded label indicating the true class.

Taking the negative log-likelihood yields the Softmax cross-entropy cost function:

$$C(\theta) = -\log L(\theta) = -\sum_{i=1}^{n} \sum_{k=1}^{c} y_{ik} \log p_k^{(i)}.$$

To find the gradient with respect to the logits $z_j^{(i)}$, we use:

$$\frac{\partial C}{\partial z_j^{(i)}} = \sum_{k=1}^{c} \frac{\partial C}{\partial p_k^{(i)}} \frac{\partial p_k^{(i)}}{\partial z_j^{(i)}}.$$

Simplifying, we obtain:

$$\frac{\partial C}{\partial z_j^{(i)}} = p_j^{(i)} - y_{ij}.$$

Thus, in vector form:

$$\boxed{\frac{\partial C}{\partial Z} = P - Y,}$$

where P is the predicted probability matrix and Y the true label matrix.

In Part B, we first define and implement a simple feed-forward neural network to approximate the Runge function, comparing its performance with the OLS baseline.

The weights are initialized from a standard normal distribution and the biases are set to a small constant $b = 0.01$, avoiding the symmetry issues that occur with zero-initialization. Each layer computes:

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

where $\sigma(\cdot)$ is the activation function.

Backpropagation computes the gradients of each parameter using:

$$\frac{\partial z_k^{(\ell+1)}}{\partial z_j^{(\ell)}} = W_{kj}^{(\ell+1)} \sigma'(z_j^{(\ell)}), \qquad \delta_j^{(\ell)} = \sum_k \delta_k^{(\ell+1)} W_{kj}^{(\ell+1)} \sigma'(z_j^{(\ell)}).$$

During training, the weights and biases are updated by gradient descent:

$$W_{jk}^{(\ell)} = W_{jk}^{(\ell)} - \eta\, \delta_j^{(\ell)} a_k^{(\ell-1)}, \qquad b_j^{(\ell)} = b_j^{(\ell)} - \eta\, \delta_j^{(\ell)},$$

where η is the learning rate.

The Runge function's output range $(0, 1)$ makes both Sigmoid and ReLU suitable choices. Although Sigmoid is typically used for binary classification, it provides a smooth, bounded response compatible with this regression task. Because the FFNN can learn nonlinear features directly, no polynomial design matrix is required — unlike OLS, which depends on polynomial degree selection.

We first establish the OLS benchmark by varying the polynomial degree $d \in [1, 23]$ and selecting the degree that minimizes the test MSE. Figure 1 shows the train and test MSE curves for different polynomial degrees, with the minimum test MSE occurring at $d = 9$ (**Test MSE = 0.0043**), which serves as the baseline for all subsequent comparisons.

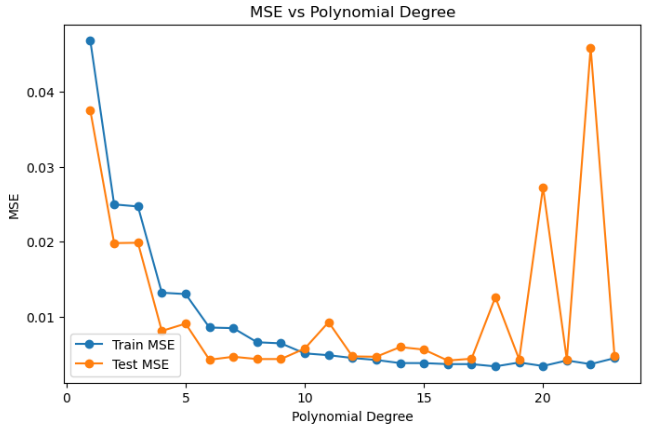


Figure 1: **T**rain and test MSE as a function of polynomial degree for the 1D Runge function using Ordinary Least Squares (OLS). The test MSE follows a U-shaped trend, reaching its minimum at degree 9, which is therefore used as the OLS baseline for comparison with neural network models.

In Part E, to evaluate the effect of regularization on model performance, we first establish baseline benchmark errors using classical **Ridge** and **LASSO** regression, before extending these concepts to our neural network. Ridge applies an L_2 penalty on large weights, whereas LASSO applies an L_1 penalty that promotes sparsity. Both methods act as references for the neural network with integrated L_1 and L_2 regularization.

We then modify our RMSProp-based neural network by adding both regularization terms directly to the loss function:

$$\text{Total Loss} = \text{Base Loss} + \lambda_1 \sum |W| + \lambda_2 \sum W^2$$

and adjust the weight gradients during backpropagation as:

$$\Delta W = \Delta W + \lambda_1 \operatorname{sign}(W) + 2\lambda_2 W$$

Iterating over logarithmically spaced values of $\lambda_1, \lambda_2 \in [0, 1]$, we determine the optimal strengths that minimize the Test Mean Squared Error (MSE).

Part C validates our custom neural network implementation. To do so, we compare it against the **TensorFlow/Keras** library using an equivalent architecture and training setup. A sequential model is constructed with the same number of hidden layers, neuron counts, activation functions, and optimizer (`RMSprop`). Each model is trained for 500 epochs on the two-dimensional Runge function dataset, while the learning rate η is varied across $\{0.0001, 0.001, 0.01, 0.1\}$ to identify the optimal configuration.

The Keras network employs the Glorot/Xavier weight initialization scheme, which scales the initial variance according to the number of neurons in each layer. In contrast, our custom network uses a fixed standard-normal initialization with biases set to 0.01. This difference in initialization strategy provides an opportunity to evaluate how adaptive scaling affects convergence and generalization.

In Part D, we investigate how architectural design choices affect neural-network performance on the 2-D Runge function. Specifically, we examine:

[label=(i)]

1. **Activation functions:** homogeneous (`sigmoid`, `ReLU`, `Leaky ReLU`) and mixed combinations of activations across layers.

2. **Depth:** the number of hidden layers ($L = 1$–15) while fixing width at 50 neurons per layer.

3. **Width:** the number of neurons per layer (10–300) while fixing depth at 5 layers.

4. **Joint depth–width search:** simultaneous variation of both hyperparameters to produce a 2-D bias–variance heatmap.

Each configuration is trained for 500 epochs using the RMSprop optimizer ($\eta = 0.01$, $\rho = 0.9$) and the Leaky ReLU–Sigmoid–Sigmoid combination unless otherwise stated. All inputs are standardized, and results are averaged over 50 bootstrap resamples to estimate test error (MSE), bias^2, and variance.

1. *Part F: Feedforward Neural Network for MNIST Classification*

We implemented a Feedforward Neural Network (FFNN) for classifying handwritten digits from the MNIST database, using the softmax cross-entropy loss function as defined in earlier sections. The objective was to correctly classify each 28×28 pixel image into its corresponding digit class (0–9).

The MNIST dataset consists of 70,000 images—60,000 for training and 10,000 for testing—with a standard 20% validation split from the training data. All input features were normalized to the range $[0, 1]$ to improve gradient-based optimization performance.

Architecture and Configuration: The neural network followed the structure 784 → 128 → 64 → 10, where 784 corresponds to the flattened input pixels, two hidden layers contain 128 and 64 neurons respectively, and the output layer represents the 10 digit classes. Hidden layers employed the Leaky ReLU activation function to mitigate the vanishing gradient problem, while the output layer used Softmax activation for multi-class probability distribution. The model was trained with a learning rate of 0.001 using gradient descent optimization.

Training Procedure: The network was initialized with randomized weights and trained for 30 epochs. Each epoch involved forward propagation to compute predictions and loss, followed by backpropagation to update

weights. Accuracy and training loss were monitored throughout to track convergence.
Evaluation: Final model performance was evaluated on the test set using classification accuracy, confirming that the implementation reproduced expected learning behavior and performance trends for an FFNN on MNIST classification.

2. Part G: Algorithm Summary and Comparison

We implemented and analyzed multiple algorithms for regression and classification tasks. For regression, these included Ordinary Least Squares (OLS), OLS with Adam optimization, and neural networks trained with Gradient Descent, Stochastic Gradient Descent (SGD), RMSProp, and Adam. For classification, we implemented Logistic Regression and neural networks with Softmax output layers, experimenting with various activation functions including Sigmoid, ReLU, Leaky ReLU, and Softmax.
Key Findings:

- **Regression:** Neural networks trained with adaptive optimizers (Adam or RMSProp) performed best, providing stable training and effectively capturing non-linear patterns that OLS could not.
- **Classification:** Neural networks with Softmax outputs yielded superior performance, leveraging their ability to learn hierarchical features from raw image data combined with stable probabilistic multi-class outputs.

B. Implementation

In Part B, the FFNN was implemented entirely in `NumPy`. Each core routine is kept within 10 lines, corresponding to initialization, forward propagation, backpropagation, and the Adam update rule.

```
# 1. Create layers and initialize parameters
def create_layers_batch(in_size, out_sizes):
    layers, i = [], in_size
    for o in out_sizes:
        W = np.random.randn(o, i)
        b = np.full((o,1), 0.01)
        layers.append((W, b)); i = o
    return layers

# 2. Forward propagation
def feed_forward_batch(X, layers, acts):
    a = X
    for (W,b), act in zip(layers, acts):
        z = W @ a.T + b
        a = act(z).T
    return a

# 3. Backpropagation
def backpropagation_batch(X, layers, acts,
T, ders):
    A, Z, a = [], [], X
    for (W,b), act in zip(layers, acts):
        A.append(a); Z.append(W @ a.T + b);
        a = act(Z[-1]).T
    grads, delta = [None]*len(layers), None
    for i in reversed(range(len(layers))):
        dCdz = ((a-T)/len(X)
        if i==len(layers)-1
        else delta @ layers[i+1][0]) \
              * ders[i](Z[i]).T
        grads[i] = (dCdz.T @ A[i],
        np.sum(dCdz.T, axis=1, keepdims=True))
        delta = dCdz
    return grads

# 4. Adam optimizer update
def train_network_sgd_adam(X, T,
layers, acts, ders, lr=1e-3, b1=0.9, b2=0.999,
eps=1e-8, epochs=500):
    m = [(np.zeros_like(W),
    np.zeros_like(b)) for (W,b) in layers]
    v = [(np.zeros_like(W),
    np.zeros_like(b)) for (W,b) in layers]
    for ep in range(epochs):
        Y = feed_forward_batch(X, layers, acts)
        grads = backpropagation_batch(X, layers,
        acts, T, ders)
        for i, ((W,b),(dW,db),(mW,mb),
        (vW,vb)) in enumerate(zip(layers,grads,m,v)):
            mW = b1*mW+(1-b1)*dW;
            vW=b2*vW+(1-b2)*(dW**2)
            mb = b1*mb+(1-b1)*db;
            vb=b2*vb+(1-b2)*(db**2)
            W -= lr*mW/(np.sqrt(vW)+eps);
            b -= lr*mb/(np.sqrt(vb)+eps)
            layers[i] = (W,b)
    return layers
```

Part E: **Benchmark: Ridge and LASSO**

```
def ridge_reg(X, y, lam):
    I = np.eye(X.shape[1])
    return np.linalg.pinv(X.T @ X + lam * I)
    @ X.T @ y

beta = ridge_reg(X_train, y_train, lam=0.01)
y_pred = X_test @ beta
print(mse(y_test, y_pred))
```

Neural Network Regularization

```
layers, _ = train_network_sgd_rmsprop_regularization(
    inputs, targets, layers,
    activation_funcs, activation_ders,
    lr=0.001, epochs=300, l1_lambda=1, l2_lambda=2)
plt.plot(lambdas, test_mses, ’o-’)
```

Part C, The Keras model:

```
def create_neural_network_keras_2d(n1, n2,
n_out, eta, lmbd):
    model = Sequential([
        Dense(n1, activation='sigmoid',
        kernel_regularizer=l2(lmbd),
              input_shape=(2,)),
        Dense(n2, activation='sigmoid',
        kernel_regularizer=l2(lmbd)),
        Dense(n_out, activation='sigmoid')
    ])
    model.compile(optimizer=RMSprop
    (learning_rate=eta, rho=0.9),
    loss='mse', metrics=['mse'])
    return model
```

The network is trained using the `fit()` routine with a batch size of 32 and validation data included to monitor both training and test loss over epochs.

Part D:

```
for config in configs:
    layers = create_layers_batch(input_size, layer_sizes)
    layers, _ = train_network_sgd_rmsprop(
        X_train, y_train, layers,
        activations=config['funcs'], act_ders=config['ders'],
        lr=0.01, decay_rate=0.9, epochs=500)
    y_pred = feed_forward_batch(X_test, layers, config['funcs'])
    test_mse = mse(y_test, y_pred)
```

C. Use of AI tools

- ChatGPT was used to support code refinement, documentation, drafting and clarification of results.

III. RESULTS AND DISCUSSION

In Part B, with the use of standardized inputs, the network with architecture `[50, 1]` (Sigmoid + Adam, $\eta = 0.001$) converged steadily from loss $\approx 0.24 \rightarrow 0.041$ and achieved a test MSE of **0.0032**, outperforming OLS. A deeper model `[50, 100, 1]` reached the same test MSE, indicating that the simpler structure already captured the Runge nonlinearity.

Extending the input to (x, y), the same network (`[50, 100, 1]`) achieved train MSE = 0.00093 and test MSE = 0.00136. The training loss curve in Figure 2 shows a smooth, monotonic convergence, which confirms that the network is generalized well without overfitting.

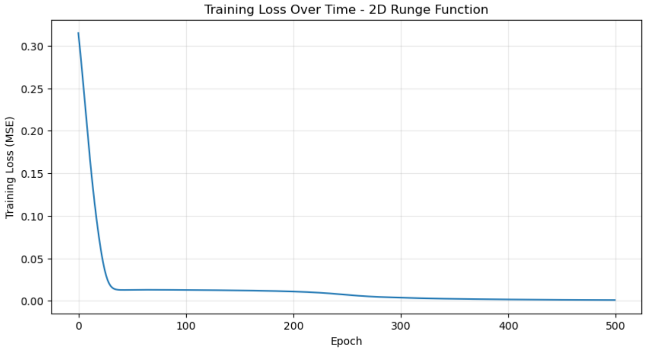


Figure 2: Training loss (MSE) over epochs for a feed-forward neural network with architecture [50, 100, 1] trained using the Adam optimizer on the 2D Runge dataset. The loss decreases smoothly from approximately 0.31 to 0.0009, demonstrating stable convergence and minimal overfitting (train MSE 0.0009, test MSE 0.00136).

We compared four optimizers—Plain GD, SGD + Momentum (0.9), RMSProp (0.9), and Adam—across a range of learning rates. Figure 3 plots train/test MSE versus η for each method on the 2D Runge task, revealing the following best results:

Optimizer	Best LR	Test MSE
Plain GD	0.5	0.00223
SGD Momentum	0.01	0.01223
RMSProp	0.001	**0.00058**
Adam	0.001	0.00136

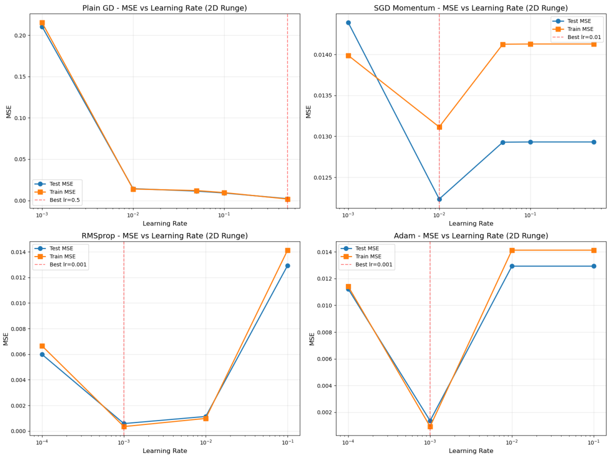


Figure 3: MSE versus learning rate for four gradient-based optimizers (Plain GD, SGD with Momentum, RMSProp, and Adam) on the 2D Runge regression task. Each sub-plot uses a logarithmic x-axis. RMSProp with a learning rate of 0.001 achieves the lowest test MSE (0.000582), followed by Adam (0.00136), confirming the superior stability of adaptive methods for this task.

- Plain GD – simple but slow; requires large η (0.5) to converge.

- SGD + Momentum – smoother convergence but still learning-rate sensitive.
- RMSProp – adaptive per-parameter learning rates; fastest and most stable.
- Adam – combines RMSProp + Momentum; stable but occasionally plateaus when β_1 is high.

In summary, RMSProp with $\eta = 10^{-3}$ gave the best MSE test (**0.00058**), followed by Adam (**0.00136**). These results confirm that adaptive optimizers outperform fixed-step methods for nonlinear regression, and that even a simple FFNN can surpass OLS on both 1D and 2D Runge functions.

In Part E, **Ridge and LASSO Benchmarks:** Ridge regression achieved a **Test MSE of 0.00384**, while the best LASSO configuration using scikit-learn yielded **Test MSE = 0.00112** at degree 15 and $\lambda = 0.0001$. Our custom ISTA implementation gave **Test MSE = 0.00240** at degree 6 and $\lambda = 0.00033$, demonstrating that LASSO's sparsity effect can outperform Ridge when approximating localized variations in the Runge function.

Neural Network Regularization: Figure 4 shows the effect of varying L_2 regularization on the Test MSE, and Figure 5 illustrates the corresponding L_1 trend. The best performance occurs at $\lambda_1 = 0.0001$ and $\lambda_2 = 0.0001$, with Test MSE values of 0.00267 and 0.00277 respectively. Larger penalty values ($\lambda > 0.01$) led to noticeable underfitting and rapidly increasing Test MSE.

Comparing these results with the Ridge and LASSO baselines:

- The L_2-regularized neural network achieved a lower Test MSE than Ridge, indicating effective suppression of weight magnitudes without over-penalization.
- The L_1-regularized network underperformed relative to LASSO, as the harsher sparsity constraint removed useful weight connections.

The U-shaped relationship between λ and MSE in both Figures 4 and 5 highlights the classic bias–variance tradeoff: smaller λ values reduce overfitting, while overly large penalties restrict the model's capacity to learn complex nonlinear patterns.

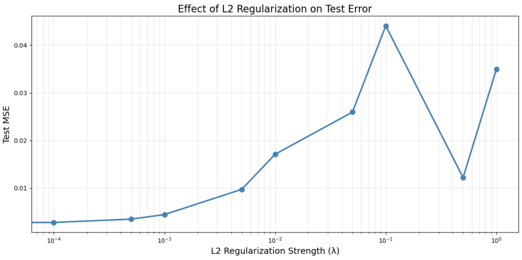


Figure 4: Effect of L_2 Regularization on Test Error. A small λ_2 (10^{-4}) minimizes MSE before the model underfits at higher strengths.

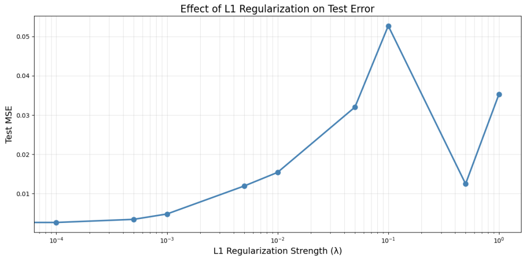


Figure 5: Effect of L_1 Regularization on Test Error. Excessive λ_1 causes weight sparsity and degraded performance.

Overall, our neural network with L_2 regularization achieves comparable or superior generalization performance relative to Ridge regression, while maintaining flexibility in representing the non-polynomial Runge function. The L_1 penalty, although effective in simpler regression settings, proves too restrictive for multi-layer networks that rely on distributed representations.

All subsequent evaluations (Parts C and D) will employ the two-dimensional Runge function and the optimally regularized network configuration derived here.

In Part C, Figure 6 presents the training and validation loss curves for all tested learning rates. The optimal configuration occurs at a learning rate of $\eta = 0.01$, achieving a **Test MSE = 0.000338**. This performance significantly surpasses our custom implementation, with best MSE 0.000927, highlighting the benefits of adaptive weight initialization and numerically optimized gradient updates within TensorFlow.

The observed improvement can be attributed primarily to Keras' *Glorot/Xavier* initialization, which scales weights by layer fan-in and fan-out to maintain stable activation variance during forward propagation. Our fixed-variance initialization, while sufficient for small networks, introduces higher variance in deeper layers and slows early-epoch convergence. Additionally, TensorFlow's internal RMSProp optimizer incorporates highly optimized numerical kernels that improve stability over our manual vectorized implementation.

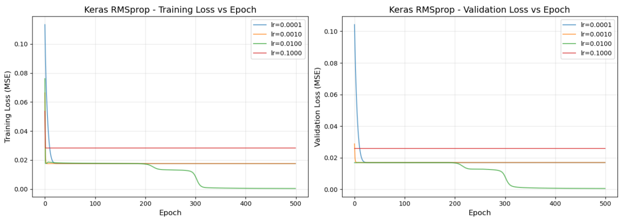


Figure 6: Keras RMSprop training results. (Left) Training Loss vs Epoch. (Right) Validation Loss vs Epoch. The optimal learning rate $\eta = 0.01$ yields the lowest Test MSE (0.000338), clearly outperforming the custom neural network implementation.

In summary, while our custom neural network implementation provides full control over gradient flow, activation functions, and optimization dynamics, standard libraries such as Keras deliver superior computational efficiency and performance due to carefully engineered initialization schemes and automatic gradient handling.

In Part D, Figure 7 compares 10 activation-layer combinations. The homogeneous sigmoid network achieves a smooth, monotonic loss decrease and the best performance among single-activation models, while ReLU and Leaky ReLU display unstable oscillations due to gradient discontinuities. Among mixed activations, the **Leaky ReLU → Sigmoid → Sigmoid** stack attains the lowest test MSE (≈ 0.00109), slightly outperforming the pure sigmoid case. This suggests that the first layer benefits from Leaky ReLU's quasi-linear behaviour, helping gradient propagation before the subsequent sigmoid compression.

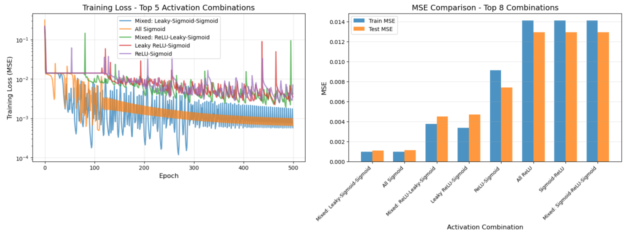


Figure 7: (Left) Training loss for the five best activation combinations. (Right) Train/test MSE for the top eight combinations. The Leaky ReLU–Sigmoid–Sigmoid architecture yields the lowest test MSE.

Next, we conduct the Bias–Variance Analysis Across Depth. Using 50 bootstraps, we train networks with 1–15 hidden layers of 50 neurons each. Figure 8 shows that bias^2 decreases initially but variance increases steadily with depth, producing a characteristic U-shaped test-error curve. Underfitting occurs at very shallow depths (high bias), while overfitting dominates as the network becomes deeper. The optimal trade-off is observed at **two hidden layers**.

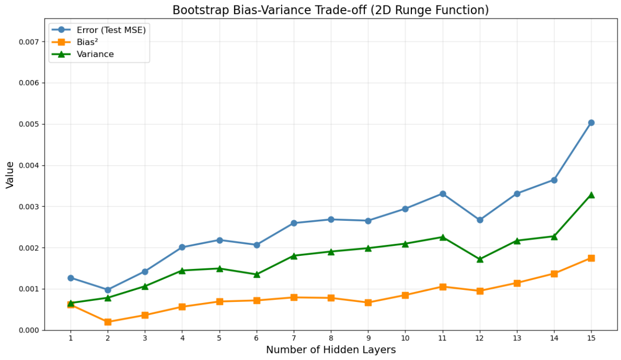


Figure 8: Bootstrap bias–variance–error decomposition versus number of hidden layers. Error is minimized around two layers, indicating an optimal bias–variance balance.

Next, we conduct the Bias–Variance Analysis Across Depth. Keeping the depth fixed at 5 layers, we vary the number of neurons per layer (10–300). As seen in Figure 9, bias^2 decreases slightly at small widths but increases again beyond 100 neurons, while variance grows rapidly. The best generalization occurs at **10–25 neurons per layer**, after which the network's increasing complexity leads to overfitting and optimization instability.

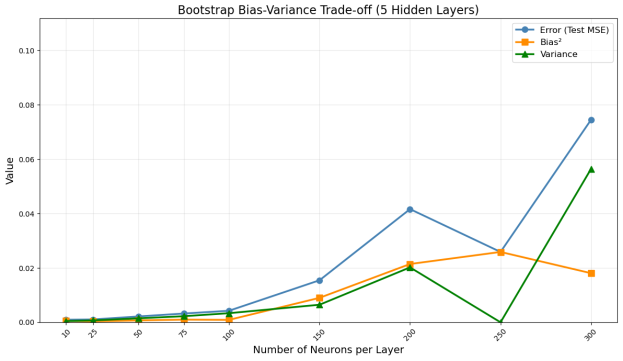


Figure 9: Bootstrap bias–variance–error versus neurons per layer (5 hidden layers). Test MSE remains lowest near 10–25 neurons per layer.

Finally, we examine the Joint Depth–Width Heatmap. We perform a 2-D grid search across hidden layers ($L \in \{1, 2, 3, 4, 5, 6, 8, 10\}$) and neurons per layer ($W \in \{10, 25, 50, 75, 100, 150, 200\}$) with 20 bootstrap iterations per cell. Figure 10 visualizes the resulting test MSE, bias^2, and variance. The optimal region is clearly located at **(2 layers, 10–25 neurons)**, consistent with the individual sweeps. As network complexity increases in either dimension, variance grows monotonically while bias first decreases then rebounds, confirming the classical bias–variance trade-off in over-parameterized models.

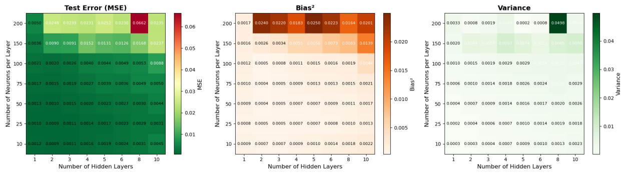
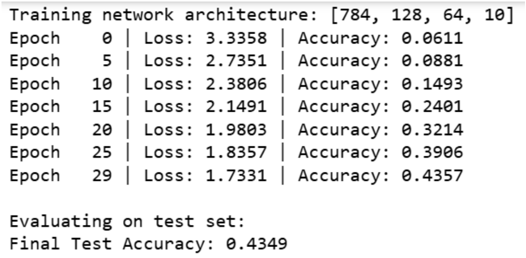


Figure 10: Joint depth–width search: test MSE (left), bias^2 (middle), and variance (right). Best overall performance occurs at two hidden layers with 10–25 neurons per layer.

Part F: From Figure 11, there is a steady decrease in training loss and an increase in accuracy rates. This indicates the neural network's successful learning from the dataset. Nevertheless, the final test accuracy of approximately 43% indicates that the model is underperforming compared to typical benchmarks for the MNIST dataset, which usually exceed 90%. This suggests the presence of *underfitting*, where the model's capacity or training process may be insufficient to capture the complexity of the dataset.



```
Training network architecture: [784, 128, 64, 10]
Epoch    0 | Loss: 3.3358 | Accuracy: 0.0611
Epoch    5 | Loss: 2.7351 | Accuracy: 0.0881
Epoch   10 | Loss: 2.3806 | Accuracy: 0.1493
Epoch   15 | Loss: 2.1491 | Accuracy: 0.2401
Epoch   20 | Loss: 1.9803 | Accuracy: 0.3214
Epoch   25 | Loss: 1.8357 | Accuracy: 0.3906
Epoch   29 | Loss: 1.7331 | Accuracy: 0.4357

Evaluating on test set:
Final Test Accuracy: 0.4349
```

Figure 11: Training loss and accuracy across epochs.

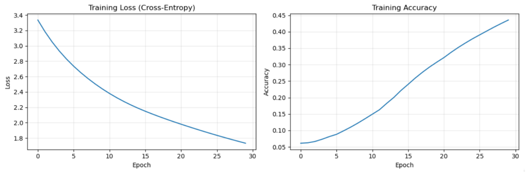


Figure 12: Final test accuracy comparison.

a. In order to improve the accuracy and performance of the model, adding more hidden layers can typically achieve higher accuracy rates. Since the accuracy values were still increasing at epoch 30, the model has yet to converge; hence, increasing the number of epochs beyond 30 may raise accuracy values further.

b. From Figure 12, the smooth downward-sloping line of the training loss curve shows the model's continuous improvement before gradually beginning to flatten by epoch 30. The smooth curve also suggests that the learning rate of 0.001 is stable. However, since the curve has not fully flattened by epoch 30, it shows that the model has yet to converge, reinforcing our earlier finding that additional training epochs could improve accuracy. The training accuracy curve also exhibits a smooth shape, but it slopes upward instead. The curve is steeper at the beginning before becoming gentler as training progresses, indicating that the neural network is learning patterns from the training data successfully. Nevertheless, the low final accuracy of the model suggests that more complex architectures may be required to handle the complexity of the MNIST dataset.

c. However, the model's accuracy (approximately 43%) is still relatively low compared to optimised libraries such as TensorFlow or Scikit-learn's `MLPClassifier`, which can achieve up to 97% accuracy using the Adam optimiser and the full 60,000 training images.

d. Part G: In conclusion, while this study demonstrates correct gradient-based learning behaviour, it remains limited in computational efficiency and performance compared to more mature machine learning libraries. For future studies, performance can be improved by introducing adaptive optimisers such as Adam and by increasing the sample size of training data to enhance the model's generalisation capability.

IV. CONCLUSION

- Overall, our results provided valuable insights and experience with implementing forward and backward propagation algorithms. However, future improvements could include increasing the number of hidden neurons or layers, and/or increasing the number of epochs, fine tuning the learning rate, and/or incorporating more advanced optimisers such as Adam.