IEOR E4742 Homework 1

Mark Ma km4054

Problem 1

We assume that the feed-forward neural network has L hidden layers, where each layer applies a linear activation function.

Assume the input is x, then we have:

$$h^{(1)} = W_1 x + b_1$$

$$h^{(2)} = W_2 h^{(1)} + b_2 = W_2 W_1 x + W_2 b_1 + b_2$$

$$h^{(3)} = W_3 h^{(2)} + b_3 = W_3 (W_2 W_1 x + W_2 b_1 + b_2) + b_3$$

$$= W_3 W_2 W_1 x + \underbrace{W_3 W_2 b_1 + W_3 b_2 + b_3}_{\text{bias terms}}$$

$$\vdots$$

Then, $h^{(L)} = W_L W_{L-1} \cdots W_1 x + \text{bias terms}$

$$= W_{\text{combined}} \cdot x + b_{\text{combined}}$$

where $W_{\text{combined}} = W_L W_{L-1} \cdots W_1$ and b_{combined} is a combination of weighted bias.

Therefore, even though we have multiple layers, since the activation function is linear, the network behaves exactly like a single linear transformation with no hidden layer.

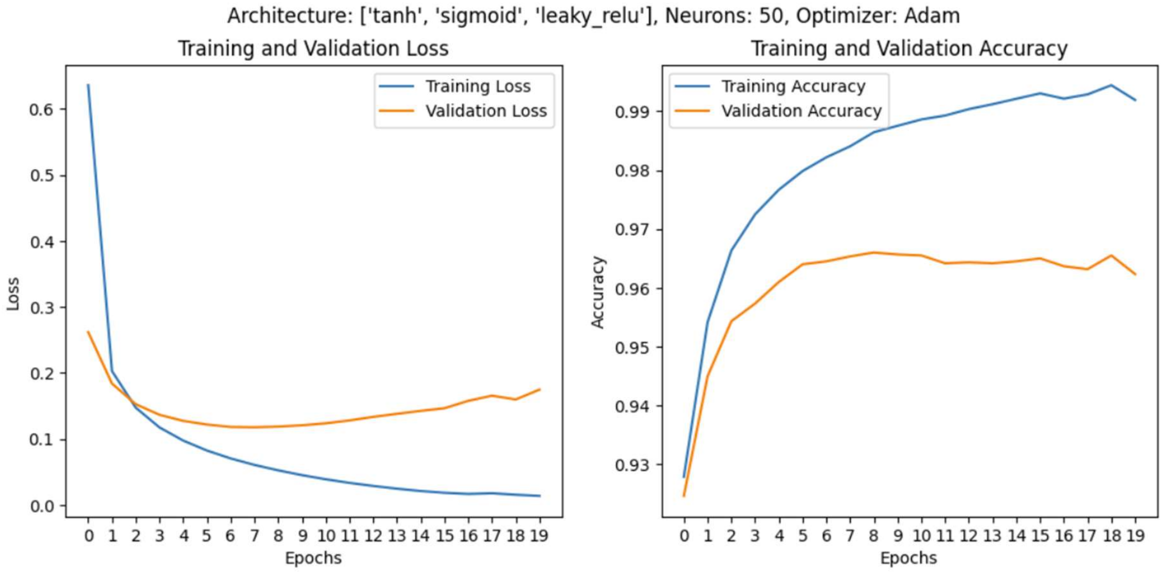
Problem 2

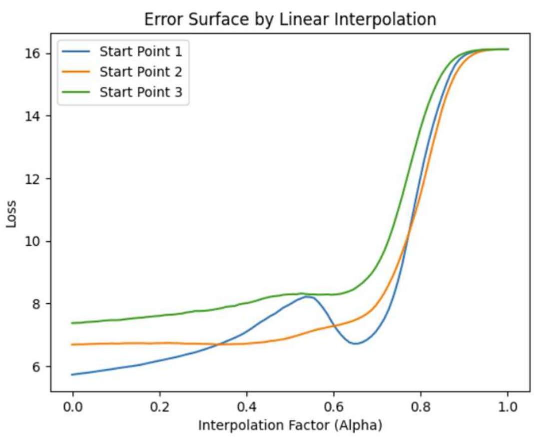
Description: For this question, I extended the 2-layer logistic regression to 6 architectures stated in the question. The neuron choices are 50 or 100. The optimizers are Adam or RMSProp. Then I plotted the loss & accuracy curves as well as linear interpolation plot for each setting. Finally, I discussed the impact of different hyperparameters on learning rates and activation functions.

Results

Architecture 1: 1st layer: tanh, 2nd layer: sigmoid, 3rd layer: leaky ReLU

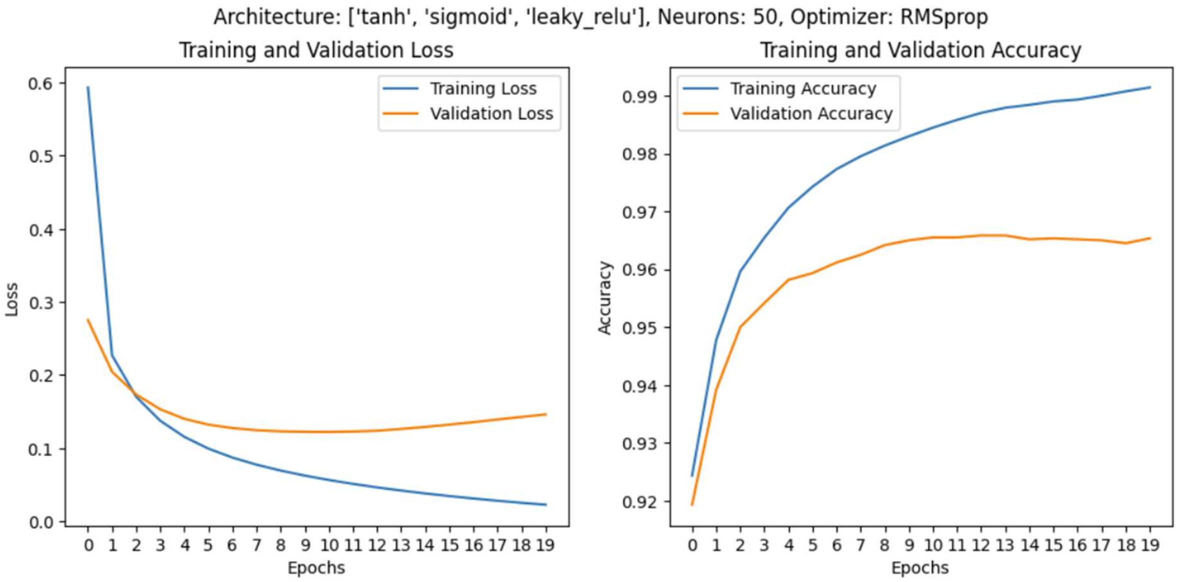
Adam as optimizer with 50 neurons:

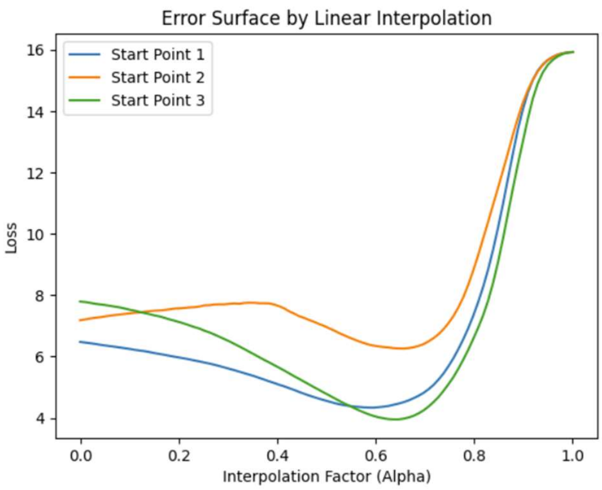




- Final Validation Accuracy: 96.54%

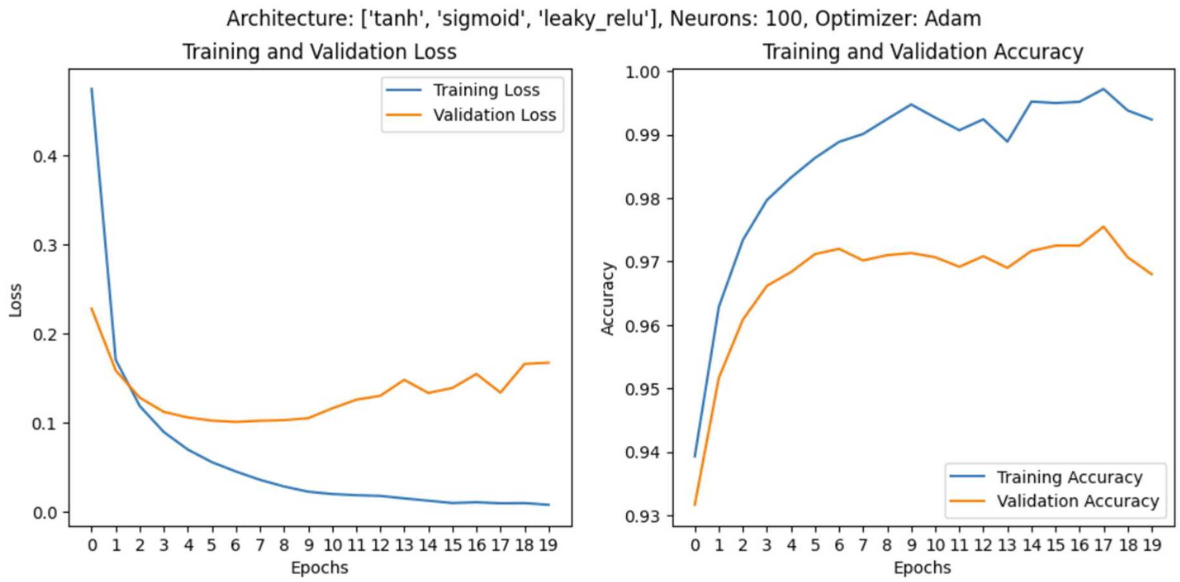
RMSProp as optimizer with 50 neurons:

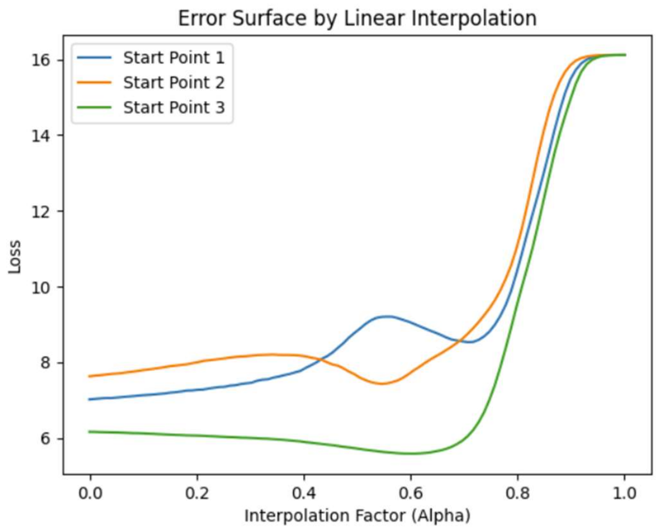




- Final Validation Accuracy: 96.86%

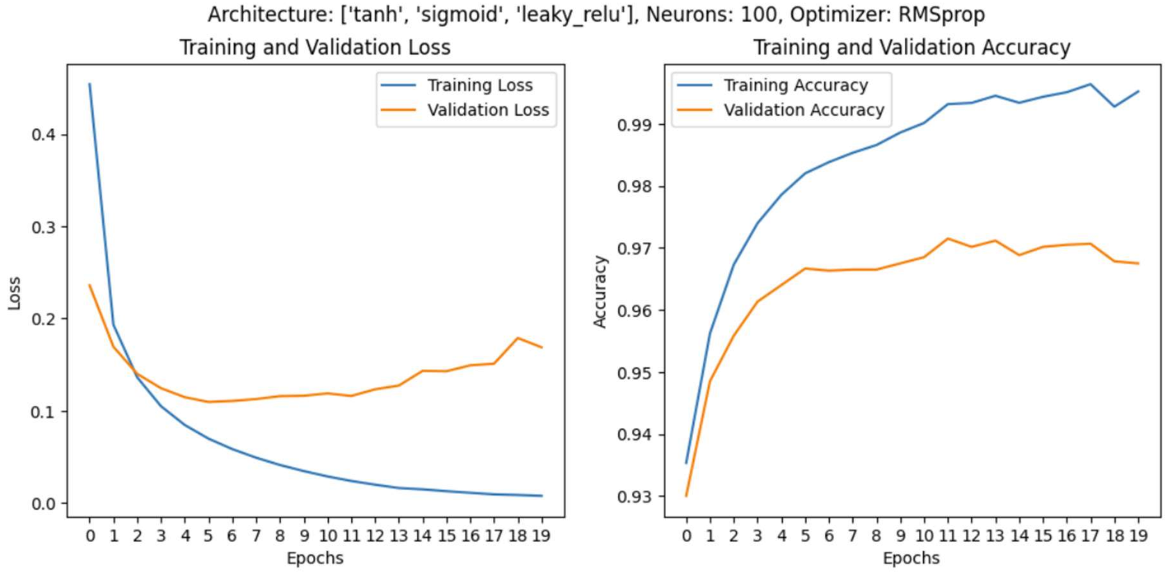
Adam as optimizer with 100 neurons:

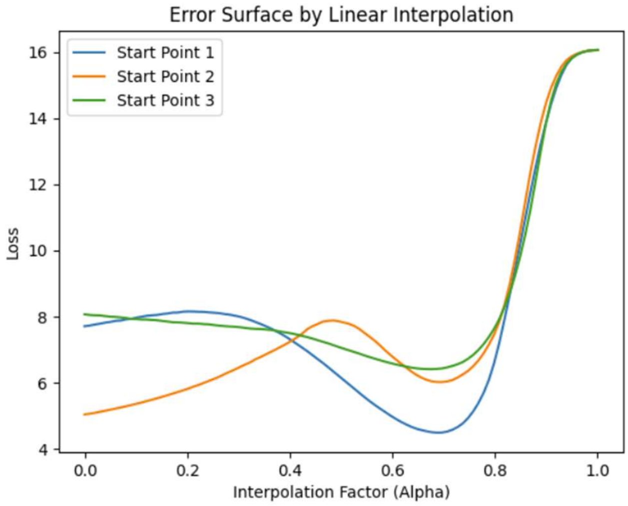




- Final Validation Accuracy: 97.10%

RMSProp as optimizer with 100 neurons:

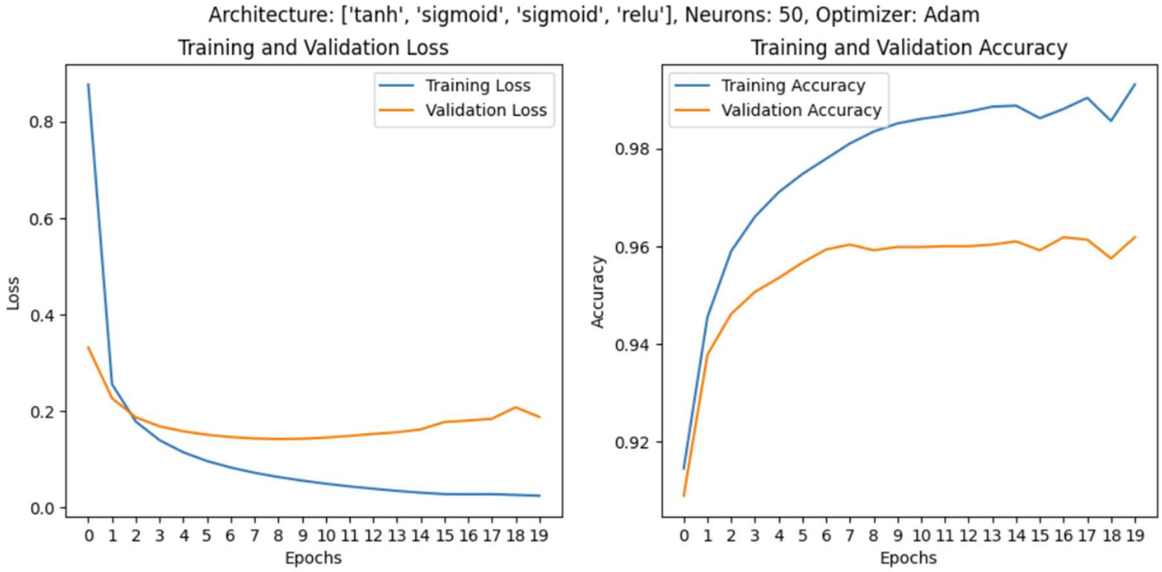


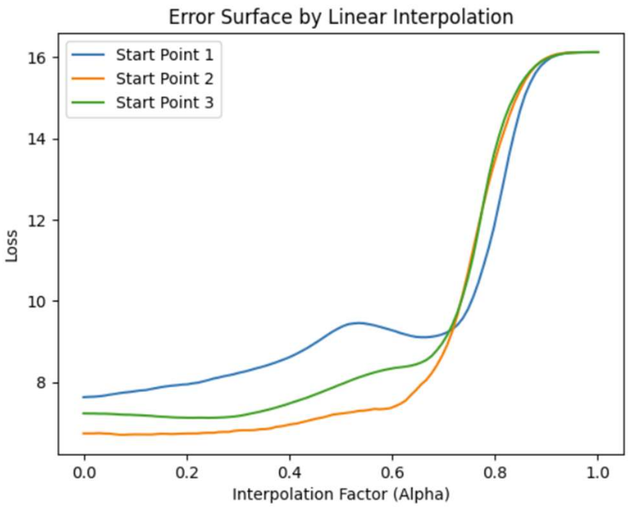


- Final Validation Accuracy: 97.28%

Architecture 2: 1st layer: tanh, 2nd layer: sigmoid, 3rd layer: sigmoid, 4th layer: ReLU

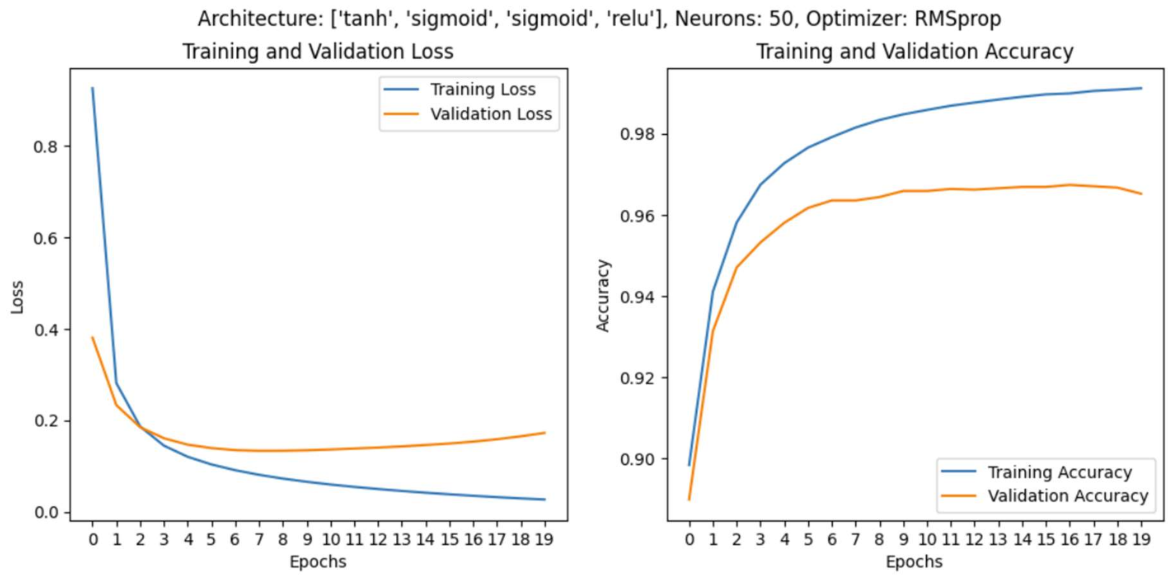
Adam as optimizer with 50 neurons:

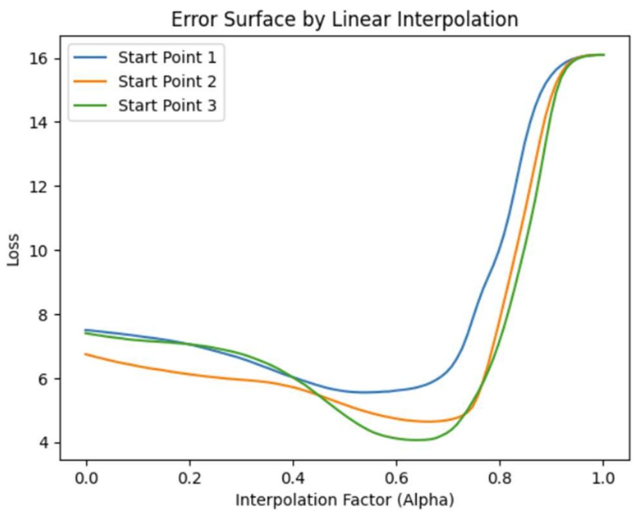




- Final Validation Accuracy: 96.52%

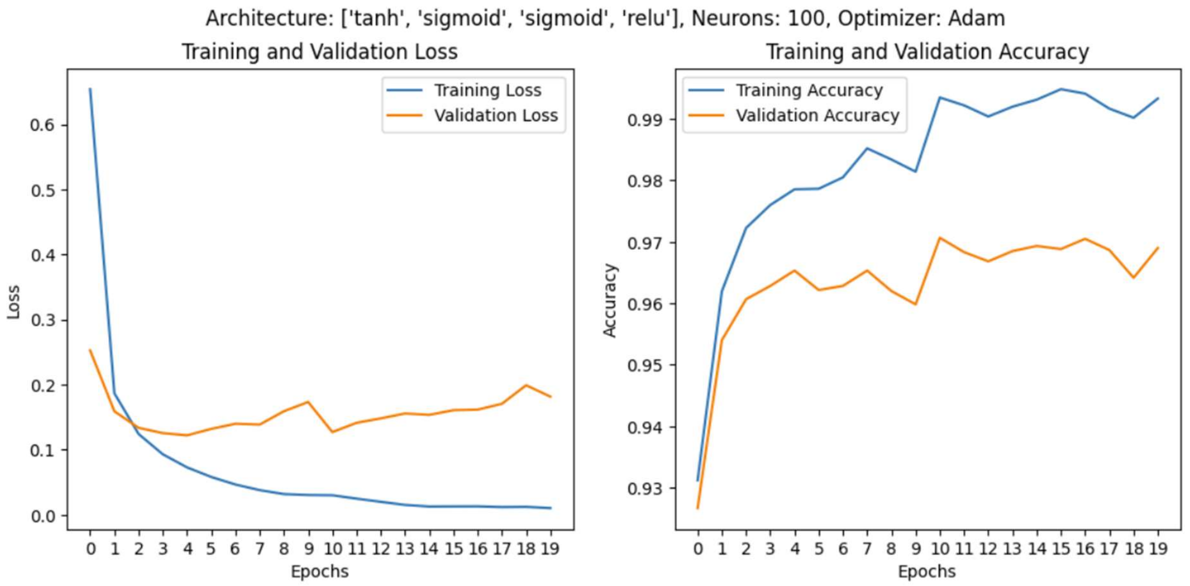
RMSProp as optimizer with 50 neurons:

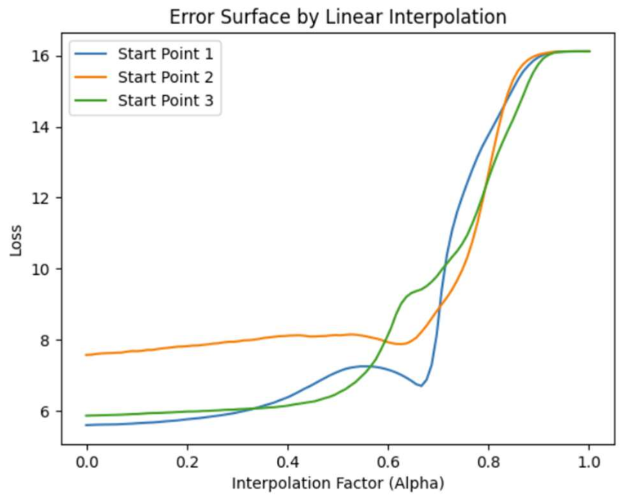




- Final Validation Accuracy: 96.71%

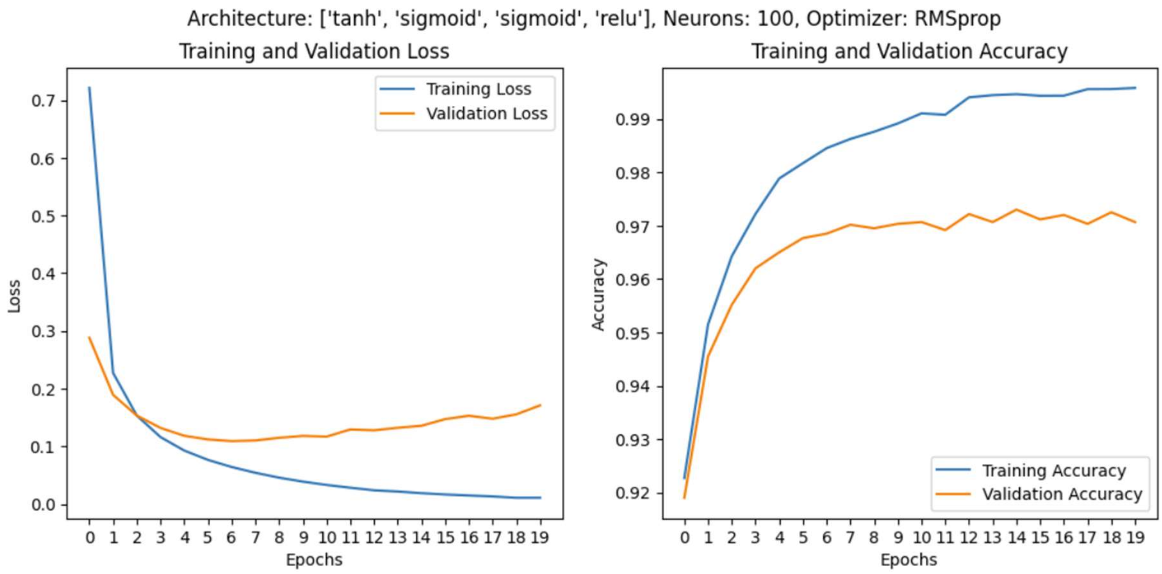
Adam as optimizer with 100 neurons:

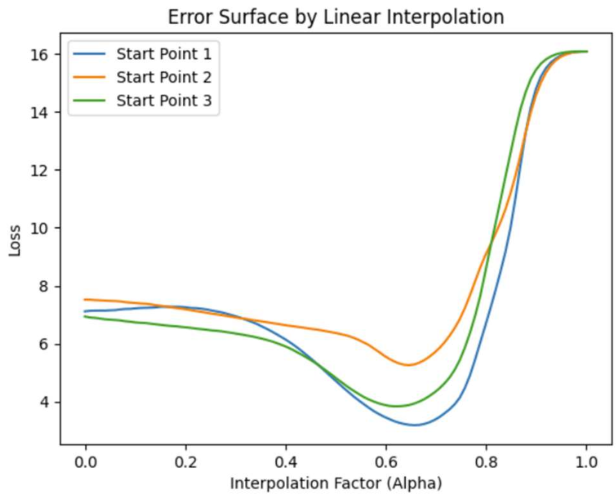




- Final Validation Accuracy: 96.95%

RMSProp as optimizer with 100 neurons:

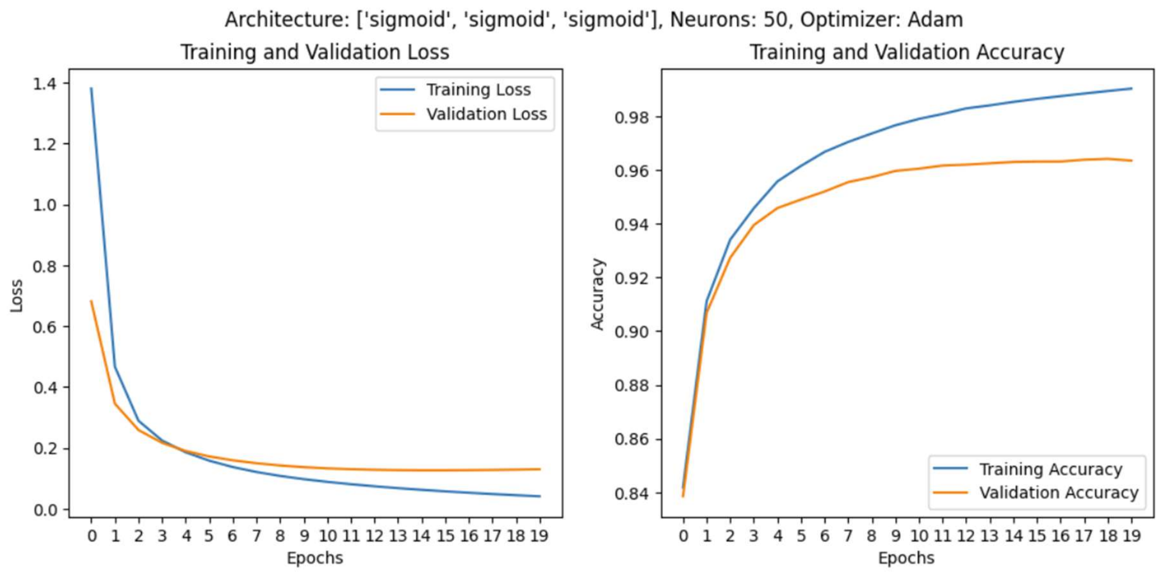


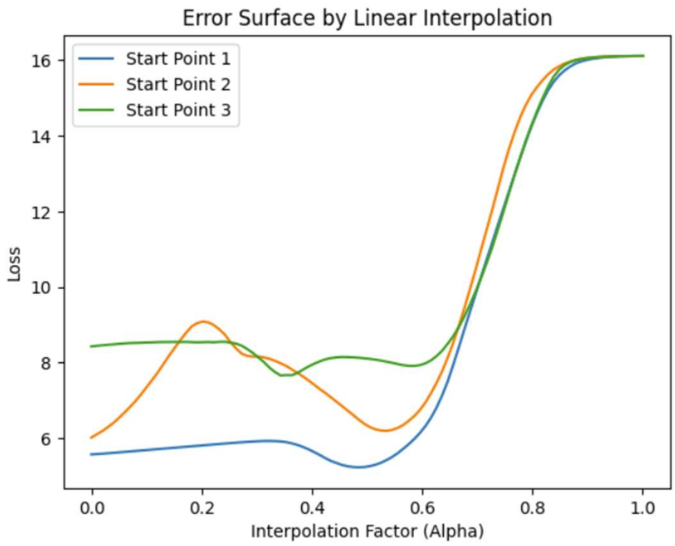


- Final Validation Accuracy: 97.36%

Architecture 3: 3 layers, all sigmoid

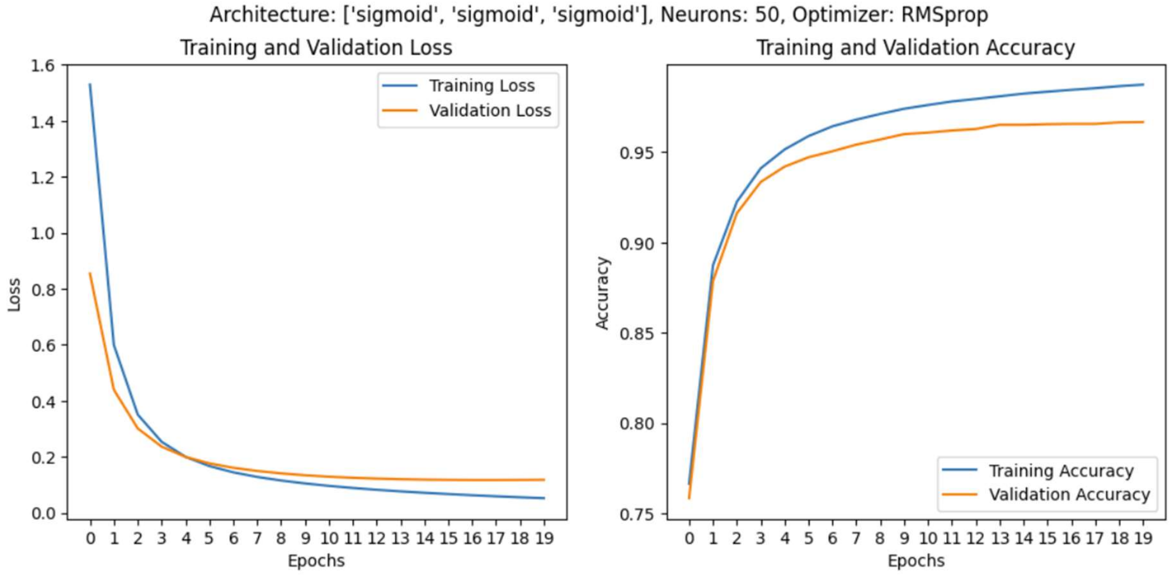
Adam as optimizer with 50 neurons:

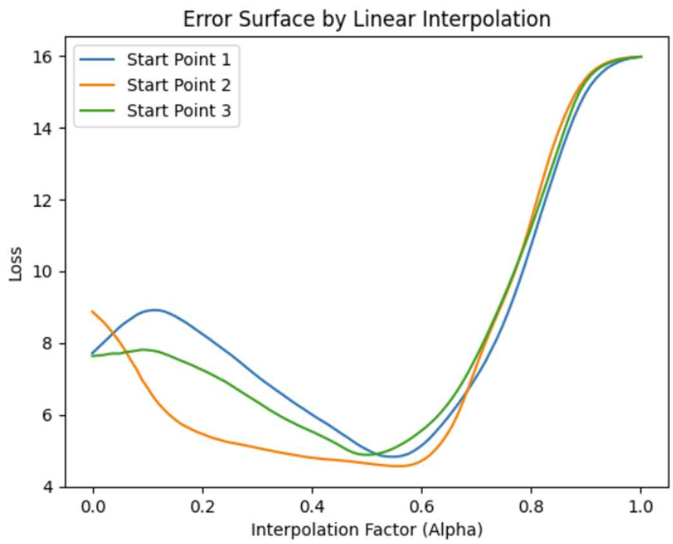




- Final Validation Accuracy: 96.90%

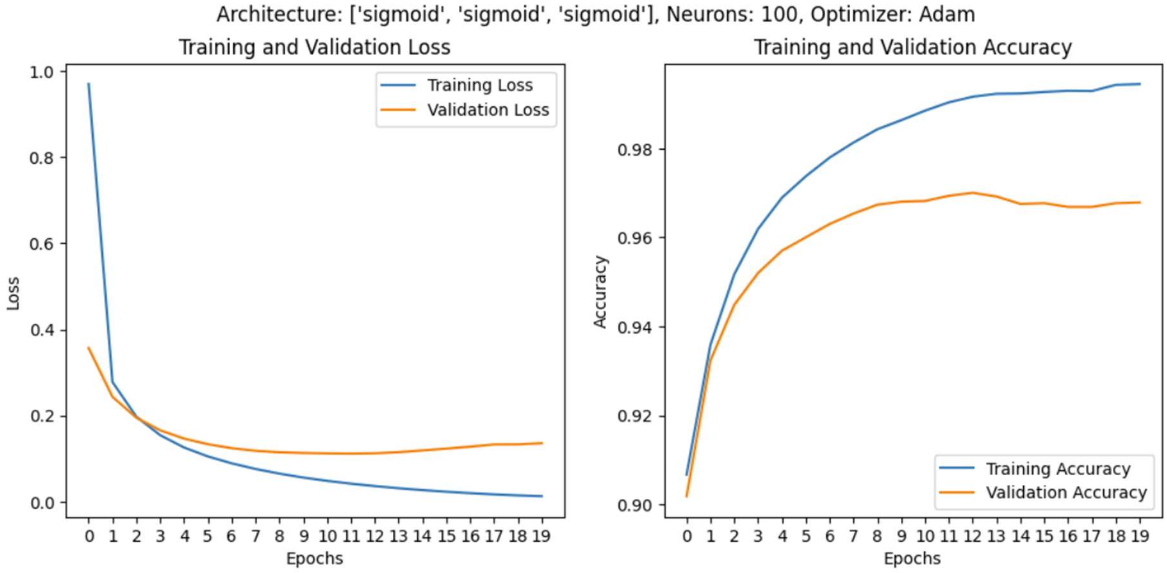
RMSProp as optimizer with 50 neurons:

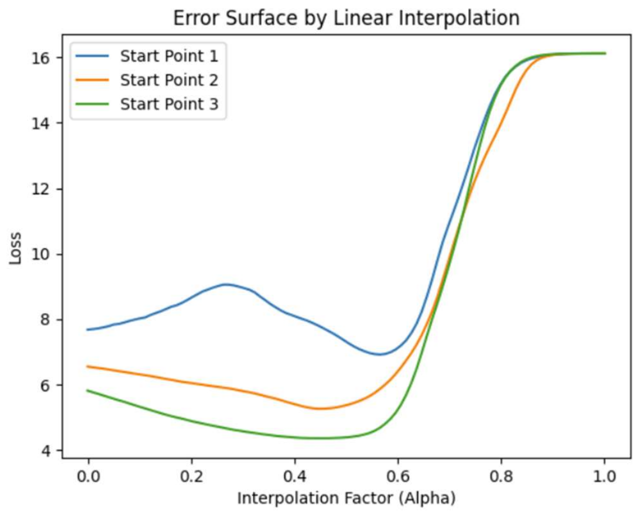




- Final Validation Accuracy: 96.84%

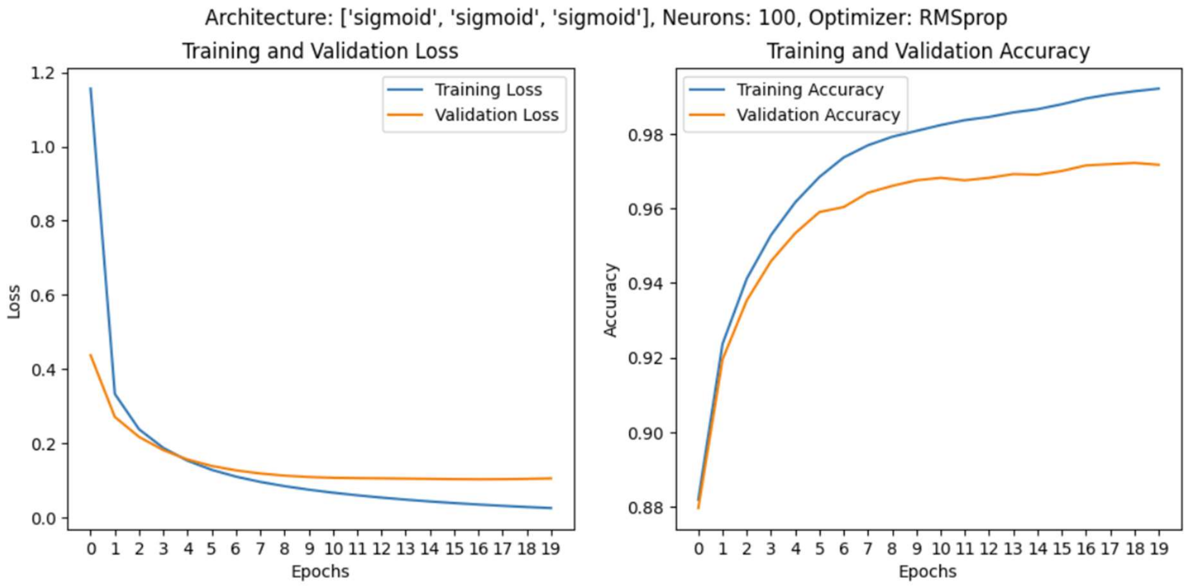
Adam as optimizer with 100 neurons:

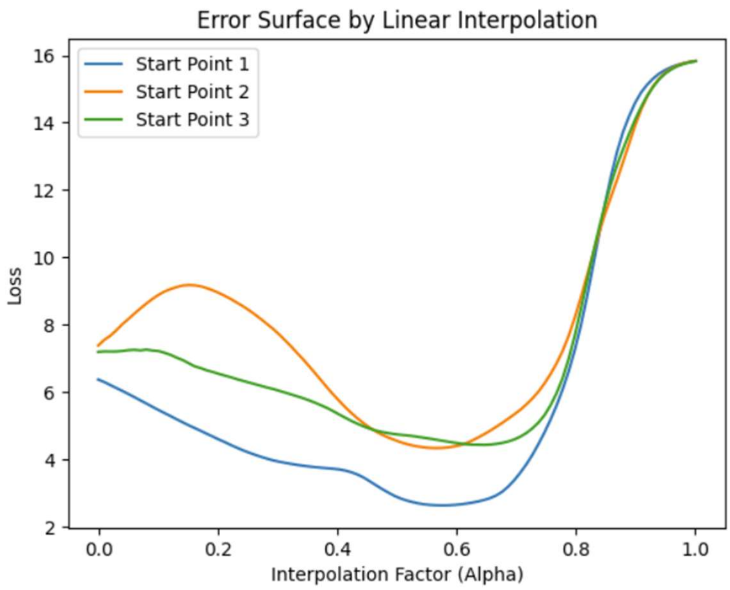




- Final Validation Accuracy: 97.25%

RMSProp as optimizer with 100 neurons:

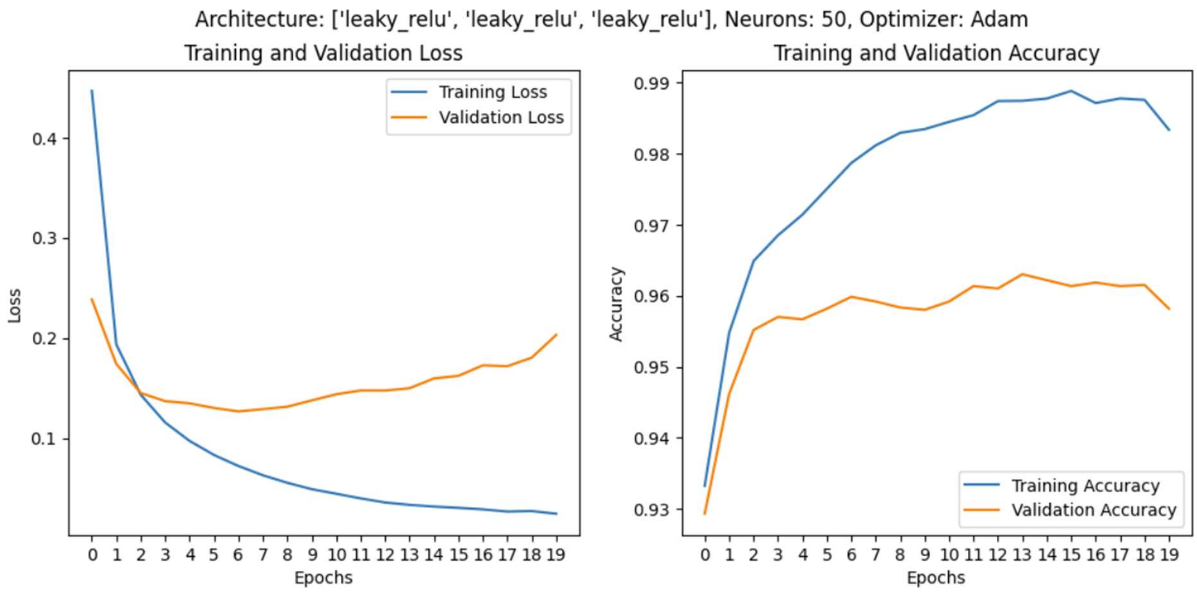


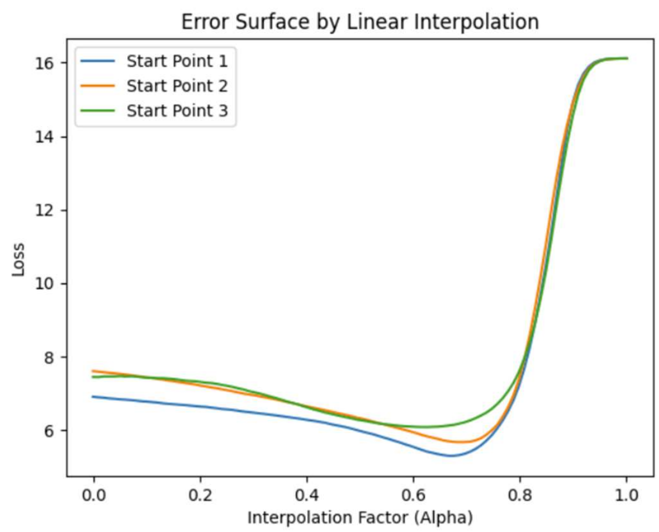


- Final Validation Accuracy: 97.63%

Architecture 4: 3 layers, all leaky ReLU

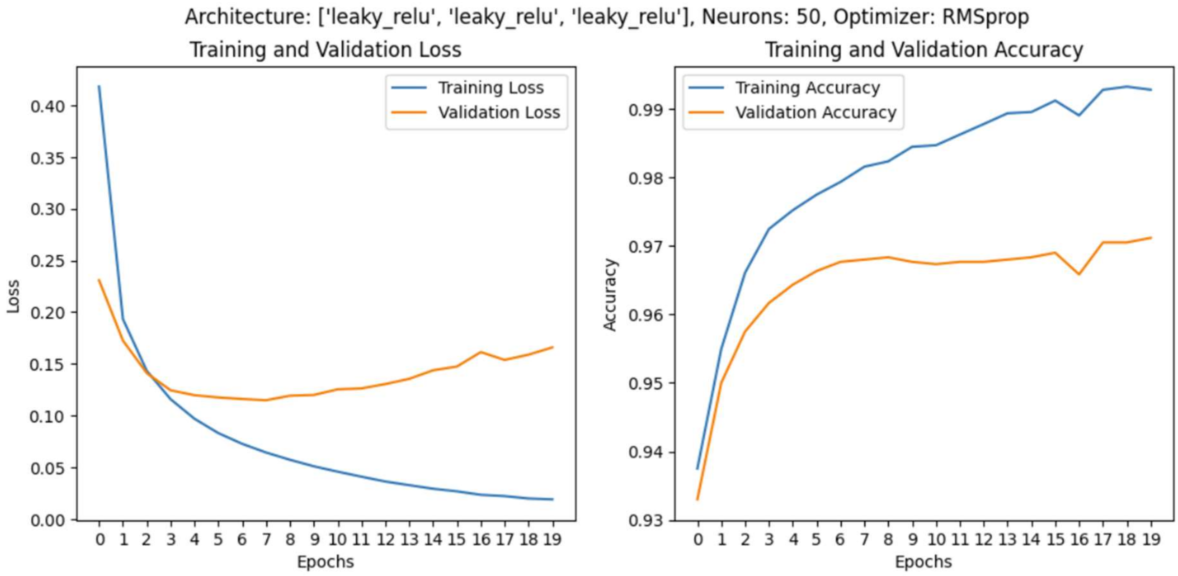
Adam as optimizer with 50 neurons:

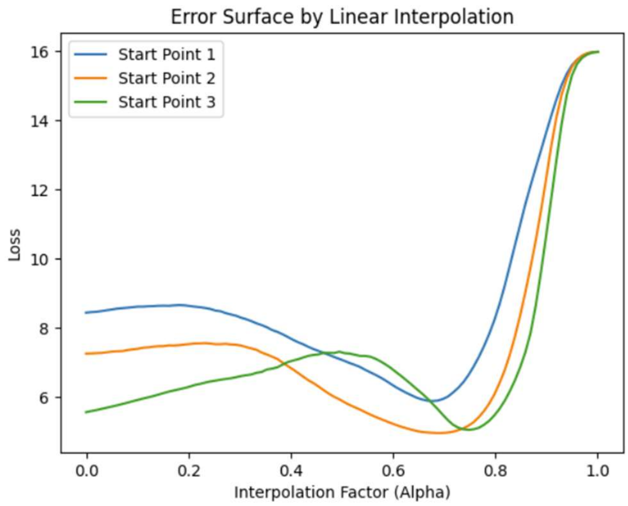




- Final Validation Accuracy: 96.45%

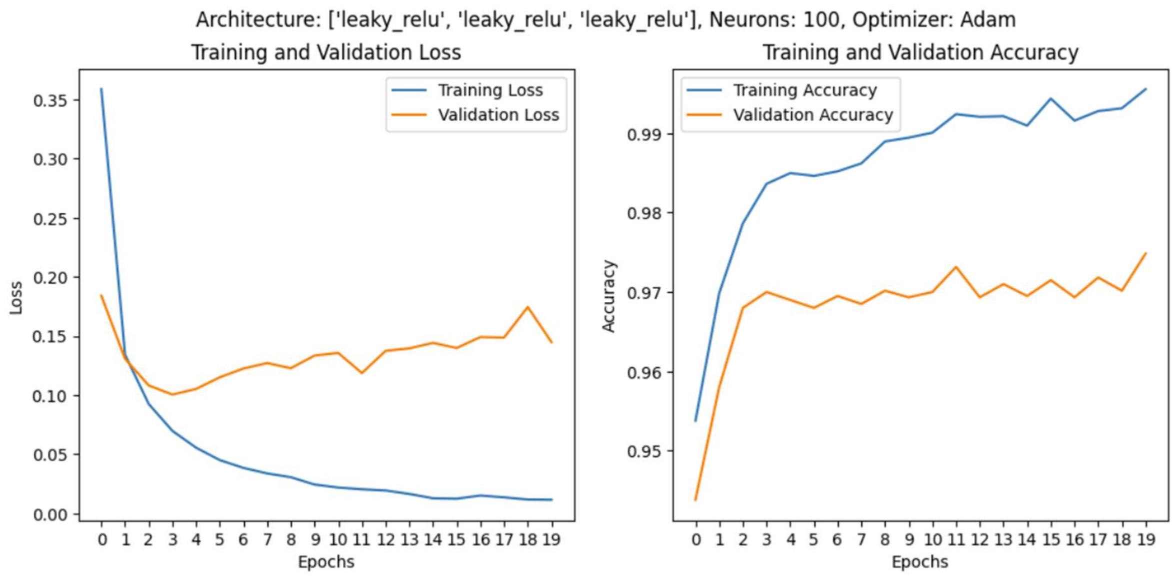
RMSProp as optimizer with 50 neurons:

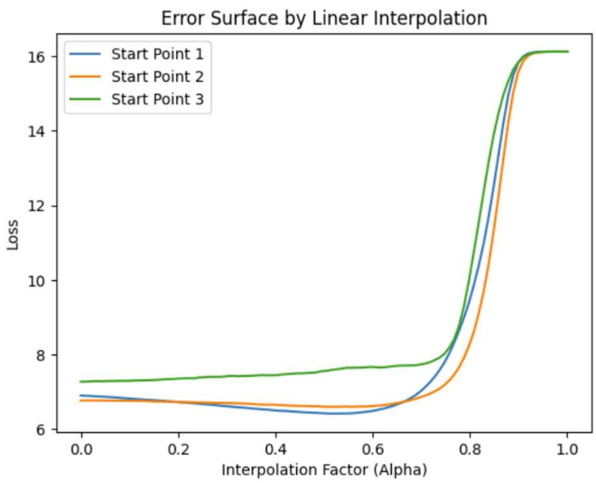




- Final Validation Accuracy: 96.97%

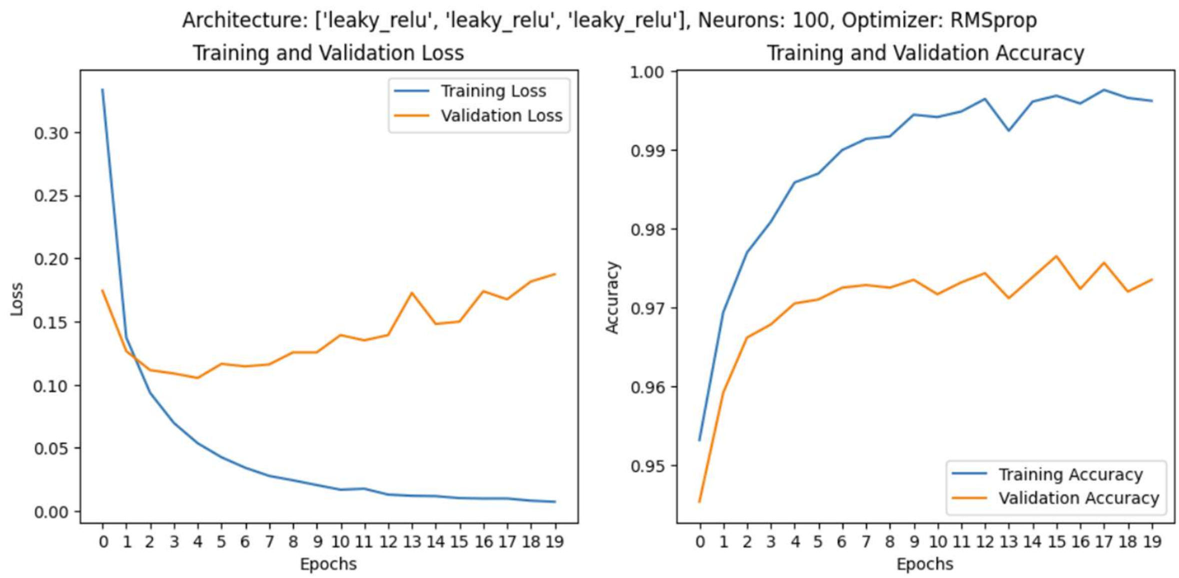
Adam as optimizer with 100 neurons:

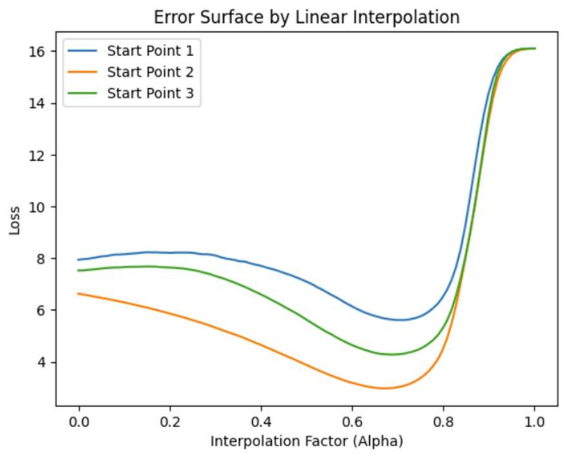




- Final Validation Accuracy: 97.55%

RMSProp as optimizer with 100 neurons:

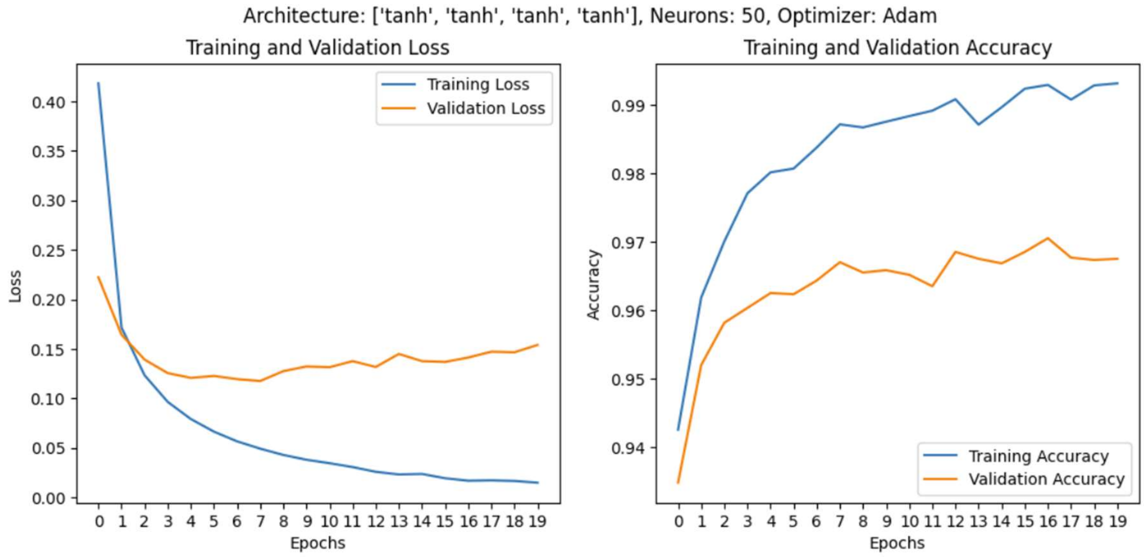


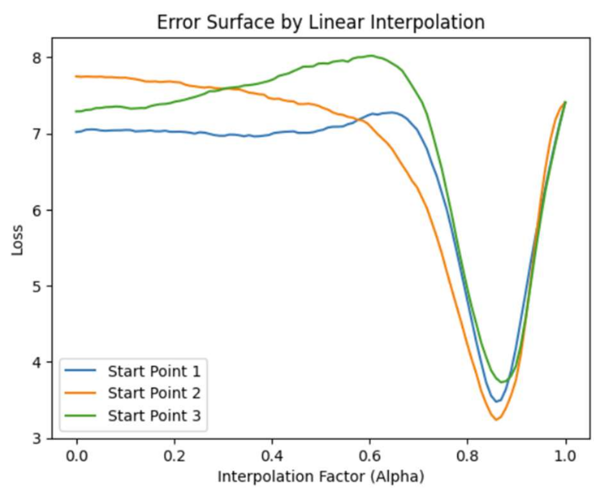


- Final Validation Accuracy: 97.75%

Architecture 5: 4 layers, all tanh

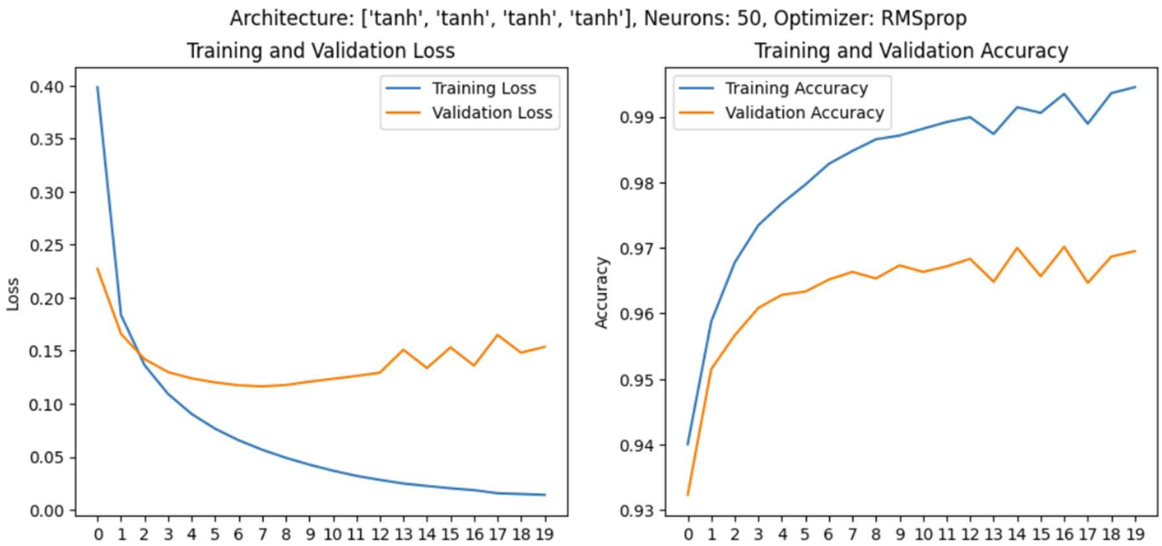
Adam as optimizer with 50 neurons:

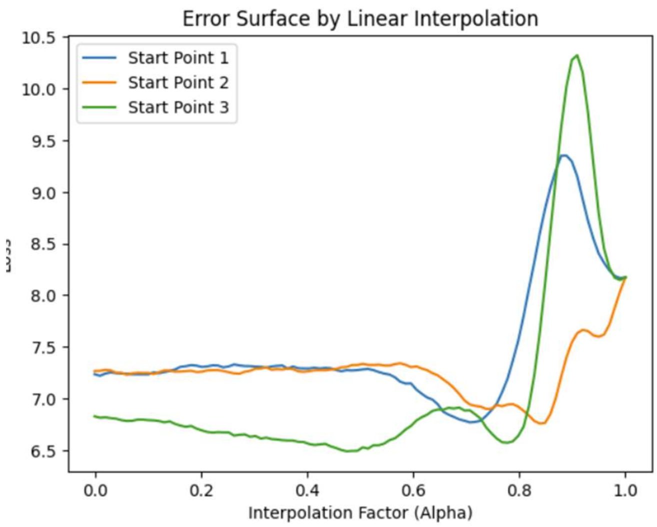




- Final Validation Accuracy: 96.77%

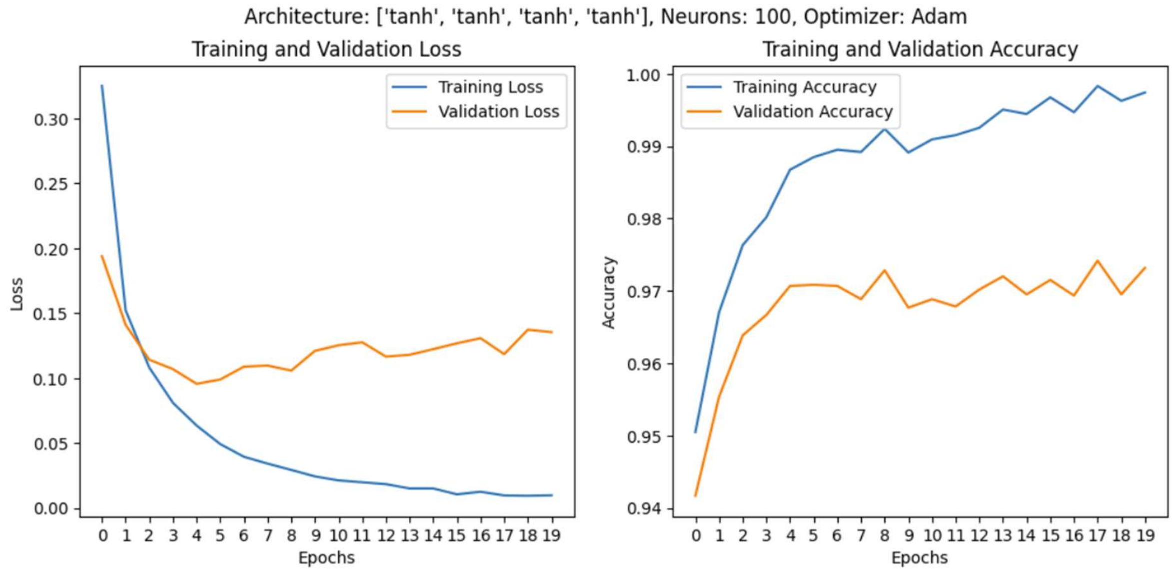
RMSProp as optimizer with 50 neurons:

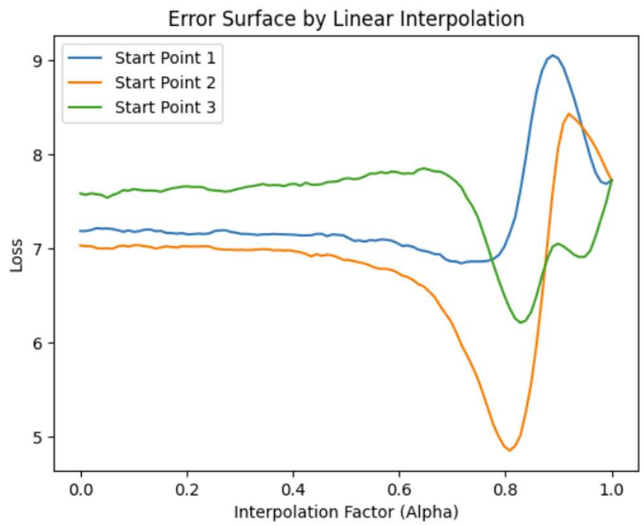




- Final Validation Accuracy: 97.19%

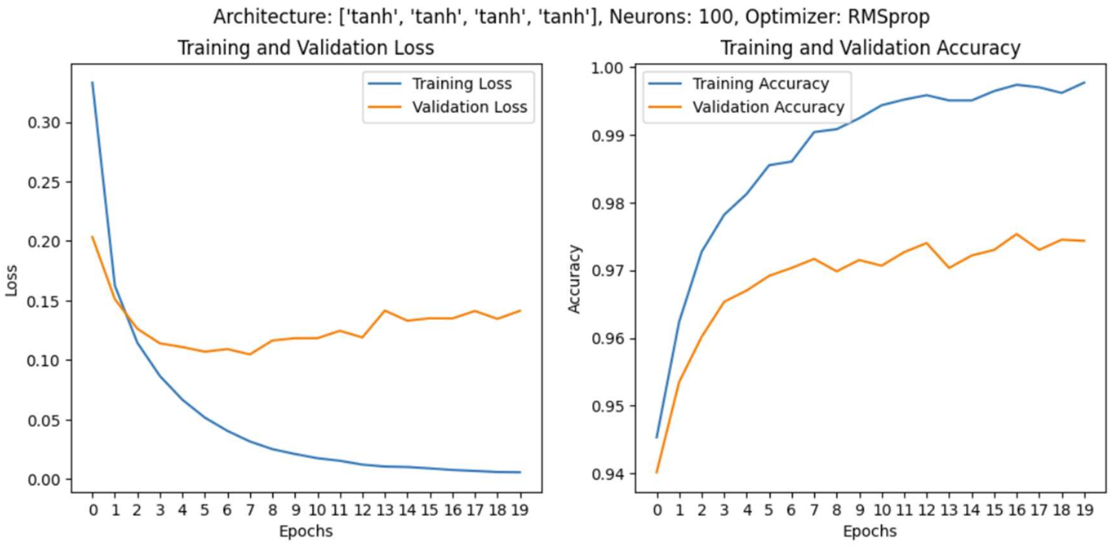
Adam as optimizer with 100 neurons:

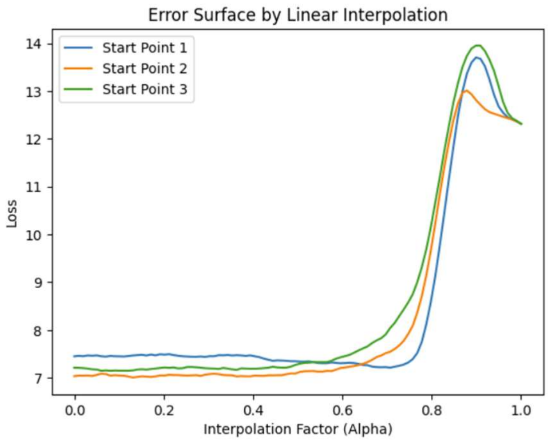




- Final Validation Accuracy: 97.57%

RMSProp as optimizer with 100 neurons:

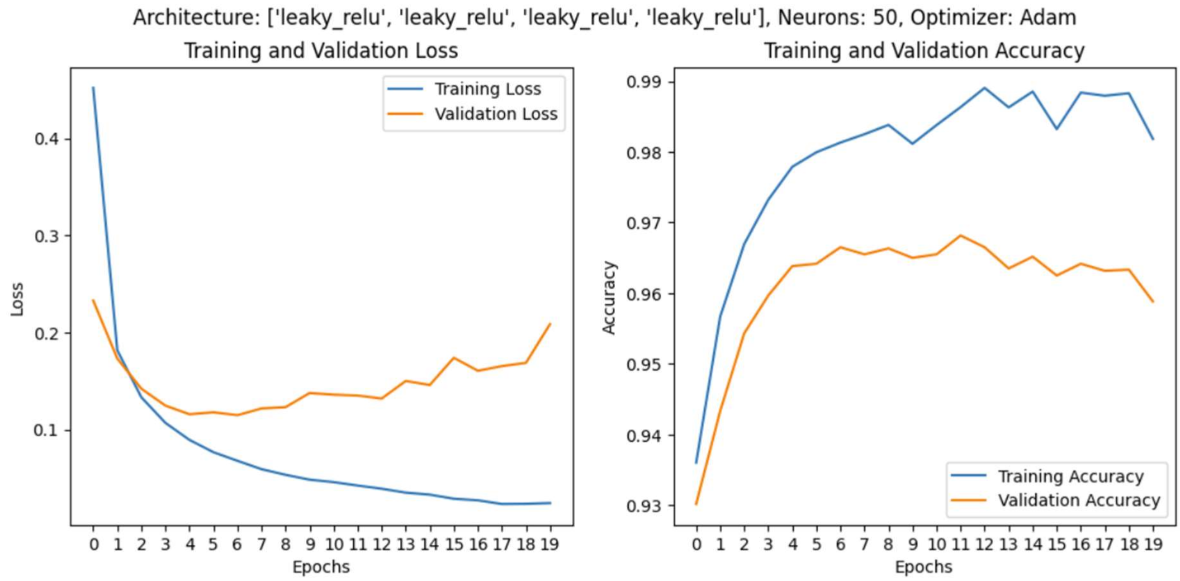


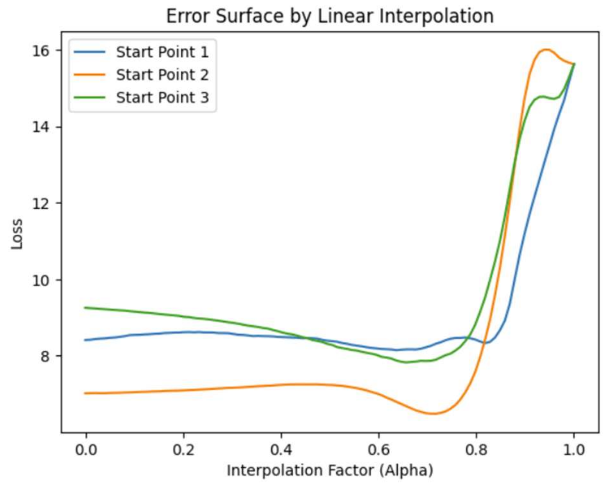


- Final Validation Accuracy: 97.32%

Architecture 6: 4 layers, all leaky ReLU

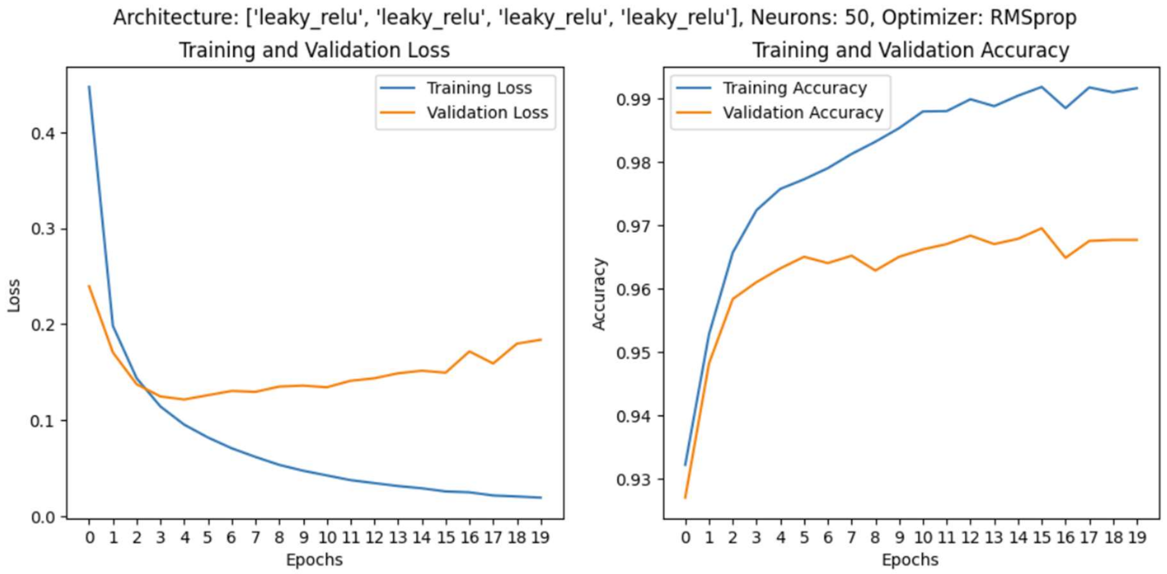
Adam as optimizer with 50 neurons:

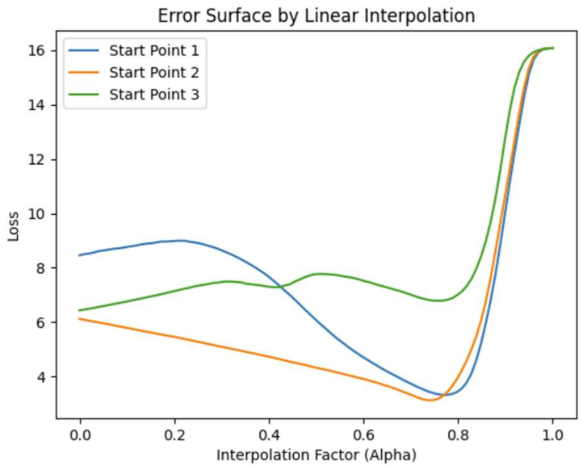




- Final Validation Accuracy: 96.27%

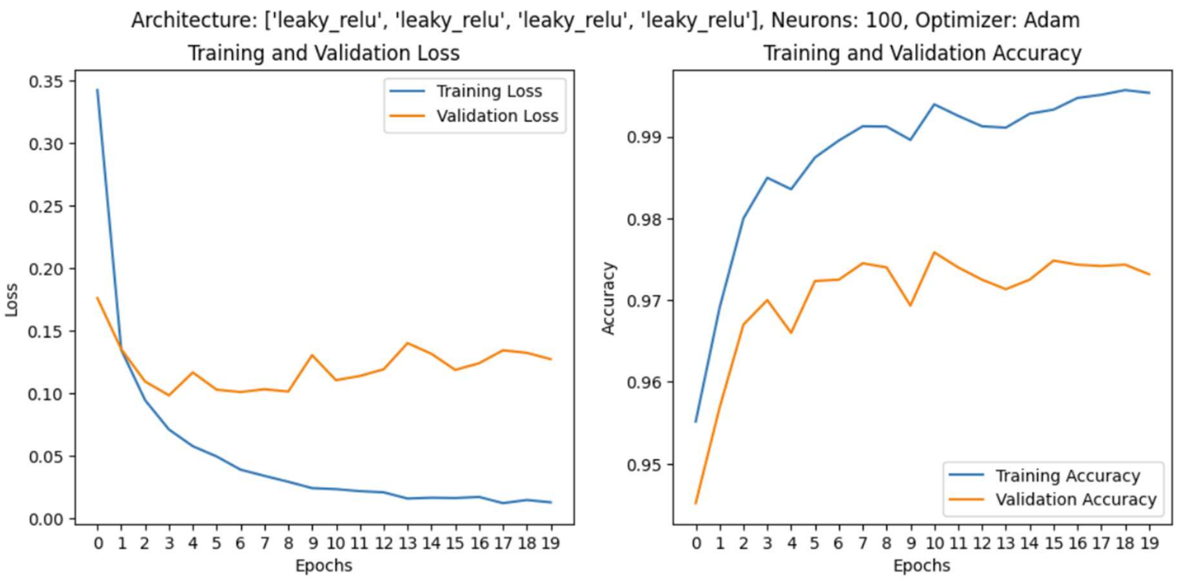
RMSProp as optimizer with 50 neurons:

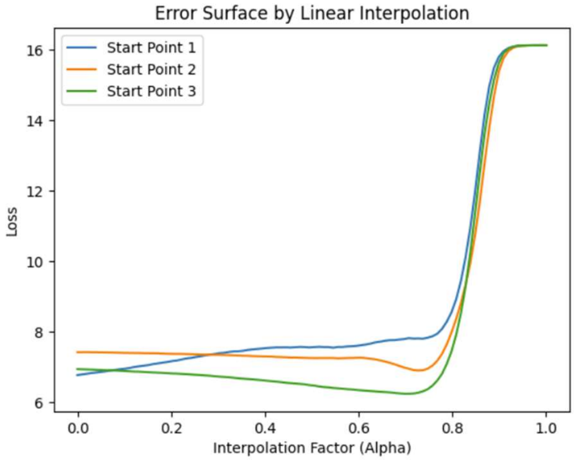




- Final Validation Accuracy: 96.95%

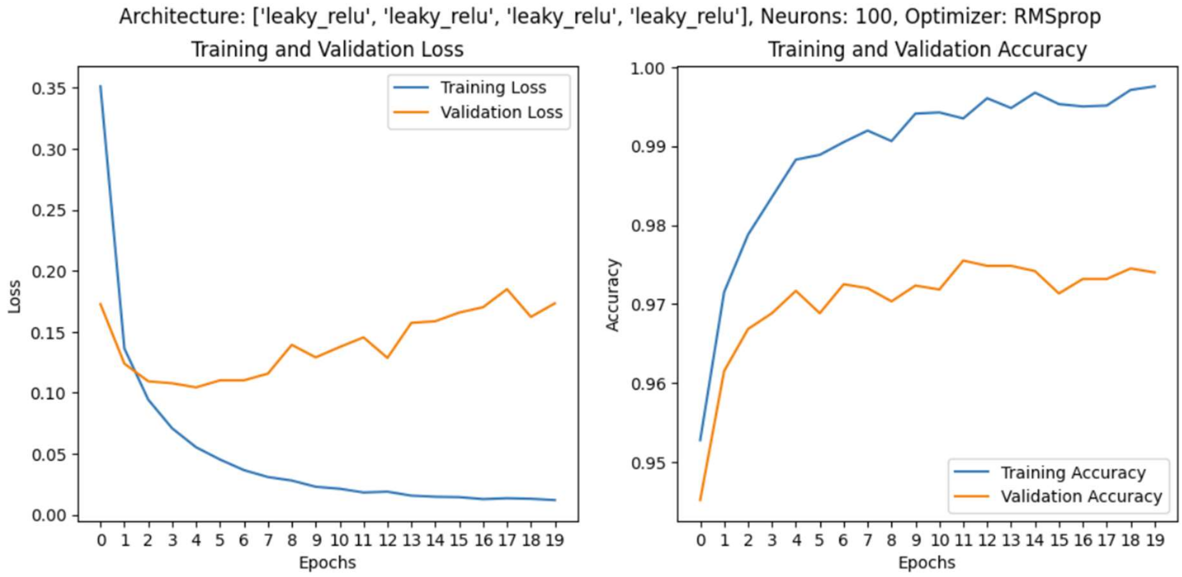
Adam as optimizer with 100 neurons:

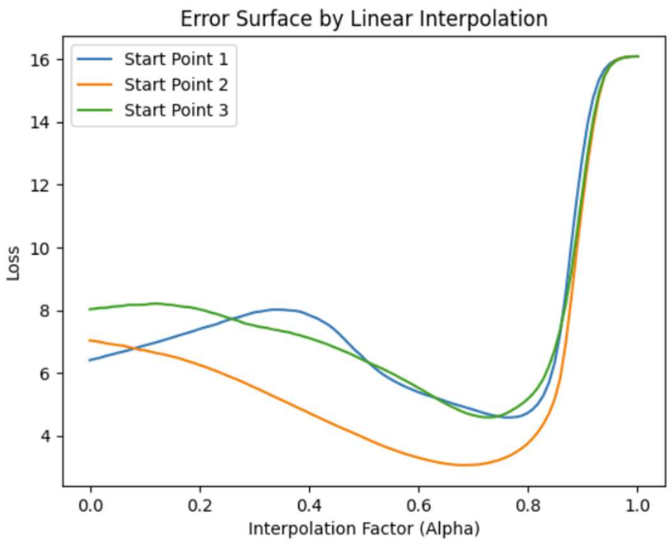




- Final Validation Accuracy: 97.57%

RMSProp as optimizer with 100 neurons:





- Final Validation Accuracy: 97.78%

About the Error Surface

For the 2D error surface by linear interpolation, we interpolated between a few different randomly initialized starting weights and the final optimized weight to visualize the error surface.

Note: For the linear interpolation plots, I use the formula below:

$$interpolated\ weights = (1-\alpha) * start\ weight + \alpha * final\ weight$$

Overall Accuracy Results on Validation data

Architecture	50 Neurons (Adam/RMSProp)	100 Neurons (Adam/RMSProp)
1	0.9654 / 0.9686	0.9710 / 0.9728
2	0.9652 / 0.9671	0.9695 / 0.9736
3	0.9690 / 0.9684	0.9725 / 0.9763
4	0.9645 / 0.9697	0.9755 / 0.9775
5	0.9677 / 0.9719	0.9757 / 0.9732
6	0.9627 / 0.9695	0.9757 / 0.9778

Conclusion & Analysis

1. Impact of Neurons per Layer: Increasing the number of neurons from 50 to 100 consistently improved the validation accuracy across all architectures. This suggests that higher neuron counts allow the model to capture more complex patterns in the data, leading to better generalization.

2. Optimizer Performance: The RMSProp optimizer slightly outperformed Adam in most configurations, particularly with higher neuron counts (100). This indicates that RMSProp may be more effective for these architectures and the problem dataset, potentially due to better handling of adaptive learning rates in layered structures.

3. Architecture and Activation Function Influence: Among the architectures tested, the 4-layer tanh (architecture 5) and 4-layer leaky ReLU (architecture 6) achieved the highest validation accuracy with 100 neurons, indicating that deeper architectures with non-linear activations (tanh, leaky ReLU) help in capturing complex non-linear relationships, which may contribute to better overall model performance.

Different Learning Rate

For this part, I choose the architecture 1 with 50 neurons and Adam as optimizer to train with different learning rates [0.0001, 0.001, 0.01, 0.1, 0.5]. We want to discover how the learning rate affects the training performance over time.

Result

Learning rate	Validation accuracy	Testing accuracy	Running time (in second)
0.0001	0.9523	0.9506	460.09
0.001	0.9648	0.9648	456.17
0.01	0.9624	0.9642	436.34
0.1	0.1141	0.1134	434.87
0.5	0.1141	0.1134	432.27

Conclusion

1. Optimal Learning Rate Range: Learning rates of 0.001 and 0.01 achieved the highest validation and testing accuracy, suggesting that these values strike a good balance between convergence speed and stability for this architecture and optimizer combination.

2. Effect of Very Low Learning Rate (0.0001): While the lowest learning rate (0.0001) provided relatively high accuracy, it took the longest time to converge, indicating that very low learning rates can increase training duration without significant accuracy benefits over slightly higher learning rates.

3. Instability with High Learning Rates (0.1 and 0.5): Both the 0.1 and 0.5 learning rates resulted in significantly lower accuracy scores, which indicates that these high values lead to unstable training, potentially causing the model to diverge or fail to find an optimal solution.

4. Marginal Time Differences Across Learning Rates: The running time slightly decreased as the learning rate increased, with only marginal differences across learning rates, suggesting that higher learning rates reduce the number of required iterations but may compromise accuracy if set too high.

code.py

```python
import tensorflow as tf
print(tf.__version__)

import os
import time

import numpy as np # linear algebra
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense, LeakyReLU
from tensorflow.keras.optimizers import Adam, RMSprop
import warnings
warnings.filterwarnings('ignore')


# generate original training and test data
img_size = 28
n_classes = 10

#MNIST data image of shape 28*28=784
input_size = 784

# 0-9 digits recognition (labels)
output_size = 10

#-------------------------------------------------------------
#option 1: load MNIST dataset
#from tensorflow.examples.tutorials.mnist import input_data
#mnist = input_data.read_data_sets("data/", one_hot=True)


#-------------------------------------------------------------
#option 2: load MNIST dataset
print('\nLoading MNIST')
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = np.reshape(x_train, [-1, img_size*img_size])
x_train = x_train.astype(np.float32)/255

x_test = np.reshape(x_test, [-1, img_size*img_size])
x_test = x_test.astype(np.float32)/255

to_categorical = tf.keras.utils.to_categorical
y_train = to_categorical(y_train)
y_test  = to_categorical(y_test)

print('\nSpliting data')

ind = np.random.permutation(x_train.shape[0])
x_train, y_train = x_train[ind], y_train[ind]

```

```
# 10% for validation
validatationPct = 0.1
n = int(x_train.shape[0] * (1-validatationPct))
x_valid = x_train[n:]
x_train = x_train[:n]
#
y_valid = y_train[n:]
y_train = y_train[:n]

train_num_examples = x_train.shape[0]
valid_num_examples = x_valid.shape[0]
test_num_examples  = x_test.shape[0]

print(train_num_examples, valid_num_examples, test_num_examples)

# Global Parameters
#--------------------------------
# learning rate
learning_rate = 0.05

#training_epochs = 1000
#batch_size = 30

training_epochs = 100
batch_size = 50

display_step = 10

#Network Architecture
# ----------------------------------------
#
# Two hidden layers
#
#-----------------------------------------
# number of neurons in layer 1
n_hidden_1 = 200
# number of neurons in layer 2
n_hidden_2 = 300

#MNIST data image of shape 28*28=784
input_size = 784

# 0-9 digits recognition (labels)
output_size = 10

def loss_2(output, y):
    """
    Computes softmax cross entropy between logits and labels and returns the loss.

    Input:
        - output: the output (logits) of the inference function (shape: batch_size * num_of_classes)
        - y: true labels for the sample batch (shape: batch_size * num_of_classes)
    Output:
```

```
        - loss: the scalar loss value for the batch
    """
    # Computes softmax cross entropy between logits (output) and true labels (y)
    xentropy = tf.nn.softmax_cross_entropy_with_logits(logits=output, labels=y)

    # Return the mean cross-entropy loss across the batch
    loss = tf.reduce_mean(xentropy)

    return loss

def evaluate(output, y):
    """
    Evaluates the accuracy on the validation set.
    Input:
        - output: prediction vector of the network for the validation set
        - y: true value for the validation set
    Output:
        - accuracy: accuracy on the validation set (scalar between 0 and 1)
    """
    # Check if the predicted class equals the true class
    correct_prediction = tf.equal(tf.argmax(output, 1), tf.argmax(y, 1))

    # Compute accuracy as the mean of correct predictions
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    # Log validation accuracy using TensorFlow summary (if needed)
    with tf.summary.create_file_writer('./logs/validation').as_default():
        tf.summary.scalar("validation_error", 1.0 - accuracy, step=0)

    return accuracy

def build_model(architecture, neurons_per_layer):
    model = tf.keras.Sequential()
    input_shape = (input_size,)

    for i, activation in enumerate(architecture):
        if activation == 'leaky_relu':
            model.add(Dense(neurons_per_layer, input_shape=input_shape if i == 0 else None))
            model.add(LeakyReLU(negative_slope=0.01))
        else:
            model.add(Dense(neurons_per_layer, activation=activation,
input_shape=input_shape if i == 0 else None))

    model.add(Dense(output_size))
    return model

# Function to plot training history
def plot_training_history(history, architecture_name, neurons_per_layer, optimizer_name,
epochs):
    plt.figure(figsize=(12, 5))

    # Add supertitle for the architecture configuration
    plt.suptitle(f"Architecture: {architecture_name}, Neurons: {neurons_per_layer},
Optimizer: {optimizer_name}")
```

```
    # Plot training and validation loss
    plt.subplot(1, 2, 1)
    plt.plot(history['loss'], label='Training Loss')
    plt.plot(history['val_loss'], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Training and Validation Loss')
    plt.xticks(np.arange(0, epochs, 1))

    # Plot training and validation accuracy
    plt.subplot(1, 2, 2)
    plt.plot(history['accuracy'], label='Training Accuracy')
    plt.plot(history['val_accuracy'], label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Training and Validation Accuracy')
    plt.xticks(np.arange(0, epochs, 1))

    plt.show()

def visualize_error_surface(model, x_train, y_train, start_points):
    """
    Visualizes the error surface by interpolating between the starting weights and final trained weights.

    Parameters:
    - model: Trained model to use for error surface visualization.
    - x_train: Training data features.
    - y_train: Training data labels.
    - start_points: List of initial weights (random starting points) to interpolate from.
    """
    loss_function = tf.keras.losses.CategoricalCrossentropy()

    # Get the final trained weights
    final_weights = model.get_weights()

    # Define the interpolation factor (alpha) values
    alphas = np.linspace(0, 1, 100)

    # Plot the error surface for each starting point
    for idx, start_weights in enumerate(start_points):
        # Store the losses along the interpolation path
        losses = []

        # Interpolate between the starting weights and final weights
        for alpha in alphas:
            # Compute interpolated weights
            interpolated_weights = [(1 - alpha) * start + alpha * final
                                    for start, final in zip(start_weights, final_weights)]
            # Set interpolated weights in the model
            model.set_weights(interpolated_weights)

```

```
            # Compute loss for the interpolated model
            y_pred = model(x_train)
            loss = loss_function(y_train, y_pred).numpy()
            losses.append(loss)

        # Plot the losses for this interpolation path
        plt.plot(alphas, losses, label=f"Start Point {idx + 1}")

    plt.xlabel("Interpolation Factor (Alpha)")
    plt.ylabel("Loss")
    plt.title("Error Surface by Linear Interpolation")
    plt.legend()
    plt.show()

def training_testing(architecture_name, neurons_per_layer, optimizer_name):
    start_time = time.time()

    # Define inputs directly (no need for placeholders)
    input_size = 784
    output_size = 10
    batch_size = 128
    training_epochs = 20
    display_step = 5

    # Instantiate the model with the architecture parameters
    model = build_model(architecture_name, neurons_per_layer)

    # Define optimizer
    if optimizer_name == 'Adam':
        optimizer = tf.optimizers.Adam()
    elif optimizer_name == 'RMSprop':
        optimizer = tf.optimizers.RMSprop()

    # Define the checkpoint manager
    checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
    checkpoint_manager = tf.train.CheckpointManager(checkpoint, './logs/multi_layer',
max_to_keep=5)

    # Training loop
    history = {'loss': [], 'accuracy': [], 'val_loss': [], 'val_accuracy': []}

    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int((train_num_examples + batch_size - 1) / batch_size)

        for i in range(total_batch):
            start = i * batch_size
            end = min(train_num_examples, start + batch_size)
            minibatch_x = x_train[start:end]
            minibatch_y = y_train[start:end]

            # Define training step using GradientTape
            with tf.GradientTape() as tape:
                output = model(minibatch_x)
```

```python
                cost = loss_2(output, minibatch_y)

            # Compute gradients and apply them
            gradients = tape.gradient(cost, model.trainable_variables)
            optimizer.apply_gradients(zip(gradients, model.trainable_variables))

            avg_cost += cost.numpy() / total_batch

        # Append metrics for plotting
        history['loss'].append(avg_cost)
        accuracy = evaluate(model(x_train), y_train)
        history['accuracy'].append(accuracy)

        val_loss = loss_2(model(x_valid), y_valid).numpy()
        val_accuracy = evaluate(model(x_valid), y_valid)
        history['val_loss'].append(val_loss)
        history['val_accuracy'].append(val_accuracy)

        if (epoch+1) % display_step == 0:
            print(f"Epoch: {(epoch+1):2d}, cost={avg_cost:.7f}, Validation Error={1-accuracy:.7f}, Training Accuracy={accuracy}, Validation Accuracy={val_accuracy}")
            checkpoint_manager.save()

    # Final test accuracy
    accuracy = evaluate(model(x_test), y_test)
    print("Test Accuracy:", accuracy)

    elapsed_time = time.time() - start_time
    print(f'Execution time (seconds) was {elapsed_time:.3f}')

    # Call the plotting function to visualize training history
    plot_training_history(history, architecture_name, neurons_per_layer, optimizer_name, training_epochs)

    # Generate a few random starting points for error surface visualization
    start_points = [ [np.random.normal(size=w.shape) for w in model.get_weights()] for _ in range(3)]

    # Visualize error surface by linear interpolation
    visualize_error_surface(model, x_train, y_train, start_points)

def training_testing_diff_lr(architecture_name, neurons_per_layer, optimizer_name, learning_rates):
    results = []  # Store validation accuracy and elapsed time for each learning rate

    # Define inputs
    input_size = 784
    output_size = 10
    batch_size = 128
    training_epochs = 20
    display_step = 5

    # Loop through different learning rates
    for lr in learning_rates:
```

```python
        print(f"\nTraining with learning rate: {lr}")

        # Start timer for this learning rate
        start_time = time.time()

        # Instantiate the model with the architecture parameters
        model = build_model(architecture_name, neurons_per_layer)

        # Define optimizer with the specific learning rate
        if optimizer_name == 'Adam':
            optimizer = tf.optimizers.Adam(learning_rate=lr)
        elif optimizer_name == 'RMSprop':
            optimizer = tf.optimizers.RMSprop(learning_rate=lr)

        # Define the checkpoint manager
        checkpoint = tf.train.Checkpoint(optimizer=optimizer, model=model)
        checkpoint_manager = tf.train.CheckpointManager(checkpoint, './logs/multi_layer', max_to_keep=5)

        # Training loop
        history = {'loss': [], 'accuracy': [], 'val_loss': [], 'val_accuracy': []}

        for epoch in range(training_epochs):
            avg_cost = 0.
            total_batch = int((train_num_examples + batch_size - 1) / batch_size)

            for i in range(total_batch):
                start = i * batch_size
                end = min(train_num_examples, start + batch_size)
                minibatch_x = x_train[start:end]
                minibatch_y = y_train[start:end]

                # Define training step using GradientTape
                with tf.GradientTape() as tape:
                    output = model(minibatch_x)
                    cost = loss_2(output, minibatch_y)

                # Compute gradients and apply them
                gradients = tape.gradient(cost, model.trainable_variables)
                optimizer.apply_gradients(zip(gradients, model.trainable_variables))

                avg_cost += cost.numpy() / total_batch

            # Append metrics for plotting
            history['loss'].append(avg_cost)
            accuracy = evaluate(model(x_train), y_train)
            history['accuracy'].append(accuracy)

            val_loss = loss_2(model(x_valid), y_valid).numpy()
            val_accuracy = evaluate(model(x_valid), y_valid)
            history['val_loss'].append(val_loss)
            history['val_accuracy'].append(val_accuracy)

            if (epoch+1) % display_step == 0:
```

```
                print(f"Epoch: {(epoch+1):2d}, cost={avg_cost:.7f}, Validation Error={1-accuracy:.7f}, Training Accuracy={accuracy}, Validation Accuracy={val_accuracy}")
                checkpoint_manager.save()

        # Final test accuracy
        test_accuracy = evaluate(model(x_test), y_test)
        print("Test Accuracy:", test_accuracy)

        # Record elapsed time
        elapsed_time = time.time() - start_time
        print(f'Execution time (seconds) was {elapsed_time:.3f}')

        # Store results for this learning rate
        results.append({
            'learning_rate': lr,
            'final_val_accuracy': val_accuracy,
            'test_accuracy': test_accuracy,
            'time': elapsed_time
        })

    # Display summary of results for all learning rates
    for result in results:
        print(f"Learning Rate: {result['learning_rate']}, Final Validation Accuracy: {result['final_val_accuracy']}, Test Accuracy: {result['test_accuracy']}, Time Taken: {result['time']} seconds")

    # Return results for further analysis if needed
    return results

if __name__ == '__main__':
    architectures = {
    "1_tanh_2_sigmoid_3_leaky_relu": ['tanh', 'sigmoid', 'leaky_relu'],
    "1_tanh_2_sigmoid_3_sigmoid_4_relu": ['tanh', 'sigmoid', 'sigmoid', 'relu'],
    "3_layers_sigmoid": ['sigmoid', 'sigmoid', 'sigmoid'],
    "3_layers_leaky_relu": ['leaky_relu', 'leaky_relu', 'leaky_relu'],
    "4_layers_tanh": ['tanh', 'tanh', 'tanh', 'tanh'],
    "4_layers_leaky_relu": ['leaky_relu', 'leaky_relu', 'leaky_relu', 'leaky_relu']}

    experiment_configurations = [
    ("1_tanh_2_sigmoid_3_leaky_relu", 50, "Adam"),
    ("1_tanh_2_sigmoid_3_leaky_relu", 50, "RMSprop"),
    ("1_tanh_2_sigmoid_3_leaky_relu", 100, "Adam"),
    ("1_tanh_2_sigmoid_3_leaky_relu", 100, "RMSprop"),
    ("1_tanh_2_sigmoid_3_sigmoid_4_relu", 50, "Adam"),
    ("1_tanh_2_sigmoid_3_sigmoid_4_relu", 50, "RMSprop"),
    ("1_tanh_2_sigmoid_3_sigmoid_4_relu", 100, "Adam"),
    ("1_tanh_2_sigmoid_3_sigmoid_4_relu", 100, "RMSprop"),
    ("3_layers_sigmoid", 50, "Adam"),
    ("3_layers_sigmoid", 50, "RMSprop"),
    ("3_layers_sigmoid", 100, "Adam"),
    ("3_layers_sigmoid", 100, "RMSprop"),
    ("3_layers_leaky_relu", 50, "Adam"),
    ("3_layers_leaky_relu", 50, "RMSprop"),
    ("3_layers_leaky_relu", 100, "Adam"),
```

```
    ("3_layers_leaky_relu", 100, "RMSprop"),
    ("4_layers_tanh", 50, "Adam"),
    ("4_layers_tanh", 50, "RMSprop"),
    ("4_layers_tanh", 100, "Adam"),
    ("4_layers_tanh", 100, "RMSprop"),
    ("4_layers_leaky_relu", 50, "Adam"),
    ("4_layers_leaky_relu", 50, "RMSprop"),
    ("4_layers_leaky_relu", 100, "Adam"),
    ("4_layers_leaky_relu", 100, "RMSprop")]

    for configuration in experiment_configurations:
        architecture_name, neurons_per_layer, optimizer_name = configuration
        training_testing(architectures[architecture_name], neurons_per_layer, optimizer_name)

    learning_rates = [0.0001, 0.001, 0.01, 0.1, 0.5]
    results = training_testing_diff_lr(['tanh', 'sigmoid', 'leaky_relu'], 50, "Adam", learning_rates)
```