Amarthya Kathula ASU Id: 1229583071 akathula@asu.edu

CSE 569: Fundamentals of Statistical Learning and Pattern Recognition

Project Part 2

TASK-1:

Introduction:

This project focuses on building a Multi-Layer Perceptron (MLP) from the beginning to classify input into two categories. The dataset is divided into two classes, and the goal is to explore the effect of different numbers of hidden nodes (nH) on the performance of the MLP. The MLP is trained using 1500 samples from each class and validated using the remaining 500 examples. The mean and standard deviation of the training data are used for feature normalization. The MLP's classification accuracy on testing data is evaluated for various nH setups

Methodology and Implementation:

Step 1: Constructing a 2-nH-1 MLP

During this part of the project, a 2-nH-1 Multi-Layer Perceptron (MLP) was built from scratch to meet the specifications. A variety of hidden layer sizes (nH) with varied activation functions were used in the implementation. For training the network, the Mean Squared Error (MSE) was chosen as the loss function.

The Neural Network class is the central element in the given code for building a multi-layer perceptron (MLP). This class contains the neural network's structure and activity, offering crucial capabilities for training, prediction, and assessment. When the class is instantiated, it enables for the modification of the neural network's architecture by defining the input size, hidden layer size, and output size. One hidden layer and one output layer comprise the network architecture. The class includes methods for initializing the network, performing forward and reverse propagation, making predictions, and calculating errors.

The b_prop method manages backward propagation, which is critical for updating weights and minimizing errors during training. The error is calculated at the output layer and propagated backward through the network, with weights updated based on the ReLU activation function. The train method manages the training process by iterating over epochs, performing backpropagation, and adjusting weights to minimize mean squared error. It also monitors and reports mistakes in training, validation, and testing across epochs.

```
def f_prop(self, sample):
    inputs = np.hstack(([1], sample[:]))
    for layer in self.layers:
        outputs = [max(0, np.dot(neuron['weights'], inputs)) for neuron in layer]
        for i, neuron in enumerate(layer):
            neuron['output'] = outputs[i]
        inputs = np.hstack(([1], outputs[:]))
    return inputs[1:]
```

The provided code's f_prop method is responsible for performing the forward propagation step across the neural network. The process of passing input data through the network's layers to obtain the final output is known as forward propagation.

```
def b_prop(self, true_label):
    # Perform backpropagation to update weights based on the error
    Out_L = self.layers[self.Out_L_idx]

    for neuron in Out_L:
        error = true_label - neuron['output']
        neuron['delta'] = error if neuron['output'] > 0 else 0

    Hidden_L = self.layers[self.Hidden_L_idx]

    for n in range(len(Hidden_L)):
        error = sum(o_neuron['weights'][n + 1] * o_neuron['delta'] for o_neuron in Out_L)
        Hidden_L[n]['delta'] = error if Hidden_L[n]['output'] > 0 else 0
```

The backpropagation step in the neural network is handled by the b_prop method in the given code. Backpropagation is a supervised learning algorithm that modifies the neural network weights based on the forward propagation error.

Step 2: Testing and Training

The network was trained with the first 1500 examples from each class in Train1.txt and Train2.txt, with the remaining 500 samples used as a "validation set." Using mean and standard deviation estimations from the training data, feature normalization was performed. The trained network was then tested on the testing data from Test1.txt and Test2.txt.

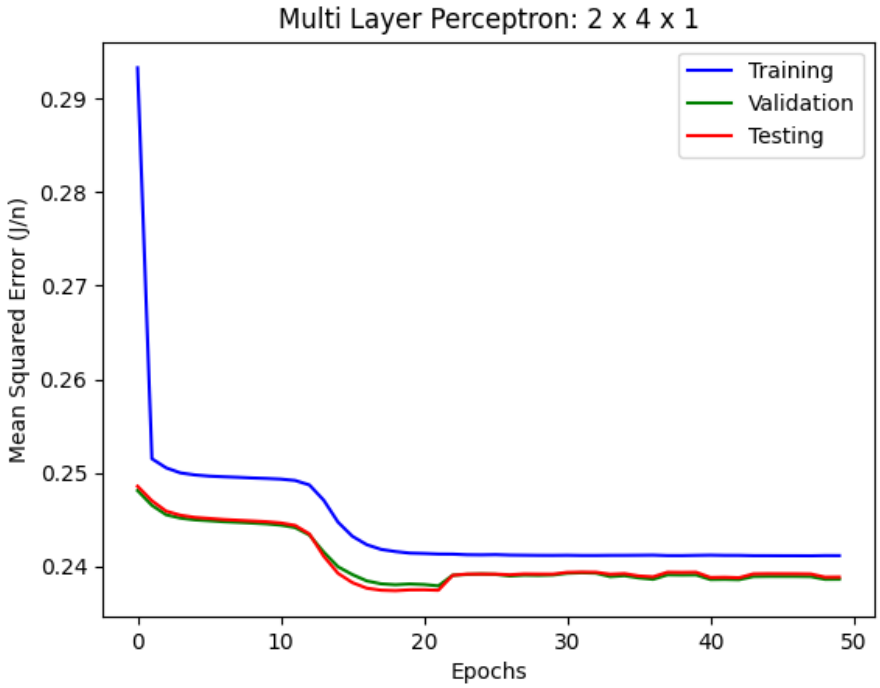
```
x = (x - x.mean(axis=0))/x.std(axis=0) if normalize else

y = np.zeros((x.shape[0], 1), dtype='int') if label == 0 else np.ones((x.shape[0], 1), dtype='int')

return np.hstack((x, y))
```

The above code lines are used to normalize the data and then store the normalized data in a NumPy stack.

The '*train*' function encompasses the entire process of training a neural network and evaluating its performance on the testing dataset. The function begins by retrieving the relevant data, including training, validation, and testing sets, using the *'DataHandler'* class and assuring randomization in the training and validation datasets. The neural network is then configured by assigning training, validation, and testing data, as well as their accompanying labels. The neural network is trained using the '*train'* technique, which uses backpropagation to update weights and reduce mean squared error. Finally, the function assesses the trained neural network on the testing data, computes accuracy by comparing predicted labels to actual labels, and reports the attained accuracy on the testing set. This unified strategy allows for the smooth execution of the entire project.

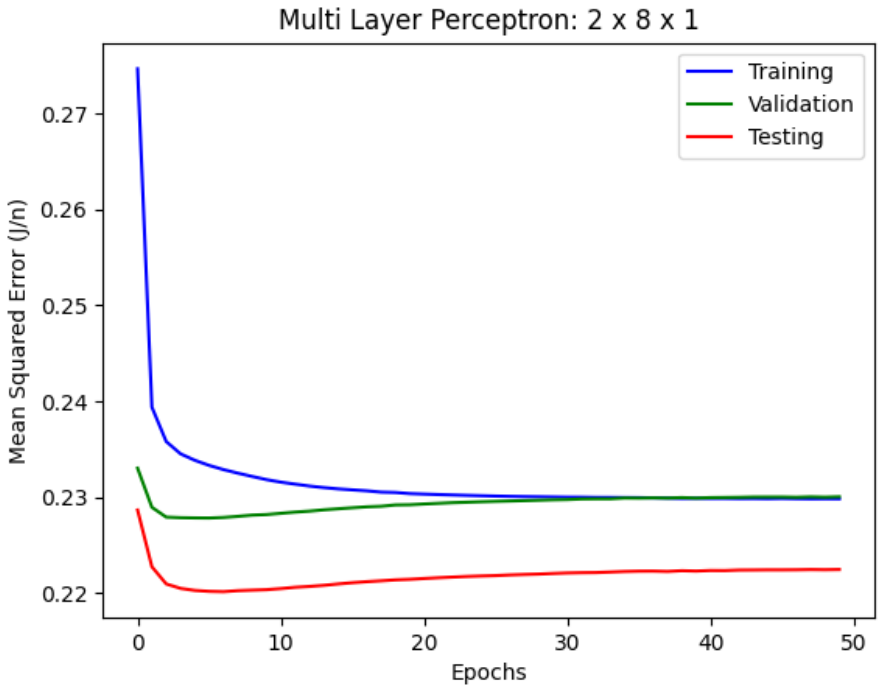
Step 3: Learning Curves

Learning curves were generated for each nH value, illustrating the Mean Squared Error across epochs for the training, validation, and testing sets. The following curves offer insights into the convergence behavior and generalization capabilities of the network.



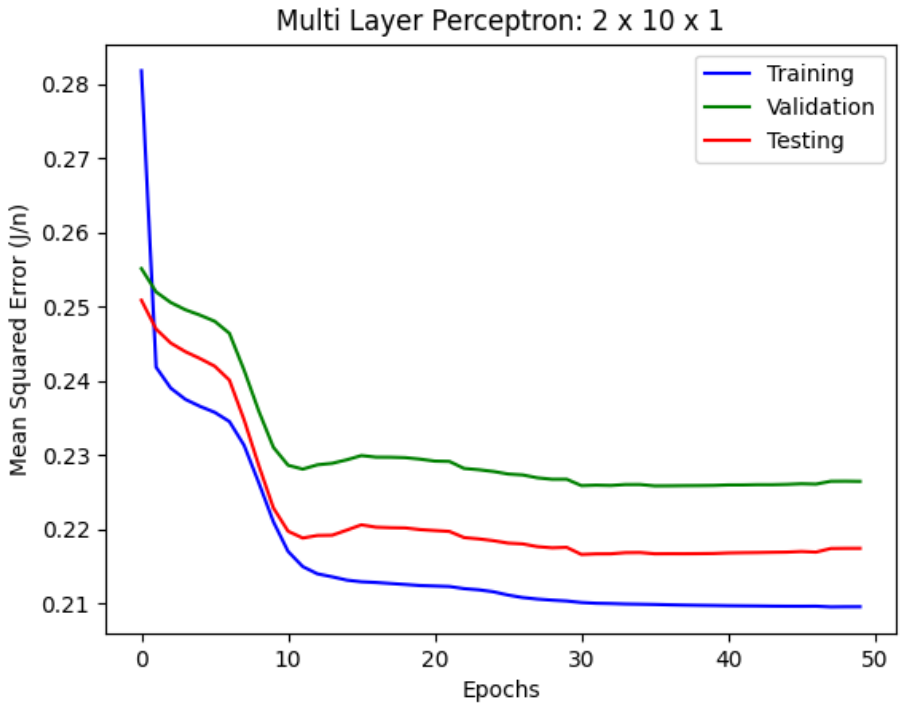
Accuracy: 0.523

On the testing set, the model with nH = 4 has relatively poor accuracy, implying that the network may lack the complexity to capture intricate patterns in the data. This could be a sign of underfitting.



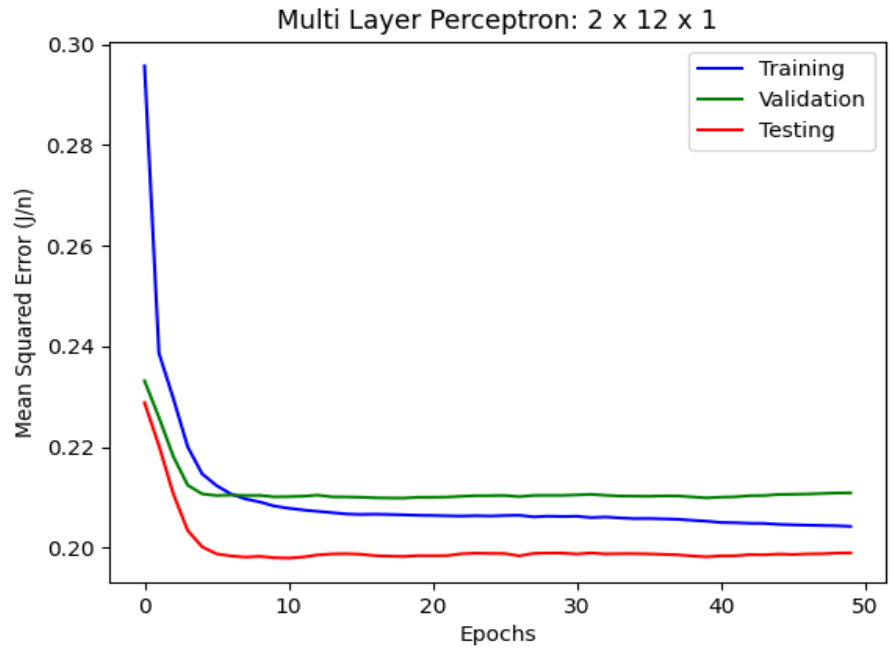
Accuracy: 0.655

Increasing the number of hidden nodes to 8 improves accuracy over nH = 4. The model with nH = 8 performs better in terms of generalization, capturing more complex aspects of the data.



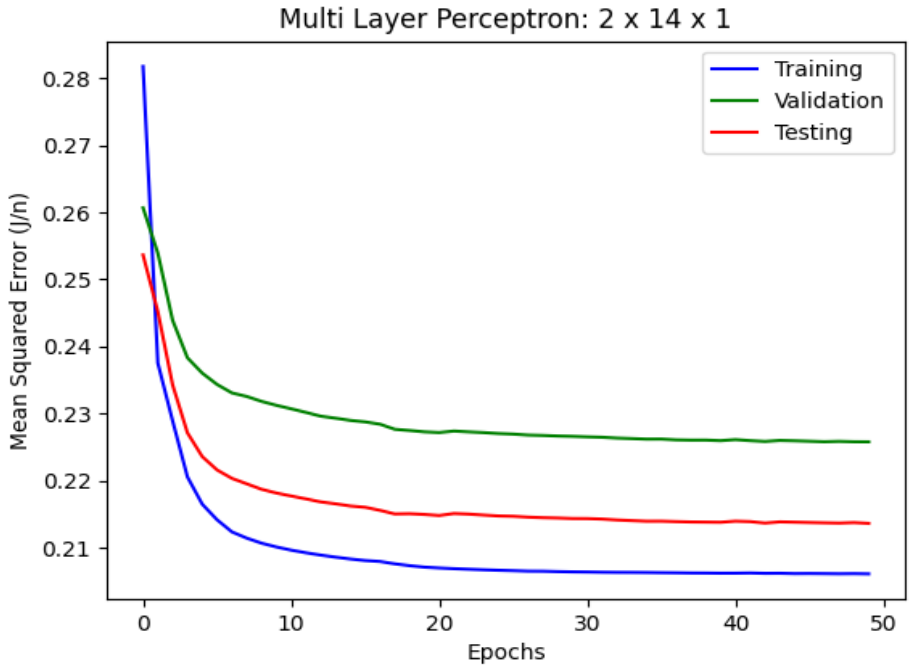
Accuracy: 0.6865

With nH = 10, the accuracy improves further, indicating that a moderate increase in the number of hidden nodes contributes to improved model performance. The learning curve indicates that accuracy is improving.



Accuracy: 0.7095

The model with nH = 12 has a higher accuracy, demonstrating that a bigger hidden layer size improves the network's ability to learn complicated representations from data.



Accuracy: 0.689

Surprisingly, the accuracy drops slightly compared to nH = 12. This may be due to the risk of overfitting as the network grows increasingly complicated. The model may begin to capture noise in the training data, resulting in less generalization on the testing set.

Results and Observations:

Based on this we can conclude that increasing the number of hidden nodes up to a specific point (nH = 12) improves the model's ability to learn and generalize. A higher nH value (nH = 14) may result in overfitting, underscoring the need to establish an ideal balance between model complexity and generalization.

TASK-2:

Introduction:

This project focuses on the use of Convolutional Neural Networks (CNNs) for Handwritten Digit Recognition with the MNIST dataset. The goal is to examine the effect of various parameters on the performance of the CNN. Convolutional layers, max-pooling layers, and fully connected layers are all part of the CNN architecture. Kernel size, the number of feature maps, the number of neurons in fully connected layers, and the learning rate are all systematically modified to see how they affect test accuracy.

Methodology and Implementation:

Convolutional layers with max-pooling are followed by fully linked layers and a SoftMax output layer in the CNN. The architecture consists of two convolutional layers followed by max-pooling and two fully linked layers.

The corresponding code snippet implements a method called 'create_model' that uses the Keras library to build a Convolutional Neural Network (CNN). This function allows the CNN architecture to be customized by specifying parameters such as the number of filters in the convolutional layers (filters), the kernel size for convolution (kernel_size), the number of neurons in the fully connected layers (fc1_neurons and fc2_neurons), the learning rate (learning_rate), the input shape (input_shape), and the number of output classes (num_classes). Convolutional layers with ReLU activation functions are used in the CNN design, followed by max-pooling for spatial down sampling.

```
def create_model(filters, kernel_size, fc1_neurons, fc2_neurons, learning_rate, input_shape,
num_classes=10):
  model = Sequential([
    Conv2D(filters, kernel_size=kernel_size, activation='relu', input_shape=input_shape),
    MaxPooling2D(pool_size=(2, 2), strides=(1, 1)),
    Conv2D(2 * filters, kernel_size=kernel_size, activation='relu', strides=(1, 1)),
    MaxPooling2D(pool_size=(2, 2), strides=(1, 1)),
    Flatten(),
    Dense(fc1_neurons, activation='relu'),
    Dense(fc2_neurons, activation='relu'),
    Dense(num_classes, activation='softmax')
  ])
```

The result is then flattened to create a 1D array, ready for fully connected layers with ReLU activation. Using the softmax activation function, the final dense layer reflects the output classes. The Adam optimizer is used to assemble the model, with categorical crossentropy as the loss function. Based on the parameters and structure specified, this architecture is intended for picture classification tasks, notably handwritten digit recognition.

The train_and_evaluate_model function is in charge of training and evaluating a specified neural network model using the training and testing datasets provided. The function accepts numerous parameters, including the neural network model (model), training data (x_train and y_train), testing data (x_test and y_test), training batch size (batch_size), and epoch count (epochs).

```
def train_and_evaluate_model(model, x_train, y_train, x_test, y_test, batch_size=64, epochs=5):
  model.fit(x_train, y_train,
        batch_size=batch_size,
        epochs=epochs,
        verbose=1,
        validation_data=(x_test, y_test))

  score = model.evaluate(x_test, y_test, verbose=0)
  print('Test loss:', score[0])
  print('Test accuracy:', score[1])
```

The fit method is used within the function to train the model on the training data. It sets the batch size, the number of epochs, and the validation data for assessing the model's performance during training on the testing set. To provide feedback on the learning process, the training progress is displayed (verbose=1). The model is assessed on the testing data after training using the evaluate technique. The evaluation findings are then printed to the console, including the test loss and accuracy.

Training and Evaluating the Model:

To train the model, we load the images from MNIST, and then we consider different parameters of feature maps, number of neurons, learning rate, etc., as described below.

```
# Experiment 1: Explore the impact of kernel size on accuracy
model1 = create_model(filters=32, kernel_size=(3, 3), fc1_neurons=128, fc2_neurons=64,
learning_rate=0.001,
                      input_shape=input_shape)
train_and_evaluate_model(model1, x_train, y_train, x_test, y_test)
```

Observations:

For this particular experiment, we have the parameters as: 32 filters, kernel size (3, 3), fc1_neurons=128, fc2_neurons=64, learning_rate=0.001.

```
Experiment 1:
Exploring the impact of kernel size on accuracy.
Epoch 1/5
938/938 [==============================] - 77s 82ms/step - loss: 0.1097 - accuracy: 0.9654 - val_loss: 0.0449 - val_accuracy: 0.9855
Epoch 2/5
938/938 [==============================] - 79s 85ms/step - loss: 0.0371 - accuracy: 0.9887 - val_loss: 0.0421 - val_accuracy: 0.9862
Epoch 3/5
938/938 [==============================] - 73s 78ms/step - loss: 0.0252 - accuracy: 0.9921 - val_loss: 0.0409 - val_accuracy: 0.9882
Epoch 4/5
938/938 [==============================] - 72s 77ms/step - loss: 0.0173 - accuracy: 0.9945 - val_loss: 0.0371 - val_accuracy: 0.9888
Epoch 5/5
938/938 [==============================] - 73s 77ms/step - loss: 0.0153 - accuracy: 0.9951 - val_loss: 0.0353 - val_accuracy: 0.9896
Test loss: 0.035328906029462814
Test accuracy: 0.9896000027656555
```

A smaller kernel size (3x3) leads to greater accuracy. This means that a narrower receptive field captures local features more effectively, and the model benefits from more detailed feature extraction.

Similarly, for the other experiments, the parameters are as follows:

Experiment 2 - Parameters: 64 filters, kernel size (3, 3), fc1_neurons=128, fc2_neurons=64, learning_rate=0.001.

```
Experiment 2:
Exploring the impact of the number of feature maps on accuracy.
Epoch 1/5
938/938 [==============================] - 153s 162ms/step - loss: 0.1117 - accuracy: 0.9658 - val_loss: 0.0518 - val_accuracy: 0.9823
Epoch 2/5
938/938 [==============================] - 158s 169ms/step - loss: 0.0366 - accuracy: 0.9888 - val_loss: 0.0378 - val_accuracy: 0.9889
Epoch 3/5
938/938 [==============================] - 165s 176ms/step - loss: 0.0244 - accuracy: 0.9920 - val_loss: 0.0310 - val_accuracy: 0.9900
Epoch 4/5
938/938 [==============================] - 163s 174ms/step - loss: 0.0169 - accuracy: 0.9948 - val_loss: 0.0376 - val_accuracy: 0.9903
Epoch 5/5
938/938 [==============================] - 165s 176ms/step - loss: 0.0143 - accuracy: 0.9954 - val_loss: 0.0416 - val_accuracy: 0.9877
Test loss: 0.0416429303586483
Test accuracy: 0.9876999855041504
```

Increasing the amount of feature maps increases accuracy. This implies that a more diverse set of learned properties is advantageous for recognizing complicated patterns in handwritten digits.

Experiment 3 - Parameters: 32 filters, kernel size (3, 3), fc1_neurons=64, fc2_neurons=32, learning_rate=0.001.

```
Experiment 3:
Exploring the impact of the number of neurons in the fully connected layers on accuracy.
Epoch 1/5
938/938 [==============================] - 72s 76ms/step - loss: 0.1226 - accuracy: 0.9625 - val_loss: 0.0462 - val_accuracy: 0.9851
Epoch 2/5
938/938 [==============================] - 74s 79ms/step - loss: 0.0388 - accuracy: 0.9880 - val_loss: 0.0371 - val_accuracy: 0.9887
Epoch 3/5
938/938 [==============================] - 72s 76ms/step - loss: 0.0271 - accuracy: 0.9914 - val_loss: 0.0354 - val_accuracy: 0.9902
Epoch 4/5
938/938 [==============================] - 70s 75ms/step - loss: 0.0197 - accuracy: 0.9936 - val_loss: 0.0370 - val_accuracy: 0.9888
Epoch 5/5
938/938 [==============================] - 72s 76ms/step - loss: 0.0150 - accuracy: 0.9951 - val_loss: 0.0379 - val_accuracy: 0.9887
Test loss: 0.03792642802000046
Test accuracy: 0.9886999726295471
```

A little reduction in the number of neurons in completely linked layers has no discernible effect on accuracy. This suggests that in the completely connected layers, the network can sustain performance with a less complex topology.

Experiment 4 - Parameters: 32 filters, kernel size (3, 3), fc1_neurons=128, fc2_neurons=64, learning_rate=0.0001.

```
Experiment 4:
Exploring the impact of learning rate on accuracy.
Epoch 1/5
938/938 [==============================] - 77s 82ms/step - loss: 0.2529 - accuracy: 0.9280 - val_loss: 0.0785 - val_accuracy: 0.9766
Epoch 2/5
938/938 [==============================] - 79s 84ms/step - loss: 0.0715 - accuracy: 0.9792 - val_loss: 0.0518 - val_accuracy: 0.9829
Epoch 3/5
938/938 [==============================] - 77s 82ms/step - loss: 0.0493 - accuracy: 0.9853 - val_loss: 0.0389 - val_accuracy: 0.9878
Epoch 4/5
938/938 [==============================] - 83s 88ms/step - loss: 0.0382 - accuracy: 0.9887 - val_loss: 0.0363 - val_accuracy: 0.9867
Epoch 5/5
938/938 [==============================] - 77s 82ms/step - loss: 0.0307 - accuracy: 0.9907 - val_loss: 0.0325 - val_accuracy: 0.9895
Test loss: 0.03247303143143654
Test accuracy: 0.9894999861717224
```

Reduced learning rate reduces accuracy. This emphasizes the significance of an adequate learning rate in allowing for successful weight changes throughout training. A poor learning rate may impede convergence and the model's capacity to adapt.

Experiment 5 - Parameters: 64 filters, kernel size (5, 5), fc1_neurons=64, fc2_neurons=32, learning_rate=0.0001.

```
Experiment 5:
Exploring a combination of changes in parameters.
Epoch 1/5
938/938 [==============================] - 194s 206ms/step - loss: 0.2502 - accuracy: 0.9250 - val_loss: 0.0702 - val_accuracy: 0.9777
Epoch 2/5
938/938 [==============================] - 178s 189ms/step - loss: 0.0692 - accuracy: 0.9800 - val_loss: 0.0474 - val_accuracy: 0.9858
Epoch 3/5
938/938 [==============================] - 179s 191ms/step - loss: 0.0507 - accuracy: 0.9853 - val_loss: 0.0375 - val_accuracy: 0.9879
Epoch 4/5
938/938 [==============================] - 182s 194ms/step - loss: 0.0397 - accuracy: 0.9877 - val_loss: 0.0363 - val_accuracy: 0.9884
Epoch 5/5
938/938 [==============================] - 187s 199ms/step - loss: 0.0321 - accuracy: 0.9906 - val_loss: 0.0335 - val_accuracy: 0.9896
Test loss: 0.03350479528307915
Test accuracy: 0.9896000027656555
```

A combination of modifications results in competitive accuracy. This demonstrates how significant changes to various parameters can improve the overall performance of the CNN. It underlines the importance of taking a deliberate and balanced approach to parameter tuning.

Conclusion:

At last, the tests provide insight into the behavior of a Convolutional Neural Network (CNN) for handwritten digit recognition. The appropriate kernel size, number of feature maps, neurons in fully connected layers, and learning rate are determined by the dataset and task features. The findings emphasize the significance of fine-tuning parameters to attain the greatest performance and show the trade-offs associated with various options. This can be critical in creating successful CNN architectures for image classification applications.