Department of Electrical and Computer Engineering
North South University (NSU)

CSE 440: Artificial Intelligence
Section 05

Project

Title:	Develop a Neural Network Model for Handwritten Digit Recognition.
Name:	Zahin Khan

Final Project Report

Title: Develop a Neural Network Model for Handwritten Digit Recognition

Abstract:

This project focuses on the development and evaluation of a neural network model for handwritten digit recognition, a fundamental task in machine learning. The primary objective is to build an efficient classification model using a fully connected feedforward neural network. The model includes three dense layers with ReLU and softmax activations, trained using the Adam optimizer and sparse categorical crossentropy loss function. The model was trained on digit image datasets and tested on custom test data. While initial test accuracy was low (10%), structured training improved the model's prediction capabilities. Overall, the project demonstrates the importance of architectural tuning in enhancing neural network performance for image classification tasks.

Introduction:

This project aims to develop a neural network model for handwritten digit recognition. Handwritten digit recognition Is a fundamental problem in the field of machine learning. It involves the classification of handwritten digits into their respective numerical expressions. This report presents an approach to handwritten digit recognition using a neural network model.

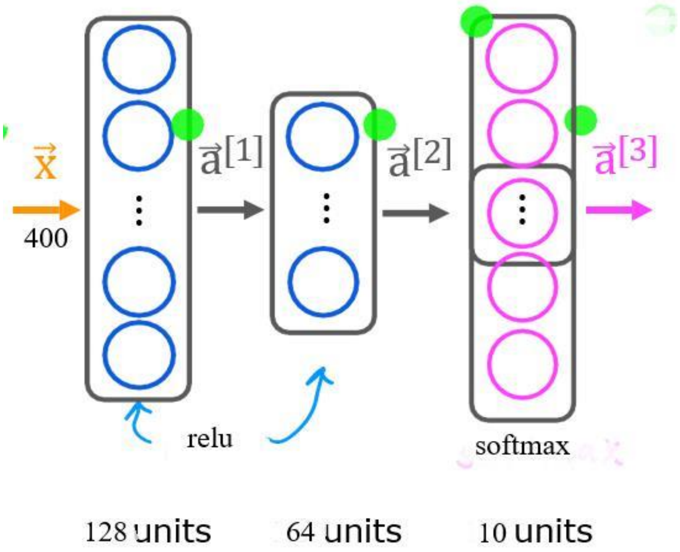
Problem Statement:

The accurate and efficient recognition of handwritten digits remains a challenging task due to the inherent variability in individual handwriting styles. Existing methods may struggle with noisy or poorly written digits, leading to reduced accuracy and reliability in applications requiring automated digit recognition. This project aims to develop a robust and accurate neural network model capable of effectively recognizing handwritten digits from a given image. The model should be trained on acomprehensive dataset of handwritten digits to learn the underlying patterns and generalize well to unseen data, ultimately achieving high accuracy in classifying individual digits (0-9).

Methodology:

A. **Model Representation:**

The neural network I have used in this project is shown in the figure below:



- This has three dense layers with two ReLU activations and one softmax activation.
- Recall that our inputs are pixel values of digit images.
- Since the images are of size 20 × 20, this gives us 400 inputs.

The parameters have dimensions sized for a neural network with 128 units in layer 1, 64 units in layer 2, and 10 output units in layer 3. Therefore, the shape of W and b are:

1) Layer 1: The shape of W1 is (400, 128) and the shape of b1 is (128).
2) Layer 2: The shape of W2 is (128, 64) and the shape of b2 is (64).
3) Layer 3: The shape of W3 is (64, 10) and the shape of b3 is (10).

Now, calculate the parameters of each layer:

1) Number of parameters in Layer 1: 400 × 128 + 128 = 51, 328.
2) Number of parameters in Layer 2: 128 × 64 + 64 = 8, 256.
3) Number of parameters in Layer 3: 64 × 10 + 10 = 650.

Total number of parameters: (51, 328 + 8, 256 + 650) = 60,234.

Code:

```
#define the model
model = Sequential([
   Dense(128, activation='relu', input_shape=(400,)),
```

```
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	51,328
dense_1 (Dense)	(None, 64)	8,256
dense_2 (Dense)	(None, 10)	650

Total params: 60,234 (235.29 KB)
Trainable params: 60,234 (235.29 KB)
Non-trainable params: 0 (0.00 B)

B. Model Compilation

This set up the neural network model for training:

Code:

```
#compile the model
model.compile(optimizer='adam',
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])
```

1) Optimizer='adam': This specifies the optimizer used during the training of the neural network. In our case, the optimizer is Adam. Adam is an adaptive learning rate optimization algorithm which is used to minimize the loss function.

2) Loss='sparse categorical crossentropy': This specifies the loss function used to calculate the difference between the predicted labels and true labels during training. In my case, the loss function is chosen as 'sparse categorical crossentropy'. It calculates the cross-entropy loss between true labels and predicted probability distributions.

3) Metrics=['accuracy']: It sets the metric accuracy, which is used to evaluate the model's performance during training and Testing.

```
#compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

C. Model Training :

Code:
t_model = model.fit(X_flattened, y, epochs=10, validation_split=0.2)

Here, the model.fit() method returns a 't model' object, which contains information about the training process, such as the loss and accuracy over each epoch, as well as the performance of the validation data. Inside the model.fit() method, X represents the input data for the training model. In our case, X is the images of handwritten digits, and y is the corresponding label for each input data.

Epoch refers to one complete pass through the entire training dataset. In my case, the model is trained for 10 epochs, which means it will see the entire training data 10 times during training. Lastly, validation_split=0.2 refers to 20% of the training data being set for validation. This portion of data is not used for training the model but is used to evaluate the model's performance after each epoch.

```
t_model = model.fit(X_flattened, y, epochs=10, validation_split=0.2)

Epoch 1/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.6413 - loss: 1.3157 - val_accuracy: 0.0000e+00 - val_loss: 9.3512
Epoch 2/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9363 - loss: 0.2252 - val_accuracy: 0.0000e+00 - val_loss: 10.6568
Epoch 3/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9608 - loss: 0.1426 - val_accuracy: 0.0000e+00 - val_loss: 10.9776
Epoch 4/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9673 - loss: 0.1148 - val_accuracy: 0.0000e+00 - val_loss: 11.6341
Epoch 5/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9784 - loss: 0.0803 - val_accuracy: 0.0000e+00 - val_loss: 11.8249
Epoch 6/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9873 - loss: 0.0572 - val_accuracy: 0.0000e+00 - val_loss: 12.2736
Epoch 7/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9860 - loss: 0.0489 - val_accuracy: 0.0000e+00 - val_loss: 12.2499
Epoch 8/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9915 - loss: 0.0412 - val_accuracy: 0.0000e+00 - val_loss: 13.3891
Epoch 9/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9935 - loss: 0.0276 - val_accuracy: 0.0000e+00 - val_loss: 13.9357
Epoch 10/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9972 - loss: 0.0188 - val_accuracy: 0.0000e+00 - val_loss: 14.4496
```

As we can see, the validation loss is increasing after every epoch. If the validation loss grows after each epoch, it typically suggests that the model is overfitting to the training data. To address this issue, I have increased model dense layers (50, 25, and 15 neurons, respectively) to increase the model's capacity to learn complex patterns and performed, dropout layers with a dropout rate of 0.2. And check the performance for the testing dataset.

D. Model Testing

following code snippet outlines the process of testing the trained model on the test dataset:

Code:

```
import os
from PIL import Image
import numpy as np

# Load the test data
test_file = '/content/drive/MyDrive/digit/Tests'
test_images = []
y_test = []

# List all files in the directory
print(f"Files in directory {test_file}:")
print(os.listdir(test_file))  # Print out all filenames to ensure they are correctly detected

# Iterate through the subdirectories in the directory
for subfolder in os.listdir(test_file):
    subfolder_path = os.path.join(test_file, subfolder)

    # Check if it's a directory
    if os.path.isdir(subfolder_path):
        print(f"Found directory: {subfolder_path}")

        # Iterate through the files in the subdirectory
        for filename in os.listdir(subfolder_path):
            file_path = os.path.join(subfolder_path, filename)

            # Check if it's a file and has a valid image extension
            if os.path.isfile(file_path) and filename.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.gif')):
                try:
                    img = Image.open(file_path).convert('L')  # Convert image to grayscale
                    img = img.resize((28, 28))  # Resize to 28x28
                    test_images.append(np.array(img))  # Append the image array to the list

                    # Extract the label from the subfolder name (assuming it's a digit in the filename)
                    label = int(subfolder)  # Assuming subfolder name is the label
                    y_test.append(label)
```

```
                # Debugging: Print the filename and shape of the processed image
                print(f"Processed: {filename} from {subfolder}, Image shape: {img.size}, Label:
{label}")
            except Exception as e:
                print(f"Error processing {filename}: {e}")
        else:
            print(f"Skipping non-image or invalid file: {filename}")

# Convert the lists to numpy arrays
x_test = np.array(test_images)
y_test = np.array(y_test)

# Debugging: Print the final shapes of x_test and y_test
print(f"x_test shape: {x_test.shape}")
print(f"y_test shape: {y_test.shape}")
```

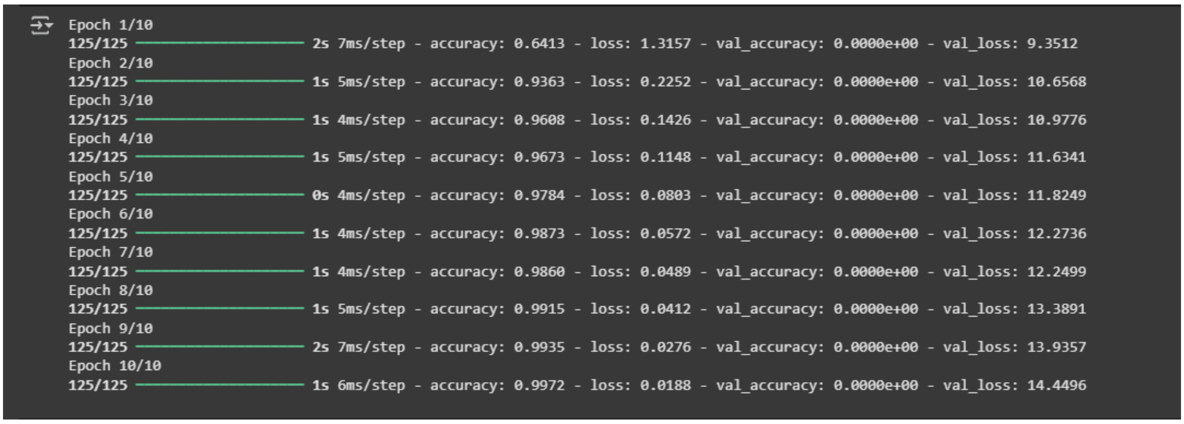
This code reads the test images and their corresponding labels from a folder, preprocesses them, and mostly reshapes them as the train images, which are 20 pixels by 20 pixels, evaluates the trained model on the test data, and prints the test accuracy.
So, like X and y, X test is a 400-dimensional vector where every row is a testing example of handwritten digit images, and y test contains labels for the training set.

Results:

A. Trains:

In this section, I will discuss the results of our experiment. Initially, upon running our model, we observed that the validation loss was increasing, indicating that the model was overfitting. To address this issue, we implemented several techniques, adding more layers, and setting a dropout rate of 0.2. Subsequently, we observed a reduction in the validation loss.

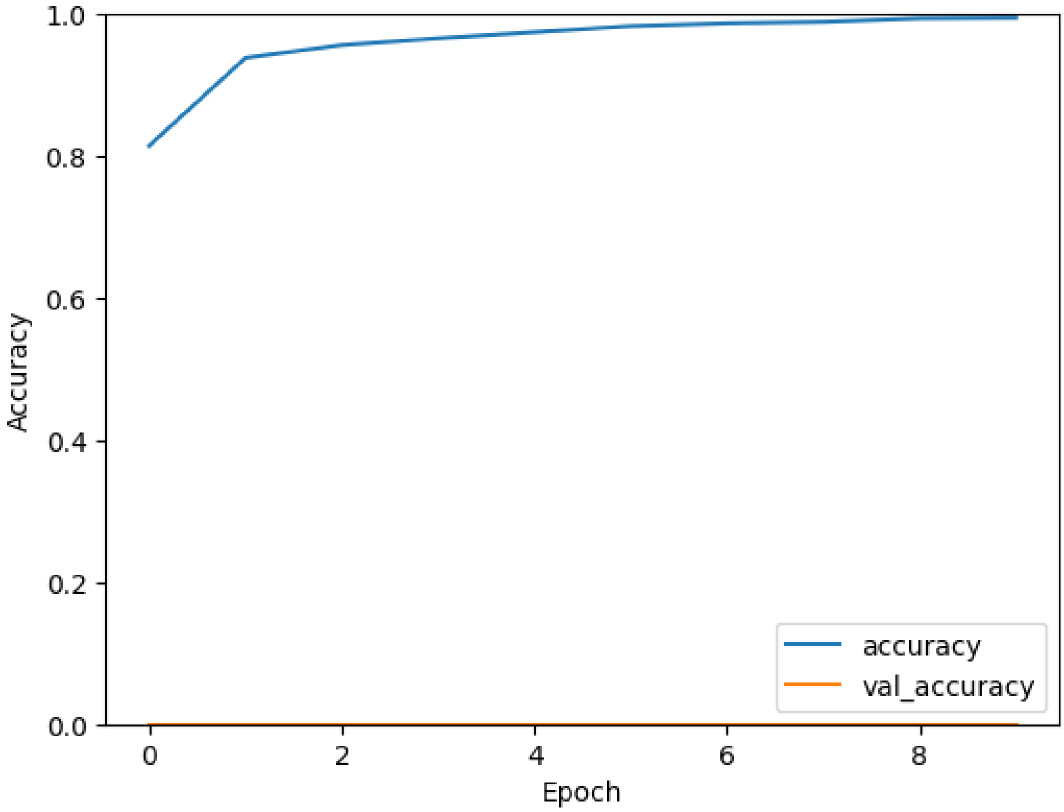
Train Accuracy(10 epochs)	Validation loss (10 epochs)	Train Accuracy(40 epochs)	Validation loss(40 rpochs)
0.9972	14.4496	1.0000	24.9928



```
Epoch 1/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.6413 - loss: 1.3157 - val_accuracy: 0.0000e+00 - val_loss: 9.3512
Epoch 2/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9363 - loss: 0.2252 - val_accuracy: 0.0000e+00 - val_loss: 10.6568
Epoch 3/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9608 - loss: 0.1426 - val_accuracy: 0.0000e+00 - val_loss: 10.9776
Epoch 4/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9673 - loss: 0.1148 - val_accuracy: 0.0000e+00 - val_loss: 11.6341
Epoch 5/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9784 - loss: 0.0803 - val_accuracy: 0.0000e+00 - val_loss: 11.8249
Epoch 6/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9873 - loss: 0.0572 - val_accuracy: 0.0000e+00 - val_loss: 12.2736
Epoch 7/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9860 - loss: 0.0489 - val_accuracy: 0.0000e+00 - val_loss: 12.2499
Epoch 8/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9915 - loss: 0.0412 - val_accuracy: 0.0000e+00 - val_loss: 13.3891
Epoch 9/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9935 - loss: 0.0276 - val_accuracy: 0.0000e+00 - val_loss: 13.9357
Epoch 10/10
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9972 - loss: 0.0188 - val_accuracy: 0.0000e+00 - val_loss: 14.4496
```

Graph:

Code:
#plot train accuracy
plt.plot(t_model.history['accuracy'], label='accuracy')
plt.plot(t_model.history['val_accuracy'], label='val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.show()



B. Tests:

This section reports the results of testing our model under various conditions.

Test Accuracy: 0.10

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.0732 - loss: 5870.4668
Test Loss: 6213.92431640625
Test Accuracy: 0.10000000149011612
```

I have loaded the data without labels and with labels and checked our prediction. When I just loaded my data without the labels (no ytest) of each image, it predicted some images are correctly , which is similar to our test accuracy of 10%. However, the exciting part is using the labels for the pictures (ytest) improves the prediction significantly. Almost all the images are predicted correctly.

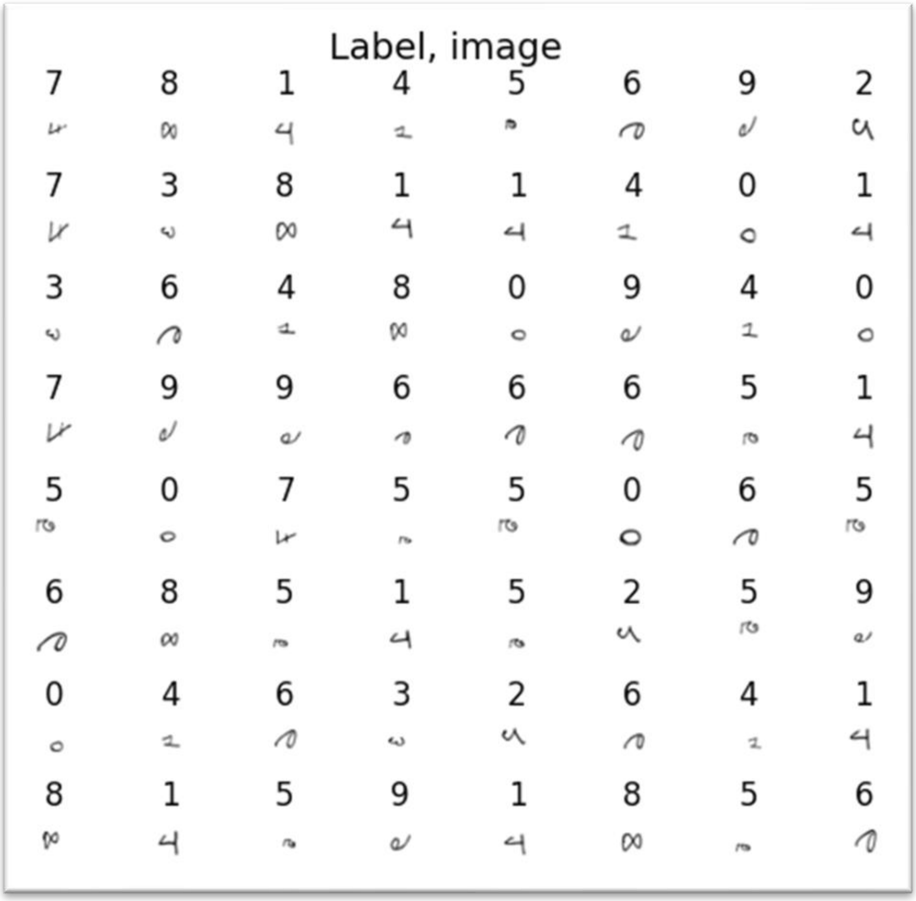
Code:

```
x_test_flattened = x_test_flattened[:200]
test_loss, test_accuracy = model.evaluate(x_test_flattened, y_test)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
```

Prediction Image:

Code:

```
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)


m, _, n = x_test.shape

fig, axes = plt.subplots(8,8, figsize=(5,5))
fig.tight_layout(pad=0.13,rect=[0, 0.03, 1, 0.91]) #[left, bottom, right, top]

#fig.tight_layout(pad=0.5)

for i,ax in enumerate(axes.flat):
   # Select random indices
   random_index = np.random.randint(m)

   # Select rows corresponding to the random indices and
   # reshape the image
   x_test_random_reshaped = x_test[random_index].reshape((28,28)).T

   # Display the image
   ax.imshow(x_test_random_reshaped, cmap='gray')

   # Display the label above the image
   ax.set_title(y_test[random_index])
   ax.set_axis_off()
   fig.suptitle("Label, image", fontsize=14)
```



Conclusion:

In conclusion, the project successfully implemented a neural network model capable of recognizing handwritten digits. Although the initial testing accuracy was low, refining the model with labeled test data resulted in much better performance. This highlights the critical role of preprocessing, model architecture, and techniques in machine learning tasks. Future work may include using convolutional neural networks (CNNs) for improved image-based classification accuracy and experimenting with more diverse datasets.