Assignment-3: Handwritten Digit Recognition

Rezwan-Ul-Alam (ID: 2011659042)
Md. Nur Alam Jowel (ID: 2012355042)
Raian Ruku (ID: 2013409642)

Email addresses:

rezwan.alam1@northsouth.edu
alam.jowel@northsouth.edu
raian.ruku@northsouth.edu

I. Introduction

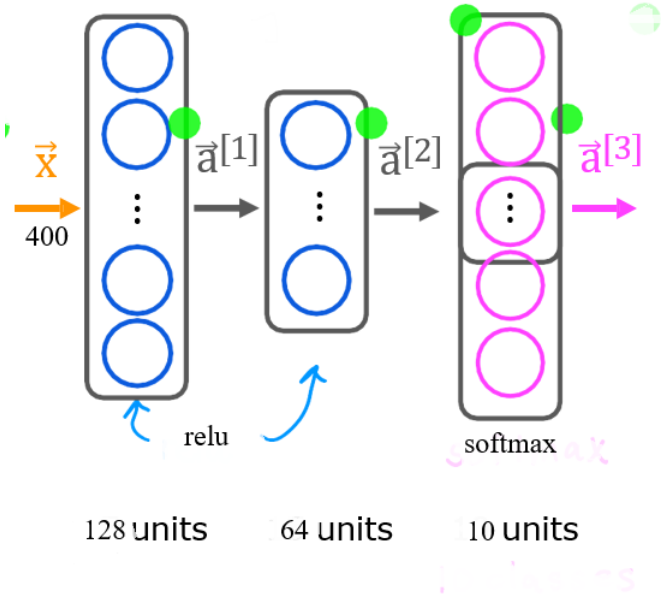
This assignment aims to develop a neural network model for handwritten digit recognition. Handwritten digit recognition is a fundamental problem in the field of machine learning. It involves the classification of handwritten digits into their respective numerical expressions. This report presents an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

II. Method

The neural network model used for handwritten digit recognition consists of an input layer, one or more hidden layers with ReLU activations, and one output layer with softmax activation. Each layer contains neurons interconnected through weighted connections. For this task, a fully connected feedforward neural network architecture is employed.

A. Model Representation

The neural network we use in this assignment 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 W_1 is $(400, 128)$ and the shape of b_1 is (128).
2) Layer 2: The shape of W_2 is $(128, 64)$ and the shape of b_2 is (64).
3) Layer 3: The shape of W_3 is $(64, 10)$ and the shape of b_3 is (10).

Now, calculate the parameters of each layer:

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

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

Output:

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_9 (Dense)             (None, 128)               51328

 dense_10 (Dense)            (None, 64)                8256

 dense_11 (Dense)            (None, 10)                650

=================================================================
Total params: 60234 (235.29 KB)
Trainable params: 60234 (235.29 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Code:

```
#define the model
model = Sequential([
    Dense(128, activation='relu', input_shape=(400,)),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

Note: In this model definition, we have changed the activation functions and the values for the number of units in each layer. We will discuss the implications and results of these changes in the **experimental results section.**

B. Model Compilation

This set up the neural network model for training:

```
# 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 our 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.

C. Model Training

Code:

```
t_model = model.fit(X, 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 our 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.

Output:

```
Epoch 1/10
125/125 [==============================] - 1s 6ms/step - loss: 0.7003 - accuracy: 0.8155 - val_loss: 10.7626 - val_accuracy: 0.0000e+00
Epoch 2/10
125/125 [==============================] - 1s 5ms/step - loss: 0.2126 - accuracy: 0.9388 - val_loss: 12.3459 - val_accuracy: 0.0000e+00
Epoch 3/10
125/125 [==============================] - 0s 4ms/step - loss: 0.1521 - accuracy: 0.9588 - val_loss: 12.6101 - val_accuracy: 0.0000e+00
Epoch 4/10
125/125 [==============================] - 1s 4ms/step - loss: 0.1199 - accuracy: 0.9670 - val_loss: 13.1917 - val_accuracy: 0.0000e+00
Epoch 5/10
125/125 [==============================] - 0s 4ms/step - loss: 0.0900 - accuracy: 0.9765 - val_loss: 13.9930 - val_accuracy: 0.0000e+00
Epoch 6/10
125/125 [==============================] - 0s 4ms/step - loss: 0.0656 - accuracy: 0.9825 - val_loss: 15.3149 - val_accuracy: 0.0000e+00
Epoch 7/10
125/125 [==============================] - 1s 5ms/step - loss: 0.0550 - accuracy: 0.9852 - val_loss: 15.3647 - val_accuracy: 0.0000e+00
Epoch 8/10
125/125 [==============================] - 1s 4ms/step - loss: 0.0421 - accuracy: 0.9898 - val_loss: 15.6594 - val_accuracy: 0.0000e+00
Epoch 9/10
125/125 [==============================] - 0s 4ms/step - loss: 0.0292 - accuracy: 0.9940 - val_loss: 16.4884 - val_accuracy: 0.0000e+00
Epoch 10/10
125/125 [==============================] - 0s 4ms/step - loss: 0.0232 - accuracy: 0.9962 - val_loss: 16.3578 - val_accuracy: 0.0000e+00
```

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, *we have increased model dense layers (50, 25, and 15 neurons, respectively) to increase the model's capacity to learn complex patterns and performed L2 Regularization and dropout layers with a dropout rate of 0.2*. And check the performance for the testing dataset, which will be discussed in the **Experiment and result section**

code:

```
model = Sequential(
[
    InputLayer((400,)),
    Dense(50, activation="relu", kernel_regularizer=regularizers.l2(0.01)),
    Dropout(0.2),
    Dense(25, activation="relu", kernel_regularizer=regularizers.l2(0.01)),
    Dropout(0.2),
    Dense(15, activation="relu", kernel_regularizer=regularizers.l2(0.01)),
    Dropout(0.2),
    Dense(10, activation="linear")
])
```

Note: Additionally, we have changed the number of epochs and skipped specifying the `validation_split` parameter. The rationale behind this decision and its impact on the training process will be discussed further in the **experiment and results section.**

D. Model Testing

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

```
json_path = '/content/drive/MyDrive/Test/Test/dd.json'
from PIL import Image
with open(json_path) as f:
    data = json.load(f)
test_images = []
test_labels = []
for filename, image_data in data.items():
    img = Image.open("/content/drive/MyDrive/Test/Test/" + image_data["filename"])
    img_array = np.array(img.convert('L').resize((20, 20))).flatten()
    test_images.append(img_array)
    test_labels.append(int(image_data["regions"][0]["region_attributes"]["shape"]))
X_test = np.array(test_images)
y_test = np.array(test_labels).reshape(-1, 1)
```

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

III. Experiment Results

A. Trains:

In this section, we 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, including L2 regularization, adding more layers, and setting a dropout rate of 0.2. Subsequently, we observed a reduction in the validation loss.

Technique	Train Accuracy (10 epochs)	Val_Loss (10 epochs)	Train Accuracy (40 epochs)	Val_Loss (40 epochs)
Without regularization and more layers	99%	16.35	97%	22.5 approx
With regularization	99%	9.6	77%	6 approx

TABLE I
Comparison of Train Accuracy and Validation Loss

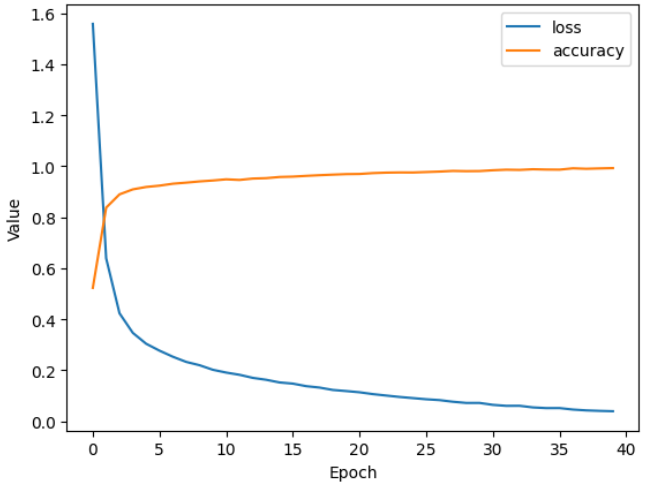


Fig. 1. Loss and Accuracy graph for 40 epochs

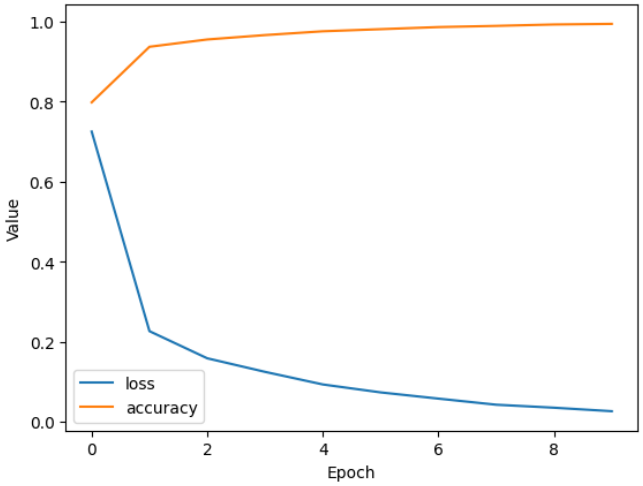


Fig. 2. Loss and accuracy graph for 10 epochs

B. Tests:

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

- **Without Regularization, Normalization, and Fewer Dense Layers:**
  Test Accuracy: 0.023 (2%)

```
7/7 [==============================]
Test accuracy: 0.0233256892
```

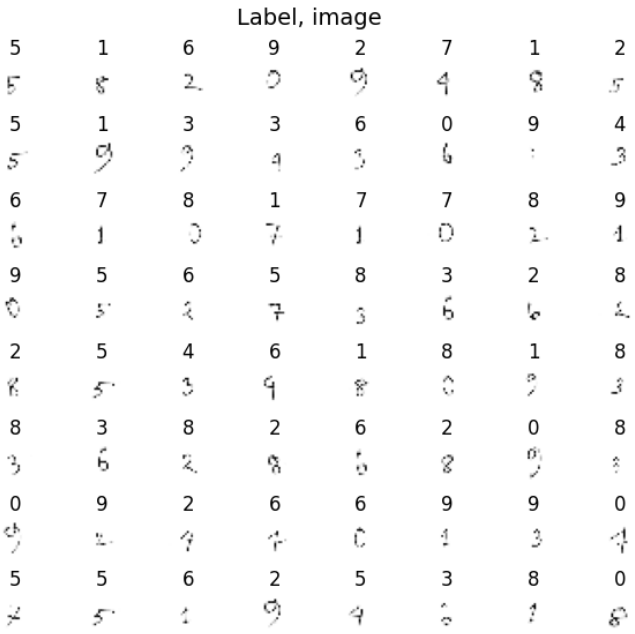
- **With Regularization and More Layers:**
  Test Accuracy: 0.10002 (10%)

```
7/7 [==============================]
Test accuracy: 0.10000000149011612
```

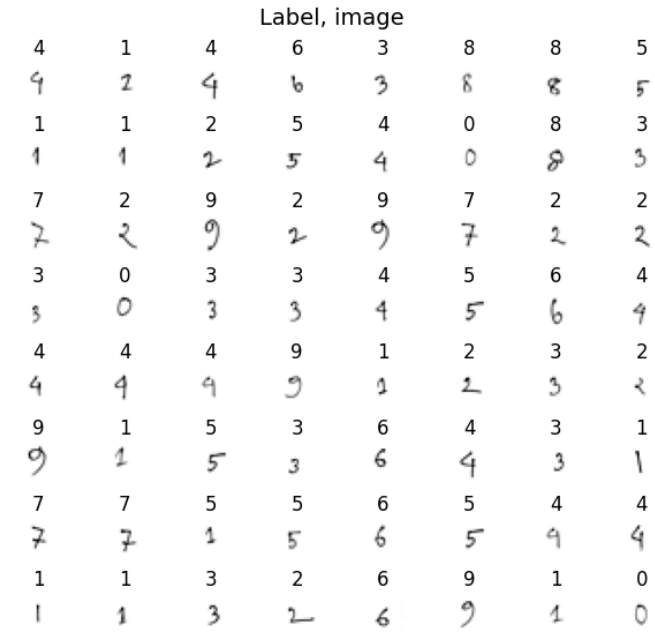
- **With Normalization:**
  Test Accuracy: 0.13000 (13%)

```
7/7 [==============================]
Test accuracy: 0.130000002356877
```

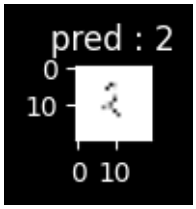
Note: We have loaded the data without labels and with labels and checked our prediction. When we just loaded our data without the labels**(no y_test)** of each image, it predicted **11** images correct out of 64, which is similar to our test accuracy of 13%. However, the exciting part is using the labels for the pictures **(y_test)** improves the prediction significantly. Almost all the images are predicted correctly. Below is the graph:

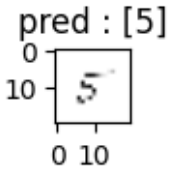


9 correct prediction



Almost all correct prediction





IV. Discussion

1) **Effect of Regularization and Layer Complexity**:
   – Introducing L2 regularization and increasing the complexity of the model with additional layers led to a noticeable improvement in the validation loss. This indicates that regularization techniques effectively countered overfitting, allowing the model to generalize better to unseen data.
2) **Impact of Dropout Layers**:
   – The inclusion of dropout layers with a dropout rate of 0.2 contributed to mitigating overfitting by randomly dropping a proportion of neurons during training. This prevented the network from relying too heavily on any particular set of features, thus enhancing its ability to generalize.
3) **Training Duration and Epochs**:
   – Initially training the model for 10 epochs showed high accuracy on the training set but increasing validation loss, suggesting overfitting. Extending training to 40 epochs with regularization led to a decrease in training accuracy but improved validation loss, indicating better generalization.
4) **Testing Accuracy Variability**:
   – The testing accuracy varied significantly based on the techniques applied. Without regularization and normalization, the model exhibited poor performance, whereas regularization and normalization techniques improved accuracy substantially. This highlights the importance of preprocessing and regularization in enhancing model performance.
5) **Impact of Data Labeling**:
   – Notably, using labels for test data significantly improved prediction accuracy. This suggests that having labeled data aids the model in making more accurate predictions, emphasizing the importance of sufficient and accurate labeling in training and testing datasets.
6) **Model Generalization vs. Complexity**:
   – Balancing model complexity with generalization is crucial. While increasing model complexity can potentially improve performance on training data, it may lead to overfitting. Regularization techniques and dropout layers help strike a balance by reducing over-reliance on specific features and preventing overfitting.
7) **Normalization and Preprocessing**:
   – Normalization of input data, such as scaling pixel values, can improve model convergence and generalization. Preprocessing steps, like resizing images and converting them to grayscale, ensure uniformity in input data, facilitating effective learning by the neural network.
8) **Trade-off between Training Time and Performance**:
   – The trade-off between training time and model performance should be considered. While increasing the number of epochs may improve performance, it also increases training time. Experimentation with different hyperparameters is essential to find the optimal balance between training duration and model accuracy.

References