In the name of GOD

Neural Network Project Report

By Seyed Mohammad Eshagh Maibodi (40313414)

Teacher: Dr. Sadatee

Winter 1403

Description: In this project, I develop a model that recognizes numbers from 0 to 9 using the MNIST dataset. To achieve better results, I use different configurations to find the best model with the highest accuracy.

Step One: Import Libraries and Load Dataset

```
# Import Library
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.layers import Input, Dense, Flatten
from tensorflow.keras import Model
from tensorflow.keras.optimizers import SGD
import matplotlib.pyplot as plt
```

```
# Load MNIST Dataset
(x_train, y_train), (x_test, y_test) =
keras.datasets.mnist.load_data(path='mnist.npz')
```

* This step will remain the same in all cases.

As shown in the code, we need to import the TensorFlow library to use Keras. In “keras.utils”, we need the “to_categorical” function to apply OneHot encoding to the outputs. From “keras.layers”, we need “Input”, “Dense”, and “Flatten”. I will explain their usage further in the code. Additionally, we need “Model” from Keras to create the neural network model.

The optimizer is essentially the learning algorithm. In this case, we use SGD (Stochastic Gradient Descent), which is based on the derivative of the weights to determine the next step for minimizing or maximizing. We can adjust the magnitude of this step by multiplying it.

Finally, we need the Matplotlib library to plot accuracy and loss data.

Step Two: Normalizing Inputs

For better results, we need to normalize the inputs. This reduces the model's sensitivity to larger input values compared to outputs. We achieve normalization using the following method.

Note that for a grayscale image, each pixel value ranges from 0 to 255. By dividing these values by 255, we scale them to a range between 0 and 1.

```
# Normalize Input Data (Min:0, Max:1)
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Step Three: OneHot Outputs

As I explained, we use the "to_categorical" method to apply OneHot encoding to the outputs. There are 10 classes corresponding to the 10 digits (0 to 9), and an image cannot represent two numbers at the same time.

```
# Convert Labels to oneHot (10 Classes for 0 to 9 Numbers)
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

Step Four: Define Neural Network Layers and Model

The MNIST dataset consists of images with a 28x28 resolution. Therefore, the model's input must have a 28x28 shape. However, we need to flatten them into one-dimensional values. The "Flatten()" function takes care of this for us.

For the layers, we use the "Dense" function to create a fully connected network. In this function, we must define four properties for each layer:

1. The number of neurons.

2. The activation function (e.g., Sigmoid, ReLU, Softmax, etc.).

3. Whether to use a bias term.

4. The previous layer for the current one.

For example, in the first hidden layer, the previous layer is the input layer.

* Later, we will adjust this property to improve performance.

```
# Define Neural Network Layers
input_layer = Input(shape=(28, 28))
flatten = Flatten()(input_layer)
h1 = Dense(64, activation='sigmoid', use_bias=True)(flatten)
h2 = Dense(64, activation='sigmoid', use_bias=True)(h1)
output_layer = Dense(10, activation='sigmoid', use_bias=True)(h2)
```

After defining the layers, we need to specify the input and output layers in the "Model" function..

```
# Define Neural Network Model
model = Model(input_layer, output_layer)
```

Now, we can view the model summary using the "model.summary()" command. Figure 1 shows an example of a model summary.

```
Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 28, 28)	0
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 64)	50,240
dense_1 (Dense)	(None, 64)	4,160
dense_2 (Dense)	(None, 10)	650

```
Total params: 55,050 (215.04 KB)
Trainable params: 55,050 (215.04 KB)
Non-trainable params: 0 (0.00 B)
```

Figure 1 example of a model summary

Finally, we need to compile the model using the "model.compile()" function. In this method, we must set three properties:

1. Optimizer – The learning algorithm. Here, we use SGD with a learning rate of 0.01.

2. Loss – The loss function. We use mean squared error(mse) for this.

3. Metrics – The evaluation metrics that need to be set..

```
# Compile Model
model.compile(optimizer=SGD(learning_rate=0.1), loss='mse',
metrics=['accuracy'])
```

Step Five: Train and Plot

The training process begins with the "model.fit()" method. In this method, we need to specify the input data, output labels, and the number of epochs for iterations. For validation, we can use the test split of our dataset.

```
# Train Model
result = model.fit(x_train, y_train, epochs=100, validation_data=(x_test,
y_test))
```

For better visualization, we plot the "Accuracy", "Validation Accuracy", and "Loss" values over epochs.

```
# Plot Training Progress
plt.plot(result.history['accuracy'], label='Accuracy')
plt.plot(result.history['val_accuracy'], label='Validation Accuracy')
plt.plot(result.history['loss'], label='Loss')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend()
plt.show()
```

Step Five: Changing Hyper-Parameters For Better Result

ID	layers No.	Neuron No.	Activation Function	Optimizer	Loss Function	Epochs	Result (Acc, Loss)
1	3	64,64,10	Sigmoid(3)	SGD(0.1)	mse	100	0.9360, 0.0109
2	3	64,64,10	Sigmoid(3)	SGD(0.01)	mse	100	0.6326, 0.0616
3	3	64,64,10	Sigmoid(3)	SGD(0.01)	mse	300	0.8858, 0.0219
4	3	64,64,10	Sigmoid(2), Softmax	SGD(0.05)	mse	150	0.9381, 0.0098
5	2	128,10	ReLu, Softmax	SGD(0.01)	mse	200	0.9565, 0.0072
6	3	64,64,10	ReLu(2), Softmax	SGD(0.01)	mse	150	0.9560, 0.0072
7	3	64,128,10	ReLu(2), Softmax	SGD(0.05)	mse	100	0.9847, 0.0029
8-Best	3	128,128,10	ReLu(2), Softmax	SGD(0.05)	mse	150	0.9934, 0.0014