EXERCISE 2.07: DEEP NEURAL NETWORK ON MNIST USING KERAS

In this exercise, we will perform a multiclass classification by implementing a deep neural network (multi-layer) for the MNIST dataset where our input layer comprises 28 × 28 pixel images flattened to 784 input nodes followed by 2 hidden layers, the first with 50 neurons and the second with 20 neurons. Lastly, there will be a Softmax layer consisting of 10 neurons since we are classifying the handwritten digits into 10 classes:

1. Import the required libraries and packages:

```
import tensorflow as tf
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

# Import Keras libraries
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
```

2. Load the MNIST data:

```
mnist = tf.keras.datasets.mnist
(train_features,train_labels), (test_features,test_labels) = \
mnist.load_data()
```

   `train_features` has the training images in the form of 28 x 28 pixel values.

   `train_labels` has the training labels. Similarly, **`test_features`** has the test images in the form of 28 x 28 pixel values. **`test_labels`** has the test labels.

3. Normalize the data:

```
train_features, test_features = train_features / 255.0, \
                                test_features / 255.0
```

   The pixel values of the images range from 0-255. We need to normalize the values by dividing them by 255 so that the range goes from 0 to 1.

4. Build the **sequential** model:

```
model = Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(units = 50, activation = 'relu'))
model.add(Dense(units = 20 , activation = 'relu'))
model.add(Dense(units = 10, activation = 'softmax'))
```

   There are couple of points to note. The first layer in this case is not actually a layer of neurons but a **Flatten** function. This flattens the 28 x 28 image into a single array of **784**, which is fed to the first hidden layer of **50** neurons. The last layer has **10** neurons corresponding to the 10 classes with a **softmax** activation function.

5. Provide training parameters using the **compile** method:

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

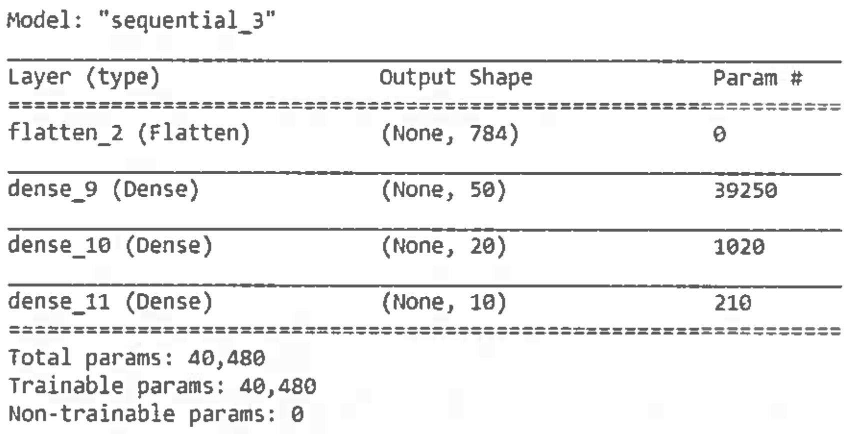
NOTE

The loss function used here is different from the binary classifier. For a multiclass classifier, the following loss functions are used: **sparse_categorical_crossentropy**, which is used when the labels are not one-hot encoded, as in this case; and, **categorical_crossentropy**, which is used when the labels are one-hot encoded.

6. Inspect the model configuration using the **summary** function:

```
model.summary()
```

The output is as follows:



```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_9 (Dense)              (None, 50)                39250
_________________________________________________________________
dense_10 (Dense)             (None, 20)                1020
_________________________________________________________________
dense_11 (Dense)             (None, 10)                210
=================================================================
Total params: 40,480
Trainable params: 40,480
Non-trainable params: 0
```

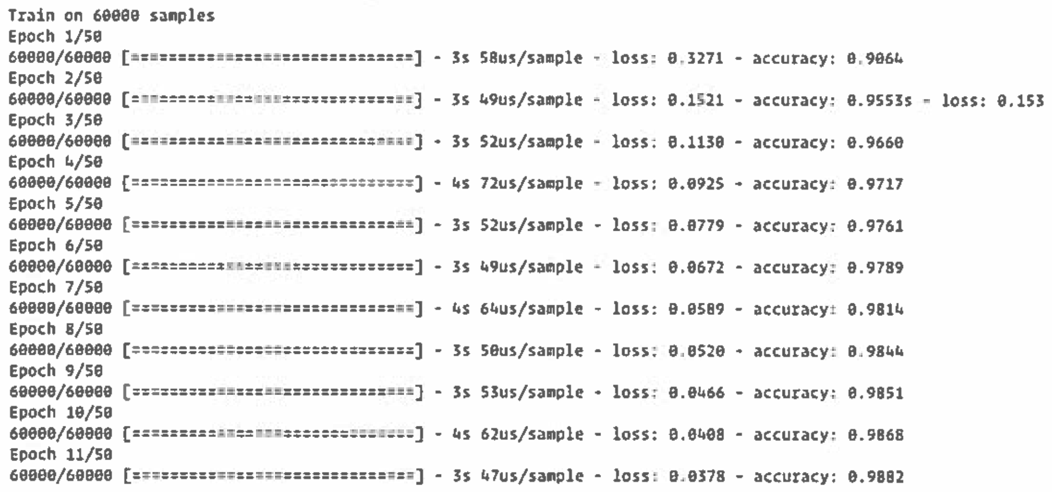
Figure 2.33: Deep neural network summary

In the model summary, we can see that there are a total of 40,480 parameters—weights and biases—to learn across the hidden layers to the output layer.

7. Train the model by calling the **`fit`** method:

```
model.fit(train_features, train_labels, epochs=50)
```

The output will be as follows:



```
Train on 60000 samples
Epoch 1/50
60000/60000 [==============================] - 3s 58us/sample - loss: 0.3271 - accuracy: 0.9064
Epoch 2/50
60000/60000 [==============================] - 3s 49us/sample - loss: 0.1521 - accuracy: 0.9553s - loss: 0.153
Epoch 3/50
60000/60000 [==============================] - 3s 52us/sample - loss: 0.1130 - accuracy: 0.9660
Epoch 4/50
60000/60000 [==============================] - 4s 72us/sample - loss: 0.0925 - accuracy: 0.9717
Epoch 5/50
60000/60000 [==============================] - 3s 52us/sample - loss: 0.0779 - accuracy: 0.9761
Epoch 6/50
60000/60000 [==============================] - 3s 49us/sample - loss: 0.0672 - accuracy: 0.9789
Epoch 7/50
60000/60000 [==============================] - 4s 64us/sample - loss: 0.0589 - accuracy: 0.9814
Epoch 8/50
60000/60000 [==============================] - 3s 50us/sample - loss: 0.0520 - accuracy: 0.9844
Epoch 9/50
60000/60000 [==============================] - 3s 53us/sample - loss: 0.0466 - accuracy: 0.9851
Epoch 10/50
60000/60000 [==============================] - 4s 62us/sample - loss: 0.0408 - accuracy: 0.9868
Epoch 11/50
60000/60000 [==============================] - 3s 47us/sample - loss: 0.0378 - accuracy: 0.9882
```

Figure 2.34: Deep neural network training logs

8. Test the model by calling the **`evaluate()`** function:

```
model.evaluate(test_features, test_labels)
```

The output will be:

```
10000/10000 [==============================] - 1s 76us/sample - loss:
  0.2072 - accuracy: 0.9718
[0.20719025060918111, 0.9718]
```

Now that the model is trained and tested, in the next few steps, we will run the prediction with some images selected randomly.

9. Load a random image from a test dataset. Let's locate the 200*th* image:

```
loc = 200
test_image = test_features[loc]
```

10. Let's see the shape of the image using the following command:

```
test_image.shape
```

The output is:

```
(28,28)
```

We can see that the shape of the image is 28 x 28. However, the model expects 3-dimensional input. We need to reshape the image accordingly.

11. Use the following code to reshape the image:

```
test_image = test_image.reshape(1,28,28)
```

12. Let's call the **`predict()`** method of the model and store the output in a variable called **`result`**:

```
result = model.predict(test_image)
print(result)
```

`result` has the output in the form of 10 probability values, as shown here:

```
[[2.9072076e-28 2.1215850e-29 1.7854708e-21
  1.0000000e+00 0.0000000e+00 1.2384960e-15
  1.2660366e-34 1.7712217e-32 1.7461657e-08
  9.6417470e-29]]
```

Baig, Mirza Rahim, et al. The Deep Learning Workshop : Learn the Skills You Need to Develop Your Own Next-Generation Deep Learning Models with TensorFlow and Keras, Packt Publishing, Limited, 2020. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/ull/detail.action?docID=6276012.

13. The position of the highest value will be the prediction. Let's use the **`argmax`** function we learned about in the previous chapter to find out the prediction:

```
result.argmax()
```

In this case, it is **3**:

```
3
```

14. In order to check whether the prediction is correct, we check the label of the corresponding image:

```
test_labels[loc]
```

Again, the value is **3**:

```
3
```

15. We can also visualize the image using **`pyplot`**:

```
plt.imshow(test_features[loc])
```

The output will be as follows:

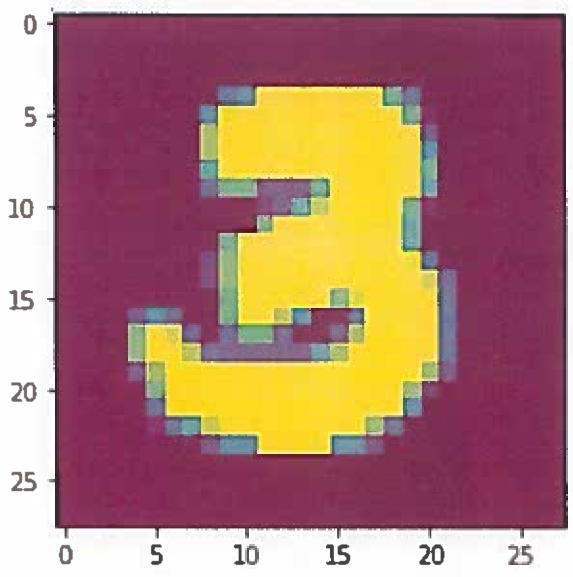


Figure 2.35: Test image visualized

And this shows that the prediction is correct.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/2O5KRgd.
>
> You can also run this example online at https://packt.live/2O8JHR0. You must execute the entire Notebook in order to get the desired result.

In this exercise, we created a multilayer multiclass neural network model using Keras to classify the MNIST data. With the model we built, we were able to correctly predict a random handwritten digit.

EXPLORING THE OPTIMIZERS AND HYPERPARAMETERS OF NEURAL NETWORKS

Training a neural network to get good predictions requires tweaking a lot of hyperparameters such as optimizers, activation functions, the number of hidden layers, the number of neurons in each layer, the number of epochs, and the learning rate. Let's go through each of them one by one and discuss them in detail.

GRADIENT DESCENT OPTIMIZERS

In an earlier section titled *Perceptron Training Process in TensorFlow*, we briefly touched upon the gradient descent optimizer without going into the details of how it works. This is a good time to explore the gradient descent optimizer in a little more detail. We will provide an intuitive explanation without going into the mathematical details.

The gradient descent optimizer's function is to minimize the loss or error. To understand how gradient descent works, you can think of this analogy: imagine a person at the top of a hill who wants to reach the bottom. At the beginning of the training, the loss is large, like the height of the hill's peak. The functioning of the optimizer is akin to the person descending the hill to the valley at the bottom, or rather, the lowest point of the hill, and not climbing up the hill that is on the other side of the valley.