July 23, 2023

[1]:
```
# Import statements

import tensorflow as tf
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from IPython.display import display
```

1 Mnist

Let's understand our mnist data. It is a collection of 70,000 handwritten images of digits ranging from 0 to 9.

Tensorflow has this dataset built into it and we could call the tensorflow api to use it into our python scripts.

[2]:
```
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
```

[3]:
```
for i in X_train, X_test, y_train, y_test:
    print(i.shape)
```

```
(60000, 28, 28)
(10000, 28, 28)
(60000,)
(10000,)
```

As we could notice, in the training dataset, we have 60,000 records and in the test set we have 10,000 records of these handwritten images.

X_train shape: 60,000 images of shape 28 by 28 pixels

X_test shape: 10,000 images of shape 28 by 28 pixels

y_train shape: 60,000 labels (as we will see soon)

y_test shape: 10,000 labels

Lets see how the first image of this dataset looks like

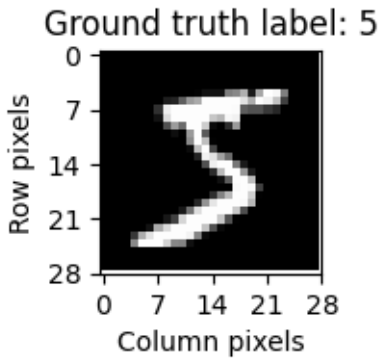
[4]:
```
# First label of training set
y_train[0]
```

[4]:
```
5
```

[5]:
```
print(type(X_train))
```

```
<class 'numpy.ndarray'>
```

[6]:
```
plt.figure(figsize=(1.5,1.5))
plt.imshow(X_train[0], cmap=plt.cm.gray)
plt.title("Ground truth label: {}".format(y_train[0]))
plt.xlabel("Column pixels")
plt.ylabel("Row pixels")
plt.xticks(range(0, 30, 7))
plt.yticks(range(0, 30, 7))
plt.show()
```



Same image of training set as a numpy array. The integers in this numpy array range from 0-255. These integers represents the brightness values of a particular pixel

[7]:
```
X_train[0]
```

[7]:
```
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
```

```
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,
  18,  18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170,
 253, 253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253,
 253, 253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253,
 253, 198, 182, 247, 241,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,  80, 156, 107, 253, 253,
 205,  11,   0,  43, 154,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,  14,   1, 154, 253,
  90,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 139, 253,
 190,   2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 190,
 253,  70,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  35,
 241, 225, 160, 108,   1,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  81, 240, 253, 253, 119,  25,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,  45, 186, 253, 253, 150,  27,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,  16,  93, 252, 253, 187,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0, 249, 253, 249,  64,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,  46, 130, 183, 253, 253, 207,   2,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  39,
```

```
 148, 229, 253, 253, 253, 250, 182,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 114, 221,
 253, 253, 253, 253, 201,  78,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,  23,  66, 213, 253, 253,
 253, 253, 198,  81,   2,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,  18, 171, 219, 253, 253, 253, 253,
 195,  80,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,  55, 172, 226, 253, 253, 253, 253, 244, 133,
  11,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0, 136, 253, 253, 253, 212, 135, 132,  16,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0],
[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0]], dtype=uint8)
```

1.1 Optional one hot encoding step for our labels

[8]:
```
# First label before one hot encoding
y_train[0]
```

[8]:
```
5
```

[9]:
```
# We could do this encoding step in multiple ways
# For demonstration we will use keras api for encodinga

y_train = tf.keras.utils.to_categorical(y_train)
y_test = tf.keras.utils.to_categorical(y_test)
```

[10]:
```
# The label 5 encoded as one hot vector
y_train[0]
```

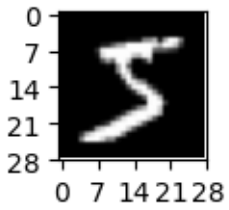
[10]:
```
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)
```

1.2 First few images in our training set

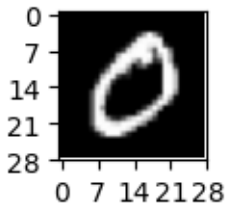
[11]:
```
for i in range(4):
    print("Ground truth label:", np.argmax(y_train[i]))

    plt.figure(figsize=(1,1))
    plt.imshow(X_train[i], cmap=plt.cm.gray)
    plt.xticks(range(0, 30, 7))
    plt.yticks(range(0, 30, 7))
    plt.show()
```

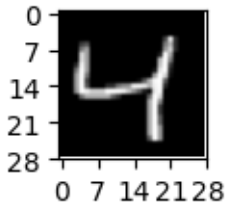
```
Ground truth label: 5
```



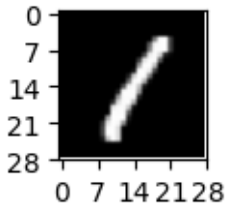

```
Ground truth label: 0
```




```
Ground truth label: 4
```




```
Ground truth label: 1
```



[12]:
```
X_train[0].shape
```

[12]:
```
(28, 28)
```

2 Model training

[13]:
```
# Loading fresh data again

(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()

y_train = tf.keras.utils.to_categorical(y_train)
y_test = tf.keras.utils.to_categorical(y_test)
```

2.1 Model architecture selection

- We will use the sequential api of keras here. When we want our model layers to be executed in a particular sequence we use the sequential api.
- Then we will use the flatten layer, to flatten the 28 by 28 pixel numpy array from 2 dimension to 1 dimension
- When dealing with image data, it is generally recommended to normalize the range of integers of brightness value. After the normalization, integers 0-255 gets compressed to the range between 0-1.

[14]:
```
# In the name argument, we could optionally specify a name for our model

model = tf.keras.models.Sequential(name="Sequential_one_hot_encoded_model")


# input_shape=(28,28) flattening the values to 28*28 = 784
model.add(tf.keras.layers.Flatten(
    input_shape=X_train[0].shape, name="input_layer"))

model.add(tf.keras.layers.BatchNormalization(
    name="minmax_normalization_1"))
```

A quick summary of our model built so far.

- We have input layer of our pixels flattened to 1 dimension numpy array
- Then we introduce batch normalization

[15]:
```
model.summary()
```

```
Model: "Sequential_one_hot_encoded_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (Flatten)       (None, 784)               0

 minmax_normalization_1 (Bat  (None, 784)              3136
 chNormalization)

=================================================================
Total params: 3,136
Trainable params: 1,568
Non-trainable params: 1,568
_________________________________________________________________
```

2.1.1 Lets demonstrate normalization

255 is the highest value for pixel brightness value. For manual normalization, we will divide each pixel by 255.

[16]:
```
(X_train_sample, y_train_sample), (X_test_sample, y_test_sample) = tf.keras.
 ↪datasets.mnist.load_data()

X_train_sample, X_test_sample = X_train_sample/255.0, X_test_sample/255.0
```

[17]:
```
np.max(X_train_sample[0])
```

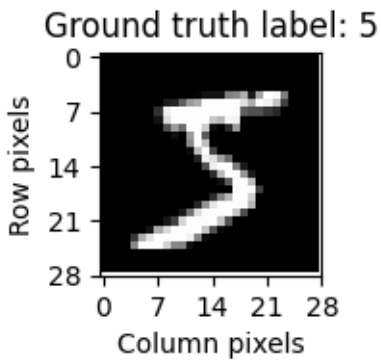
[17]: 1.0

As we can notice, the maximum value of first image in our X_train_sample is 1 after manual normalization. And it will not impact its visualization as well.

[18]:
```
plt.figure(figsize=(1.5,1.5))

plt.imshow(X_train_sample[0], cmap=plt.cm.gray)

plt.title("Ground truth label: {}".format(y_train_sample[0]))
plt.xlabel("Column pixels")
plt.ylabel("Row pixels")
plt.xticks(range(0, 30, 7))
plt.yticks(range(0, 30, 7))
plt.show()
```



2.1.2 Continue model building

- Now we will add a fully connected (Dense) layer of 256 neurons with **relu** activation and name it **hidden_layer_1**
- Then we repeat the steps to add batch normalization and add another Dense layer
- As our final layer, we will use a Dense layer with 10 neurons and name it **output_layer**

[19]:
```
model.add(tf.keras.layers.Dense(
    256, activation="relu", name="hidden_layer_1"))

model.add(tf.keras.layers.BatchNormalization(
    name="minmax_normalization_2"))

model.add(tf.keras.layers.Dense(
    256, activation="relu", name="hidden_layer_2"))

model.add(tf.keras.layers.Dense(
    10, activation="softmax", name="output_layer"))
```

This is how our model architecture looks so far.

[20]:
```
model.summary()
```

```
Model: "Sequential_one_hot_encoded_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (Flatten)       (None, 784)               0

 minmax_normalization_1 (Bat  (None, 784)              3136
 chNormalization)

 hidden_layer_1 (Dense)      (None, 256)               200960
```

```
 minmax_normalization_2 (Bat  (None, 256)              1024
 chNormalization)

 hidden_layer_2 (Dense)      (None, 256)              65792

 output_layer (Dense)        (None, 10)               2570

=================================================================
Total params: 273,482
Trainable params: 271,402
Non-trainable params: 2,080
_________________________________________________________________
```

2.1.3 Model Compile

- Loss is a measure of how good our model is doing
- To adjust weights during the back-propagation and achieve optimum results we will use adaptive moment as our optimizer

[21]:
```
model.compile(
    loss=tf.keras.losses.CategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.005),
    metrics=["accuracy"]
)

print(model.summary())
```

```
Model: "Sequential_one_hot_encoded_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (Flatten)       (None, 784)               0

 minmax_normalization_1 (Bat  (None, 784)              3136
 chNormalization)

 hidden_layer_1 (Dense)      (None, 256)               200960

 minmax_normalization_2 (Bat  (None, 256)              1024
 chNormalization)

 hidden_layer_2 (Dense)      (None, 256)               65792

 output_layer (Dense)        (None, 10)               2570

=================================================================
Total params: 273,482
```

```
Trainable params: 271,402
Non-trainable params: 2,080
_________________________________________________________________
None
```

2.2 Model training

[22]:
```
X_train.shape
```

[22]:
```
(60000, 28, 28)
```

[23]:
```
# We will use 15000 examples in our batch for model training

X_train.shape[0]//4
```

[23]:
```
15000
```

[24]:
```
# We can save our model intermidiate performance

results = model.fit(
    X_train, y_train,
    validation_split=0.2,
    batch_size=X_train.shape[0]//4,
    epochs=20
)
```

```
Epoch 1/20
4/4 [==============================] - 1s 67ms/step - loss: 1.3143 - accuracy:
0.5715 - val_loss: 3.0675 - val_accuracy: 0.7004
Epoch 2/20
4/4 [==============================] - 0s 20ms/step - loss: 0.3249 - accuracy:
0.9009 - val_loss: 1.3063 - val_accuracy: 0.8288
Epoch 3/20
4/4 [==============================] - 0s 22ms/step - loss: 0.2309 - accuracy:
0.9296 - val_loss: 0.7177 - val_accuracy: 0.8881
Epoch 4/20
4/4 [==============================] - 0s 22ms/step - loss: 0.1797 - accuracy:
0.9447 - val_loss: 0.5387 - val_accuracy: 0.9047
Epoch 5/20
4/4 [==============================] - 0s 21ms/step - loss: 0.1474 - accuracy:
0.9552 - val_loss: 0.4397 - val_accuracy: 0.9144
Epoch 6/20
4/4 [==============================] - 0s 20ms/step - loss: 0.1223 - accuracy:
0.9625 - val_loss: 0.3960 - val_accuracy: 0.9188
Epoch 7/20
4/4 [==============================] - 0s 20ms/step - loss: 0.1046 - accuracy:
0.9679 - val_loss: 0.3595 - val_accuracy: 0.9231
Epoch 8/20
```

```
4/4 [==============================] - 0s 20ms/step - loss: 0.0896 - accuracy:
0.9728 - val_loss: 0.3229 - val_accuracy: 0.9264
Epoch 9/20
4/4 [==============================] - 0s 20ms/step - loss: 0.0783 - accuracy:
0.9773 - val_loss: 0.2895 - val_accuracy: 0.9309
Epoch 10/20
4/4 [==============================] - 0s 20ms/step - loss: 0.0685 - accuracy:
0.9799 - val_loss: 0.2725 - val_accuracy: 0.9328
Epoch 11/20
4/4 [==============================] - 0s 20ms/step - loss: 0.0601 - accuracy:
0.9826 - val_loss: 0.2786 - val_accuracy: 0.9308
Epoch 12/20
4/4 [==============================] - 0s 19ms/step - loss: 0.0525 - accuracy:
0.9853 - val_loss: 0.2555 - val_accuracy: 0.9349
Epoch 13/20
4/4 [==============================] - 0s 19ms/step - loss: 0.0465 - accuracy:
0.9873 - val_loss: 0.2606 - val_accuracy: 0.9333
Epoch 14/20
4/4 [==============================] - 0s 19ms/step - loss: 0.0413 - accuracy:
0.9891 - val_loss: 0.2278 - val_accuracy: 0.9413
Epoch 15/20
4/4 [==============================] - 0s 19ms/step - loss: 0.0363 - accuracy:
0.9909 - val_loss: 0.2043 - val_accuracy: 0.9471
Epoch 16/20
4/4 [==============================] - 0s 19ms/step - loss: 0.0316 - accuracy:
0.9925 - val_loss: 0.2103 - val_accuracy: 0.9447
Epoch 17/20
4/4 [==============================] - 0s 19ms/step - loss: 0.0276 - accuracy:
0.9936 - val_loss: 0.2025 - val_accuracy: 0.9482
Epoch 18/20
4/4 [==============================] - 0s 19ms/step - loss: 0.0244 - accuracy:
0.9946 - val_loss: 0.1994 - val_accuracy: 0.9479
Epoch 19/20
4/4 [==============================] - 0s 19ms/step - loss: 0.0215 - accuracy:
0.9957 - val_loss: 0.1888 - val_accuracy: 0.9513
Epoch 20/20
4/4 [==============================] - 0s 19ms/step - loss: 0.0192 - accuracy:
0.9963 - val_loss: 0.1745 - val_accuracy: 0.9540
```

2.2.1 Model performance

Printing as dataframe the model performance at intermidiate steps.

[25]:
```
pd.DataFrame(results.history)
```

[25]:

	loss	accuracy	val_loss	val_accuracy
0	1.314288	0.571542	3.067518	0.700417
1	0.324928	0.900896	1.306302	0.828750

```
2   0.230898  0.929563  0.717749      0.888083
3   0.179746  0.944708  0.538727      0.904750
4   0.147386  0.955208  0.439678      0.914417
5   0.122348  0.962458  0.396018      0.918833
6   0.104642  0.967854  0.359503      0.923083
7   0.089602  0.972792  0.322932      0.926417
8   0.078316  0.977271  0.289508      0.930917
9   0.068542  0.979917  0.272488      0.932833
10  0.060072  0.982604  0.278643      0.930833
11  0.052521  0.985292  0.255485      0.934917
12  0.046486  0.987292  0.260558      0.933250
13  0.041332  0.989104  0.227770      0.941333
14  0.036279  0.990896  0.204261      0.947083
15  0.031562  0.992458  0.210328      0.944667
16  0.027634  0.993604  0.202501      0.948250
17  0.024414  0.994563  0.199373      0.947917
18  0.021526  0.995687  0.188845      0.951333
19  0.019219  0.996292  0.174456      0.954000
```

Our results variable has history method, and this is how things look.

results.history is a dictionary containing accuracy and loss.

[26]:
```
results
```

[26]:
```
<keras.callbacks.History at 0x212afacb790>
```

[27]:
```
results.history
```

[27]:
```
{'loss': [1.314287543296814,
  0.3249281942844391,
  0.23089845478534698,
  0.17974579334259033,
  0.14738604426383972,
  0.12234783172607422,
  0.10464172065258026,
  0.08960229158401489,
  0.07831590622663498,
  0.0685422345995903,
  0.060072094202041626,
  0.052520837634801865,
  0.04648600146174431,
  0.04133233800530434,
  0.03627898544073105,
  0.03156242519617081,
  0.02763357199728489,
  0.024413976818323135,
  0.021525533869862556,
```

```
 0.019219055771827698],
'accuracy': [0.5715416669845581,
 0.9008958339691162,
 0.929562509059906,
 0.9447083473205566,
 0.9552083611488342,
 0.9624583125114441,
 0.9678541421890259,
 0.9727916717529297,
 0.9772708415985107,
 0.9799166917800903,
 0.9826041460037231,
 0.9852916598320007,
 0.987291693687439,
 0.989104151725769,
 0.9908958077430725,
 0.9924583435058594,
 0.9936041831970215,
 0.9945625066757202,
 0.9956874847412109,
 0.9962916374206543],
'val_loss': [3.0675175189971924,
 1.3063021898269653,
 0.7177493572235107,
 0.538727343082428,
 0.43967798352241516,
 0.3960179388523102,
 0.3595034182071686,
 0.32293233275413513,
 0.28950831294059753,
 0.27248769998550415,
 0.27864331007003784,
 0.2554854154586792,
 0.26055777072906494,
 0.22776950895786285,
 0.20426109433174133,
 0.21032753586769104,
 0.20250144600868225,
 0.19937266409397125,
 0.1888447403907776,
 0.17445556819438934],
'val_accuracy': [0.7004166841506958,
 0.8287500143051147,
 0.8880833387374878,
 0.9047499895095825,
 0.9144166707992554,
 0.918833315372467,
```

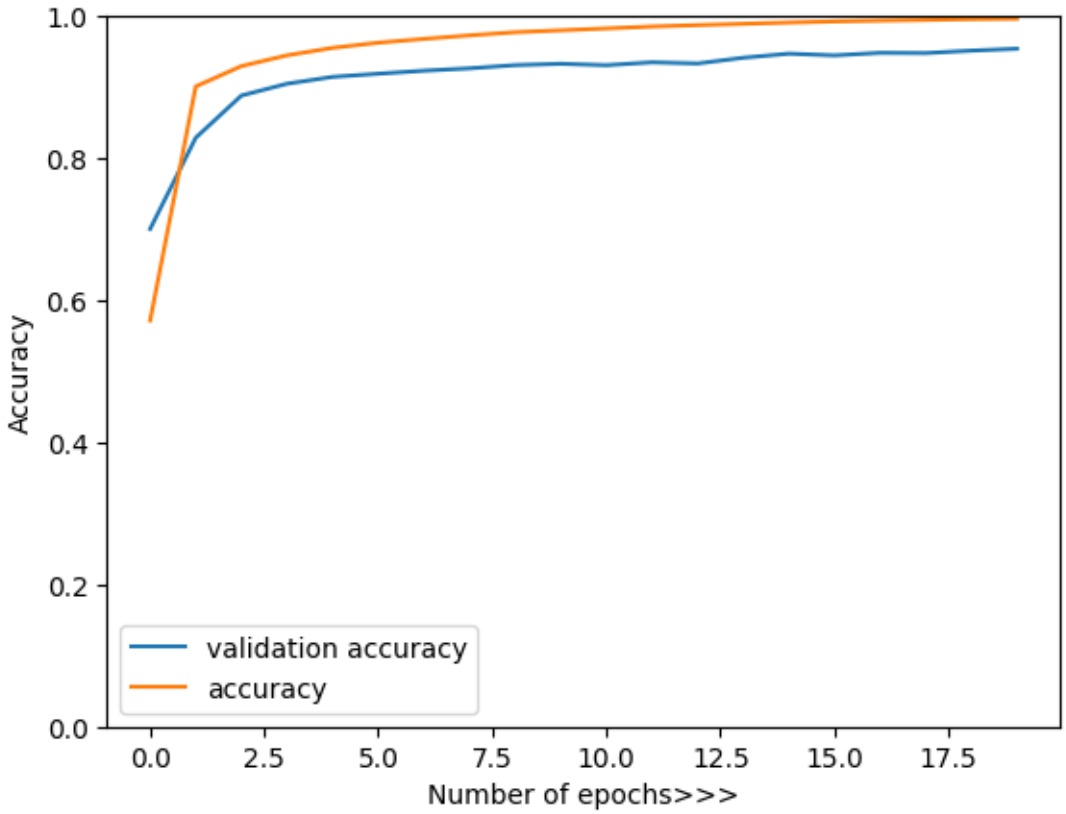
```
0.9230833053588867,
0.9264166951179504,
0.9309166669845581,
0.9328333139419556,
0.9308333396911621,
0.9349166750907898,
0.9332500100135803,
0.9413333535194397,
0.9470833539962769,
0.9446666836738586,
0.9482499957084656,
0.9479166865348816,
0.9513333439826965,
0.9539999961853027]}
```

[28]:
```
results.history.keys()
```

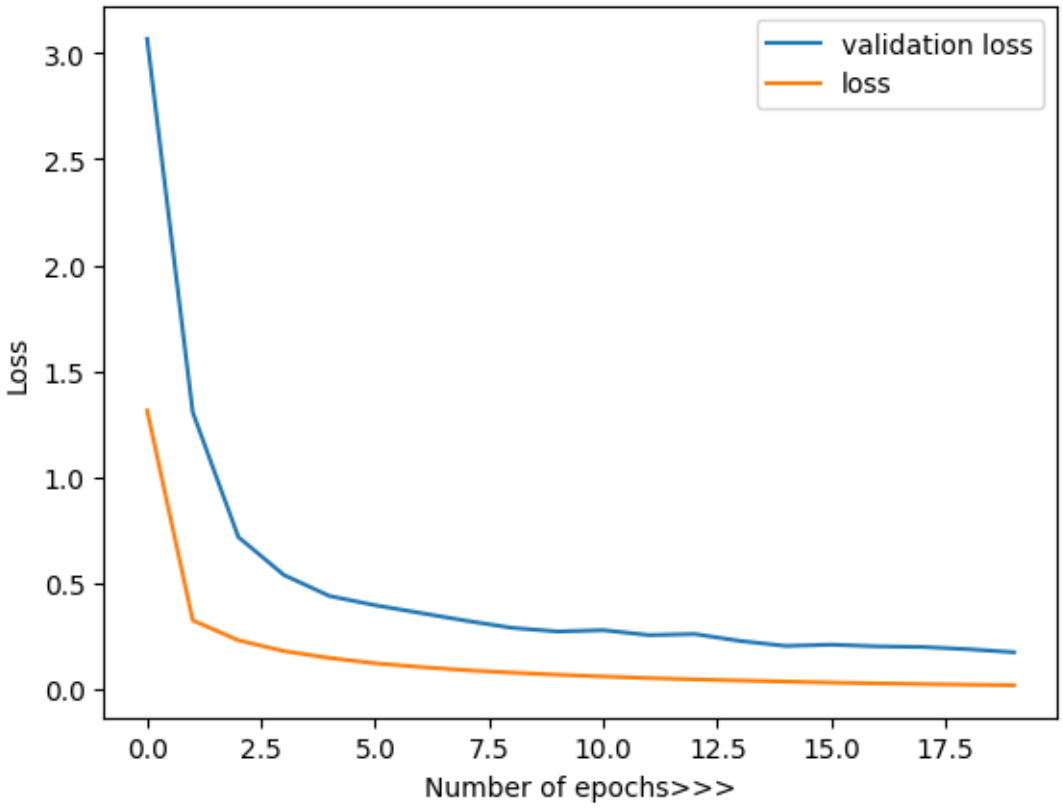
[28]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

2.2.2 Visualize accuracy and loss

[29]:
```
plt.plot(results.history["val_accuracy"], label="validation accuracy")
plt.plot(results.history["accuracy"], label="accuracy")
plt.xlabel("Number of epochs>>>")
plt.ylabel("Accuracy")
plt.ylim([0, 1])
plt.legend()
plt.show()
```



[30]:
```
plt.plot(results.history["val_loss"], label="validation loss")
plt.plot(results.history["loss"], label="loss")
plt.xlabel("Number of epochs>>>")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



2.3 Test set metrics

We have one hot encoded our labels, so we will use numpy argmax to capture the actual label.

Displaying first four labels

[31]:
```
y_test[:4]
```

[31]:
```
array([[0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

Displaying first four labels after numpy argmax transformation

[32]:
```
np.argmax(y_test, axis=1)[:4]
```

[32]:
```
array([7, 2, 1, 0], dtype=int64)
```

[33]:
```
y_test.shape
```

[33]:
```
(10000, 10)
```

[34]:
```
np.argmax(y_test, axis=1).shape
```

[34]:
```
(10000,)
```

2.3.1 Predictions

- We have a predict method in our model which will give us probabilities (which sums up to 1)

[35]:
```
first_prediction = model.predict(X_test[0:1])
first_prediction
```

```
1/1 [==============================] - 0s 76ms/step
```

[35]:
```
array([[9.9700713e-11, 4.0301193e-10, 2.6291625e-08, 5.1341972e-06,
        9.7685748e-13, 7.1036780e-13, 1.3400891e-12, 9.9999464e-01,
        9.1605938e-11, 2.7374983e-07]], dtype=float32)
```

[36]:
```
np.round(first_prediction, 4)
```

[36]:
```
array([[0., 0., 0., 0., 0., 0., 0., 1., 0., 0.]], dtype=float32)
```

Lets check the first label of our test set.

[37]:
```
y_test[0]
```

[37]:
```
array([0., 0., 0., 0., 0., 0., 0., 1., 0., 0.], dtype=float32)
```

[38]:
```
# Actual label
np.argmax(y_test, axis=1)[0]
```

[38]:
```
7
```

Lets capture rest of the predictions on test set

[39]:
```
y_pred = np.argmax(model.predict(X_test), axis=1)
y_pred[:5]
```

```
313/313 [==============================] - 0s 968us/step
```

[39]:
```
array([7, 2, 1, 0, 4], dtype=int64)
```

[40]:
```
from sklearn.metrics import accuracy_score

acc = accuracy_score(
    np.argmax(y_test, axis=1),
    y_pred
)
print("Test set accuracy:", str(acc*100) + "%")
```

```
Test set accuracy: 95.57%
```

[41]:
```
from sklearn.metrics import classification_report

print(classification_report(
    np.argmax(y_test, axis=1),
    y_pred
))
```

```
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       980
           1       0.97      0.99      0.98      1135
           2       0.94      0.97      0.95      1032
           3       0.92      0.97      0.95      1010
           4       0.99      0.93      0.96       982
           5       0.96      0.95      0.96       892
           6       0.95      0.98      0.96       958
           7       0.95      0.98      0.96      1028
           8       0.99      0.83      0.90       974
           9       0.95      0.96      0.95      1009

    accuracy                           0.96     10000
   macro avg       0.96      0.95      0.95     10000
weighted avg       0.96      0.96      0.96     10000
```

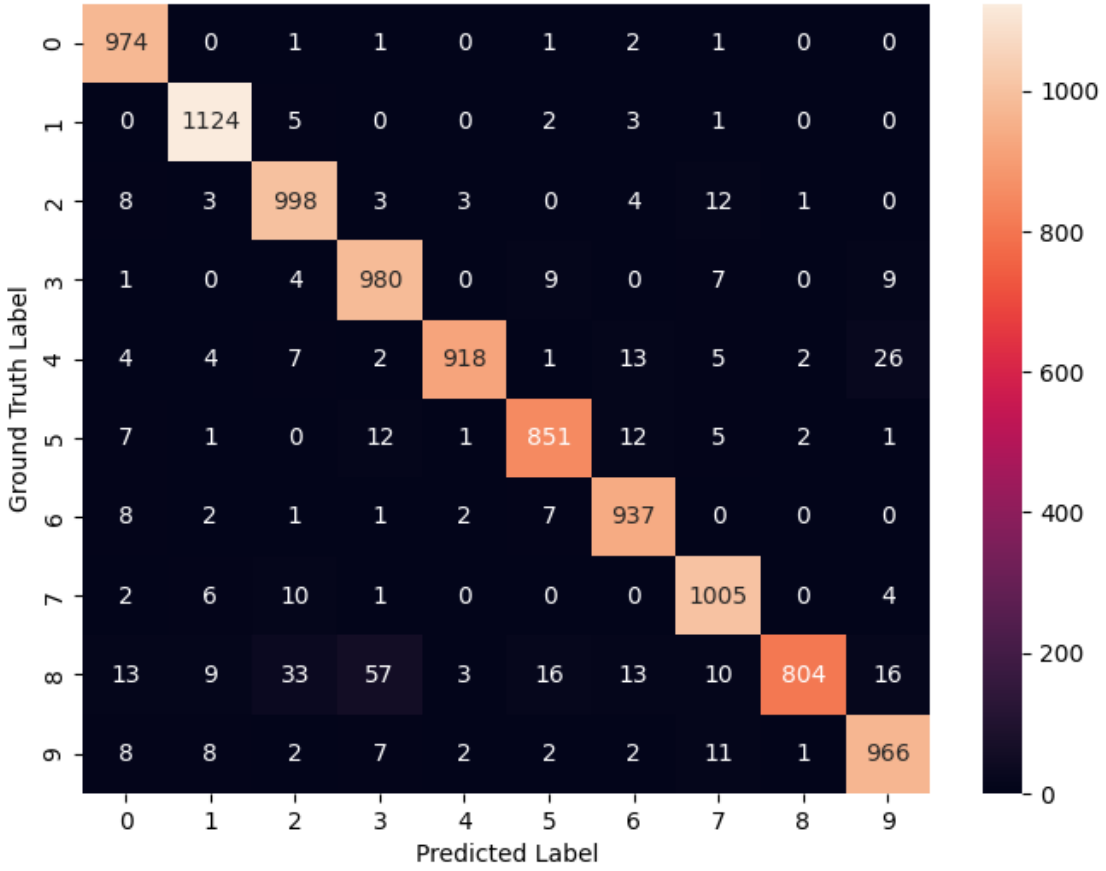
[42]:
```
from sklearn.metrics import confusion_matrix

conf_matrix = confusion_matrix(
    np.argmax(y_test, axis=1), y_pred)

print("Confusion Matrix:")
print(conf_matrix)
```

```
Confusion Matrix:
[[ 974    0    1    1    0    1    2    1    0    0]
 [   0 1124    5    0    0    2    3    1    0    0]
 [   8    3  998    3    3    0    4   12    1    0]
 [   1    0    4  980    0    9    0    7    0    9]
 [   4    4    7    2  918    1   13    5    2   26]
 [   7    1    0   12    1  851   12    5    2    1]
 [   8    2    1    1    2    7  937    0    0    0]
 [   2    6   10    1    0    0    0 1005    0    4]
 [  13    9   33   57    3   16   13   10  804   16]
 [   8    8    2    7    2    2    2   11    1  966]]
```

[43]:
```
plt.figure(figsize=(8,6))
sns.heatmap(conf_matrix, annot=True, fmt='.4g') # fmt=".0f"
plt.ylabel("Ground Truth Label")
plt.xlabel("Predicted Label")
plt.show()
```

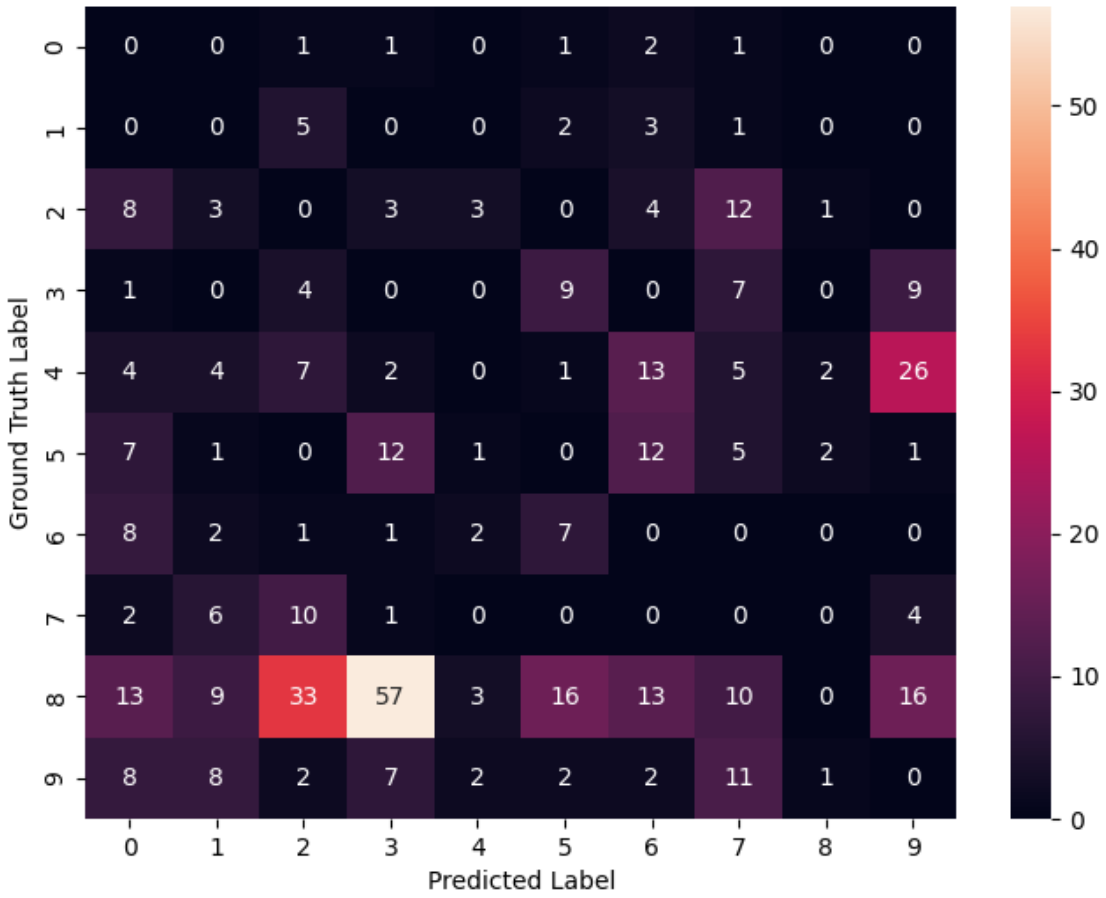


3 Error analysis using Error Matrix

This is the same confusion matrix with all true positives removed to clearly see where the model makes error

[44]:
```
np.fill_diagonal(conf_matrix, 0)

plt.figure(figsize=(8,6))
sns.heatmap(conf_matrix, annot=True, fmt='.4g') # fmt=".0f"
plt.ylabel("Ground Truth Label")
plt.xlabel("Predicted Label")
plt.show()
```



[45]:
```
np.argmax(y_test, axis=1)
```

[45]:
```
array([7, 2, 1, …, 4, 5, 6], dtype=int64)
```

3.0.1 Defining a function to visualize image based on test set indices

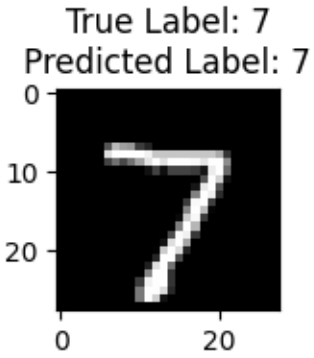
[46]:
```
def plot_digit(index):
    plt.figure(figsize=(1.5, 1.5))
    plt.imshow(X_test[index], cmap=plt.cm.gray)
    plt.title("True Label: {}\nPredicted Label: {}".format(
        np.argmax(y_test, axis=1)[index], y_pred[index]))
    plt.show()
```

In the test set, our first image is of label 7

[47]:
```
np.argmax(y_test, axis=1)[0]
```

[47]:
```
7
```

[48]:
```
plot_digit(0)
```



[49]:
```
# Index of misclassified examples

misclassified_idx = np.where(np.argmax(y_test, axis=1) != y_pred)[0]
misclassified_idx[:4]
```

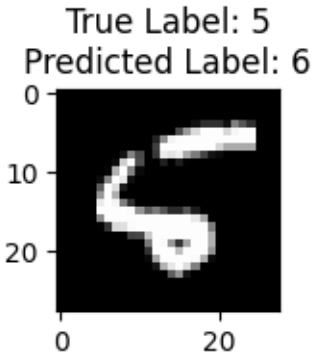
[49]:
```
array([  8, 115, 151, 179], dtype=int64)
```

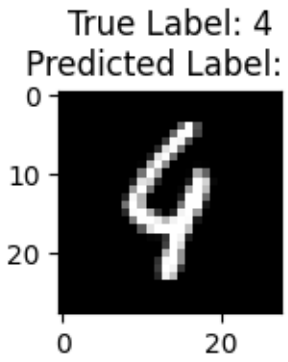
3.0.2 Misclassified examples

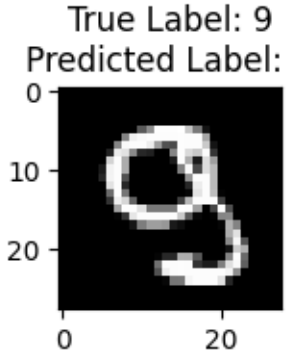
Using our user defined function plot_digit to display sample examples.

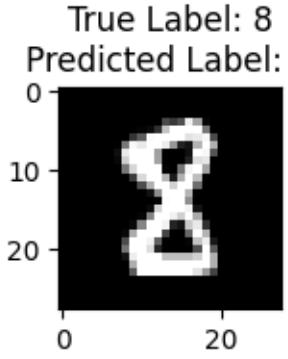
[50]:
```
# Visualizing first four misclassified examples

for i in misclassified_idx[:4]:
    plot_digit(i)
```




True Label: 4
Predicted Label: 9
0
10
20
0
20


True Label: 9
Predicted Label: 8
0
10
20
0
20


True Label: 8
Predicted Label: 2
0
10
20
0
20

4 How to find optimum parameters

[51]:
```
import scikeras
from scikeras.wrappers import KerasClassifier

from sklearn.model_selection import RandomizedSearchCV
```

[52]:
```
def build_model(
    n_hidden=1,
    n_neurons=32,
    learning_rate=0.001):


    """Helper function to build the model architecture.
    Uses: To aid in hyperparameter tuning and search.

    Parameters:
    ---------
        n_hidden: number of hidden layers
            Default: 1
        n_neurons: number of neurons in a layer
            Default: 32
        learning_rate: learning rate to use in adam optimizer
            Default: 0.001

    Returns:
    -------
        Compiled Keras Model
    """
    import tensorflow as tf

    # Sequential model
    model = tf.keras.models.Sequential()

    model.add(tf.keras.layers.Flatten(input_shape=(28,28)))

    # Add 1 dense layer according to n_hidden with dropout probability
    # according to dropout parameter
    for i in range(n_hidden):
        model.add(tf.keras.layers.BatchNormalization())
        model.add(tf.keras.layers.Dense(n_neurons, activation="relu"))

    model.add(tf.keras.layers.Dense(10, activation="softmax"))

    # Compile the model according to the learning rate provided as
    # learning_rate parameter of the helper function build_model
    model.compile(
```

```
        loss="categorical_crossentropy",
        metrics=["accuracy"],
        optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate)
    )

    return model
```

[53]:
```
# Hyper parameter combinations: 3x3x4x5
# Number of random fits selected: 8
# x 2 (cross-validation fits)
# Total fits: 16

param_distrib = {
    "n_hidden": [1, 2, 3],
    "n_neurons": [64, 128, 256],
    "learning_rate": [0.01, 0.001, 0.005, 0.0001],
}

display(param_distrib)

# Scikit learn wrapper for Keras
# It'll take a custom model building function to work
# We have build_model as our helper function
# Other parameters like n_hidden to KerasClassifier call will be
# used as arguments inside our helper function (build_model)
keras_clf = KerasClassifier(
    build_model,
    n_hidden=1,
    n_neurons=64,
    learning_rate=0.001
)

# While using scikit learn wrappers for keras
# pass a keras custom model and other search parameters
# like dict containing hyper-parameter combinations
randomized_search = RandomizedSearchCV(
    keras_clf,
    param_distrib,
    n_iter=8,
    cv=2,
    verbose=2
)

# Batch size 1024 used, please reduce if error comes
history = randomized_search.fit(
    X_train, y_train,
    epochs=10,
```

```
    validation_data=(X_test, y_test),
    batch_size=1024,
    verbose=0
)

display(randomized_search.best_params_)
display(randomized_search.best_score_)
```

```
{'n_hidden': [1, 2, 3],
 'n_neurons': [64, 128, 256],
 'learning_rate': [0.01, 0.001, 0.005, 0.0001]}
```

```
Fitting 2 folds for each of 8 candidates, totalling 16 fits
938/938 [==============================] - 1s 783us/step
[CV] END …learning_rate=0.005, n_hidden=1, n_neurons=64; total time=   2.5s
938/938 [==============================] - 1s 785us/step
[CV] END …learning_rate=0.005, n_hidden=1, n_neurons=64; total time=   2.5s
938/938 [==============================] - 1s 916us/step
[CV] END …learning_rate=0.001, n_hidden=2, n_neurons=64; total time=   3.1s
938/938 [==============================] - 1s 932us/step
[CV] END …learning_rate=0.001, n_hidden=2, n_neurons=64; total time=   3.3s
938/938 [==============================] - 1s 808us/step
[CV] END …learning_rate=0.001, n_hidden=1, n_neurons=256; total time=   2.6s
938/938 [==============================] - 1s 807us/step
[CV] END …learning_rate=0.001, n_hidden=1, n_neurons=256; total time=   2.6s
938/938 [==============================] - 1s 853us/step
[CV] END …learning_rate=0.005, n_hidden=1, n_neurons=128; total time=   2.5s
938/938 [==============================] - 1s 851us/step
[CV] END …learning_rate=0.005, n_hidden=1, n_neurons=128; total time=   2.7s
938/938 [==============================] - 1s 947us/step
[CV] END …learning_rate=0.01, n_hidden=2, n_neurons=256; total time=   3.1s
938/938 [==============================] - 1s 932us/step
[CV] END …learning_rate=0.01, n_hidden=2, n_neurons=256; total time=   3.0s
938/938 [==============================] - 1s 949us/step
[CV] END …learning_rate=0.005, n_hidden=2, n_neurons=256; total time=   3.1s
938/938 [==============================] - 1s 954us/step
[CV] END …learning_rate=0.005, n_hidden=2, n_neurons=256; total time=   3.2s
938/938 [==============================] - 1s 1ms/step
[CV] END …learning_rate=0.001, n_hidden=3, n_neurons=64; total time=   3.7s
938/938 [==============================] - 1s 1ms/step
[CV] END …learning_rate=0.001, n_hidden=3, n_neurons=64; total time=   3.6s
938/938 [==============================] - 1s 1ms/step
[CV] END …learning_rate=0.0001, n_hidden=3, n_neurons=64; total time=   3.6s
938/938 [==============================] - 1s 1ms/step
[CV] END …learning_rate=0.0001, n_hidden=3, n_neurons=64; total time=   3.8s
```

```
{'n_neurons': 256, 'n_hidden': 2, 'learning_rate': 0.005}
```

```
0.9663833333333334
```

4.1 Best hyper-parameter combination

[54]:
```
print("Best parameter combination is as follows:")
print(randomized_search.best_params_)
print("Accuracy:", randomized_search.best_score_)
```

```
Best parameter combination is as follows:
{'n_neurons': 256, 'n_hidden': 2, 'learning_rate': 0.005}
Accuracy: 0.9663833333333334
```

5 MNIST without one-hot-encoding

The only thing requiring change would be the loss function. In this case, we will use **sparse_categorical_crossentropy** as we will be using labels directly instead of onehot encoding them.

[55]:
```
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()

model = tf.keras.models.Sequential(name="MNIST_sparse_categorical_crossentropy")

model.add(tf.keras.layers.Flatten(
    input_shape=X_train[0].shape, name="input_layer"))

model.add(tf.keras.layers.BatchNormalization(
    name="minmax_normalization_1"))

model.add(tf.keras.layers.Dense(
    256, activation="relu", name="hidden_layer_1"))

model.add(tf.keras.layers.BatchNormalization(
    name="minmax_normalization_2"))

model.add(tf.keras.layers.Dense(
    256, activation="relu", name="hidden_layer_2"))

model.add(tf.keras.layers.Dense(
    10, activation="softmax", name="output_layer"))

model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
    metrics=["accuracy"]
)

print(model.summary())
```

```
Model: "MNIST_sparse_categorical_crossentropy"
_________________________________________________________________
```

```
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (Flatten)       (None, 784)               0

 minmax_normalization_1 (Bat  (None, 784)              3136
 chNormalization)

 hidden_layer_1 (Dense)      (None, 256)               200960

 minmax_normalization_2 (Bat  (None, 256)              1024
 chNormalization)

 hidden_layer_2 (Dense)      (None, 256)               65792

 output_layer (Dense)        (None, 10)                2570

=================================================================
Total params: 273,482
Trainable params: 271,402
Non-trainable params: 2,080

_________________________________________________________________
None
```

[56]:
```
results = model.fit(
    X_train, y_train,
    validation_split=0.2,
    batch_size=X_train.shape[0]//4,
    epochs=20
)
```

```
Epoch 1/20
4/4 [==============================] - 1s 60ms/step - loss: 1.3849 - accuracy:
0.5443 - val_loss: 7.3115 - val_accuracy: 0.6202
Epoch 2/20
4/4 [==============================] - 0s 20ms/step - loss: 0.3490 - accuracy:
0.8985 - val_loss: 3.3789 - val_accuracy: 0.7657
Epoch 3/20
4/4 [==============================] - 0s 19ms/step - loss: 0.2445 - accuracy:
0.9270 - val_loss: 2.2830 - val_accuracy: 0.8047
Epoch 4/20
4/4 [==============================] - 0s 21ms/step - loss: 0.1807 - accuracy:
0.9473 - val_loss: 1.6246 - val_accuracy: 0.8273
Epoch 5/20
4/4 [==============================] - 0s 21ms/step - loss: 0.1527 - accuracy:
0.9537 - val_loss: 1.4210 - val_accuracy: 0.8327
Epoch 6/20
4/4 [==============================] - 0s 21ms/step - loss: 0.1267 - accuracy:
0.9628 - val_loss: 1.1459 - val_accuracy: 0.8497
```

```
Epoch 7/20
4/4 [==============================] - 0s 21ms/step - loss: 0.1072 - accuracy:
0.9685 - val_loss: 0.9582 - val_accuracy: 0.8618
Epoch 8/20
4/4 [==============================] - 0s 21ms/step - loss: 0.0931 - accuracy:
0.9721 - val_loss: 0.7424 - val_accuracy: 0.8829
Epoch 9/20
4/4 [==============================] - 0s 21ms/step - loss: 0.0807 - accuracy:
0.9764 - val_loss: 0.5922 - val_accuracy: 0.8965
Epoch 10/20
4/4 [==============================] - 0s 21ms/step - loss: 0.0714 - accuracy:
0.9790 - val_loss: 0.4772 - val_accuracy: 0.9095
Epoch 11/20
4/4 [==============================] - 0s 20ms/step - loss: 0.0632 - accuracy:
0.9816 - val_loss: 0.4219 - val_accuracy: 0.9189
Epoch 12/20
4/4 [==============================] - 0s 21ms/step - loss: 0.0561 - accuracy:
0.9842 - val_loss: 0.3887 - val_accuracy: 0.9234
Epoch 13/20
4/4 [==============================] - 0s 23ms/step - loss: 0.0492 - accuracy:
0.9861 - val_loss: 0.3318 - val_accuracy: 0.9327
Epoch 14/20
4/4 [==============================] - 0s 21ms/step - loss: 0.0441 - accuracy:
0.9877 - val_loss: 0.3383 - val_accuracy: 0.9302
Epoch 15/20
4/4 [==============================] - 0s 21ms/step - loss: 0.0389 - accuracy:
0.9897 - val_loss: 0.2525 - val_accuracy: 0.9469
Epoch 16/20
4/4 [==============================] - 0s 20ms/step - loss: 0.0338 - accuracy:
0.9909 - val_loss: 0.2651 - val_accuracy: 0.9426
Epoch 17/20
4/4 [==============================] - 0s 20ms/step - loss: 0.0302 - accuracy:
0.9922 - val_loss: 0.2625 - val_accuracy: 0.9434
Epoch 18/20
4/4 [==============================] - 0s 20ms/step - loss: 0.0262 - accuracy:
0.9933 - val_loss: 0.2384 - val_accuracy: 0.9494
Epoch 19/20
4/4 [==============================] - 0s 20ms/step - loss: 0.0225 - accuracy:
0.9949 - val_loss: 0.2345 - val_accuracy: 0.9493
Epoch 20/20
4/4 [==============================] - 0s 21ms/step - loss: 0.0197 - accuracy:
0.9955 - val_loss: 0.2186 - val_accuracy: 0.9517
```

[57]:
```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       980
```

1	0.97	0.99	0.98	1135
2	0.94	0.97	0.95	1032
3	0.92	0.97	0.95	1010
4	0.99	0.93	0.96	982
5	0.96	0.95	0.96	892
6	0.95	0.98	0.96	958
7	0.95	0.98	0.96	1028
8	0.99	0.83	0.90	974
9	0.95	0.96	0.95	1009
accuracy			0.96	10000
macro avg	0.96	0.95	0.95	10000
weighted avg	0.96	0.96	0.96	10000