# Deep Learning

April 24, 2021

## 0.1 Deep Learning: Classifying Irish Data.

First we need to convert the species column in binary form. Next we retrieve the data as a list and convert back as an array

[234]:
```
import pandas as pd
import numpy as np
from sklearn.utils import shuffle
from numpy import loadtxt
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split

#Importing and reading data.
Iris= pd.read_csv("Iris.csv")
# Since our data is seperated by semicolons we need to do sep=";"
Iris=Iris.iloc[:,[1,2,3,4,5]]
Iris.head()
```

[234]:

| | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm | Species |
|---|---|---|---|---|---|
| 0 | 5.1 | 3.5 | 1.4 | 0.2 | Iris-setosa |
| 1 | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| 2 | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| 3 | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| 4 | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |

[235]:
```
#Reshuffling the data
Iris= shuffle(Iris)
Iris.head()
```

[235]:

| | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm | Species |
|---|---|---|---|---|---|
| 28 | 5.2 | 3.4 | 1.4 | 0.2 | Iris-setosa |
| 132 | 6.4 | 2.8 | 5.6 | 2.2 | Iris-virginica |
| 131 | 7.9 | 3.8 | 6.4 | 2.0 | Iris-virginica |
| 101 | 5.8 | 2.7 | 5.1 | 1.9 | Iris-virginica |
| 22 | 4.6 | 3.6 | 1.0 | 0.2 | Iris-setosa |

[236]:
```
Iris_list=Iris.values.tolist()
for i in range(len(Iris_list)):
    if(Iris_list[i][4]=='Iris-versicolor'):
        Iris_list[i][4]=0
    elif (Iris_list[i][4]=='Iris-setosa'):
         Iris_list[i][4]=1
    else:
         Iris_list[i][4]=2

data=pd.DataFrame(Iris_list)
X=data.iloc[:,[0,1,2,3]]
y=data.iloc[:,[4]]
```

[237]:
```
# define the keras model
model = Sequential()
model.add(Dense(5, input_dim=4, activation='relu'))
model.add(Dense(3, activation='relu'))
model.add(Dense(3, activation='softmax'))
# compile the keras model
model.compile(loss='sparse_categorical_crossentropy', optimizer='Adam',␣
 ↪metrics=['accuracy'])
```

[238]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)
model=model.fit(X_train, y_train,validation_split=0.1, epochs=100,␣
 ↪batch_size=10)
```

```
Epoch 1/100
 1/10 [==>…] - ETA: 3s - loss: 1.2662 - accuracy:
0.0000e+00WARNING:tensorflow:10 out of the last 35 calls to <function
Model.make_test_function.<locals>.test_function at 0x162b2c940> triggered
tf.function retracing. Tracing is expensive and the excessive number of tracings
could be due to (1) creating @tf.function repeatedly in a loop, (2) passing
tensors with different shapes, (3) passing Python objects instead of tensors.
For (1), please define your @tf.function outside of the loop. For (2),
@tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
10/10 [==============================] - 1s 22ms/step - loss: 1.2083 - accuracy:
0.0000e+00 - val_loss: 1.2768 - val_accuracy: 0.0000e+00
Epoch 2/100
10/10 [==============================] - 0s 6ms/step - loss: 1.1374 - accuracy:
0.0000e+00 - val_loss: 1.2477 - val_accuracy: 0.0000e+00
Epoch 3/100
10/10 [==============================] - 0s 7ms/step - loss: 1.1323 - accuracy:
0.0000e+00 - val_loss: 1.2195 - val_accuracy: 0.0000e+00
```

```
Epoch 4/100
10/10 [==============================] - 0s 7ms/step - loss: 1.0984 - accuracy:
0.0761 - val_loss: 1.1982 - val_accuracy: 0.2727
Epoch 5/100
10/10 [==============================] - 0s 6ms/step - loss: 1.1022 - accuracy:
0.3286 - val_loss: 1.1796 - val_accuracy: 0.2727
Epoch 6/100
10/10 [==============================] - 0s 6ms/step - loss: 1.0837 - accuracy:
0.3691 - val_loss: 1.1685 - val_accuracy: 0.2727
Epoch 7/100
10/10 [==============================] - 0s 8ms/step - loss: 1.0825 - accuracy:
0.3201 - val_loss: 1.1581 - val_accuracy: 0.2727
Epoch 8/100
10/10 [==============================] - 0s 7ms/step - loss: 1.0737 - accuracy:
0.3349 - val_loss: 1.1461 - val_accuracy: 0.2727
Epoch 9/100
10/10 [==============================] - 0s 7ms/step - loss: 1.0573 - accuracy:
0.3413 - val_loss: 1.1347 - val_accuracy: 0.2727
Epoch 10/100
10/10 [==============================] - 0s 7ms/step - loss: 1.0795 - accuracy:
0.2711 - val_loss: 1.1220 - val_accuracy: 0.2727
Epoch 11/100
10/10 [==============================] - 0s 7ms/step - loss: 1.0170 - accuracy:
0.4358 - val_loss: 1.1155 - val_accuracy: 0.2727
Epoch 12/100
10/10 [==============================] - 0s 6ms/step - loss: 1.0071 - accuracy:
0.3767 - val_loss: 1.1068 - val_accuracy: 0.2727
Epoch 13/100
10/10 [==============================] - 0s 7ms/step - loss: 1.0335 - accuracy:
0.3054 - val_loss: 1.0935 - val_accuracy: 0.2727
Epoch 14/100
10/10 [==============================] - 0s 7ms/step - loss: 1.0136 - accuracy:
0.3056 - val_loss: 1.0830 - val_accuracy: 0.2727
Epoch 15/100
10/10 [==============================] - 0s 8ms/step - loss: 0.9659 - accuracy:
0.3669 - val_loss: 1.0761 - val_accuracy: 0.2727
Epoch 16/100
10/10 [==============================] - 0s 7ms/step - loss: 0.9716 - accuracy:
0.3556 - val_loss: 1.0627 - val_accuracy: 0.2727
Epoch 17/100
10/10 [==============================] - 0s 7ms/step - loss: 0.9405 - accuracy:
0.3699 - val_loss: 1.0501 - val_accuracy: 0.2727
Epoch 18/100
10/10 [==============================] - 0s 6ms/step - loss: 0.9157 - accuracy:
0.3860 - val_loss: 1.0330 - val_accuracy: 0.2727
Epoch 19/100
10/10 [==============================] - 0s 6ms/step - loss: 0.9185 - accuracy:
0.3452 - val_loss: 1.0157 - val_accuracy: 0.2727
```

```
Epoch 20/100
10/10 [==============================] - 0s 7ms/step - loss: 0.8701 - accuracy:
0.3454 - val_loss: 1.0013 - val_accuracy: 0.2727
Epoch 21/100
10/10 [==============================] - 0s 9ms/step - loss: 0.8699 - accuracy:
0.3584 - val_loss: 0.9867 - val_accuracy: 0.2727
Epoch 22/100
10/10 [==============================] - 0s 11ms/step - loss: 0.8053 - accuracy:
0.4475 - val_loss: 0.9717 - val_accuracy: 0.3636
Epoch 23/100
10/10 [==============================] - 0s 14ms/step - loss: 0.8335 - accuracy:
0.6314 - val_loss: 0.9546 - val_accuracy: 0.4545
Epoch 24/100
10/10 [==============================] - 0s 8ms/step - loss: 0.7618 - accuracy:
0.7388 - val_loss: 0.9390 - val_accuracy: 0.4545
Epoch 25/100
10/10 [==============================] - 0s 8ms/step - loss: 0.7955 - accuracy:
0.6826 - val_loss: 0.9207 - val_accuracy: 0.4545
Epoch 26/100
10/10 [==============================] - 0s 7ms/step - loss: 0.7345 - accuracy:
0.7681 - val_loss: 0.9105 - val_accuracy: 0.4545
Epoch 27/100
10/10 [==============================] - 0s 6ms/step - loss: 0.6975 - accuracy:
0.7730 - val_loss: 0.8977 - val_accuracy: 0.4545
Epoch 28/100
10/10 [==============================] - 0s 7ms/step - loss: 0.6738 - accuracy:
0.7751 - val_loss: 0.8796 - val_accuracy: 0.4545
Epoch 29/100
10/10 [==============================] - 0s 6ms/step - loss: 0.7136 - accuracy:
0.6998 - val_loss: 0.8612 - val_accuracy: 0.4545
Epoch 30/100
10/10 [==============================] - 0s 7ms/step - loss: 0.6730 - accuracy:
0.7265 - val_loss: 0.8529 - val_accuracy: 0.4545
Epoch 31/100
10/10 [==============================] - 0s 7ms/step - loss: 0.6523 - accuracy:
0.7288 - val_loss: 0.8393 - val_accuracy: 0.4545
Epoch 32/100
10/10 [==============================] - 0s 7ms/step - loss: 0.6043 - accuracy:
0.7637 - val_loss: 0.8297 - val_accuracy: 0.4545
Epoch 33/100
10/10 [==============================] - 0s 6ms/step - loss: 0.5823 - accuracy:
0.7736 - val_loss: 0.8224 - val_accuracy: 0.4545
Epoch 34/100
10/10 [==============================] - 0s 6ms/step - loss: 0.5413 - accuracy:
0.8105 - val_loss: 0.8105 - val_accuracy: 0.4545
Epoch 35/100
10/10 [==============================] - 0s 6ms/step - loss: 0.5648 - accuracy:
0.7571 - val_loss: 0.7939 - val_accuracy: 0.4545
```

```
Epoch 36/100
10/10 [==============================] - 0s 5ms/step - loss: 0.5404 - accuracy:
0.7595 - val_loss: 0.7830 - val_accuracy: 0.4545
Epoch 37/100
10/10 [==============================] - 0s 5ms/step - loss: 0.5620 - accuracy:
0.7066 - val_loss: 0.7742 - val_accuracy: 0.4545
Epoch 38/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4939 - accuracy:
0.7879 - val_loss: 0.7679 - val_accuracy: 0.4545
Epoch 39/100
10/10 [==============================] - 0s 5ms/step - loss: 0.4917 - accuracy:
0.7784 - val_loss: 0.7521 - val_accuracy: 0.4545
Epoch 40/100
10/10 [==============================] - 0s 7ms/step - loss: 0.5614 - accuracy:
0.6748 - val_loss: 0.7435 - val_accuracy: 0.4545
Epoch 41/100
10/10 [==============================] - 0s 8ms/step - loss: 0.4883 - accuracy:
0.7359 - val_loss: 0.7353 - val_accuracy: 0.4545
Epoch 42/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4853 - accuracy:
0.7254 - val_loss: 0.7313 - val_accuracy: 0.4545
Epoch 43/100
10/10 [==============================] - 0s 8ms/step - loss: 0.5085 - accuracy:
0.7046 - val_loss: 0.7231 - val_accuracy: 0.4545
Epoch 44/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4731 - accuracy:
0.7416 - val_loss: 0.7160 - val_accuracy: 0.4545
Epoch 45/100
10/10 [==============================] - 0s 6ms/step - loss: 0.5037 - accuracy:
0.6889 - val_loss: 0.7080 - val_accuracy: 0.4545
Epoch 46/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4482 - accuracy:
0.7601 - val_loss: 0.7033 - val_accuracy: 0.4545
Epoch 47/100
10/10 [==============================] - 0s 7ms/step - loss: 0.4227 - accuracy:
0.7674 - val_loss: 0.6914 - val_accuracy: 0.4545
Epoch 48/100
10/10 [==============================] - 0s 8ms/step - loss: 0.4519 - accuracy:
0.7089 - val_loss: 0.6829 - val_accuracy: 0.4545
Epoch 49/100
10/10 [==============================] - 0s 5ms/step - loss: 0.4555 - accuracy:
0.7375 - val_loss: 0.6818 - val_accuracy: 0.4545
Epoch 50/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4385 - accuracy:
0.7114 - val_loss: 0.6785 - val_accuracy: 0.4545
Epoch 51/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4731 - accuracy:
0.6772 - val_loss: 0.6820 - val_accuracy: 0.4545
```

```
Epoch 52/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4882 - accuracy:
0.6600 - val_loss: 0.6837 - val_accuracy: 0.4545
Epoch 53/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4800 - accuracy:
0.6703 - val_loss: 0.6833 - val_accuracy: 0.4545
Epoch 54/100
10/10 [==============================] - 0s 9ms/step - loss: 0.4240 - accuracy:
0.7096 - val_loss: 0.6863 - val_accuracy: 0.4545
Epoch 55/100
10/10 [==============================] - 0s 11ms/step - loss: 0.4322 - accuracy:
0.7165 - val_loss: 0.6810 - val_accuracy: 0.4545
Epoch 56/100
10/10 [==============================] - 0s 7ms/step - loss: 0.4019 - accuracy:
0.7599 - val_loss: 0.6793 - val_accuracy: 0.4545
Epoch 57/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4717 - accuracy:
0.6797 - val_loss: 0.6700 - val_accuracy: 0.4545
Epoch 58/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4050 - accuracy:
0.7141 - val_loss: 0.6617 - val_accuracy: 0.4545
Epoch 59/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3748 - accuracy:
0.7748 - val_loss: 0.6616 - val_accuracy: 0.4545
Epoch 60/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3848 - accuracy:
0.7399 - val_loss: 0.6595 - val_accuracy: 0.4545
Epoch 61/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3656 - accuracy:
0.7682 - val_loss: 0.6523 - val_accuracy: 0.4545
Epoch 62/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4451 - accuracy:
0.6585 - val_loss: 0.6432 - val_accuracy: 0.4545
Epoch 63/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3977 - accuracy:
0.7094 - val_loss: 0.6447 - val_accuracy: 0.4545
Epoch 64/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4259 - accuracy:
0.7075 - val_loss: 0.6435 - val_accuracy: 0.4545
Epoch 65/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3908 - accuracy:
0.7475 - val_loss: 0.6453 - val_accuracy: 0.4545
Epoch 66/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3703 - accuracy:
0.7705 - val_loss: 0.6405 - val_accuracy: 0.4545
Epoch 67/100
10/10 [==============================] - 0s 8ms/step - loss: 0.4288 - accuracy:
0.6857 - val_loss: 0.6322 - val_accuracy: 0.4545
```

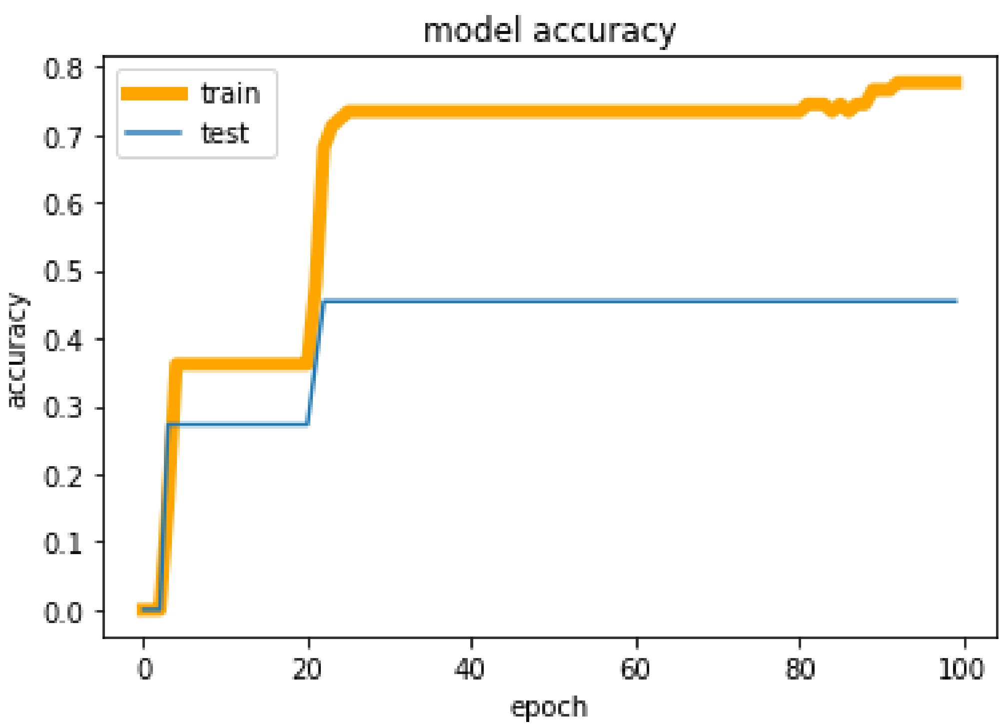
```
Epoch 68/100
10/10 [==============================] - 0s 8ms/step - loss: 0.3728 - accuracy:
0.7452 - val_loss: 0.6299 - val_accuracy: 0.4545
Epoch 69/100
10/10 [==============================] - 0s 8ms/step - loss: 0.4350 - accuracy:
0.6694 - val_loss: 0.6302 - val_accuracy: 0.4545
Epoch 70/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3781 - accuracy:
0.7345 - val_loss: 0.6308 - val_accuracy: 0.4545
Epoch 71/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3851 - accuracy:
0.7104 - val_loss: 0.6247 - val_accuracy: 0.4545
Epoch 72/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4077 - accuracy:
0.6803 - val_loss: 0.6265 - val_accuracy: 0.4545
Epoch 73/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3489 - accuracy:
0.7907 - val_loss: 0.6407 - val_accuracy: 0.4545
Epoch 74/100
10/10 [==============================] - 0s 5ms/step - loss: 0.4166 - accuracy:
0.6741 - val_loss: 0.6253 - val_accuracy: 0.4545
Epoch 75/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4156 - accuracy:
0.6778 - val_loss: 0.6245 - val_accuracy: 0.4545
Epoch 76/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3895 - accuracy:
0.7306 - val_loss: 0.6216 - val_accuracy: 0.4545
Epoch 77/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3885 - accuracy:
0.7222 - val_loss: 0.6210 - val_accuracy: 0.4545
Epoch 78/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3905 - accuracy:
0.7048 - val_loss: 0.6157 - val_accuracy: 0.4545
Epoch 79/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3698 - accuracy:
0.7447 - val_loss: 0.6180 - val_accuracy: 0.4545
Epoch 80/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3760 - accuracy:
0.7243 - val_loss: 0.6152 - val_accuracy: 0.4545
Epoch 81/100
10/10 [==============================] - 0s 6ms/step - loss: 0.4362 - accuracy:
0.6484 - val_loss: 0.6051 - val_accuracy: 0.4545
Epoch 82/100
10/10 [==============================] - 0s 7ms/step - loss: 0.3805 - accuracy:
0.7104 - val_loss: 0.6103 - val_accuracy: 0.4545
Epoch 83/100
10/10 [==============================] - 0s 7ms/step - loss: 0.3804 - accuracy:
0.7312 - val_loss: 0.6066 - val_accuracy: 0.4545
```

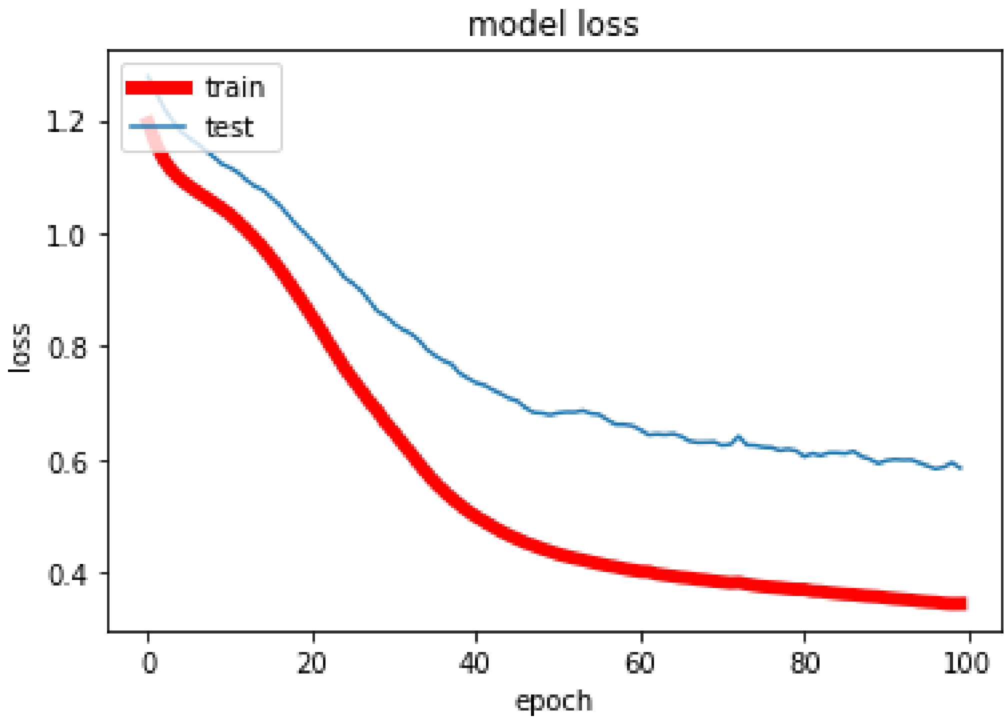
```
Epoch 84/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3676 - accuracy:
0.7299 - val_loss: 0.6116 - val_accuracy: 0.4545
Epoch 85/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3410 - accuracy:
0.7485 - val_loss: 0.6115 - val_accuracy: 0.4545
Epoch 86/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3884 - accuracy:
0.7295 - val_loss: 0.6101 - val_accuracy: 0.4545
Epoch 87/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3654 - accuracy:
0.7333 - val_loss: 0.6145 - val_accuracy: 0.4545
Epoch 88/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3389 - accuracy:
0.7634 - val_loss: 0.6049 - val_accuracy: 0.4545
Epoch 89/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3233 - accuracy:
0.7819 - val_loss: 0.6004 - val_accuracy: 0.4545
Epoch 90/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3708 - accuracy:
0.7482 - val_loss: 0.5926 - val_accuracy: 0.4545
Epoch 91/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3745 - accuracy:
0.7660 - val_loss: 0.5979 - val_accuracy: 0.4545
Epoch 92/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3669 - accuracy:
0.7441 - val_loss: 0.5999 - val_accuracy: 0.4545
Epoch 93/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3660 - accuracy:
0.7542 - val_loss: 0.5989 - val_accuracy: 0.4545
Epoch 94/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3118 - accuracy:
0.8259 - val_loss: 0.5993 - val_accuracy: 0.4545
Epoch 95/100
10/10 [==============================] - 0s 6ms/step - loss: 0.3448 - accuracy:
0.7909 - val_loss: 0.5940 - val_accuracy: 0.4545
Epoch 96/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3142 - accuracy:
0.7973 - val_loss: 0.5884 - val_accuracy: 0.4545
Epoch 97/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3490 - accuracy:
0.7748 - val_loss: 0.5832 - val_accuracy: 0.4545
Epoch 98/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3667 - accuracy:
0.7541 - val_loss: 0.5865 - val_accuracy: 0.4545
Epoch 99/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3325 - accuracy:
0.8127 - val_loss: 0.5942 - val_accuracy: 0.4545
```

```
Epoch 100/100
10/10 [==============================] - 0s 5ms/step - loss: 0.3773 - accuracy:
0.6999 - val_loss: 0.5846 - val_accuracy: 0.4545
```

[239]:

```python
import matplotlib.pyplot as plt
loss=list(model.history.values())[0]
accuracy=list(model.history.values())[1]
val_loss=list(model.history.values())[2]
val_accuracy=list(model.history.values())[3]

# summarize history for accuracy
plt.plot(accuracy,color='orange', linewidth=5)
plt.plot(val_accuracy)
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(loss, color='red', linewidth=5)
plt.plot(val_loss)
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```






### 0.2 IMAGE CLASSIFICATION: MNIST DATASET

[240]:
```
#install required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.models import Sequential
from keras.layers import Convolution2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from keras.layers import Dropout
#model evaluation packages
from sklearn.metrics import f1_score, roc_auc_score, log_loss
from sklearn.model_selection import cross_val_score, cross_validate
```

[241]:
```
#read mnist fashion dataset
mnist = keras.datasets.fashion_mnist
```

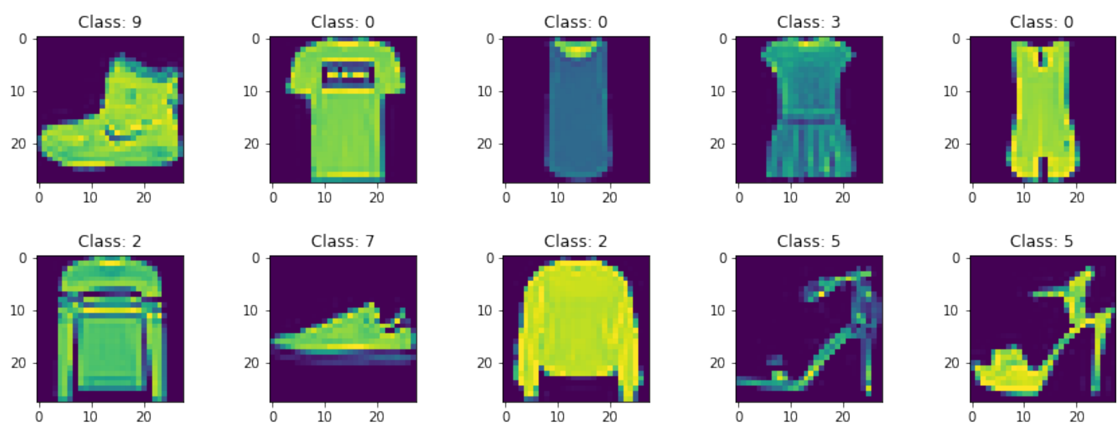
```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

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

[242]:
```
#reshape data from 3-D to 2-D array
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
#feature scaling
from sklearn.preprocessing import MinMaxScaler
minmax = MinMaxScaler()
#fit and transform training dataset
X_train = minmax.fit_transform(X_train)
X_train=X_train/255
#transform testing dataset
X_test = minmax.transform(X_test)
print('Number of unique classes: ', len(np.unique(y_train)))
print('Classes: ', np.unique(y_train))
```

```
Number of unique classes:  10
Classes:  [0 1 2 3 4 5 6 7 8 9]
```

[243]:
```
fig, axes = plt.subplots(nrows=2, ncols=5,figsize=(15,5))
ax = axes.ravel()
for i in range(10):
    ax[i].imshow(X_train[i].reshape(28,28))
    ax[i].title.set_text('Class: ' + str(y_train[i]))
plt.subplots_adjust(hspace=0.5)
plt.show()
```




[244]:
```
classifier_e25 = Sequential()
#add 1st hidden layer
```

```
classifier_e25.add(Dense(input_dim = X_train.shape[1], units = 256,␣
 ↪kernel_initializer='uniform', activation='relu'))
classifier_e25.add(Dense(256, kernel_initializer='uniform', activation='relu'))
#add output layer
classifier_e25.add(Dense(units = 10, kernel_initializer='uniform',␣
 ↪activation='softmax'))
#compile the neural network
classifier_e25.compile(optimizer='adam',␣
 ↪loss='sparse_categorical_crossentropy', metrics=['accuracy'])
#model summary
classifier_e25.summary()
```

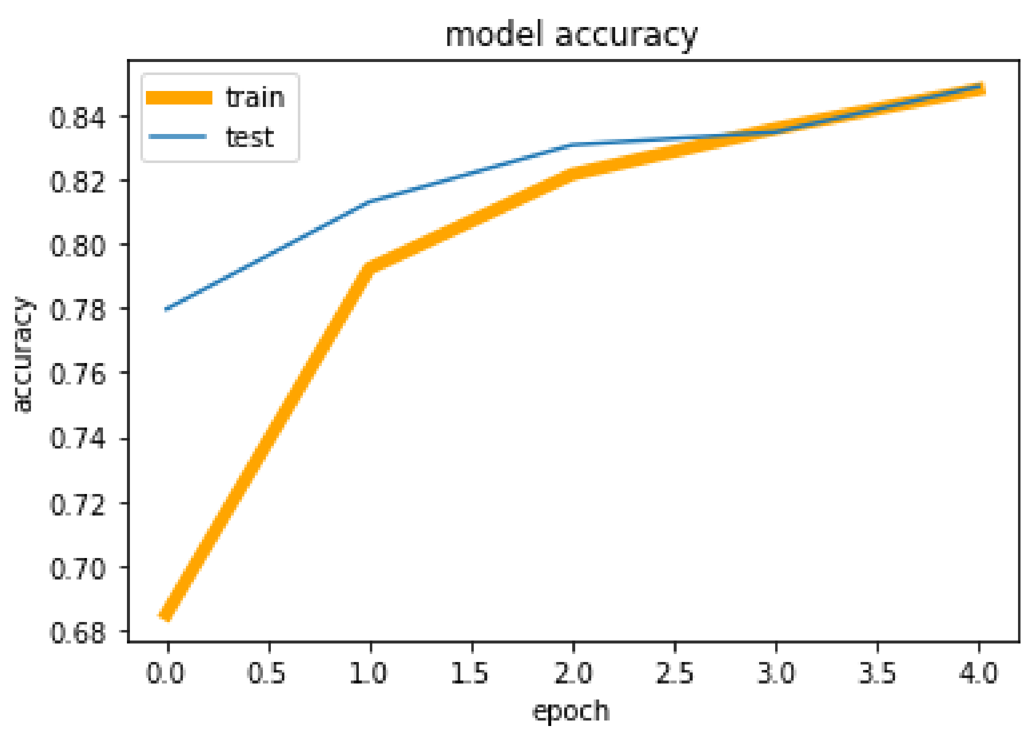
```
Model: "sequential_41"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_97 (Dense)             (None, 256)               200960
_________________________________________________________________
dense_98 (Dense)             (None, 256)               65792
_________________________________________________________________
dense_99 (Dense)             (None, 10)                2570
=================================================================
Total params: 269,322
Trainable params: 269,322
Non-trainable params: 0
_________________________________________________________________
```

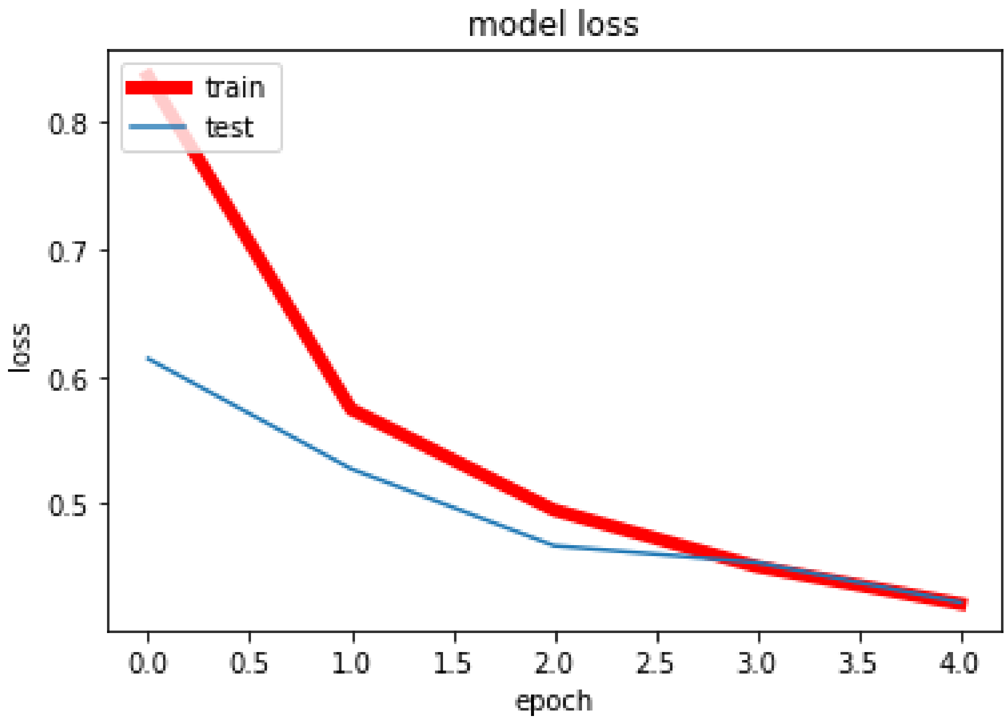
[245]:
```
model= classifier_e25.fit(X_train, y_train, validation_split=0.33, epochs=5,␣
 ↪batch_size=10)
```

```
Epoch 1/5
4020/4020 [==============================] - 11s 3ms/step - loss: 1.0933 -
accuracy: 0.5823 - val_loss: 0.6141 - val_accuracy: 0.7796
Epoch 2/5
4020/4020 [==============================] - 10s 2ms/step - loss: 0.5956 -
accuracy: 0.7827 - val_loss: 0.5275 - val_accuracy: 0.8128
Epoch 3/5
4020/4020 [==============================] - 10s 3ms/step - loss: 0.5029 -
accuracy: 0.8166 - val_loss: 0.4669 - val_accuracy: 0.8305
Epoch 4/5
4020/4020 [==============================] - 10s 3ms/step - loss: 0.4604 -
accuracy: 0.8334 - val_loss: 0.4537 - val_accuracy: 0.8345
Epoch 5/5
4020/4020 [==============================] - 10s 3ms/step - loss: 0.4329 -
accuracy: 0.8440 - val_loss: 0.4224 - val_accuracy: 0.8486
```

[246]:

```
loss=list(model.history.values())[0]
accuracy=list(model.history.values())[1]
val_loss=list(model.history.values())[2]
val_accuracy=list(model.history.values())[3]

# summarize history for accuracy
plt.plot(accuracy,color='orange', linewidth=5)
plt.plot(val_accuracy)
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(loss, color='red', linewidth=5)
plt.plot(val_loss)
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```






[247]:
```
#evaluate the model for testing dataset
test_loss_e25 = classifier_e25.evaluate(X_test, y_test, verbose=0)
#calculate evaluation parameters
f1_e25 = f1_score(y_test, classifier_e25.predict_classes(X_test),␣
↪average='micro')
roc_e25 = roc_auc_score(y_test, classifier_e25.predict_proba(X_test),␣
↪multi_class='ovo')
#create evaluation dataframe
stats_e25 = pd.DataFrame({'Test accuracy' :  round(test_loss_e25[1]*100,3),
                      'F1 score'      : round(f1_e25,3),
                      'ROC AUC score' : round(roc_e25,3),
                      'Total Loss'    : round(test_loss_e25[0],3)}, index=[0])
#print evaluation dataframe
display(stats_e25)
```

```
/opt/anaconda3/envs/Project/lib/python3.8/site-
packages/tensorflow/python/keras/engine/sequential.py:450: UserWarning:
`model.predict_classes()` is deprecated and will be removed after 2021-01-01.
Please use instead:* `np.argmax(model.predict(x), axis=-1)`,   if your model
does multi-class classification   (e.g. if it uses a `softmax` last-layer
activation).* `(model.predict(x) > 0.5).astype("int32")`,   if your model does
binary classification   (e.g. if it uses a `sigmoid` last-layer activation).
  warnings.warn('`model.predict_classes()` is deprecated and '
/opt/anaconda3/envs/Project/lib/python3.8/site-
```

```
packages/tensorflow/python/keras/engine/sequential.py:425: UserWarning:
`model.predict_proba()` is deprecated and will be removed after 2021-01-01.
Please use `model.predict()` instead.
  warnings.warn('`model.predict_proba()` is deprecated and '
```

| | Test accuracy | F1 score | ROC AUC score | Total Loss |
|---|---|---|---|---|
| 0 | 69.52 | 0.695 | 0.86 | 85.949 |

# 1 CHARACTER CLASSIFICATION: MNIST DATASET

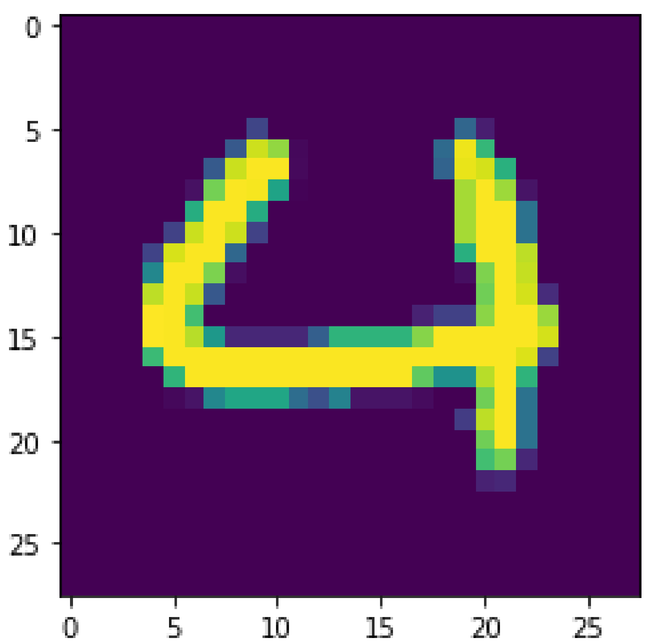
[335]:
```
#read mnist fashion dataset
mnist = keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
y_train
```

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

[335]:
```
array([5, 0, 4, …, 5, 6, 8], dtype=uint8)
```

[383]:
```
plt.imshow(X_train[60])
```

[383]:
```
<matplotlib.image.AxesImage at 0x163b32e80>
```



[338]:
```
X_train=X_train/X_train.max()
X_test=X_test/X_train.max()
```

[348]:
```
# Set random seed
tf.random.set_seed(42)

# Create the model
model = tf.keras.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(100, activation="relu"),
  tf.keras.layers.Dense(100, activation="relu"),
  tf.keras.layers.Dense(10, activation="softmax") # output shape is 10,␣
 ↪activation is softmax
])

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

# Fit the model
history = model.fit(X_train,
                                y_train,
                                epochs=10,
                                validation_data=(X_test, y_test)) # see ho
```

```
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4253 -
accuracy: 0.8758 - val_loss: 20.6599 - val_accuracy: 0.9564
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1109 -
accuracy: 0.9661 - val_loss: 19.6268 - val_accuracy: 0.9615
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0794 -
accuracy: 0.9751 - val_loss: 12.0422 - val_accuracy: 0.9768
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0534 -
accuracy: 0.9837 - val_loss: 15.5001 - val_accuracy: 0.9745
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0451 -
accuracy: 0.9856 - val_loss: 15.1063 - val_accuracy: 0.9744
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0344 -
accuracy: 0.9890 - val_loss: 17.6403 - val_accuracy: 0.9740
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0274 -
accuracy: 0.9906 - val_loss: 18.0195 - val_accuracy: 0.9763
```

```
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0197 -
accuracy: 0.9936 - val_loss: 20.0415 - val_accuracy: 0.9755
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0185 -
accuracy: 0.9944 - val_loss: 21.5667 - val_accuracy: 0.9760
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0162 -
accuracy: 0.9942 - val_loss: 21.8415 - val_accuracy: 0.9761
```

[350]:

```
model.summary()
```

```
Model: "sequential_62"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_10 (Flatten)         (None, 784)               0
_________________________________________________________________
dense_164 (Dense)            (None, 100)               78500
_________________________________________________________________
dense_165 (Dense)            (None, 100)               10100
_________________________________________________________________
dense_166 (Dense)            (None, 10)                1010
=================================================================
Total params: 89,610
Trainable params: 89,610
Non-trainable params: 0
_________________________________________________________________
```

[353]:

```
# Note: The following confusion matrix code is a remix of Scikit-Learn's
# plot_confusion_matrix function - https://scikit-learn.org/stable/modules/
↪generated/sklearn.
#metrics.plot_confusion_matrix.html
# and Made with ML's introductory notebook - https://github.com/madewithml/
↪basics/blob/master/notebooks/
#09_Multilayer_Perceptrons/09_TF_Multilayer_Perceptrons.ipynb
import itertools
from sklearn.metrics import confusion_matrix

# Our function needs a different name to sklearn's plot_confusion_matrix
def make_confusion_matrix(y_true, y_pred, classes=None, figsize=(10, 10),␣
↪text_size=15):
  """Makes a labelled confusion matrix comparing predictions and ground truth␣
↪labels.

  If classes is passed, confusion matrix will be labelled, if not, integer␣
↪class values
```

```
  will be used.

  Args:
    y_true: Array of truth labels (must be same shape as y_pred).
    y_pred: Array of predicted labels (must be same shape as y_true).
    classes: Array of class labels (e.g. string form). If `None`, integer␣
↪labels are used.
    figsize: Size of output figure (default=(10, 10)).
    text_size: Size of output figure text (default=15).

  Returns:
    A labelled confusion matrix plot comparing y_true and y_pred.

  Example usage:
    make_confusion_matrix(y_true=test_labels, # ground truth test labels
                          y_pred=y_preds, # predicted labels
                          classes=class_names, # array of class label names
                          figsize=(15, 15),
                          text_size=10)
  """
  # Create the confustion matrix
  cm = confusion_matrix(y_true, y_pred)
  cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis] # normalize it
  n_classes = cm.shape[0] # find the number of classes we're dealing with

  # Plot the figure and make it pretty
  fig, ax = plt.subplots(figsize=figsize)
  cax = ax.matshow(cm, cmap=plt.cm.Blues) # colors will represent how 'correct'␣
↪a class is, darker == better
  fig.colorbar(cax)

  # Are there a list of classes?
  if classes:
    labels = classes
  else:
    labels = np.arange(cm.shape[0])

  # Label the axes
  ax.set(title="Confusion Matrix",
         xlabel="Predicted label",
         ylabel="True label",
         xticks=np.arange(n_classes), # create enough axis slots for each class
         yticks=np.arange(n_classes),
         xticklabels=labels, # axes will labeled with class names (if they␣
↪exist) or ints
         yticklabels=labels)
```

```
  # Make x-axis labels appear on bottom
  ax.xaxis.set_label_position("bottom")
  ax.xaxis.tick_bottom()

  # Set the threshold for different colors
  threshold = (cm.max() + cm.min()) / 2.

  # Plot the text on each cell
  for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, f"{cm[i, j]} ({cm_norm[i, j]*100:.1f}%)",
             horizontalalignment="center",
             color="white" if cm[i, j] > threshold else "black",
             size=text_size)
```

[393]:
```
y_probs = model.predict(X_test) # "probs" is short for probabilities

# View the first 5 predictions
y_probs[:5]
```

[393]:
```
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.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]], dtype=float32)
```

[394]:
```
y_preds = y_probs.argmax(axis=1)

# View the first 10 prediction labels
y_preds[:10]
```

[394]:
```
array([7, 2, 1, 0, 4, 1, 4, 9, 5, 9])
```

[423]:
```
less=[(y_preds[i],y_test[i]) for i in range(9)]
less
```

[423]:
```
[(7, 7), (2, 2), (1, 1), (0, 0), (4, 4), (1, 1), (4, 4), (9, 9), (5, 5)]
```

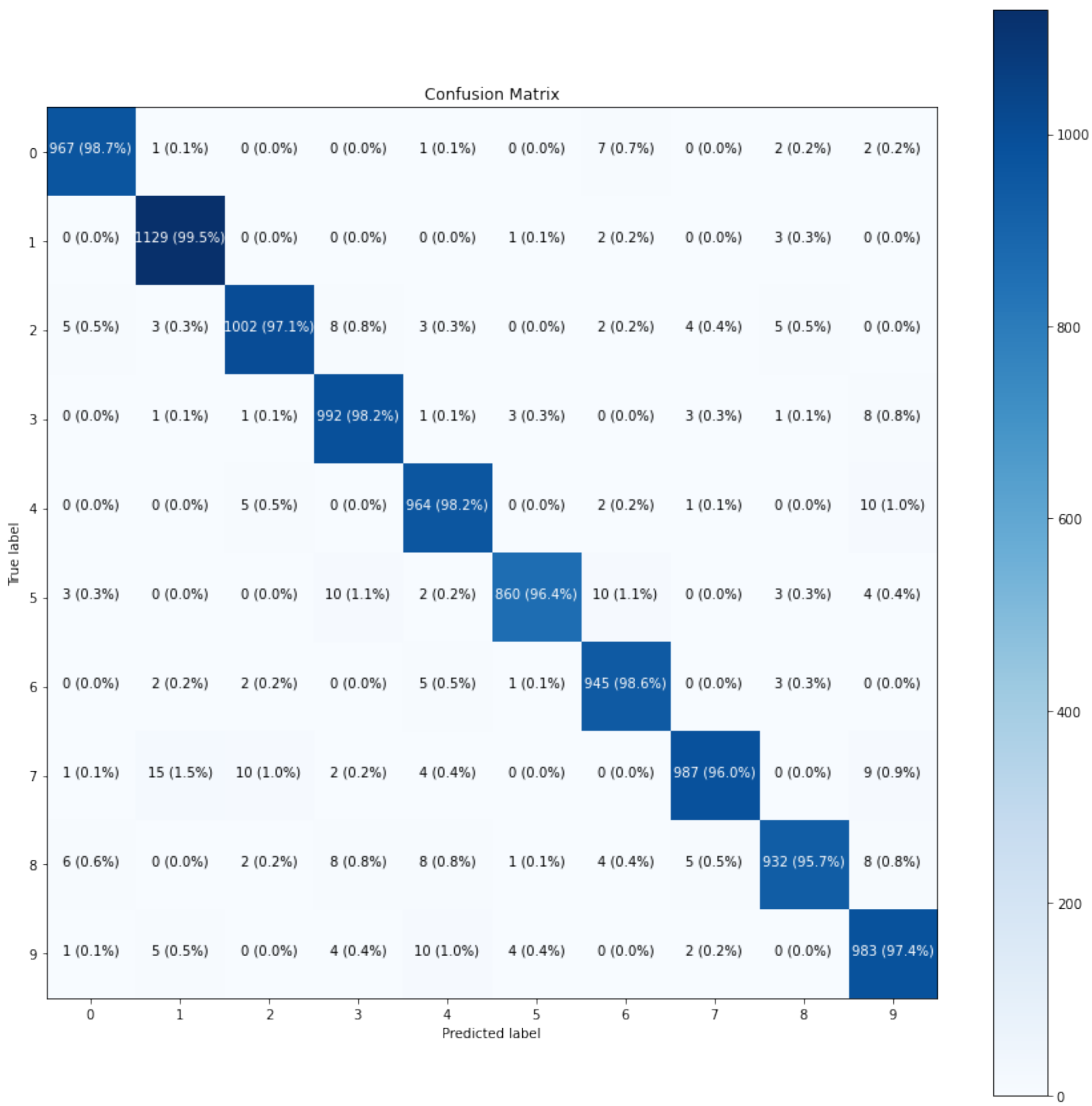
[395]:
```
# Check out the non-prettified confusion matrix
from sklearn.metrics import confusion_matrix
confusion_matrix(y_true=y_test,
                 y_pred=y_preds)
```

[395]:
```
array([[ 967,    1,    0,    0,    1,    0,    7,    0,    2,    2],
       [   0, 1129,    0,    0,    0,    1,    2,    0,    3,    0],
       [   5,    3, 1002,    8,    3,    0,    2,    4,    5,    0],
       [   0,    1,    1,  992,    1,    3,    0,    3,    1,    8],
       [   0,    0,    5,    0,  964,    0,    2,    1,    0,   10],
```

```
[   3,    0,    0,   10,    2,  860,   10,    0,    3,    4],
[   0,    2,    2,    0,    5,    1,  945,    0,    3,    0],
[   1,   15,   10,    2,    4,    0,    0,  987,    0,    9],
[   6,    0,    2,    8,    8,    1,    4,    5,  932,    8],
[   1,    5,    0,    4,   10,    4,    0,    2,    0,  983]])
```

[396]:
```python
class_names = ['0', '1', '2', '3', '4',
               '5', '6', '7', '8', '9']
```

[397]:
```python
# Make a prettier confusion matrix
make_confusion_matrix(y_true=y_test,
                      y_pred=y_preds,
                      classes=class_names,
                      figsize=(15, 15),
                      text_size=10)
```



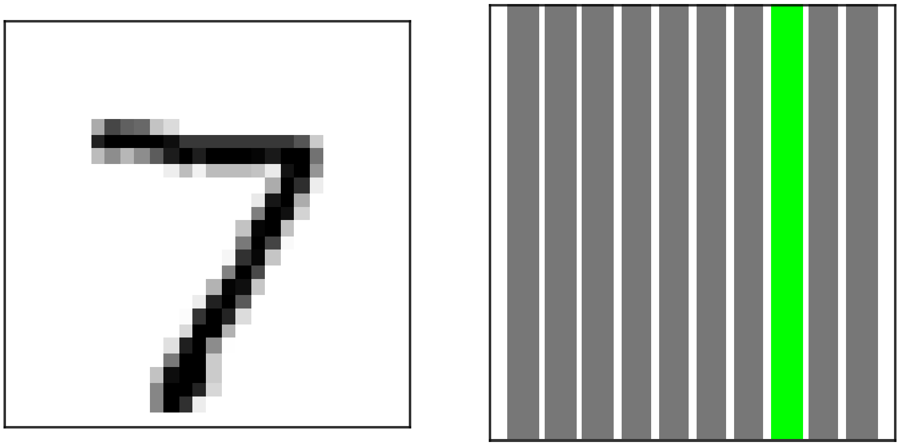
[479]:
```python
def plot_image(i, predictions_array, true_label, img):
  true_label, img = true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = predictions_array
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  #plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                #100*np.max(predictions_array),
                                #class_names[true_label]),
                                #color=color)

def plot_value_array(i, predictions_array, true_label):
  true_label = true_label[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])
  thisplot = plt.bar(range(10),predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = predictions_array

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('lime')
```

Verify predictions

With the model trained, you can use it to make predictions about some images.

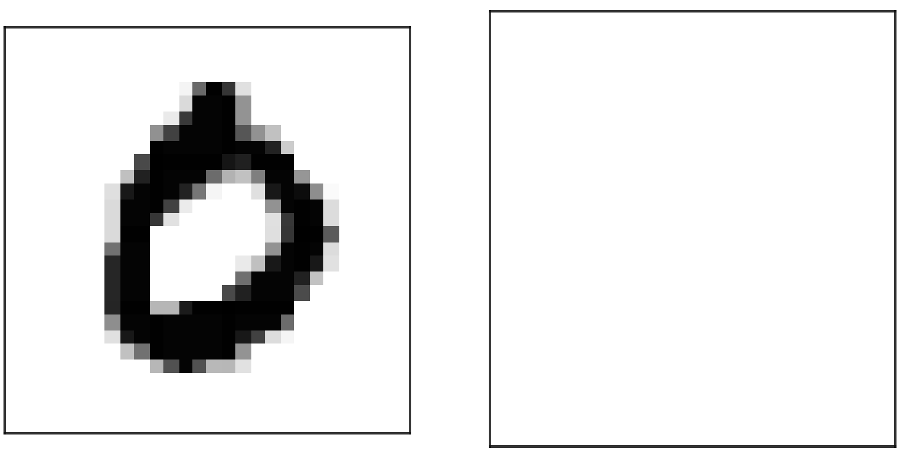
Let's look at the 0th image, predictions, and prediction array. Correct prediction labels are blue and incorrect prediction labels are red. The number gives the percentage (out of 100) for the predicted label.

[480]:
```python
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, y_preds[i], y_test, X_test)
plt.subplot(1,2,2)
plot_value_array(i, y_preds[i],  y_test)
plt.show()
```



[481]:

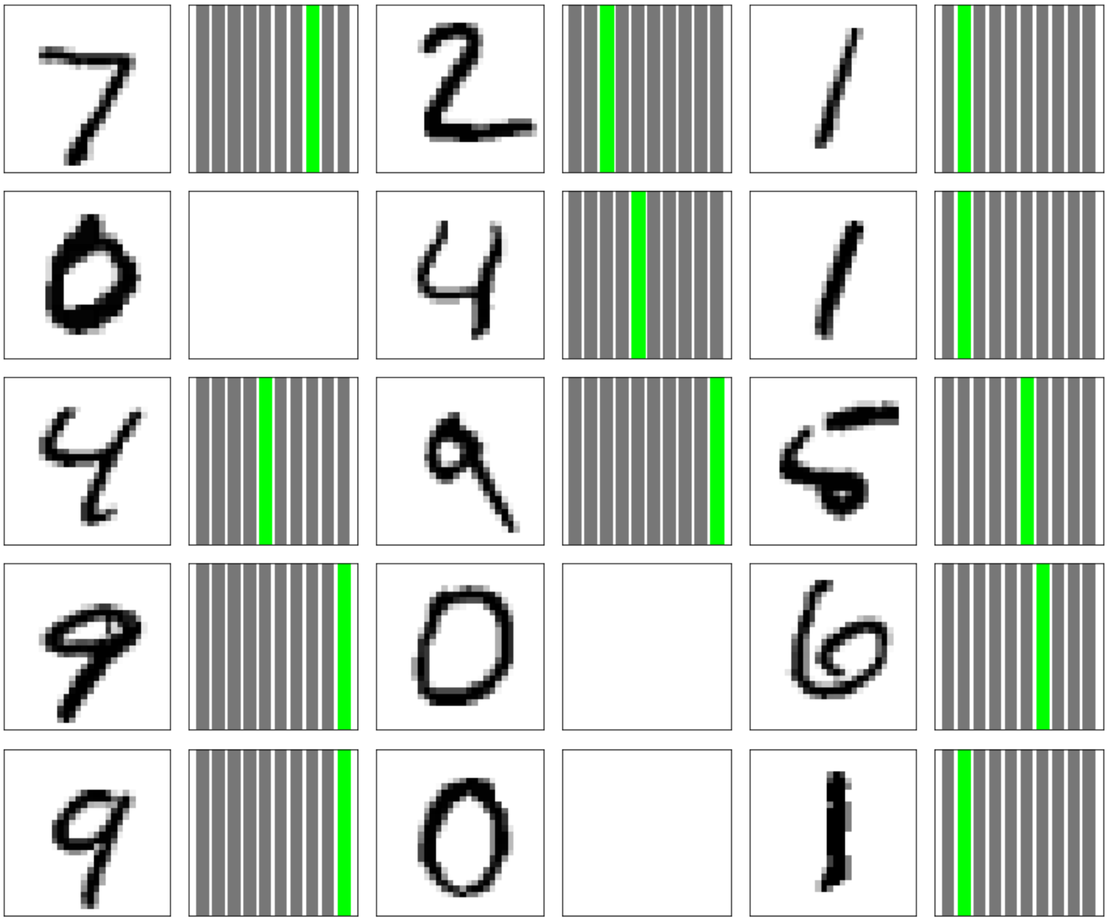
```
i = 3
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, y_preds[i], y_test, X_test)
plt.subplot(1,2,2)
plot_value_array(i, y_preds[i],  y_test)
plt.show()
```



[482]:

```
# Plot the first X test images, their predicted labels, and the true labels.
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
```

```
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, y_preds[i], y_test, X_test)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, y_preds[i], y_test)
plt.tight_layout()
plt.show()
```



## 1.1 AIR QUALITY PREDICTION USING DEEP LEARNING

[478]:

```
#Importing useful libraries
import pandas as pd
#import numpy as np
import sklearn
from sklearn import linear_model
from sklearn.utils import shuffle
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn import tree
airdata=pd.read_csv('station_day.csv')


New_data=airdata.iloc[:,[2,3,4,5,6,7,8,9,10,11,12,13,14,15]]
New_data=New_data.dropna()
New_data.head()
```

[478]:

```
   PM2.5    PM10    NO    NO2    NOx    NH3    CO    SO2      O3  Benzene  \
1  81.40  124.50  1.44  20.50  12.08  10.72  0.12  15.24  127.09     0.20
2  78.32  129.06  1.26  26.00  14.85  10.28  0.14  26.96  117.44     0.22
3  88.76  135.32  6.60  30.85  21.77  12.91  0.11  33.59  111.81     0.29
4  64.18  104.09  2.56  28.07  17.01  11.42  0.09  19.00  138.18     0.17
5  72.47  114.84  5.23  23.20  16.59  12.25  0.16  10.55  109.74     0.21

   Toluene  Xylene    AQI AQI_Bucket
1     6.50    0.06  184.0   Moderate
2     7.95    0.08  197.0   Moderate
3     7.63    0.12  198.0   Moderate
4     5.02    0.07  188.0   Moderate
5     4.71    0.08  173.0   Moderate
```

[254]:

```
import numpy as np
X=New_data.iloc[:,[0,1,2,3,4,5,6,7,8,9,10,11]]

y=New_data['AQI_Bucket'].to_list()
for i in range(len(y)) :
    if(y[i]=='Very Poor'):
        y[i]=0
    elif (y[i]=='Poor'):
        y[i]=1
    elif (y[i]=='Moderate'):
        y[i]=2
    elif (y[i]=='Satisfactory'):
        y[i]=3
    elif (y[i]=='Good'):
        y[i]=4
    else:
        y[i]=5
y=np.array(y)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=42)
```

[255]:

```
#initializing CNN model
classifier_e25 = Sequential()
```

```
#add 1st hidden layer
classifier_e25.add(Dense(input_dim = 12, units = 12,␣
 ↪kernel_initializer='uniform', activation='relu'))
#add output layer
classifier_e25.add(Dense(units = 8, kernel_initializer='uniform',␣
 ↪activation='relu'))
classifier_e25.add(Dense(units = 6,  activation='softmax'))
#compile the neural network
classifier_e25.compile(optimizer='adam',␣
 ↪loss='sparse_categorical_crossentropy', metrics=['accuracy'])
#model summary
classifier_e25.summary()
```

```
Model: "sequential_42"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_100 (Dense)            (None, 12)                156
_________________________________________________________________
dense_101 (Dense)            (None, 8)                 104
_________________________________________________________________
dense_102 (Dense)            (None, 6)                 54
=================================================================
Total params: 314
Trainable params: 314
Non-trainable params: 0
_________________________________________________________________
```

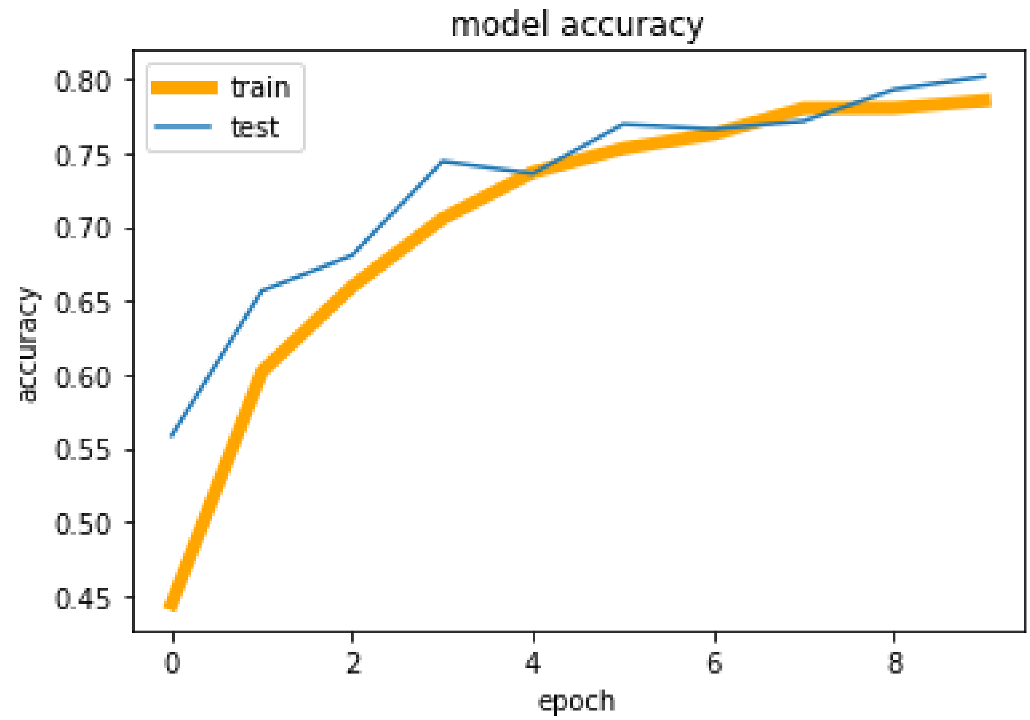
[256]:
```
model=classifier_e25.fit(X_train, y_train, validation_split=0.33, epochs=10,␣
 ↪batch_size=10)
```

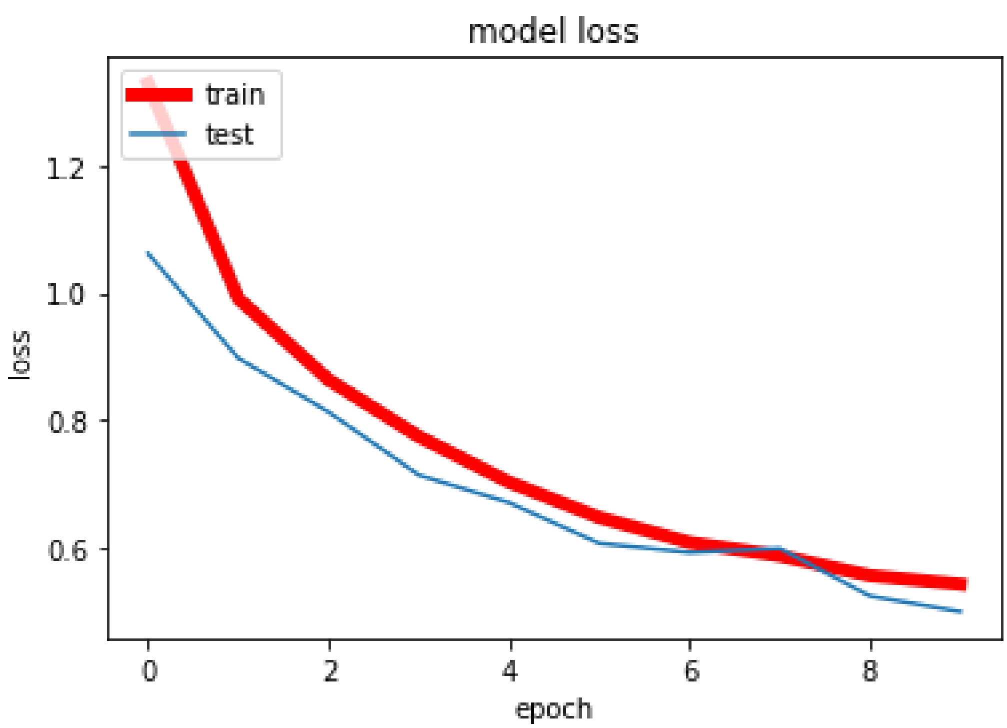
```
Epoch 1/10
484/484 [==============================] - 1s 2ms/step - loss: 1.5453 -
accuracy: 0.3567 - val_loss: 1.0616 - val_accuracy: 0.5585
Epoch 2/10
484/484 [==============================] - 1s 1ms/step - loss: 1.0431 -
accuracy: 0.5759 - val_loss: 0.8973 - val_accuracy: 0.6567
Epoch 3/10
484/484 [==============================] - 1s 1ms/step - loss: 0.8833 -
accuracy: 0.6446 - val_loss: 0.8126 - val_accuracy: 0.6807
Epoch 4/10
484/484 [==============================] - 1s 1ms/step - loss: 0.7989 -
accuracy: 0.6896 - val_loss: 0.7138 - val_accuracy: 0.7444
Epoch 5/10
484/484 [==============================] - 1s 1ms/step - loss: 0.7117 -
accuracy: 0.7435 - val_loss: 0.6705 - val_accuracy: 0.7360
Epoch 6/10
```

```
484/484 [==============================] - 1s 1ms/step - loss: 0.6706 -
accuracy: 0.7387 - val_loss: 0.6065 - val_accuracy: 0.7696
Epoch 7/10
484/484 [==============================] - 1s 1ms/step - loss: 0.6006 -
accuracy: 0.7677 - val_loss: 0.5921 - val_accuracy: 0.7663
Epoch 8/10
484/484 [==============================] - 1s 1ms/step - loss: 0.5982 -
accuracy: 0.7761 - val_loss: 0.5985 - val_accuracy: 0.7717
Epoch 9/10
484/484 [==============================] - 1s 1ms/step - loss: 0.5631 -
accuracy: 0.7776 - val_loss: 0.5236 - val_accuracy: 0.7931
Epoch 10/10
484/484 [==============================] - 1s 2ms/step - loss: 0.5470 -
accuracy: 0.7873 - val_loss: 0.4991 - val_accuracy: 0.8019
```

[257]:

```
loss=list(model.history.values())[0]
accuracy=list(model.history.values())[1]
val_loss=list(model.history.values())[2]
val_accuracy=list(model.history.values())[3]

# summarize history for accuracy
plt.plot(accuracy,color='orange', linewidth=5)
plt.plot(val_accuracy)
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(loss, color='red', linewidth=5)
plt.plot(val_loss)
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```


model accuracy
train
test
accuracy
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0
2
4
6
8
epoch


model loss
train
test
loss
1.2
1.0
0.8
0.6
0
2
4
6
8
epoch

## 1.2 Classification with Deep Learning

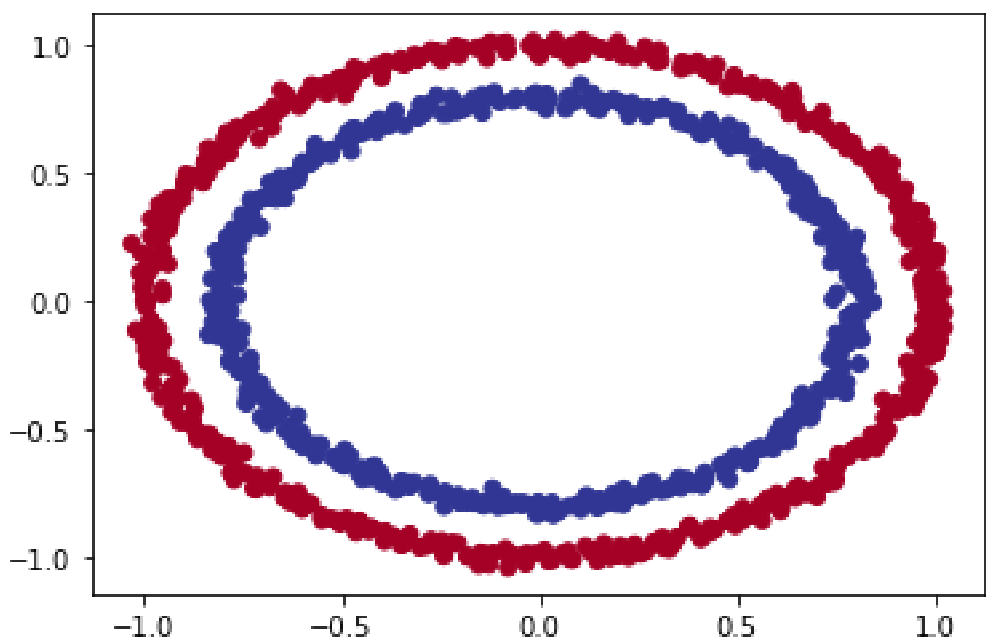
[301]:
```
from sklearn.datasets import make_circles

# Make 1000 examples
n_samples = 1000

# Create circles
X, y = make_circles(n_samples,
                    noise=0.02,
                    random_state=42)
```

[302]:
```
# Make dataframe of features and labels
import pandas as pd
circles = pd.DataFrame({"X0":X[:, 0], "X1":X[:, 1], "label":y})
circles.head()
```

[302]:

| | X0 | X1 | label |
|---|---|---|---|
| 0 | 0.760266 | 0.223878 | 1 |
| 1 | -0.767222 | 0.145542 | 1 |
| 2 | -0.808159 | 0.148944 | 1 |
| 3 | -0.376028 | 0.703209 | 1 |
| 4 | 0.440510 | -0.897617 | 0 |

[303]:
```
# Visualize with a plot
import matplotlib.pyplot as plt
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdYlBu);
```



[304]:
```
import numpy as np

def plot_decision_boundary(model, X, y):
  """
  Plots the decision boundary created by a model predicting on X.
  This function has been adapted from two phenomenal resources:
   1. CS231n - https://cs231n.github.io/neural-networks-case-study/
   2. Made with ML basics - https://github.com/madewithml/basics/blob
   /master/notebooks/09_Multilayer_Perceptrons/09_TF_Multilayer_Perceptrons.
 ↪ipynb
  """
  # Define the axis boundaries of the plot and create a meshgrid
  x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
  y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
  xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                       np.linspace(y_min, y_max, 100))

  # Create X values (we're going to predict on all of these)
  x_in = np.c_[xx.ravel(), yy.ravel()] # stack 2D arrays together: https://
 ↪numpy.org
    #/devdocs/reference/generated/numpy.c_.html

  # Make predictions using the trained model
  y_pred = model.predict(x_in)

  # Check for multi-class
  if len(y_pred[0]) > 1:
    print("doing multiclass classification...")
    # We have to reshape our predictions to get them ready for plotting
    y_pred = np.argmax(y_pred, axis=1).reshape(xx.shape)
  else:
    print("doing binary classifcation...")
    y_pred = np.round(y_pred).reshape(xx.shape)

  # Plot decision boundary
  plt.contourf(xx, yy, y_pred, cmap=plt.cm.RdYlBu, alpha=0.7)
  plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.RdYlBu)
  plt.xlim(xx.min(), xx.max())
  plt.ylim(yy.min(), yy.max())
```

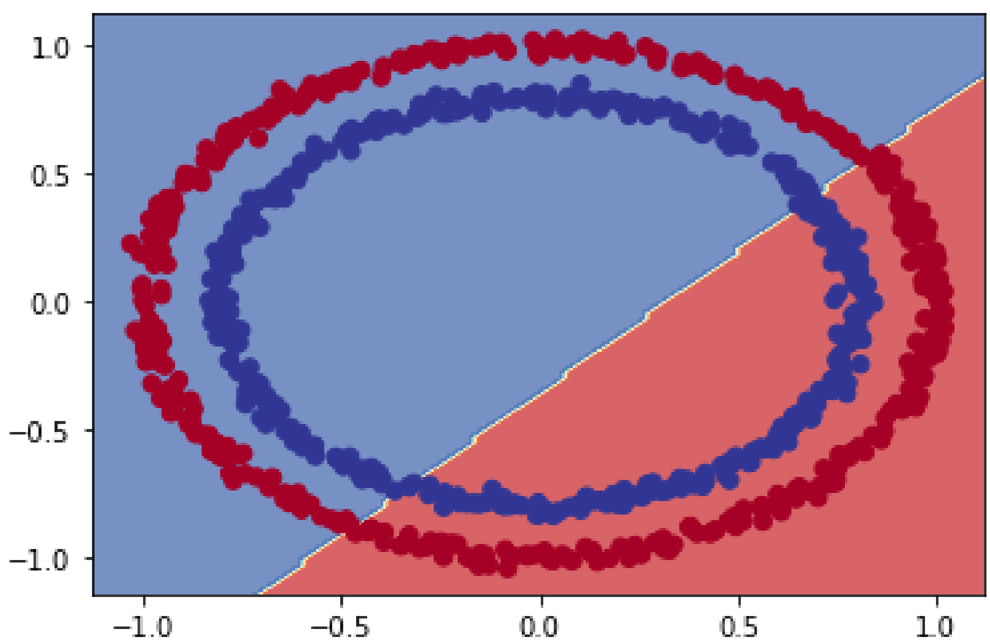
[305]:
```
X_train=X[:800]
y_train=y[:800]
X_test=X[800:]
y_test=y[800:]
```

## 1.3 Model 1

[306]:
```
tf.random.set_seed(42)

# Create a model
model_1 = tf.keras.Sequential([
  tf.keras.layers.Dense(100, activation=tf.keras.activations.linear), # hidden␣
↪layer 1, ReLU activation
  tf.keras.layers.Dense(10, activation=tf.keras.activations.linear), # hidden␣
↪layer 2, ReLU activation
  tf.keras.layers.Dense(1, activation=tf.keras.activations.linear) # ouput␣
↪layer, sigmoid activation
])

# Compile the model
model_1.compile(loss=tf.keras.losses.binary_crossentropy,
                optimizer=tf.keras.optimizers.Adam(),
                metrics=['accuracy'])

# Fit the model
history = model_1.fit(X, y, epochs=100, verbose=0)
```

[307]:
```
plot_decision_boundary(model_1, X_train, y_train)
```

```
doing binary classifcation…
```



[308]:
```
model_1.evaluate(X_test, y_test)
```

```
WARNING:tensorflow:5 out of the last 33 calls to <function
Model.make_test_function.<locals>.test_function at 0x162ff9dc0> triggered
tf.function retracing. Tracing is expensive and the excessive number of tracings
could be due to (1) creating @tf.function repeatedly in a loop, (2) passing
tensors with different shapes, (3) passing Python objects instead of tensors.
For (1), please define your @tf.function outside of the loop. For (2),
@tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
7/7 [==============================] - 0s 1ms/step - loss: 0.6899 - accuracy:
0.5750
```

[308]:
```
[0.689868152141571, 0.574999988079071]
```

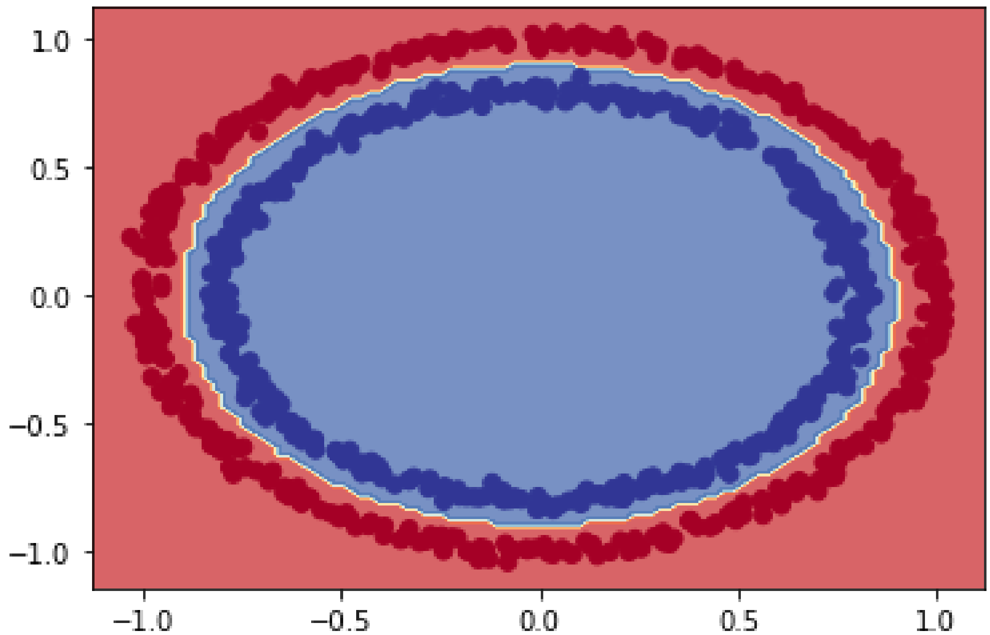
Our model 1 has a linear decision boundary and thus does not perform well in classifying our data. Now lets resort to non-linear activation functions.

### 1.4 Model 2

[309]:
```
tf.random.set_seed(42)

# Create a model
model_2 = tf.keras.Sequential([
  tf.keras.layers.Dense(100, activation=tf.keras.activations.relu), # hidden␣
 ↪layer 1, ReLU activation
  tf.keras.layers.Dense(10, activation=tf.keras.activations.relu), # hidden␣
 ↪layer 2, ReLU activation
  tf.keras.layers.Dense(1, activation=tf.keras.activations.sigmoid) # ouput␣
 ↪layer, sigmoid activation
])

# Compile the model
model_2.compile(loss=tf.keras.losses.binary_crossentropy,
                optimizer=tf.keras.optimizers.Adam(),
                metrics=['accuracy'])

# Fit the model
history = model_2.fit(X, y, epochs=100, verbose=0)
```

[310]:
```
plot_decision_boundary(model_2, X_train, y_train)
```

```
doing binary classifcation…
```




[311]:
```
model_2.evaluate(X_train, y_train)
```

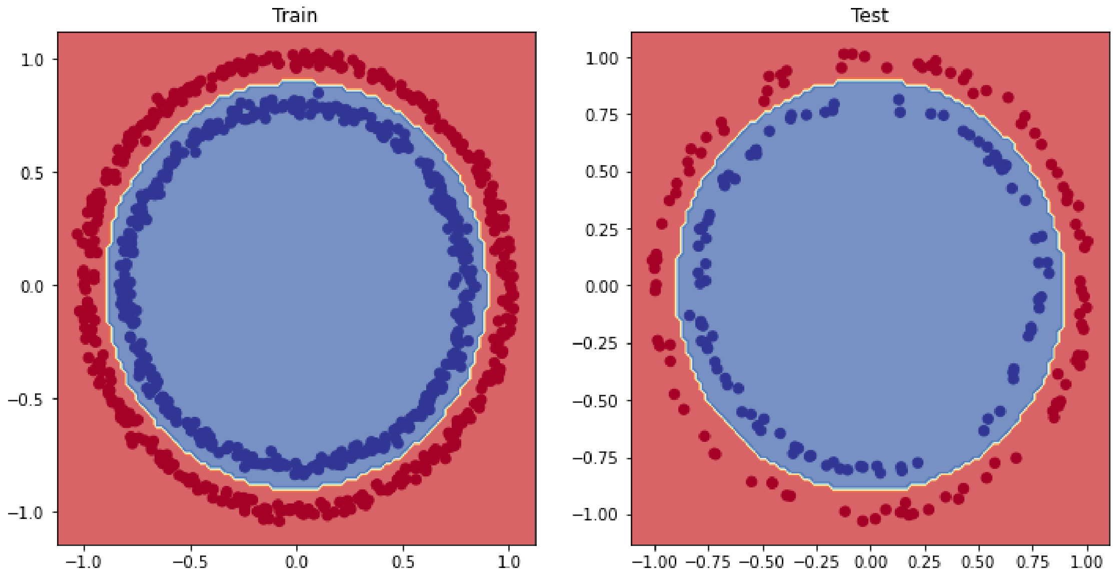
```
25/25 [==============================] - 0s 1ms/step - loss: 0.0016 - accuracy:
1.0000
```

[311]:
```
[0.001596104702912271, 1.0]
```

Performs well with an accuracy of 99%

[312]:
```
# Plot the decision boundaries for the training and test sets
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Train")
plot_decision_boundary(model_2, X=X_train, y=y_train)
plt.subplot(1, 2, 2)
plt.title("Test")
plot_decision_boundary(model_2, X=X_test, y=y_test)
plt.show()
```

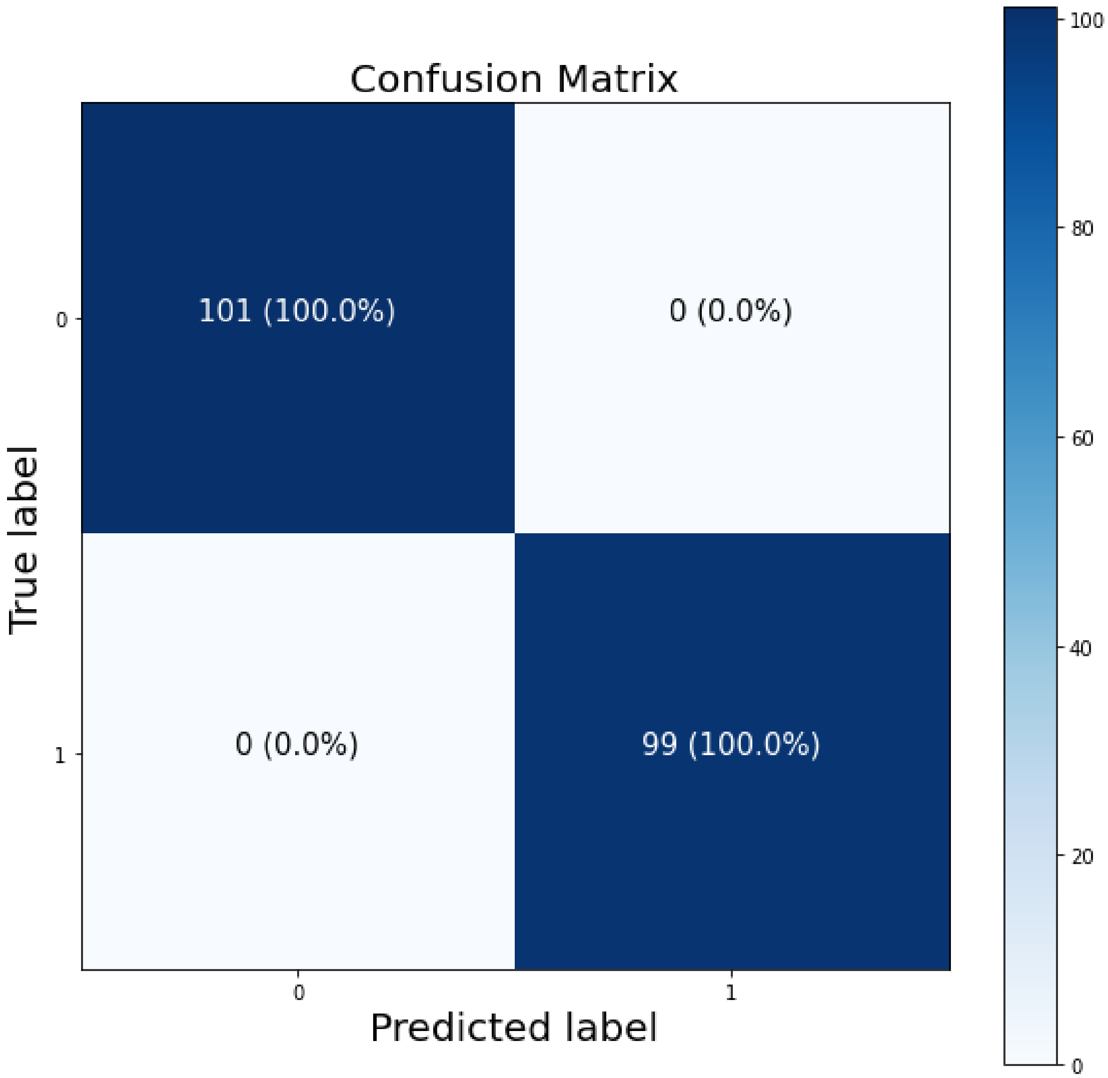
```
doing binary classifcation…
doing binary classifcation…
```




[314]:
```
y_preds=model_2.predict(X_test)
```

[319]:
```
# Note: The following confusion matrix code is a remix of Scikit-Learn's
# plot_confusion_matrix function - https://scikit-learn.org/stable/modules/
#generated/sklearn.metrics.plot_confusion_matrix.html
# and Made with ML's introductory notebook - https://github.com/madewithml/
#basics/blob/master/notebooks/09_Multilayer_Perceptrons/
↪09_TF_Multilayer_Perceptrons.ipynb
import itertools

figsize = (10, 10)

# Create the confusion matrix
cm = confusion_matrix(y_test, tf.round(y_preds))
cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis] # normalize it
n_classes = cm.shape[0]

# Let's prettify it
fig, ax = plt.subplots(figsize=figsize)
# Create a matrix plot
cax = ax.matshow(cm, cmap=plt.cm.Blues) # https://matplotlib.org/3.2.0/api/
↪_as_gen/matplotlib.axes.Axes.matshow.html
fig.colorbar(cax)

# Create classes
classes = False

if classes:
```

```
  labels = classes
else:
  labels = np.arange(cm.shape[0])

# Label the axes
ax.set(title="Confusion Matrix",
       xlabel="Predicted label",
       ylabel="True label",
       xticks=np.arange(n_classes),
       yticks=np.arange(n_classes),
       xticklabels=labels,
       yticklabels=labels)

# Set x-axis labels to bottom
ax.xaxis.set_label_position("bottom")
ax.xaxis.tick_bottom()

# Adjust label size
ax.xaxis.label.set_size(20)
ax.yaxis.label.set_size(20)
ax.title.set_size(20)

# Set threshold for different colors
threshold = (cm.max() + cm.min()) / 2.

# Plot the text on each cell
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
  plt.text(j, i, f"{cm[i, j]} ({cm_norm[i, j]*100:.1f}%)",
           horizontalalignment="center",
           color="white" if cm[i, j] > threshold else "black",
           size=15)
```




### 1.5 Another Example

[271]:
```
import pandas as pd
import seaborn as sns
data=pd.read_csv("cwbdata.csv",header=None)
data.columns =['X1', 'X2', 'label']
data.head()
```

[271]:

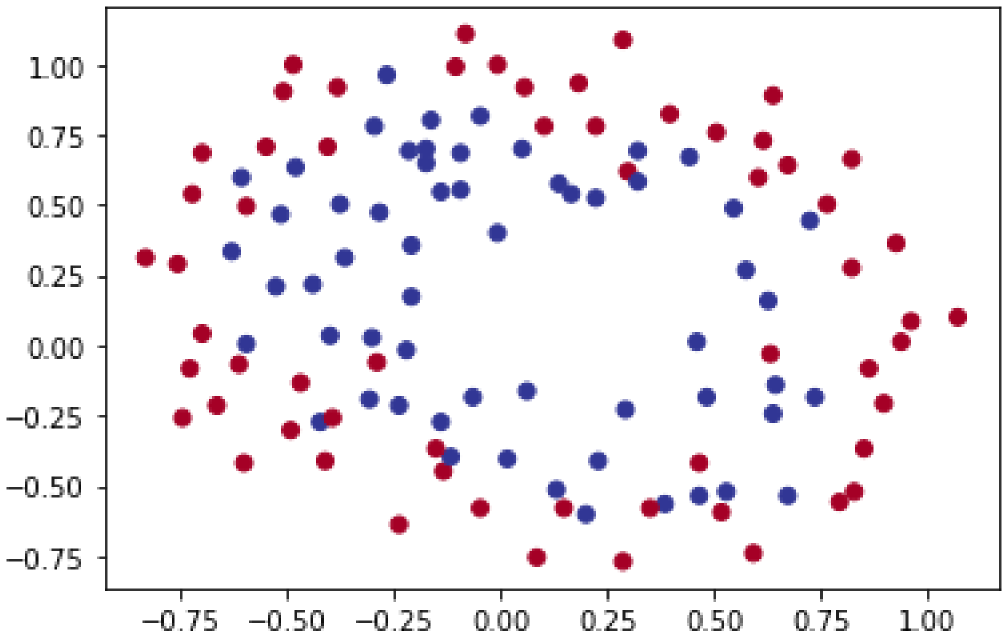
| | X1 | X2 | label |
|---|---|---|---|
| 0 | 0.051267 | 0.69956 | 1 |
| 1 | -0.092742 | 0.68494 | 1 |
| 2 | -0.213710 | 0.69225 | 1 |

```
3 -0.375000  0.50219      1
4 -0.513250  0.46564      1
```

[272]:
```
import random
f1=data['X1'].to_list()
#Reshuffling data

f2=data['X2'].to_list()
label=data['label'].to_list()

X=[[f1[i],f2[i]] for i in range(len(f1))]
y=label
```

[273]:
```
indices=[ 49,  81, 107,  25,  51,  12, 117,  13,  43,  37,  50,  60,  33,
       102,  42,  88,  99,   8,  80,  73,  97,  23, 110,  72,  10,  82,
        24,  40,  98,   1,  71,   5,  78,  84,  75,   3,   4,  46,  21,
        63,  93,  92,  68,  45, 105,  28,  31,  89,  14,  91,  20,  16,
        38,  67, 114,  17,  53,  83,  30,  56, 100,  52,  47,  57, 109,
        61,  41,  26, 111,  94,  35,   7,  95,  69,  87,  15,  11,  86,
       104,  66, 112,   9,  18,   6,  19,  48, 101,  34,  85, 103, 113,
        90,  62,  59,  76,  96,  44,  32,  77,   2, 115,   0,  27, 106,
        22, 116,  39,  36,  64,  74,  79,  54, 108,  55,  65,  70,  29,
        58]
X=[X[i] for i in indices]
y=[y[i] for i in indices]
X=np.array(X)
y=np.array(y)
```

[274]:
```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdYlBu);
```




[275]:
```
X_train=X[:90]
y_train=y[:90]
X_test=X[90:]
y_test=y[90:]
```

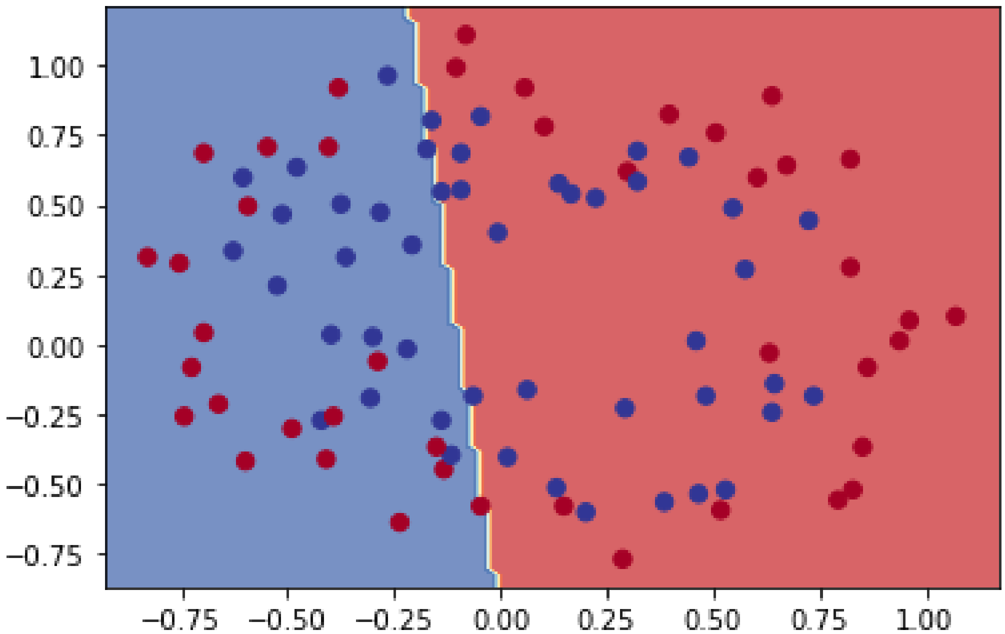
## model 3

[276]:
```
tf.random.set_seed(42)

# Create a model
model_3 = tf.keras.Sequential([
  tf.keras.layers.Dense(100, activation=tf.keras.activations.linear), # hidden␣
 ↪layer 1, ReLU activation
  tf.keras.layers.Dense(10, activation=tf.keras.activations.linear), # hidden␣
 ↪layer 2, ReLU activation
  tf.keras.layers.Dense(1, activation=tf.keras.activations.linear) # ouput␣
 ↪layer, sigmoid activation
])

# Compile the model
model_3.compile(loss=tf.keras.losses.binary_crossentropy,
                optimizer=tf.keras.optimizers.Adam(),
                metrics=['accuracy'])

# Fit the model
history = model_3.fit(X, y, epochs=100, verbose=0)
```

[277]:

```
plot_decision_boundary(model_3, X_train, y_train)
```

```
doing binary classifcation…
```




[278]:

```
model_3.evaluate(X_test, y_test)
```

```
1/1 [==============================] - 0s 125ms/step - loss: 0.6746 - accuracy:
0.6786
```

[278]: [0.6746087074279785, 0.6785714030265808]

### 1.6 Model 4

[287]:

```
tf.random.set_seed(42)

# Create a model
model_4 = tf.keras.Sequential([
  tf.keras.layers.Dense(100, activation=tf.keras.activations.relu),# hidden␣
 ↪layer 1, ReLU activation
  tf.keras.layers.Dense(80, activation=tf.keras.activations.relu),
  tf.keras.layers.Dense(10, activation=tf.keras.activations.relu), # hidden␣
 ↪layer 2, ReLU activation
  tf.keras.layers.Dense(1, activation=tf.keras.activations.sigmoid) # ouput␣
 ↪layer, sigmoid activation
])
```

```python
# Compile the model
model_4.compile(loss=tf.keras.losses.binary_crossentropy,
                optimizer=tf.keras.optimizers.Adam(lr=0.02),
                metrics=['accuracy'])

# Fit the model
history = model_4.fit(X, y, epochs=50,verbose=1)
```

```
Epoch 1/50
4/4 [==============================] - 0s 2ms/step - loss: 0.6899 - accuracy:
0.5466
Epoch 2/50
4/4 [==============================] - 0s 2ms/step - loss: 0.6612 - accuracy:
0.5406
Epoch 3/50
4/4 [==============================] - 0s 2ms/step - loss: 0.6480 - accuracy:
0.6529
Epoch 4/50
4/4 [==============================] - 0s 2ms/step - loss: 0.5962 - accuracy:
0.6550
Epoch 5/50
4/4 [==============================] - 0s 4ms/step - loss: 0.5841 - accuracy:
0.6936
Epoch 6/50
4/4 [==============================] - 0s 2ms/step - loss: 0.5421 - accuracy:
0.7520
Epoch 7/50
4/4 [==============================] - 0s 2ms/step - loss: 0.4986 - accuracy:
0.7918
Epoch 8/50
4/4 [==============================] - 0s 2ms/step - loss: 0.5532 - accuracy:
0.7798
Epoch 9/50
4/4 [==============================] - 0s 3ms/step - loss: 0.4826 - accuracy:
0.7993
Epoch 10/50
4/4 [==============================] - 0s 2ms/step - loss: 0.4865 - accuracy:
0.7973
Epoch 11/50
4/4 [==============================] - 0s 2ms/step - loss: 0.4869 - accuracy:
0.7803
Epoch 12/50
4/4 [==============================] - 0s 2ms/step - loss: 0.4543 - accuracy:
0.7952
Epoch 13/50
4/4 [==============================] - 0s 2ms/step - loss: 0.5242 - accuracy:
```

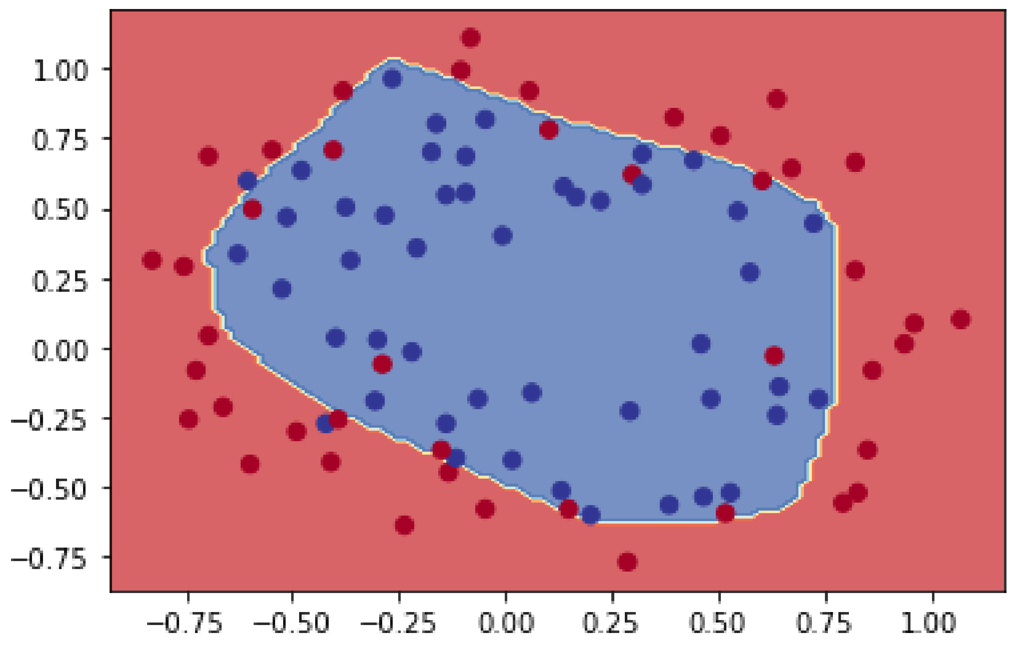
```
0.7598
Epoch 14/50
4/4 [==============================] - 0s 2ms/step - loss: 0.5122 - accuracy:
0.7835
Epoch 15/50
4/4 [==============================] - 0s 2ms/step - loss: 0.5038 - accuracy:
0.7775
Epoch 16/50
4/4 [==============================] - 0s 2ms/step - loss: 0.4516 - accuracy:
0.8017
Epoch 17/50
4/4 [==============================] - 0s 2ms/step - loss: 0.4405 - accuracy:
0.8322
Epoch 18/50
4/4 [==============================] - 0s 2ms/step - loss: 0.4388 - accuracy:
0.8176
Epoch 19/50
4/4 [==============================] - 0s 2ms/step - loss: 0.4205 - accuracy:
0.7989
Epoch 20/50
4/4 [==============================] - 0s 6ms/step - loss: 0.3790 - accuracy:
0.8679
Epoch 21/50
4/4 [==============================] - 0s 2ms/step - loss: 0.4113 - accuracy:
0.8403
Epoch 22/50
4/4 [==============================] - 0s 3ms/step - loss: 0.3577 - accuracy:
0.8489
Epoch 23/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3575 - accuracy:
0.8874
Epoch 24/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3465 - accuracy:
0.8835
Epoch 25/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3982 - accuracy:
0.8390
Epoch 26/50
4/4 [==============================] - 0s 3ms/step - loss: 0.4133 - accuracy:
0.8014
Epoch 27/50
4/4 [==============================] - 0s 3ms/step - loss: 0.4718 - accuracy:
0.7978
Epoch 28/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3569 - accuracy:
0.8710
Epoch 29/50
4/4 [==============================] - 0s 2ms/step - loss: 0.4283 - accuracy:
```

```
0.8033
Epoch 30/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3638 - accuracy:
0.8442
Epoch 31/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3813 - accuracy:
0.8127
Epoch 32/50
4/4 [==============================] - 0s 3ms/step - loss: 0.3550 - accuracy:
0.8853
Epoch 33/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3556 - accuracy:
0.8523
Epoch 34/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3903 - accuracy:
0.8132
Epoch 35/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3236 - accuracy:
0.8609
Epoch 36/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3286 - accuracy:
0.8708
Epoch 37/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3257 - accuracy:
0.8695
Epoch 38/50
4/4 [==============================] - 0s 4ms/step - loss: 0.3087 - accuracy:
0.8570
Epoch 39/50
4/4 [==============================] - 0s 2ms/step - loss: 0.2866 - accuracy:
0.9085
Epoch 40/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3562 - accuracy:
0.8531
Epoch 41/50
4/4 [==============================] - 0s 2ms/step - loss: 0.2745 - accuracy:
0.8906
Epoch 42/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3279 - accuracy:
0.8439
Epoch 43/50
4/4 [==============================] - 0s 2ms/step - loss: 0.2633 - accuracy:
0.8793
Epoch 44/50
4/4 [==============================] - 0s 4ms/step - loss: 0.2756 - accuracy:
0.8992
Epoch 45/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3498 - accuracy:
```

```
0.8398
Epoch 46/50
4/4 [==============================] - 0s 3ms/step - loss: 0.3521 - accuracy:
0.8317
Epoch 47/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3785 - accuracy:
0.8335
Epoch 48/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3913 - accuracy:
0.8041
Epoch 49/50
4/4 [==============================] - 0s 2ms/step - loss: 0.3409 - accuracy:
0.8728
Epoch 50/50
4/4 [==============================] - 0s 2ms/step - loss: 0.4052 - accuracy:
0.8124
```

[288]:
```
plot_decision_boundary(model_4, X_train, y_train)
```

```
doing binary classifcation…
```



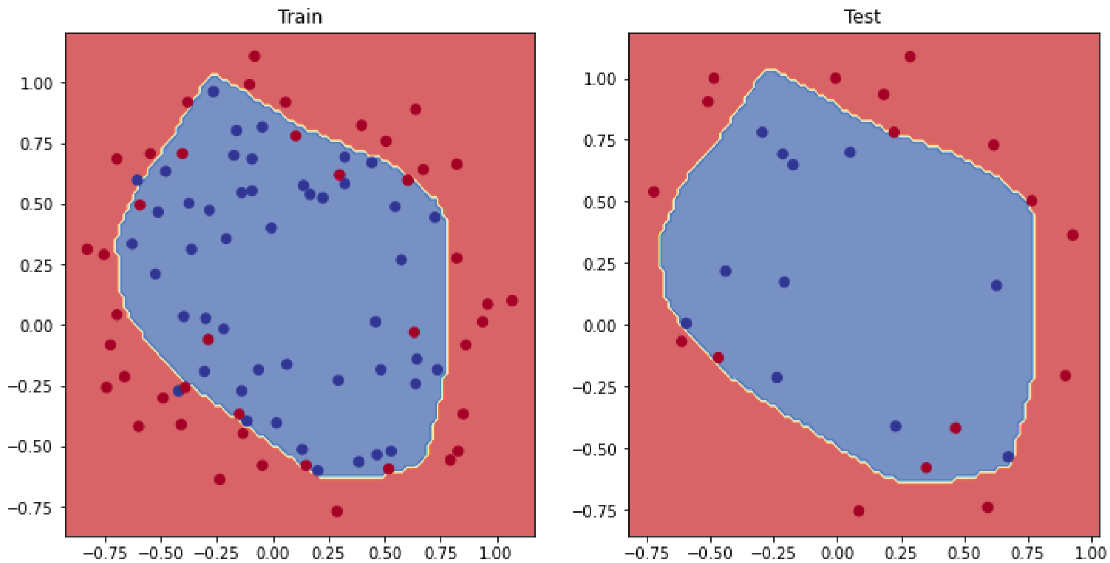

[292]:
```
model_4.evaluate(X_test, y_test)
```

```
1/1 [==============================] - 0s 17ms/step - loss: 0.3195 - accuracy:
0.8929
```

[292]: [0.3194580078125, 0.8928571343421936]

[293]:
```
# Plot the decision boundaries for the training and test sets
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Train")
plot_decision_boundary(model_4, X=X_train, y=y_train)
plt.subplot(1, 2, 2)
plt.title("Test")
plot_decision_boundary(model_4, X=X_test, y=y_test)
plt.show()
```

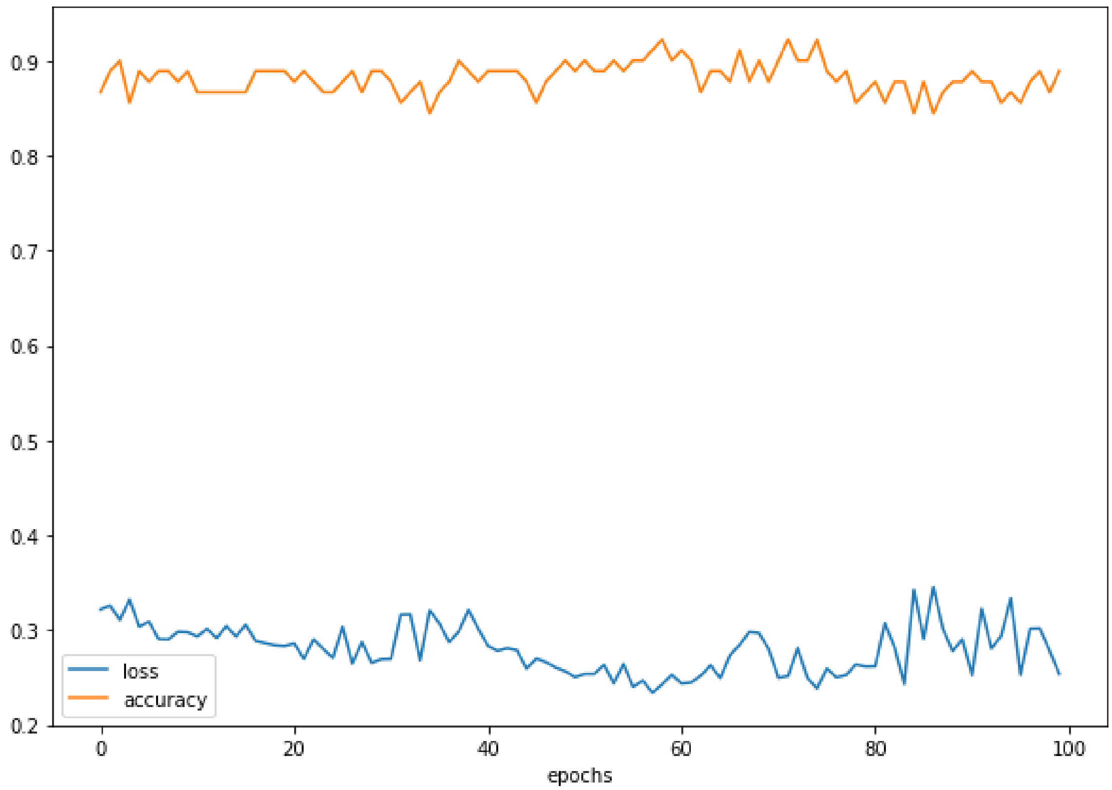
doing binary classifcation…
doing binary classifcation…




[294]:
```
history = model_4.fit(X_train, y_train, epochs=100,verbose=0)
```

[298]:
```
y_preds = model_4.predict(X_test)
```

[295]:
```
# Checkout the history
pd.DataFrame(history.history).plot(figsize=(10,7), xlabel="epochs");
```

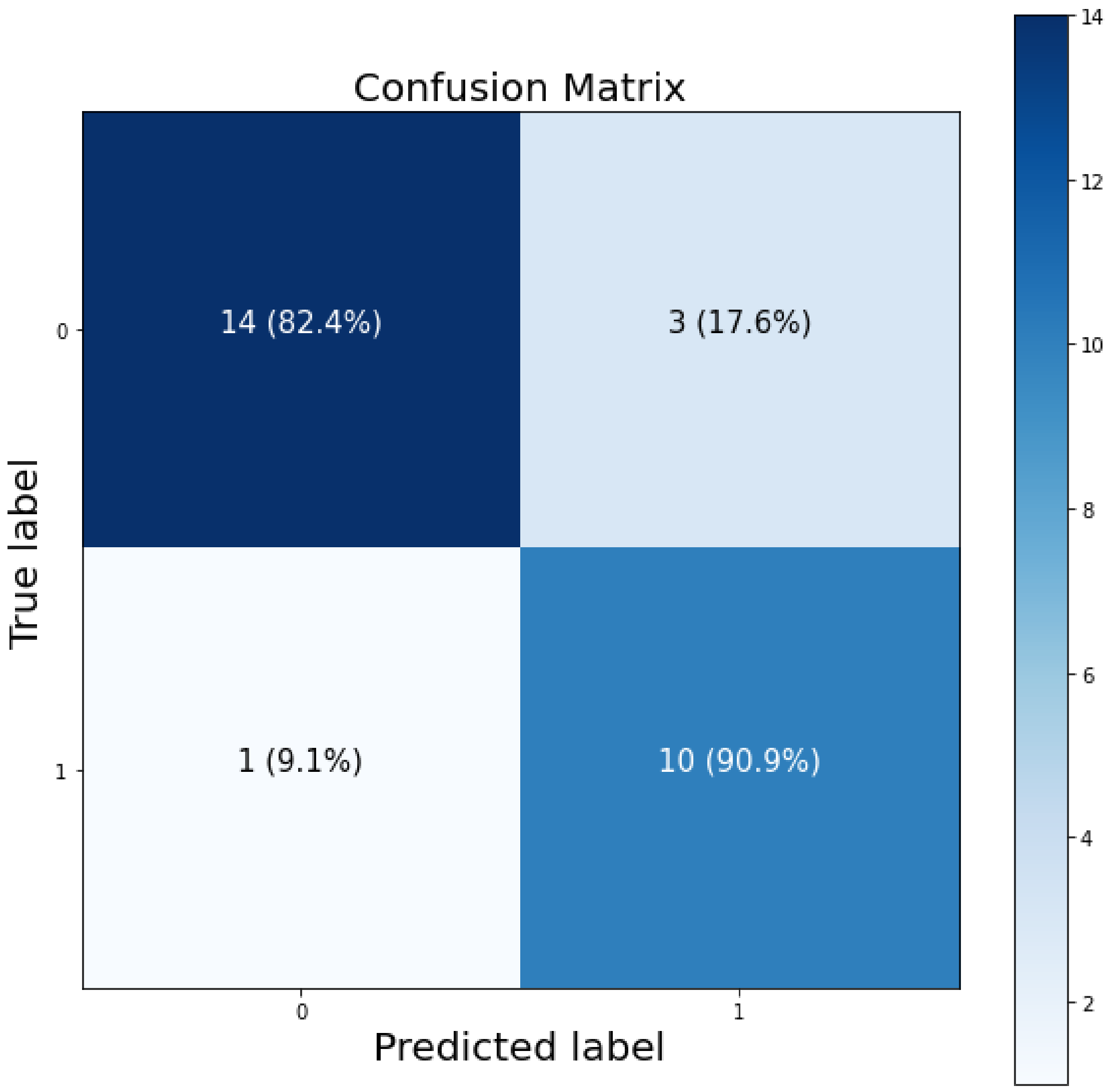



[299]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, tf.round(y_preds))
```

[299]:
```
array([[14,  3],
       [ 1, 10]])
```

[300]:
```
# Note: The following confusion matrix code is a remix of Scikit-Learn's
# plot_confusion_matrix function - https://scikit-learn.org/stable/modules/
#generated/sklearn.metrics.plot_confusion_matrix.html
# and Made with ML's introductory notebook - https://github.com/madewithml/
#basics/blob/master/notebooks/09_Multilayer_Perceptrons/
↪09_TF_Multilayer_Perceptrons.ipynb
import itertools

figsize = (10, 10)

# Create the confusion matrix
cm = confusion_matrix(y_test, tf.round(y_preds))
cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis] # normalize it
n_classes = cm.shape[0]

# Let's prettify it
```

```
fig, ax = plt.subplots(figsize=figsize)
# Create a matrix plot
cax = ax.matshow(cm, cmap=plt.cm.Blues) # https://matplotlib.org/3.2.0/api/
↪_as_gen/matplotlib.axes.Axes.matshow.html
fig.colorbar(cax)

# Create classes
classes = False

if classes:
  labels = classes
else:
  labels = np.arange(cm.shape[0])

# Label the axes
ax.set(title="Confusion Matrix",
       xlabel="Predicted label",
       ylabel="True label",
       xticks=np.arange(n_classes),
       yticks=np.arange(n_classes),
       xticklabels=labels,
       yticklabels=labels)

# Set x-axis labels to bottom
ax.xaxis.set_label_position("bottom")
ax.xaxis.tick_bottom()

# Adjust label size
ax.xaxis.label.set_size(20)
ax.yaxis.label.set_size(20)
ax.title.set_size(20)

# Set threshold for different colors
threshold = (cm.max() + cm.min()) / 2.

# Plot the text on each cell
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
  plt.text(j, i, f"{cm[i, j]} ({cm_norm[i, j]*100:.1f}%)",
           horizontalalignment="center",
           color="white" if cm[i, j] > threshold else "black",
           size=15)
```


Confusion Matrix
14 (82.4%)
3 (17.6%)
1 (9.1%)
10 (90.9%)
True label
Predicted label
0
1
0
1
14
12
10
8
6
4
2

```
[ ]:
```