Deep Learning Project in Python with Keras in Multiclass Classification Dataset

Keras is a powerful and easy-to-use free open source Python library for developing and evaluating deep learning models.

It is part of the TensorFlow library and allows you to define and train neural network models in just a few lines of code.

It is easy-to-use Neural Network Library written in Python that runs at top of Theano or Tensorflow. Tensorflow provides low-level as well as high-level API, indeed Keras only provide High-level API.

Dataset Description

This is a multi-class classification problem, meaning that there are more than two classes to be predicted. In fact, there are seven classes in the outcome column. The datasets consists of several medical predictor variables and one target variable (Classification Final). Predictor variables includes age, sex, patient type, different pregnancy-related situations, patients with various conditions, and more.

We can summarize the construction of deep learning models in Keras as follows:

- Define your model: Create a sequence and add layers.
  - Compile your model: Specify loss functions and optimizers.
  - Fit your model: Execute the model using data.
  - Evaluate your model: Evaluate the model on training dataset
  - Make predictions. Use the model to generate predictions on new data.

1. Import Necessary Libraries

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
```

2. Load Data

In [2]:
```
data = pd.read_csv('Covid_Dataset.csv')
data.head()
```

Out[2]:

	USMER	MEDICAL_UNIT	SEX	PATIENT_TYPE	DATE_DIED	INTUBED	PNEUMONIA	AGE	PREGNANT	DIABETES
0	2	1	1	1	03-05-2020	3	1	65	2	0
1	2	1	2	1	03-06-2020	3	1	72	3	0
2	2	1	2	2	09-06-2020	1	2	55	3	1
3	2	1	1	1	12-06-2020	3	2	53	2	0
4	2	1	2	1	21-06-2020	3	2	68	3	1

5 rows × 21 columns

In [3]:

```
# Total number of rows and columns
data.shape
```

Out[3]:

```
(199999, 21)
```

In [4]:

```
# Getting information of the dataframe
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 199999 entries, 0 to 199998
Data columns (total 21 columns):
 #   Column                Non-Null Count   Dtype 
---  ------                --------------   ----- 
 0   USMER                 199999 non-null  int64 
 1   MEDICAL_UNIT          199999 non-null  int64 
 2   SEX                   199999 non-null  int64 
 3   PATIENT_TYPE          199999 non-null  int64 
 4   DATE_DIED             199999 non-null  object
 5   INTUBED               199999 non-null  int64 
 6   PNEUMONIA             199999 non-null  int64 
 7   AGE                   199999 non-null  int64 
 8   PREGNANT              199999 non-null  int64 
 9   DIABETES              199999 non-null  int64 
 10  COPD                  199999 non-null  int64 
 11  ASTHMA                199999 non-null  int64 
 12  INMSUPR               199999 non-null  int64 
 13  HIPERTENSION          199999 non-null  int64 
 14  OTHER_DISEASE         199999 non-null  int64 
 15  CARDIOVASCULAR        199999 non-null  int64 
 16  OBESITY               199999 non-null  int64 
 17  RENAL_CHRONIC         199999 non-null  int64 
 18  TOBACCO               199999 non-null  int64 
 19  CLASIFFICATION_FINAL  199999 non-null  int64 
 20  ICU                   199999 non-null  int64 
dtypes: int64(20), object(1)
memory usage: 32.0+ MB
```

In [5]:

```
# Check the total missing values in each column
data.isnull().sum()
```

Out[5]:
```
USMER                   0
MEDICAL_UNIT            0
SEX                     0
PATIENT_TYPE            0
DATE_DIED               0
INTUBED                 0
PNEUMONIA               0
AGE                     0
PREGNANT                0
DIABETES                0
COPD                    0
ASTHMA                  0
INMSUPR                 0
HIPERTENSION            0
OTHER_DISEASE           0
CARDIOVASCULAR          0
OBESITY                 0
RENAL_CHRONIC           0
TOBACCO                 0
CLASIFFICATION_FINAL    0
ICU                     0
dtype: int64
```

In [6]:
```python
# Check the duplicate value
data.duplicated(keep='last').sum()
```

Out[6]:
```
125590
```

In [7]:
```python
# Dropping the duplicate values
data.drop_duplicates(keep='last',inplace=True)
```

In [8]:
```python
# Checking for dropped duplicate values
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 74409 entries, 0 to 199998
Data columns (total 21 columns):
 #   Column                Non-Null Count  Dtype 
---  ------                --------------  ----- 
 0   USMER                 74409 non-null  int64 
 1   MEDICAL_UNIT          74409 non-null  int64 
 2   SEX                   74409 non-null  int64 
 3   PATIENT_TYPE          74409 non-null  int64 
 4   DATE_DIED             74409 non-null  object
 5   INTUBED               74409 non-null  int64 
 6   PNEUMONIA             74409 non-null  int64 
 7   AGE                   74409 non-null  int64 
 8   PREGNANT              74409 non-null  int64 
 9   DIABETES              74409 non-null  int64 
 10  COPD                  74409 non-null  int64 
 11  ASTHMA                74409 non-null  int64 
 12  INMSUPR               74409 non-null  int64 
 13  HIPERTENSION          74409 non-null  int64 
 14  OTHER_DISEASE         74409 non-null  int64 
 15  CARDIOVASCULAR        74409 non-null  int64 
 16  OBESITY               74409 non-null  int64 
 17  RENAL_CHRONIC         74409 non-null  int64 
 18  TOBACCO               74409 non-null  int64 
 19  CLASIFFICATION_FINAL  74409 non-null  int64 
 20  ICU                   74409 non-null  int64 
dtypes: int64(20), object(1)
memory usage: 12.5+ MB
```

In [9]:
```
# Renaming columns
data.rename({'HIPERTENSION':'HYPERTENSION','CLASIFFICATION_FINAL':'CLASSIFICATION_FINAL','INMSUP
                    inplace=True)
```

In [10]:
```
data.columns
```

Out[10]:
```
Index(['USMER', 'MEDICAL_UNIT', 'SEX', 'PATIENT_TYPE', 'DATE_DIED', 'INTUBED',
       'PNEUMONIA', 'AGE', 'PREGNANT', 'DIABETES', 'COPD', 'ASTHMA',
       'IMMUNOSUPRS', 'HYPERTENSION', 'OTHER_DISEASE', 'CARDIOVASCULAR',
       'OBESITY', 'RENAL_CHRONIC', 'TOBACCO', 'CLASSIFICATION_FINAL', 'ICU'],
      dtype='object')
```

In [11]:
```
# Adding a new column in the dataframe and creating that column with patient survived or died
data['PATIENT_SURVIVED'] = (data['DATE_DIED'] == '9999-99-99').astype(int)
```

In [12]:
```
data.drop(['DATE_DIED'],axis=1,inplace=True)
```

In [13]:
```
x = data.drop('CLASSIFICATION_FINAL',axis=1).astype(float)
y = data['CLASSIFICATION_FINAL']
```

3. Encode the Output Variable

When modeling multi-class classification problems using neural networks, it is good practice to reshape the output attribute from a vector that contains values for each class value to a matrix with a Boolean for each class value and whether a given instance has that class value or not.

This is called one-hot encoding or creating dummy variables from a categorical variable. First encode the strings consistently to integers using the scikit-learn class LabelEncoder. Then convert the vector of integers to a one-hot encoding using the Keras function to_categorical().

In [14]:
```
from tensorflow.keras.utils import to_categorical
# encode class values as integers
from sklearn.preprocessing import LabelEncoder
label = LabelEncoder()
label.fit(y)
encoded_y = label.transform(y)
# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = to_categorical(encoded_y)
```

4. Data split

In [15]:
```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test = train_test_split(x,dummy_y,test_size=0.2,random_state=40)
```

In [16]:
```
n_features = len(x_train.columns)
n_features
```

Out[16]:
```
20
```

5. Data Normalization

In [17]:
```
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
x_train = ss.fit_transform(x_train)
x_test = ss.transform(x_test)
```

Build Deep Learning Models with Keras

6. Define Model

Models in Keras are defined as a sequence of layers. Created a Sequential model and added layers one at a time for the computation to be performed.

In [18]:
```
from keras.models import Sequential
from keras.layers import Dense
```

In [19]:
```
model = Sequential()
model.add(Dense(128, activation='relu', input_dim = n_features, kernel_initializer='uniform'))
model.add(Dense(64, activation='relu', kernel_initializer='uniform'))
model.add(Dense(8, activation='relu', kernel_initializer='uniform'))
model.add(Dense(7, activation='softmax'))
```

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

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	2,688
dense_1 (Dense)	(None, 64)	8,256
dense_2 (Dense)	(None, 8)	520
dense_3 (Dense)	(None, 7)	63

```
Total params: 11,527 (45.03 KB)
Trainable params: 11,527 (45.03 KB)
Non-trainable params: 0 (0.00 B)
```

7. Compile Model

Compile the model which makes use of the underlying framework to optimize the computation to be performed by your model. In this you can specify the loss function and the optimizer to be used.

In [21]:

```
# Compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

8. Fit Model

The model must be fit to data. This can be done one batch of data at a time or by firing off the entire model training regime. This is where all the compute happens.

We can train or fit our model by calling the fit() function on the model. This requires requires the training data to be specified, both a matrix of input patterns, X, and an array of matching output patterns, y. The training process will run for a fixed number of iterations through the dataset called epochs, that we must specify using the epochs argument. We can also set the number of instances that are evaluated before a weight update in the network is performed, called the batch size and set using the batch_size argument. For this problem, we will run for a small number of iterations (200) and use a relatively small batch size of 10. Again, these can be chosen experimentally by trial and error.

In [22]:

```
# Fit the model
history = model.fit(x_train, y_train, epochs=200, batch_size=10, verbose=1)
```

```
Epoch 1/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6379 - loss: 1.0636
Epoch 2/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6546 - loss: 0.9389
Epoch 3/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6560 - loss: 0.9285
Epoch 4/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6568 - loss: 0.9211
Epoch 5/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6588 - loss: 0.9169
Epoch 6/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6592 - loss: 0.9182
Epoch 7/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6561 - loss: 0.9199
Epoch 8/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6568 - loss: 0.9176
Epoch 9/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6590 - loss: 0.9121
Epoch 10/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6590 - loss: 0.9064
Epoch 11/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6591 - loss: 0.9107
Epoch 12/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6630 - loss: 0.8994
Epoch 13/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6584 - loss: 0.9091
Epoch 14/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6592 - loss: 0.9030
Epoch 15/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6635 - loss: 0.8944
Epoch 16/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6605 - loss: 0.8989
Epoch 17/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6610 - loss: 0.9014
Epoch 18/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6617 - loss: 0.8934
Epoch 19/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6667 - loss: 0.8879
Epoch 20/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6604 - loss: 0.8990
Epoch 21/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6623 - loss: 0.8893
Epoch 22/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6670 - loss: 0.8882
Epoch 23/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6677 - loss: 0.8846
Epoch 24/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6635 - loss: 0.8821
Epoch 25/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6656 - loss: 0.8746
Epoch 26/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6638 - loss: 0.8800
Epoch 27/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6630 - loss: 0.8740
Epoch 28/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6624 - loss: 0.8737
Epoch 29/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6654 - loss: 0.8773
Epoch 30/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6684 - loss: 0.8714
Epoch 31/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6687 - loss: 0.8708
```

```
Epoch 32/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6668 - loss: 0.8770
Epoch 33/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6676 - loss: 0.8726
Epoch 34/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6681 - loss: 0.8689
Epoch 35/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6662 - loss: 0.8693
Epoch 36/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6688 - loss: 0.8672
Epoch 37/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6675 - loss: 0.8664
Epoch 38/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6663 - loss: 0.8673
Epoch 39/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6650 - loss: 0.8693
Epoch 40/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6657 - loss: 0.8648
Epoch 41/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6698 - loss: 0.8611
Epoch 42/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6685 - loss: 0.8612
Epoch 43/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6685 - loss: 0.8596
Epoch 44/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6695 - loss: 0.8579
Epoch 45/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6656 - loss: 0.8614
Epoch 46/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6716 - loss: 0.8562
Epoch 47/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6731 - loss: 0.8504
Epoch 48/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6721 - loss: 0.8530
Epoch 49/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6677 - loss: 0.8605
Epoch 50/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6707 - loss: 0.8565
Epoch 51/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6705 - loss: 0.8562
Epoch 52/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6713 - loss: 0.8554
Epoch 53/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6740 - loss: 0.8487
Epoch 54/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6712 - loss: 0.8532
Epoch 55/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6682 - loss: 0.8625
Epoch 56/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6732 - loss: 0.8518
Epoch 57/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6727 - loss: 0.8496
Epoch 58/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.6717 - loss: 0.8461
Epoch 59/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6708 - loss: 0.8516
Epoch 60/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6735 - loss: 0.8463
Epoch 61/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6718 - loss: 0.8487
Epoch 62/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6745 - loss: 0.8409
```

```
Epoch 63/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6693 - loss: 0.8495
Epoch 64/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6710 - loss: 0.8522
Epoch 65/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6725 - loss: 0.8487
Epoch 66/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6737 - loss: 0.8460
Epoch 67/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6696 - loss: 0.8480
Epoch 68/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6688 - loss: 0.8517
Epoch 69/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6741 - loss: 0.8444
Epoch 70/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6770 - loss: 0.8371
Epoch 71/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6767 - loss: 0.8383
Epoch 72/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6752 - loss: 0.8403
Epoch 73/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6749 - loss: 0.8403
Epoch 74/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6757 - loss: 0.8390
Epoch 75/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6728 - loss: 0.8429
Epoch 76/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6724 - loss: 0.8435
Epoch 77/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6732 - loss: 0.8418
Epoch 78/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6713 - loss: 0.8466
Epoch 79/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6765 - loss: 0.8370
Epoch 80/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6751 - loss: 0.8406
Epoch 81/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6749 - loss: 0.8398
Epoch 82/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6745 - loss: 0.8377
Epoch 83/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6737 - loss: 0.8391
Epoch 84/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6744 - loss: 0.8337
Epoch 85/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.6753 - loss: 0.8354
Epoch 86/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6741 - loss: 0.8393
Epoch 87/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6759 - loss: 0.8331
Epoch 88/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6768 - loss: 0.8385
Epoch 89/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6731 - loss: 0.8339
Epoch 90/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6765 - loss: 0.8297
Epoch 91/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6764 - loss: 0.8300
Epoch 92/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6788 - loss: 0.8257
Epoch 93/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6748 - loss: 0.8353
```

```
Epoch 94/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6729 - loss: 0.8404
Epoch 95/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6781 - loss: 0.8308
Epoch 96/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6790 - loss: 0.8307
Epoch 97/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6777 - loss: 0.8344
Epoch 98/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6761 - loss: 0.8281
Epoch 99/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6769 - loss: 0.8316
Epoch 100/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 11s 2ms/step - accuracy: 0.6783 - loss: 0.8274
Epoch 101/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6779 - loss: 0.8305
Epoch 102/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6771 - loss: 0.8324
Epoch 103/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6729 - loss: 0.8406
Epoch 104/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6760 - loss: 0.8318
Epoch 105/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6752 - loss: 0.8361
Epoch 106/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6741 - loss: 0.8329
Epoch 107/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6774 - loss: 0.8294
Epoch 108/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6774 - loss: 0.8271
Epoch 109/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6785 - loss: 0.8252
Epoch 110/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6779 - loss: 0.8275
Epoch 111/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6784 - loss: 0.8287
Epoch 112/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6811 - loss: 0.8273
Epoch 113/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6772 - loss: 0.8312
Epoch 114/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6776 - loss: 0.8280
Epoch 115/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6759 - loss: 0.8324
Epoch 116/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6757 - loss: 0.8319
Epoch 117/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6802 - loss: 0.8275
Epoch 118/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6784 - loss: 0.8280
Epoch 119/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6750 - loss: 0.8271
Epoch 120/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6791 - loss: 0.8237
Epoch 121/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6809 - loss: 0.8209
Epoch 122/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6753 - loss: 0.8272
Epoch 123/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6773 - loss: 0.8231
Epoch 124/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6752 - loss: 0.8311
```

```
Epoch 125/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6828 - loss: 0.8175
Epoch 126/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6790 - loss: 0.8271
Epoch 127/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6795 - loss: 0.8205
Epoch 128/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6794 - loss: 0.8251
Epoch 129/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6783 - loss: 0.8236
Epoch 130/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6785 - loss: 0.8260
Epoch 131/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6797 - loss: 0.8226
Epoch 132/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6805 - loss: 0.8202
Epoch 133/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6772 - loss: 0.8233
Epoch 134/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6802 - loss: 0.8192
Epoch 135/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6775 - loss: 0.8208
Epoch 136/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6746 - loss: 0.8274
Epoch 137/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6781 - loss: 0.8178
Epoch 138/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6810 - loss: 0.8177
Epoch 139/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6826 - loss: 0.8160
Epoch 140/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6808 - loss: 0.8178
Epoch 141/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6817 - loss: 0.8211
Epoch 142/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6841 - loss: 0.8135
Epoch 143/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6790 - loss: 0.8194
Epoch 144/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6783 - loss: 0.8239
Epoch 145/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6786 - loss: 0.8230
Epoch 146/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6790 - loss: 0.8168
Epoch 147/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6799 - loss: 0.8226
Epoch 148/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6809 - loss: 0.8203
Epoch 149/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6772 - loss: 0.8229
Epoch 150/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6809 - loss: 0.8176
Epoch 151/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6792 - loss: 0.8209
Epoch 152/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6774 - loss: 0.8275
Epoch 153/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6825 - loss: 0.8186
Epoch 154/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6806 - loss: 0.8201
Epoch 155/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6827 - loss: 0.8172
```

```
Epoch 156/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6804 - loss: 0.8172
Epoch 157/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6814 - loss: 0.8174
Epoch 158/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6799 - loss: 0.8185
Epoch 159/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6792 - loss: 0.8189
Epoch 160/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6813 - loss: 0.8128
Epoch 161/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6781 - loss: 0.8181
Epoch 162/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6772 - loss: 0.8217
Epoch 163/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6810 - loss: 0.8198
Epoch 164/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6772 - loss: 0.8181
Epoch 165/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6796 - loss: 0.8178
Epoch 166/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6793 - loss: 0.8187
Epoch 167/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6839 - loss: 0.8131
Epoch 168/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6791 - loss: 0.8228
Epoch 169/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6816 - loss: 0.8173
Epoch 170/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6794 - loss: 0.8172
Epoch 171/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6841 - loss: 0.8147
Epoch 172/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6878 - loss: 0.8047
Epoch 173/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6806 - loss: 0.8177
Epoch 174/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6816 - loss: 0.8160
Epoch 175/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6794 - loss: 0.8194
Epoch 176/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6817 - loss: 0.8116
Epoch 177/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6811 - loss: 0.8190
Epoch 178/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6830 - loss: 0.8108
Epoch 179/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6816 - loss: 0.8143
Epoch 180/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6822 - loss: 0.8138
Epoch 181/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 2ms/step - accuracy: 0.6817 - loss: 0.8093
Epoch 182/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6793 - loss: 0.8183
Epoch 183/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 10s 1ms/step - accuracy: 0.6794 - loss: 0.8193
Epoch 184/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6821 - loss: 0.8108
Epoch 185/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6802 - loss: 0.8205
Epoch 186/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6819 - loss: 0.8111
```

```
Epoch 187/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6804 - loss: 0.8155
Epoch 188/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6830 - loss: 0.8154
Epoch 189/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6827 - loss: 0.8105
Epoch 190/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6789 - loss: 0.8133
Epoch 191/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6846 - loss: 0.8148
Epoch 192/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6805 - loss: 0.8152
Epoch 193/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6826 - loss: 0.8132
Epoch 194/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 2ms/step - accuracy: 0.6840 - loss: 0.8091
Epoch 195/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6787 - loss: 0.8195
Epoch 196/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6829 - loss: 0.8085
Epoch 197/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6831 - loss: 0.8092
Epoch 198/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6835 - loss: 0.8137
Epoch 199/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 8s 1ms/step - accuracy: 0.6853 - loss: 0.8067
Epoch 200/200
5953/5953 ━━━━━━━━━━━━━━━━━━━━ 9s 1ms/step - accuracy: 0.6824 - loss: 0.8095
```

Once fit, a history object is returned that provides a summary of the performance of the model during training. This includes both the loss and any additional metrics specified when compiling the model, recorded each epoch.

9. Evaluate Model

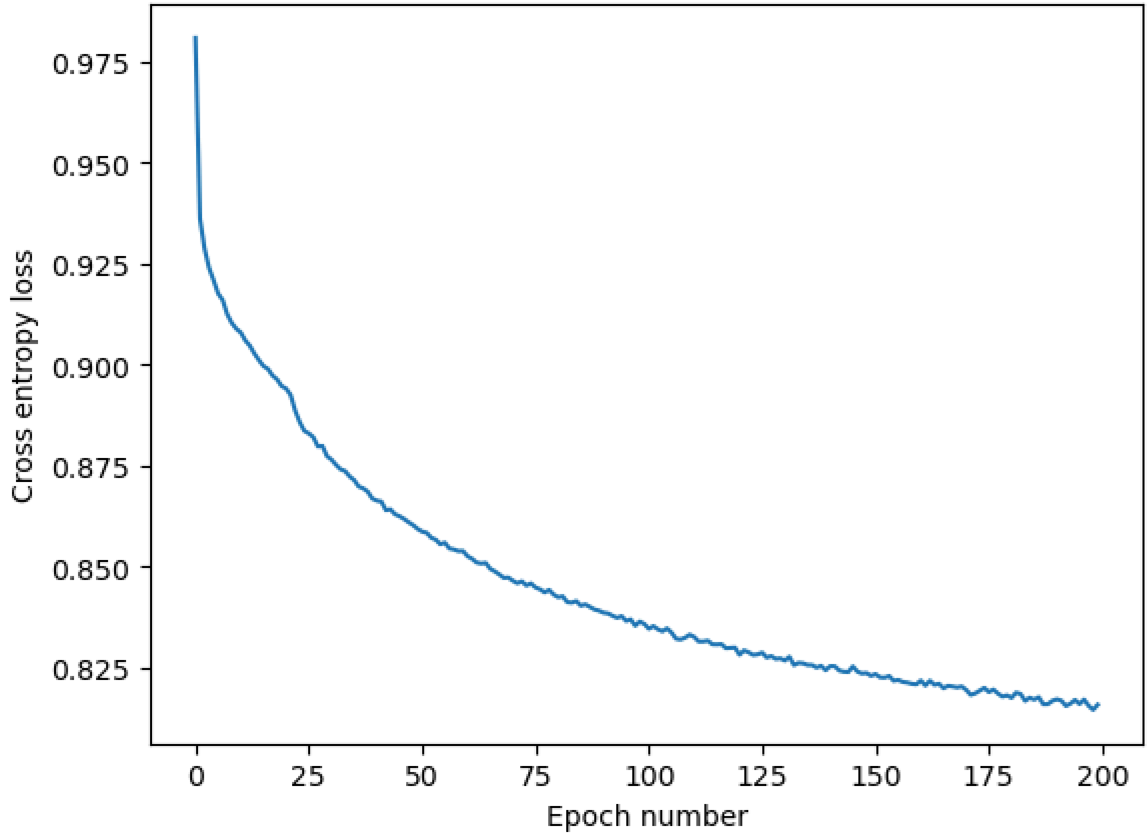
We have trained our neural network on the entire dataset and we can evaluate the performance of the network on the same dataset.

In [23]:
```python
# evaluate the model
scores = model.evaluate(x_train, y_train)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

```
1861/1861 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step - accuracy: 0.6844 - loss: 0.7996
compile_metrics: 68.46%
```

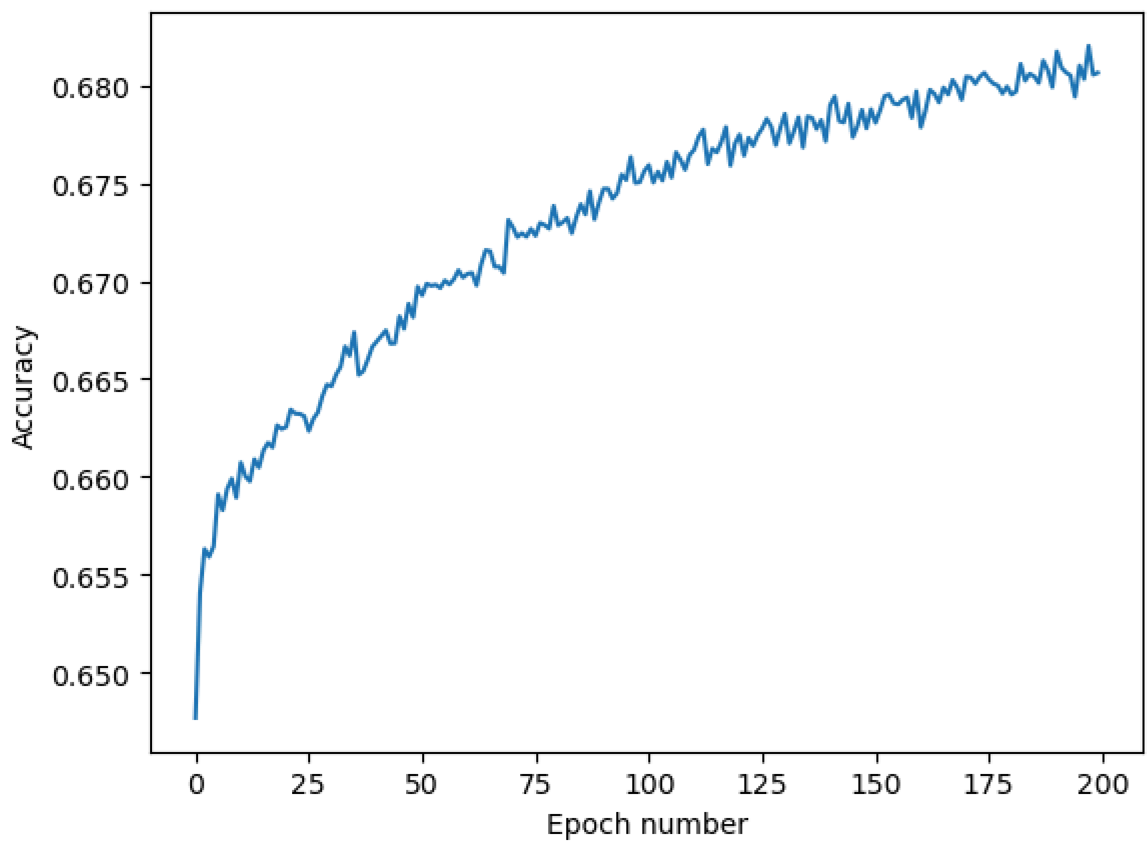
In [24]:
```python
#Plot history of Loss
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], label='categorical cross entropy loss')
plt.xlabel('Epoch number')
plt.ylabel('Cross entropy loss')
```

Out[24]:
```
Text(0, 0.5, 'Cross entropy loss')
```



In [25]:
```python
#Plot history of Accuracy
plt.plot(history.history['accuracy'], label='accuracy')
plt.xlabel('Epoch number')
plt.ylabel('Accuracy')
```

Out[25]:
```
Text(0, 0.5, 'Accuracy')
```

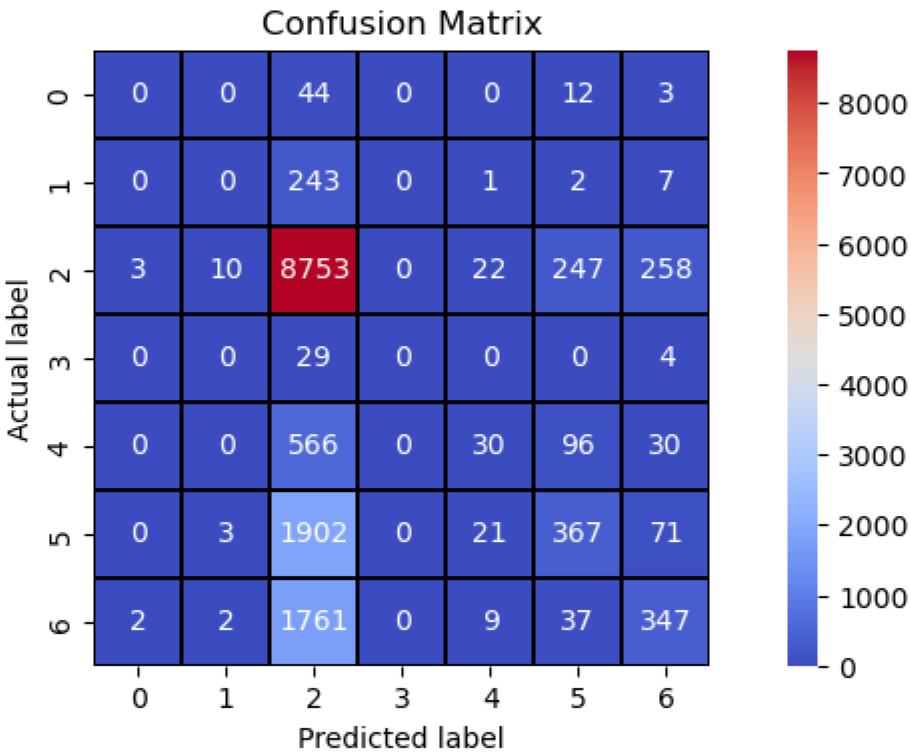


10. Make Predictions with Confusion Matrix and Classification Report

In [26]:
```
from sklearn.metrics import confusion_matrix, classification_report, multilabel_confusion_matrix

y_pred = model.predict(x_test)

print('Confusion Matrix is: ')
mat = multilabel_confusion_matrix(y_test.argmax(axis=1), y_pred.argmax(axis=1))
print(mat)


print('Classification Report is: ')
print(classification_report(y_test.argmax(axis=1), y_pred.argmax(axis=1)))
```

```
466/466 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step
Confusion Matrix is: 
[[[14818     5]
  [   59     0]]

 [[14614    15]
  [  253     0]]

 [[ 1044  4545]
  [  540  8753]]

 [[14849     0]
  [   33     0]]

 [[14107    53]
  [  692    30]]

 [[12124   394]
  [ 1997   367]]

 [[12351   373]
  [ 1811   347]]]
Classification Report is: 
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        59
           1       0.00      0.00      0.00       253
           2       0.66      0.94      0.77      9293
           3       0.00      0.00      0.00        33
           4       0.36      0.04      0.07       722
           5       0.48      0.16      0.23      2364
           6       0.48      0.16      0.24      2158

    accuracy                           0.64     14882
   macro avg       0.28      0.19      0.19     14882
weighted avg       0.58      0.64      0.56     14882
```

In [27]:
```
plt.figure(figsize=(10,4))
sns.heatmap(confusion_matrix(y_test.argmax(axis=1), y_pred.argmax(axis=1)), square=True, annot=T
            fmt='d', cmap='coolwarm',linewidths=0.3, linecolor='black')
plt.title('Confusion Matrix')
plt.xlabel('Predicted label')
plt.ylabel('Actual label')
plt.show()
```



The predictions will be returned in the format provided by the output layer of the network. For a multiclass classification problem, the results may be in the form of an array of probabilities (use argmax() Function)