Assignment 4 - Multiclass Classifier Neural Network

In [ ]:
```
import pandas as pd
```

Step 1: Load the Data

In [ ]:
```
df = pd.read_csv('a4-data/train.csv')
df.head()
```

Out[ ]:

	Feature_1	Feature_2	Feature_3	Feature_4	Feature_5	Feature_6	Feature_7	Fea
0	3289	22	19	240	93	1708	205	
1	2963	21	18	134	27	1243	206	
2	3037	185	9	127	10	6462	222	
3	3113	203	13	190	22	2125	213	
4	3128	346	9	120	36	552	203	

5 rows × 55 columns

we need to check for any null or missing values and deal with them accordinly, here we have none

In [ ]:
```
df.isnull().sum().sum()
```

Out[ ]:
```
0
```

Step 2: Split the Data into Features (X) and Target (Y)

Here we split the training data into X and y variables where X containsa feature matrix and y contains a target vector, this is done so the model has a target value to compare to its predictions, this enables us to calculate the loss which utimately allows the model to learn.

In [ ]:
```
# split dataset into features and target
X = df.drop('Target', axis=1)
y = df['Target']
X.shape, y.shape
```

Out[ ]:
```
((464809, 54), (464809,))
```

Now that we have our target and features separated, we need to normalize the feature matrix which will help the model converge to a minima faster by leveling out the gradient. The reason for not normalizing the target vector is because it contains categorical data which should not be normalized.

Normalize the features

When no normalization is done the models accuracy is poor with a value less than 60%, but when all features arn normalized the models preformance dramitacally increases to over 90%

We can take this one step further by only normalizing the non-binary features which improves the models performance even more, even when the model has a significantly higher accuracy of 93.6%, only normalizing the non-binary features gives us a 1% increase with a 94.6%

Here we get only the non-binary columns and normalize them using z-score normalization

In [ ]:

```
binary_columns = [col for col in df.columns if df[col].dropna().isin([0, 1])

non_binary_columns = [col for col in X.columns if col not in binary_columns]

# Normalize non-binary columns
X[non_binary_columns] = (X[non_binary_columns] - X[non_binary_columns].mean(
X.head()
```

Out[ ]:

e_10	...	Feature_45	Feature_46	Feature_47	Feature_48	Feature_49	Feature_50	F
5718	...	0	0	1	0	0	0	
4309	...	0	0	0	0	0	0	
5933	...	0	0	0	0	0	0	
3645	...	0	1	0	0	0	0	
7276	...	0	0	0	0	0	0	

Step 3: Data Splitting

We need to split our data into training and test datasets so that we can evaulate and validate our models performance, if we tested on the entire dataset, we would not be testing the model on unseen data which would not be a valid test of its performance.

In [ ]:
```
from sklearn.model_selection import train_test_split

# Split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rar
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[ ]:
```
((325366, 54), (139443, 54), (325366,), (139443,))
```

Step 4: Build the Neural Network Model

Here we build the model. In order to have optimal parameters, I decided to use the Hyperband algorithm to tune my parameters using the kerastuner library. I first started with a small depth and width, letting the algorithm find the optimal learning rate. The smaller models performed well, but by having a larger neural network lead to better models.

The best model from the Hyperband process has the following parameters:

- 54 node input layer, for the 54 features in X
- 120 node hidden layer, relu activation
- 112 node hidder layer, relu activation
- 104 node hidden layer, relu activation
- 80 node hidden layer, relu activation
- 7 node output layer, sofmax actiavtion
- learned rate of 0.00085410586

I tested using few layers but utimately ended up with 4 hidden layers which seemed to give good results for the model

Relu was used as the activation fucntion for hidden layers because it introduces non-linearity into the model which allows the model to learn complex patterns and relationships which is what we are trying to achieve with are classifier.

Softmax was used for the output layers activation function as it converts the raw output from a neural network into probabilites which we can then select the class with the highest probabilty, this is perfect for our classifiers output layer.

Since the number of nodes in the output layer of the model needs to match the range for the taget, we use the targets unique value count plus 1 as the output layers shape

Here we set up the model with our found widths for each layer from the hyperparameter tuning

In [ ]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
```

```
# Define the model
model = Sequential([
    Input(shape=(X_train.shape[1],)),
    Dense(units=120, activation='relu'),
    Dense(units=112, activation='relu'),
    Dense(units=104, activation='relu'),
    Dense(units=80, activation='relu'),
    Dense(units=(y_train.nunique()+1) , activation='softmax')
])
```

```
/Users/bradymitchelmore/Library/Mobile Documents/com~apple~CloudDocs/MUN/Yea
r 3/Term 7/COMP 3401/Assignments/comp3401/lib/python3.9/site-packages/urllib
3/__init__.py:35: NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1
+, currently the 'ssl' module is compiled with 'LibreSSL 2.8.3'. See: http
s://github.com/urllib3/urllib3/issues/3020
  warnings.warn(
```

Step 5: Train the Model

Here we train the model using the learning rate from our hyperparameter tuning and 100 epochs

- I choose the Adam optimizer due to its adaptive learning rates and efficiency which help the model converge faster.
- The loss fucntion used was sparse_categorical_crossentropy, which is a function that is good when dealing with multiclass claissification where each sample belongs to one of many classes and the labels are numeric, these are 2 properties of our target vector which deems this loss function suitable.

In [ ]:
```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers.schedules import ExponentialDecay

initial_learning_rate = 0.00085410586

optimizer = Adam(learning_rate=initial_learning_rate)

model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, m
history = model.fit(X_train, y_train, epochs=100, validation_data=(X_test, y
```

```
Epoch 1/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 9s 828us/step - accuracy: 0.7519 - loss: 0.
5861 - val_accuracy: 0.8319 - val_loss: 0.4034
Epoch 2/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 761us/step - accuracy: 0.8403 - loss: 0.
3805 - val_accuracy: 0.8606 - val_loss: 0.3378
Epoch 3/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 702us/step - accuracy: 0.8671 - loss: 0.
3204 - val_accuracy: 0.8728 - val_loss: 0.3068
Epoch 4/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 798us/step - accuracy: 0.8812 - loss: 0.
2870 - val_accuracy: 0.8860 - val_loss: 0.2772
Epoch 5/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 733us/step - accuracy: 0.8916 - loss: 0.
2620 - val_accuracy: 0.8945 - val_loss: 0.2582
Epoch 6/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 714us/step - accuracy: 0.8991 - loss: 0.
2450 - val_accuracy: 0.8917 - val_loss: 0.2592
Epoch 7/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 694us/step - accuracy: 0.9059 - loss: 0.
2303 - val_accuracy: 0.9028 - val_loss: 0.2371
Epoch 8/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 720us/step - accuracy: 0.9100 - loss: 0.
2191 - val_accuracy: 0.9088 - val_loss: 0.2243
Epoch 9/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 705us/step - accuracy: 0.9143 - loss: 0.
2079 - val_accuracy: 0.9135 - val_loss: 0.2169
Epoch 10/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 703us/step - accuracy: 0.9180 - loss: 0.
2021 - val_accuracy: 0.9110 - val_loss: 0.2208
Epoch 11/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 707us/step - accuracy: 0.9200 - loss: 0.
1955 - val_accuracy: 0.9129 - val_loss: 0.2159
Epoch 12/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 738us/step - accuracy: 0.9237 - loss: 0.
1872 - val_accuracy: 0.9168 - val_loss: 0.2066
Epoch 13/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 697us/step - accuracy: 0.9260 - loss: 0.
1824 - val_accuracy: 0.9192 - val_loss: 0.2006
Epoch 14/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 722us/step - accuracy: 0.9273 - loss: 0.
1787 - val_accuracy: 0.9218 - val_loss: 0.1959
Epoch 15/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 721us/step - accuracy: 0.9293 - loss: 0.
1737 - val_accuracy: 0.9226 - val_loss: 0.1965
Epoch 16/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 696us/step - accuracy: 0.9310 - loss: 0.
1699 - val_accuracy: 0.9198 - val_loss: 0.2021
Epoch 17/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 740us/step - accuracy: 0.9315 - loss: 0.
1684 - val_accuracy: 0.9244 - val_loss: 0.1916
Epoch 18/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 688us/step - accuracy: 0.9341 - loss: 0.
1641 - val_accuracy: 0.9255 - val_loss: 0.1894
Epoch 19/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 727us/step - accuracy: 0.9350 - loss: 0.
```

```
1611 - val_accuracy: 0.9241 - val_loss: 0.1950
Epoch 20/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 715us/step - accuracy: 0.9351 - loss: 0.
1596 - val_accuracy: 0.9276 - val_loss: 0.1831
Epoch 21/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 693us/step - accuracy: 0.9369 - loss: 0.
1557 - val_accuracy: 0.9286 - val_loss: 0.1809
Epoch 22/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 717us/step - accuracy: 0.9375 - loss: 0.
1547 - val_accuracy: 0.9288 - val_loss: 0.1833
Epoch 23/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 717us/step - accuracy: 0.9384 - loss: 0.
1524 - val_accuracy: 0.9293 - val_loss: 0.1805
Epoch 24/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 703us/step - accuracy: 0.9394 - loss: 0.
1499 - val_accuracy: 0.9314 - val_loss: 0.1778
Epoch 25/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 704us/step - accuracy: 0.9403 - loss: 0.
1472 - val_accuracy: 0.9307 - val_loss: 0.1813
Epoch 26/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 696us/step - accuracy: 0.9410 - loss: 0.
1463 - val_accuracy: 0.9315 - val_loss: 0.1772
Epoch 27/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 718us/step - accuracy: 0.9420 - loss: 0.
1423 - val_accuracy: 0.9343 - val_loss: 0.1699
Epoch 28/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 724us/step - accuracy: 0.9431 - loss: 0.
1426 - val_accuracy: 0.9317 - val_loss: 0.1796
Epoch 29/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 685us/step - accuracy: 0.9434 - loss: 0.
1408 - val_accuracy: 0.9335 - val_loss: 0.1719
Epoch 30/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 716us/step - accuracy: 0.9430 - loss: 0.
1398 - val_accuracy: 0.9333 - val_loss: 0.1726
Epoch 31/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 713us/step - accuracy: 0.9442 - loss: 0.
1384 - val_accuracy: 0.9308 - val_loss: 0.1815
Epoch 32/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 746us/step - accuracy: 0.9441 - loss: 0.
1378 - val_accuracy: 0.9349 - val_loss: 0.1718
Epoch 33/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 746us/step - accuracy: 0.9457 - loss: 0.
1343 - val_accuracy: 0.9318 - val_loss: 0.1847
Epoch 34/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 729us/step - accuracy: 0.9460 - loss: 0.
1350 - val_accuracy: 0.9358 - val_loss: 0.1668
Epoch 35/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 754us/step - accuracy: 0.9462 - loss: 0.
1335 - val_accuracy: 0.9363 - val_loss: 0.1707
Epoch 36/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 766us/step - accuracy: 0.9469 - loss: 0.
1313 - val_accuracy: 0.9334 - val_loss: 0.1779
Epoch 37/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 717us/step - accuracy: 0.9476 - loss: 0.
1301 - val_accuracy: 0.9369 - val_loss: 0.1654
Epoch 38/100
```

```
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 728us/step - accuracy: 0.9482 - loss: 0.
1291 - val_accuracy: 0.9380 - val_loss: 0.1677
Epoch 39/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 729us/step - accuracy: 0.9476 - loss: 0.
1295 - val_accuracy: 0.9374 - val_loss: 0.1662
Epoch 40/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 727us/step - accuracy: 0.9486 - loss: 0.
1284 - val_accuracy: 0.9372 - val_loss: 0.1668
Epoch 41/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 714us/step - accuracy: 0.9487 - loss: 0.
1282 - val_accuracy: 0.9371 - val_loss: 0.1678
Epoch 42/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 718us/step - accuracy: 0.9498 - loss: 0.
1253 - val_accuracy: 0.9390 - val_loss: 0.1633
Epoch 43/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 722us/step - accuracy: 0.9494 - loss: 0.
1260 - val_accuracy: 0.9373 - val_loss: 0.1652
Epoch 44/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 737us/step - accuracy: 0.9509 - loss: 0.
1236 - val_accuracy: 0.9386 - val_loss: 0.1683
Epoch 45/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 729us/step - accuracy: 0.9504 - loss: 0.
1242 - val_accuracy: 0.9365 - val_loss: 0.1696
Epoch 46/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 750us/step - accuracy: 0.9503 - loss: 0.
1231 - val_accuracy: 0.9393 - val_loss: 0.1655
Epoch 47/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 718us/step - accuracy: 0.9506 - loss: 0.
1234 - val_accuracy: 0.9389 - val_loss: 0.1628
Epoch 48/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 721us/step - accuracy: 0.9516 - loss: 0.
1206 - val_accuracy: 0.9372 - val_loss: 0.1709
Epoch 49/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 719us/step - accuracy: 0.9517 - loss: 0.
1207 - val_accuracy: 0.9361 - val_loss: 0.1751
Epoch 50/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 739us/step - accuracy: 0.9520 - loss: 0.
1200 - val_accuracy: 0.9402 - val_loss: 0.1628
Epoch 51/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 728us/step - accuracy: 0.9528 - loss: 0.
1183 - val_accuracy: 0.9405 - val_loss: 0.1608
Epoch 52/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 737us/step - accuracy: 0.9523 - loss: 0.
1201 - val_accuracy: 0.9401 - val_loss: 0.1643
Epoch 53/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 706us/step - accuracy: 0.9529 - loss: 0.
1188 - val_accuracy: 0.9393 - val_loss: 0.1652
Epoch 54/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 732us/step - accuracy: 0.9529 - loss: 0.
1189 - val_accuracy: 0.9399 - val_loss: 0.1639
Epoch 55/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 765us/step - accuracy: 0.9533 - loss: 0.
1169 - val_accuracy: 0.9350 - val_loss: 0.1754
Epoch 56/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 738us/step - accuracy: 0.9535 - loss: 0.
1172 - val_accuracy: 0.9405 - val_loss: 0.1600
```

```
Epoch 57/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 723us/step - accuracy: 0.9539 - loss: 0.
1160 - val_accuracy: 0.9374 - val_loss: 0.1715
Epoch 58/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 736us/step - accuracy: 0.9540 - loss: 0.
1162 - val_accuracy: 0.9399 - val_loss: 0.1669
Epoch 59/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 763us/step - accuracy: 0.9547 - loss: 0.
1138 - val_accuracy: 0.9381 - val_loss: 0.1702
Epoch 60/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 792us/step - accuracy: 0.9541 - loss: 0.
1155 - val_accuracy: 0.9413 - val_loss: 0.1637
Epoch 61/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 723us/step - accuracy: 0.9542 - loss: 0.
1149 - val_accuracy: 0.9375 - val_loss: 0.1742
Epoch 62/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 769us/step - accuracy: 0.9545 - loss: 0.
1149 - val_accuracy: 0.9376 - val_loss: 0.1782
Epoch 63/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 749us/step - accuracy: 0.9546 - loss: 0.
1138 - val_accuracy: 0.9389 - val_loss: 0.1671
Epoch 64/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 734us/step - accuracy: 0.9550 - loss: 0.
1137 - val_accuracy: 0.9395 - val_loss: 0.1715
Epoch 65/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 701us/step - accuracy: 0.9541 - loss: 0.
1153 - val_accuracy: 0.9416 - val_loss: 0.1642
Epoch 66/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 773us/step - accuracy: 0.9552 - loss: 0.
1129 - val_accuracy: 0.9399 - val_loss: 0.1694
Epoch 67/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 737us/step - accuracy: 0.9554 - loss: 0.
1129 - val_accuracy: 0.9391 - val_loss: 0.1754
Epoch 68/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 738us/step - accuracy: 0.9552 - loss: 0.
1137 - val_accuracy: 0.9394 - val_loss: 0.1702
Epoch 69/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 726us/step - accuracy: 0.9556 - loss: 0.
1114 - val_accuracy: 0.9435 - val_loss: 0.1588
Epoch 70/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 9s 874us/step - accuracy: 0.9555 - loss: 0.
1113 - val_accuracy: 0.9384 - val_loss: 0.1705
Epoch 71/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 730us/step - accuracy: 0.9558 - loss: 0.
1117 - val_accuracy: 0.9401 - val_loss: 0.1722
Epoch 72/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 755us/step - accuracy: 0.9565 - loss: 0.
1108 - val_accuracy: 0.9415 - val_loss: 0.1645
Epoch 73/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 769us/step - accuracy: 0.9563 - loss: 0.
1107 - val_accuracy: 0.9414 - val_loss: 0.1677
Epoch 74/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 736us/step - accuracy: 0.9565 - loss: 0.
1094 - val_accuracy: 0.9413 - val_loss: 0.1753
Epoch 75/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 719us/step - accuracy: 0.9565 - loss: 0.
```

```
1109 - val_accuracy: 0.9426 - val_loss: 0.1622
Epoch 76/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 788us/step - accuracy: 0.9570 - loss: 0.
1100 - val_accuracy: 0.9422 - val_loss: 0.1637
Epoch 77/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 752us/step - accuracy: 0.9568 - loss: 0.
1097 - val_accuracy: 0.9416 - val_loss: 0.1683
Epoch 78/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 740us/step - accuracy: 0.9561 - loss: 0.
1122 - val_accuracy: 0.9422 - val_loss: 0.1663
Epoch 79/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 730us/step - accuracy: 0.9580 - loss: 0.
1079 - val_accuracy: 0.9435 - val_loss: 0.1624
Epoch 80/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 778us/step - accuracy: 0.9577 - loss: 0.
1071 - val_accuracy: 0.9432 - val_loss: 0.1643
Epoch 81/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 734us/step - accuracy: 0.9566 - loss: 0.
1109 - val_accuracy: 0.9418 - val_loss: 0.1727
Epoch 82/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 734us/step - accuracy: 0.9576 - loss: 0.
1069 - val_accuracy: 0.9437 - val_loss: 0.1625
Epoch 83/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 737us/step - accuracy: 0.9567 - loss: 0.
1119 - val_accuracy: 0.9442 - val_loss: 0.1637
Epoch 84/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 737us/step - accuracy: 0.9579 - loss: 0.
1081 - val_accuracy: 0.9419 - val_loss: 0.1675
Epoch 85/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 729us/step - accuracy: 0.9575 - loss: 0.
1077 - val_accuracy: 0.9405 - val_loss: 0.1723
Epoch 86/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 750us/step - accuracy: 0.9578 - loss: 0.
1083 - val_accuracy: 0.9436 - val_loss: 0.1630
Epoch 87/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 736us/step - accuracy: 0.9578 - loss: 0.
1079 - val_accuracy: 0.9408 - val_loss: 0.1696
Epoch 88/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 742us/step - accuracy: 0.9584 - loss: 0.
1073 - val_accuracy: 0.9429 - val_loss: 0.1687
Epoch 89/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 736us/step - accuracy: 0.9588 - loss: 0.
1061 - val_accuracy: 0.9431 - val_loss: 0.1680
Epoch 90/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 732us/step - accuracy: 0.9588 - loss: 0.
1056 - val_accuracy: 0.9432 - val_loss: 0.1633
Epoch 91/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 727us/step - accuracy: 0.9575 - loss: 0.
1093 - val_accuracy: 0.9419 - val_loss: 0.1749
Epoch 92/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 732us/step - accuracy: 0.9587 - loss: 0.
1055 - val_accuracy: 0.9447 - val_loss: 0.1627
Epoch 93/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 744us/step - accuracy: 0.9590 - loss: 0.
1051 - val_accuracy: 0.9434 - val_loss: 0.1654
Epoch 94/100
```

```
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 753us/step - accuracy: 0.9584 - loss: 0.
1081 - val_accuracy: 0.9427 - val_loss: 0.1679
Epoch 95/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 736us/step - accuracy: 0.9578 - loss: 0.
1091 - val_accuracy: 0.9406 - val_loss: 0.1759
Epoch 96/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 741us/step - accuracy: 0.9588 - loss: 0.
1056 - val_accuracy: 0.9408 - val_loss: 0.1734
Epoch 97/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 7s 734us/step - accuracy: 0.9583 - loss: 0.
1057 - val_accuracy: 0.9435 - val_loss: 0.1681
Epoch 98/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 736us/step - accuracy: 0.9584 - loss: 0.
1063 - val_accuracy: 0.9418 - val_loss: 0.1713
Epoch 99/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 747us/step - accuracy: 0.9583 - loss: 0.
1071 - val_accuracy: 0.9435 - val_loss: 0.1696
Epoch 100/100
10168/10168 ━━━━━━━━━━━━━━━━━━━━ 8s 806us/step - accuracy: 0.9595 - loss: 0.
1051 - val_accuracy: 0.9444 - val_loss: 0.1742
```

Step 6: Model Evaluation

Evaluating the model on the test set allows us to assess its performance on unseen data, giving us an indication of its generalization ability. Accuracy is used as a metric to quantify the percentage of correctly predicted instances, while the loss metric gives us an indication of how close our predictions are.

In [ ]:
```python
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Loss: {loss}, Accuracy: {accuracy}")
```

```
4358/4358 ━━━━━━━━━━━━━━━━━━━━ 2s 338us/step - accuracy: 0.9436 - loss: 0.17
69
Loss: 0.17416204512119293, Accuracy: 0.9443858861923218
```

Here we can see our model gives us a 94.36% accuracy on the test set with a loss of just 0.1705

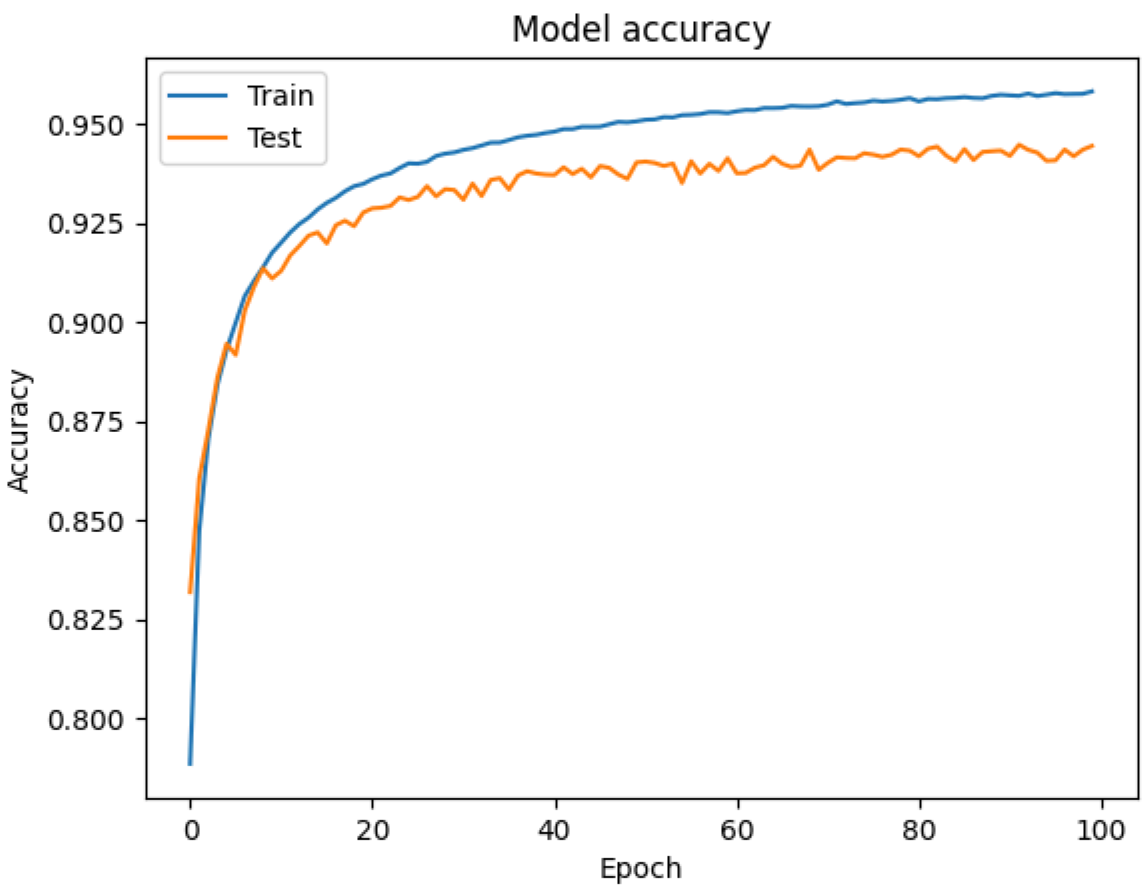
When training we used X_test and y_test as a validation set which gave us the val_accuaracy and val_loss which measure the accuracy and loss in terms of the test set. We can plot the validation vs the training accuracy and loss to visualize how the model fits to its training data better than unseen data

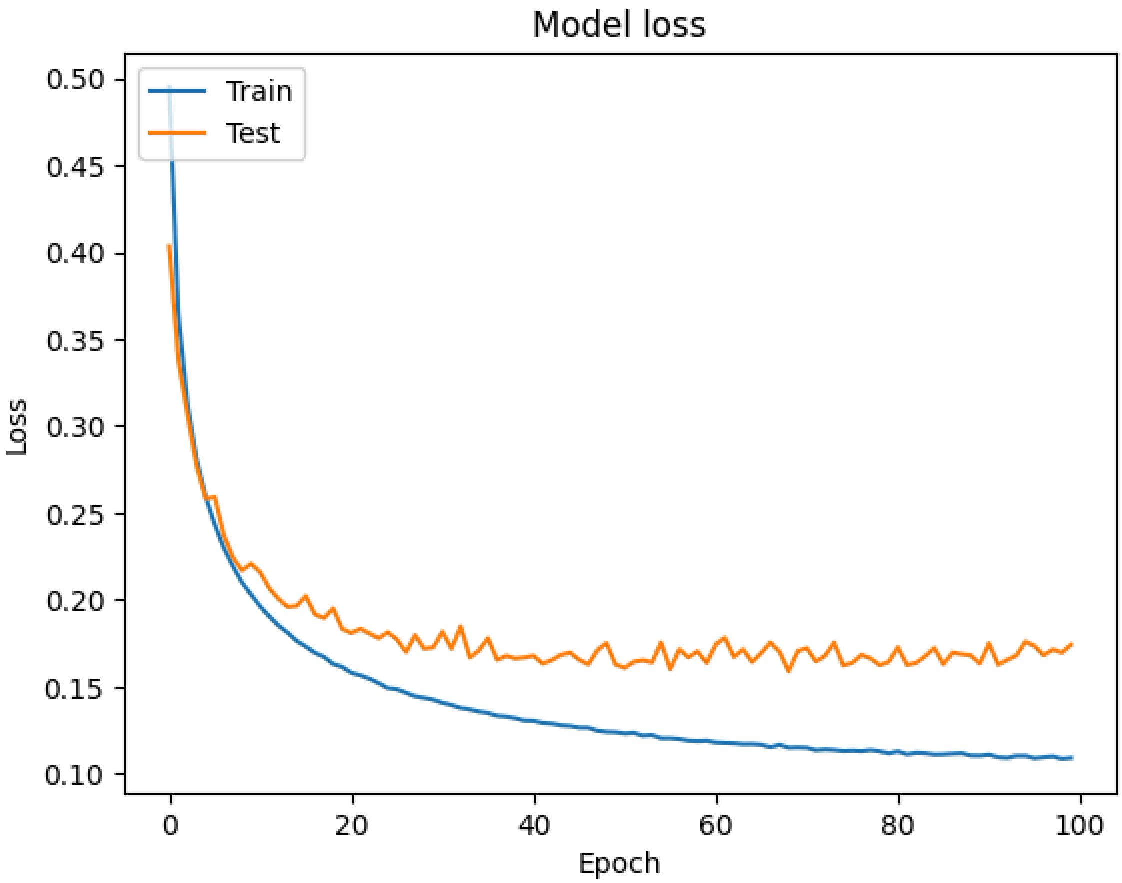
In [ ]:
```python
import matplotlib.pyplot as plt

# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
```

```
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```





In both graphs we see that the validation results arent quite as good which is expected as it is unseen data

Step 7: Make Predictions

Making predictions involves feeding new data into the trained model and using the softmax probabilities to determine the most likely class for each instance. This process demonstrates the model's practical utility in classifying new, unseen data.

In [ ]:
```
# load the test set
test_set = pd.read_csv('a4-data/test.csv')
test_set.head()
```

Out[ ]:

	Feature_1	Feature_2	Feature_3	Feature_4	Feature_5	Feature_6	Feature_7	Fea
0	3351	206	27	726	124	3813	192	
1	2732	129	7	212	1	1082	231	
2	2572	24	9	201	25	957	216	
3	2824	69	13	417	39	3223	233	
4	2529	84	5	120	9	1092	227	

5 rows × 54 columns

We need to only scale binary columns like we did in the training data in order to get accurate predictions

In [ ]:

```
binary_columns = [col for col in test_set.columns if test_set[col].dropna().

non_binary_columns = [col for col in test_set.columns if col not in binary_c

# Normalize non-binary columns
test_set[non_binary_columns] = (test_set[non_binary_columns] - test_set[non_
test_set.head()
```

Out[ ]:

	Feature_1	Feature_2	Feature_3	Feature_4	Feature_5	Feature_6	Feature_7	Fea
0	1.397080	0.448163	1.717586	2.142548	1.325494	0.939452	-0.749856	1.
1	-0.817890	-0.239629	-0.950993	-0.271829	-0.781223	-0.813558	0.704324	0
2	-1.390419	-1.177526	-0.684135	-0.323499	-0.370156	-0.893795	0.145024	-0.
3	-0.488686	-0.775570	-0.150419	0.691103	-0.130367	0.560735	0.778898	-0
4	-1.544286	-0.641585	-1.217851	-0.703975	-0.644200	-0.807139	0.555178	0

5 rows × 54 columns

Now we use `model.predit()` to get a matrix with 7 columns and each row representing a probability distribution of the classes, we then get the highest probabilties index which is our prediction

In [ ]:

```
import numpy as np

# make predictions and convert them to classes
predictions = model.predict(test_set)
predicted_classes = np.argmax(predictions, axis=1)
```

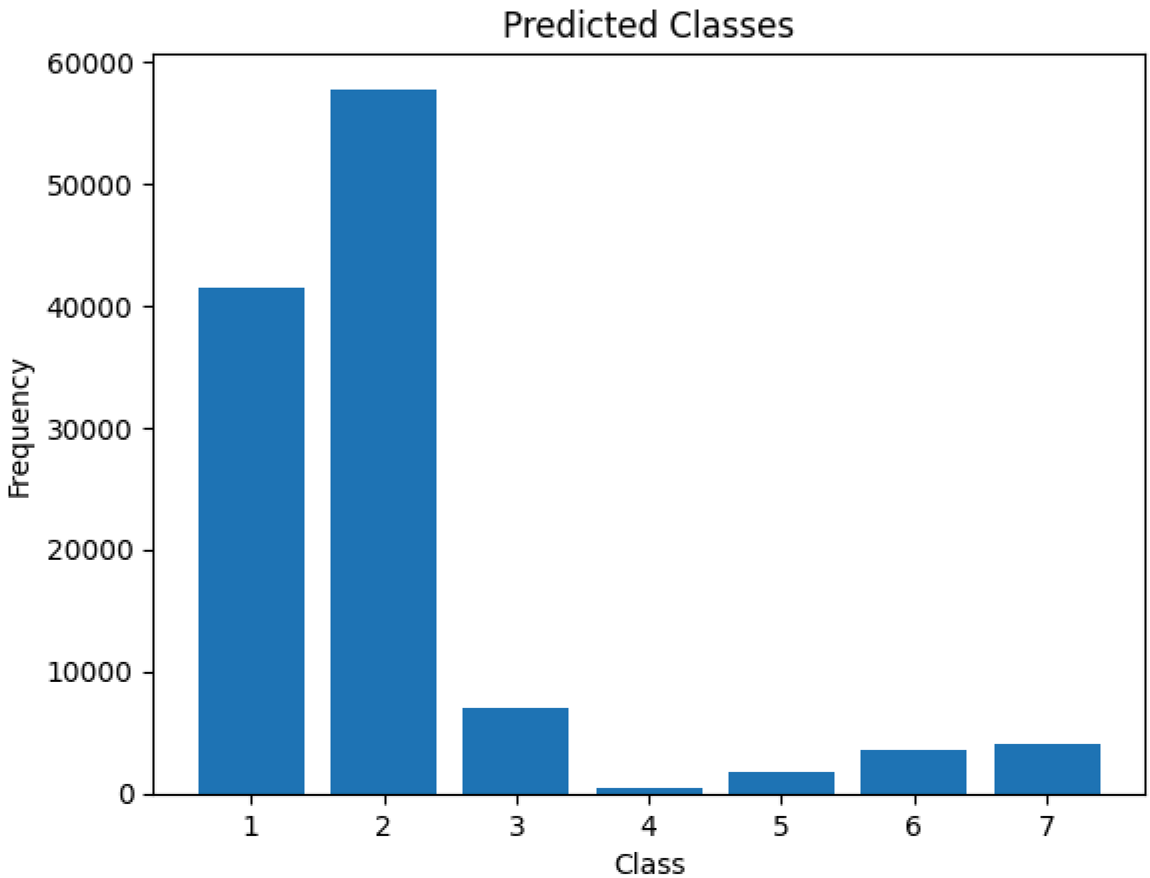
```
3632/3632 ━━━━━━━━━━━━━━━━━━━━ 1s 293us/step
```

In [ ]:

```
import matplotlib.pyplot as plt

# plot the predicted classes
plt.hist(predicted_classes, bins=range(1, 9), rwidth=0.8, align='left')
```

```
plt.title('Predicted Classes')
plt.xlabel('Class')
plt.ylabel('Frequency')
plt.show()
```



Here we can see the the a large majority of predictions were either 1 or 2, this is a useful insight

Step 8: Generate Submission File

In [ ]:
```
# save the predictions to a CSV file
submission = pd.DataFrame({'Target': predicted_classes})
submission.to_csv('submission.csv')
```