General Data Collection and Pre-processing:

In this dataset, I have used the ‘Celebrity Face Recognition - VGGFace Model’ by Vinayak Shanawad. The dataset was downloaded from Kaggle containing 17,534 images for 100 people. All images in the dataset were taken from 'Pinterest' and aligned using the dlib library. The dataset was further narrowed down to 1,113 images for 20 people with each person having about 47 – 60 images each.

Methodology

The methods used in this research were trained and tested with google collab, and Jupyter notebook with the use of some basic libraries. These algorithms were trained using regular CPU’S, therefore high power GPU’s are not essential to run them.

1. CNN Sequential Model Approach

The first approach used is the custom sequential convolutional neural networks (CNN) model. The convolutional neural networks were deemed quite distinguishable from other neural networks due to their superior performance using image inputs (IBM, no date). The convolutional neural networks are made up of three layers: convolutional, pooling and fully connected (FC) layers.

The convolutional layer is a crucial block of the CNN which applies a set of filters to the input images (IBM, no date). The convolutional layer also converts the input images into numerical values which allows the neural network extract relevant patterns (IBM, no date). In our model, we use three convolutional layers with increasing complexity. Each convolutional layer consists of multiple filters that convolve across the input images, extracting spatial features such as edges, textures, and patterns, and produces a feature map as an output. The activation function used in these layers is Rectified Linear Unit (ReLU), which introduces non-linearity to the model, enabling it to learn complex patterns.

The pooling layer conducts dimensionality reduction by reducing the input parameters. This is also known as ‘downsampling’ (IBM, no date). In our model, we utilize three max pooling layers, which selects the maximum value pixels, gotten from the output of the preceding convolutional layer, and sends it to an output array, effectively choosing the key features within a feature map. This process of ‘downsampling’ helps in reducing computational complexity and controls overfitting (IBM, no date).

Lastly, the fully connected (FC) layer performs classification functions based on the features extracted in the previous layers (IBM, no date). The softmax layer computes the probability distribution over the classes, providing the final classification output.

Specific Pre-processing:

For the CNN sequential model, the training and test datasets were pre-processed using the ImageDataGenerator class from the Keras library. ‘The Keras preprocessing layers API allows developers to build Keras-native input processing pipelines. These input processing pipelines can be used as independent preprocessing code in non-Keras workflows…’ (TensorFlow, 2024, para. 1). The ImageDataGenerator class preprocesses image data through data transformation and augmentation. In this model, I normalized the dataset by rescaling all image pixel values, to a number between 0 and 1, by dividing by 255. I also used the ImageDataGenerator class to split the dataset into training and validation sets using a validation split of 0.2.

Image Resizing: All images in the dataset have a target size of 160 x 16`0.

Data Augmentation: Random horizontal and vertical flips, a rotation range value of 45 degrees, and a shear and zoom range of 0.2 each were applied to augment the dataset. The model also has a Dropout rate of 0.5.

. Potential Use of a grayscale.

CNN Sequential Model Attempt1

In the first Attempt, I started the process having a very limited knowledge on training custom face recognition models therefore I had some difficulty setting the parameter values for the preprocessing pipeline and model architecture. I decided to set somewhat of a basic architecture for this first attempt. The model architecture constituted of 3 convolutional layers with the first convolutional layer having 32 filters, the second convolutional layer having 64 filters and the third having 128 filters. In between the convolutional layers, I had three MaxPooling2D layers with a pool size of (2,2). MaxPooling2D layer is a very common type of pooling layer. I also employed two FC layers, which are the dense layers. The first FC layer had a 'ReLU' activation function, while the last layer, which is the output layer, had a 'softmax' activation function. Lastly, In between the FC layers, I incorporated one dropout layer with a parameter of 0.5.

```
num_classes = 20
# Define the CNN model
CNNmodel = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')  # Output layer with the number of classes
])
```

```
from keras.preprocessing.image import ImageDataGenerator
#preprocessing
train_datagen = ImageDataGenerator(rescale=1./255,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   rotation_range=45,
                                   horizontal_flip=True,
                                   vertical_flip=True,
                                   validation_split = .2)

test_datagen = ImageDataGenerator(rescale=1./255,
                                  validation_split = .2)
```

```
[21] # Extract features for each image
training_set = train_datagen.flow_from_directory(datasetDir,
                                         target_size=(150, 150),
                                                 batch_size=32,
                                       class_mode='categorical',
                                              subset='training')

validation_set = test_datagen.flow_from_directory(datasetDir,
                                           target_size=(150, 150),
                                                   batch_size=32,
                                         class_mode='categorical',
                                                  shuffle = False,
                                              subset='validation')
```

Loss & Accuracy Results:

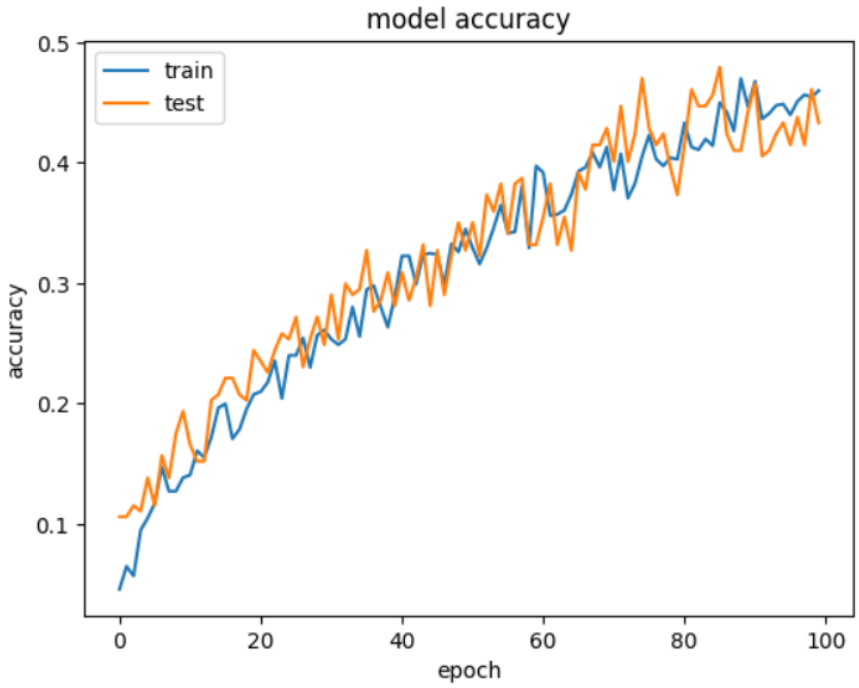
In this attempt, I decided to use 100 epochs, a batch-size of 32 and a steps per epoch equal to the length of the training set. When compiling the model, I used the 'adam' optimiser and 'categorical_crossentropy' for the loss function. I decided to use the specified batch size and epochs because it felt like a standard parameter and would not be too time consuming for the size of this dataset.

```
28/28 [==============================] - 6s 208ms/step - loss: 1.5932 - accuracy: 0.4699 - val_loss: 1.8046 - val_accuracy: 0.4101
Epoch 90/100
28/28 [==============================] - 5s 184ms/step - loss: 1.6698 - accuracy: 0.4464 - val_loss: 1.6786 - val_accuracy: 0.4424
Epoch 91/100
28/28 [==============================] - 6s 215ms/step - loss: 1.5863 - accuracy: 0.4676 - val_loss: 1.6373 - val_accuracy: 0.4654
Epoch 92/100
28/28 [==============================] - 6s 217ms/step - loss: 1.6373 - accuracy: 0.4364 - val_loss: 1.8223 - val_accuracy: 0.4055
Epoch 93/100
28/28 [==============================] - 5s 182ms/step - loss: 1.6864 - accuracy: 0.4408 - val_loss: 1.7386 - val_accuracy: 0.4101
Epoch 94/100
28/28 [==============================] - 6s 219ms/step - loss: 1.6144 - accuracy: 0.4475 - val_loss: 1.7912 - val_accuracy: 0.4240
Epoch 95/100
28/28 [==============================] - 5s 181ms/step - loss: 1.6464 - accuracy: 0.4487 - val_loss: 1.7261 - val_accuracy: 0.4332
Epoch 96/100
28/28 [==============================] - 5s 187ms/step - loss: 1.6175 - accuracy: 0.4397 - val_loss: 1.7753 - val_accuracy: 0.4147
Epoch 97/100
28/28 [==============================] - 6s 206ms/step - loss: 1.5815 - accuracy: 0.4509 - val_loss: 1.7951 - val_accuracy: 0.4378
Epoch 98/100
28/28 [==============================] - 5s 187ms/step - loss: 1.6187 - accuracy: 0.4565 - val_loss: 1.7924 - val_accuracy: 0.4147
Epoch 99/100
28/28 [==============================] - 5s 182ms/step - loss: 1.5972 - accuracy: 0.4542 - val_loss: 1.6373 - val_accuracy: 0.4608
Epoch 100/100
28/28 [==============================] - 5s 188ms/step - loss: 1.5933 - accuracy: 0.4598 - val_loss: 1.7439 - val_accuracy: 0.4332
```

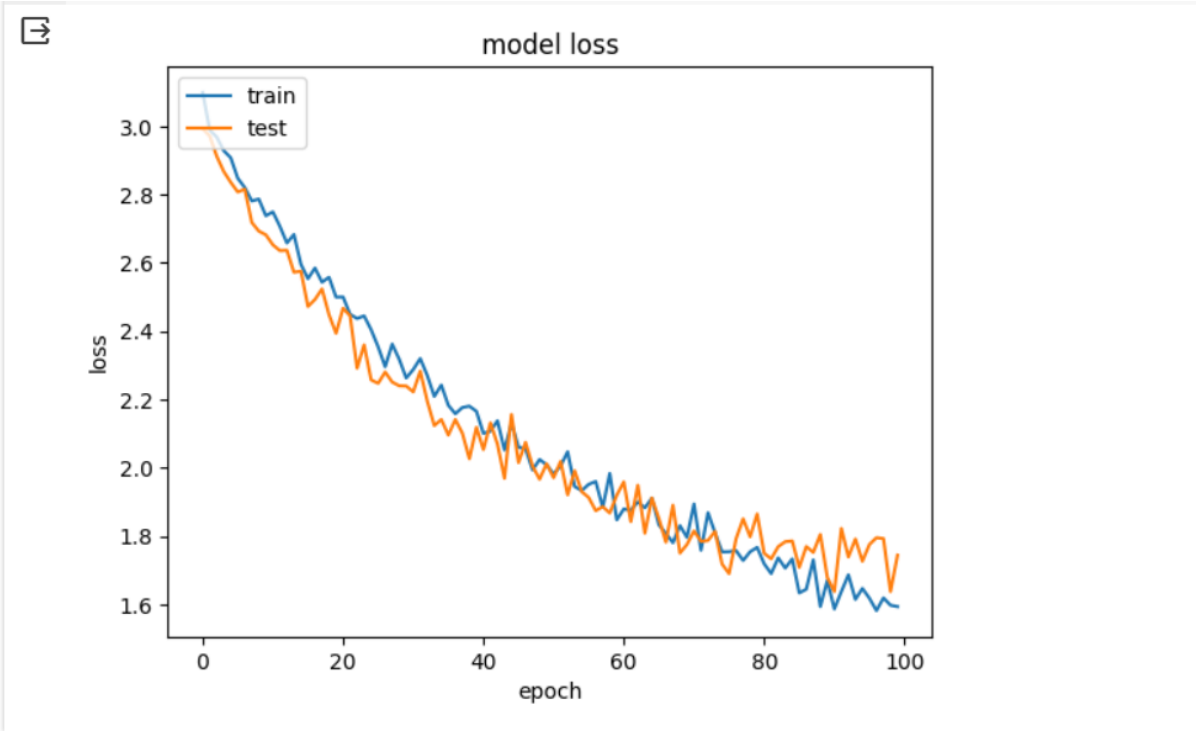
```
[26] score = CNNmodel.evaluate(validation_set, verbose=0)
     print('Test Loss:', score[0])
     print('Test Accuracy:', score[1])

     Test Loss: 1.7439130544662476
     Test Accuracy: 0.43317973613739014
```

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



The training and test accuracy turned out quite low. The training accuracy having a value of 46% and the test accuracy having a value of 43%. While this was quite disappointing, I was consoled with the fact that in this attempt I did not overfit the model.



In conjunction with the very low accuracy rate from earlier, the training and test loss consequently turned out quite high with a percent range of 160 – 170%. This basically told me that the model was performing very poorly. From this outcome, I had to go back to the drawing board and review the model's architecture.

CNN Sequential Model Attempt 2

In this attempt, I decided to change the model architecture. After watching Organisciak (2021), I decided that the model architecture needed to be more complex. I therefore created more convolutional and pooling layers within the system. I increased the convolutional layers from 3 to 5, with the first layer having 16 filters, the next three layers having 32 filters each and the last layer having 64 filters. The pooling layers also increased to 5, with each still having a pool size of (2,2). I also increased the amount of dropout layers from 1 to 5, with each layer having a dropout rate of 0.3.

```
[14] num_classes = 20
     # Define the CNN model
     CNNmodel2 = Sequential([
         Conv2D(16, (3, 3), activation='relu', input_shape=(150, 150, 3)),
         Dropout(0.3),
         MaxPooling2D((2, 2)),
         Conv2D(32, (3, 3), activation='relu'),
         Dropout(0.3),
         MaxPooling2D((2, 2)),
         Conv2D(32, (3, 3), activation='relu'),
         Dropout(0.3),
         MaxPooling2D((2, 2)),
         Conv2D(32, (3, 3), activation='relu'),
         Dropout(0.3),
         MaxPooling2D((2, 2)),
         Conv2D(64, (3, 3), activation='relu'),
         MaxPooling2D((2, 2)),
         Flatten(),
         Dense(128, activation='relu'),
         Dropout(0.3),
         Dense(num_classes, activation='softmax')  # Output layer with the number of classes
     ])
     # Compile the model
     CNNmodel2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Loss & Accuracy Results:

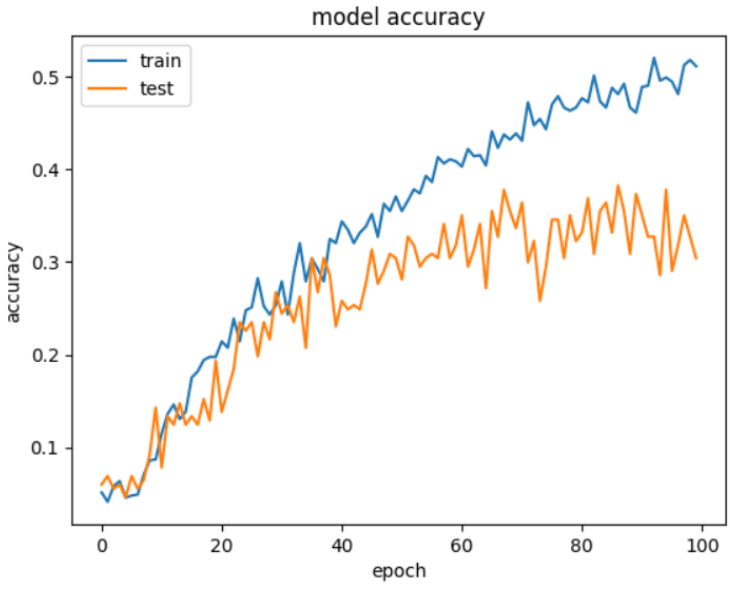
In this attempt, I used the same epochs, optimiser, batch size, and steps per epoch as in the first attempt.

```
[15] 28/28 [==============================] - 6s 214ms/step - loss: 1.5058 - accuracy: 0.4900 - val_loss: 2.0669 - val_accuracy: 0.3272
     Epoch 93/100
     28/28 [==============================] - 5s 179ms/step - loss: 1.5129 - accuracy: 0.5201 - val_loss: 2.1283 - val_accuracy: 0.3272
     Epoch 94/100
     28/28 [==============================] - 6s 219ms/step - loss: 1.4854 - accuracy: 0.4955 - val_loss: 2.1287 - val_accuracy: 0.2857
     Epoch 95/100
     28/28 [==============================] - 5s 180ms/step - loss: 1.4962 - accuracy: 0.4989 - val_loss: 2.0410 - val_accuracy: 0.3779
     Epoch 96/100
     28/28 [==============================] - 6s 218ms/step - loss: 1.5375 - accuracy: 0.4944 - val_loss: 2.1844 - val_accuracy: 0.2903
     Epoch 97/100
     28/28 [==============================] - 5s 181ms/step - loss: 1.4826 - accuracy: 0.4810 - val_loss: 2.1200 - val_accuracy: 0.3180
     Epoch 98/100
     28/28 [==============================] - 6s 201ms/step - loss: 1.4718 - accuracy: 0.5123 - val_loss: 2.0725 - val_accuracy: 0.3502
     Epoch 99/100
     28/28 [==============================] - 5s 183ms/step - loss: 1.4571 - accuracy: 0.5179 - val_loss: 2.0933 - val_accuracy: 0.3272
     Epoch 100/100
     28/28 [==============================] - 6s 219ms/step - loss: 1.4971 - accuracy: 0.5112 - val_loss: 2.1473 - val_accuracy: 0.3041
```

```
score1 = CNNmodel2.evaluate(validation_set, verbose=0)
print('Test Loss:', score1[0])
print('Test Accuracy:', score1[1])
```

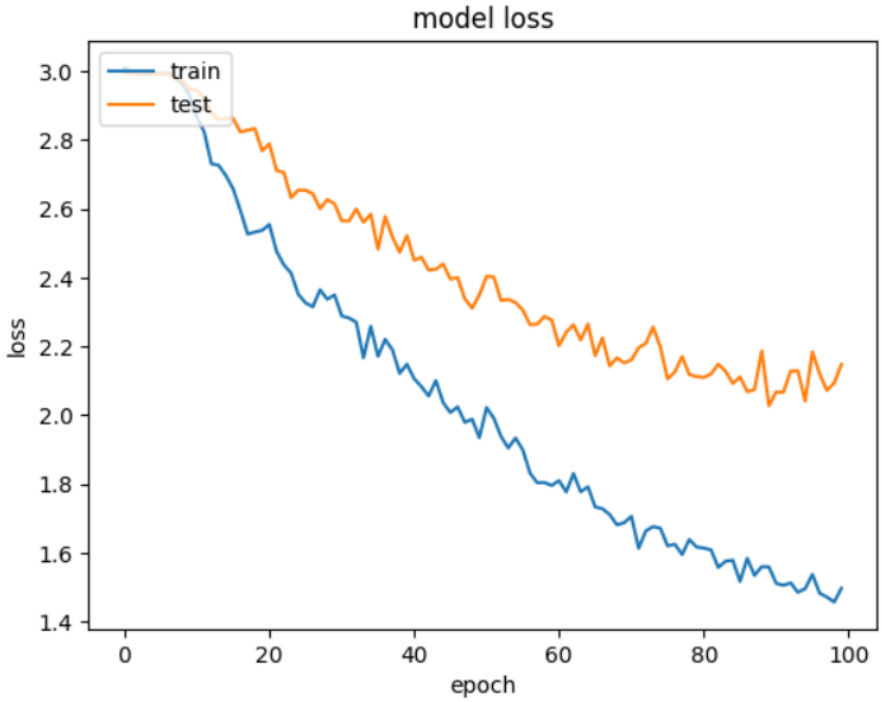
```
Test Loss: 2.147327423095703
Test Accuracy: 0.3041474521160126
```

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



In this attempt, the training and test accuracy also turned out quite low. The training accuracy having a value of 51.1% and the test accuracy having a value of 30.4%. In this attempt, the training accuracy was higher, while the test accuracy was lower than that of attempt 1. Therefore, I concluded that now not only was the model failing but it was also clearly overfitting.

```
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



In conjunction with the very low accuracy rate from earlier, the training and test loss consequently turned out quite high. The training loss had a value of 1.5, while the test loss had a value of 2.1. This basically told me that the model was completely failing and overfitting. From this outcome, I decided to go back to the drawing board and not only review the model's architecture but to also review some of the preprocessing techniques applied to it.

Sequential Model Attempt 3

In this attempt, I decided to modify the model's preprocessing techniques and architecture. For the preprocessing techniques, I disabled vertical flip, changed the rotation range from 45 to 30. I also included a width and height shift range of 0.1, a fill mode of 'nearest' and converted the images to grayscale. I also decided to change the image resizing to 160x160, there was no specific reason for this I just felt to experiment with the image size to see if it would have an impact on the model's accuracy.

```
from keras.preprocessing.image import ImageDataGenerator
#preprocessing
train_datagen1 = ImageDataGenerator(rescale=1./255,
                                    shear_range=0.2,
                                    zoom_range=0.2,
                                    rotation_range=30,
                                    width_shift_range=0.1,
                                    # randomly shift images vertically
                                    height_shift_range=0.1,
                                    horizontal_flip=True,
                                    vertical_flip=False,
                                    fill_mode='nearest',
                                    validation_split = 0.2,
                                    )

test_datagen1 = ImageDataGenerator(rescale=1./255,
                                   validation_split = 0.2)
```

```
[ ] # Extract features for each image
    training_set1 = train_datagen1.flow_from_directory(datasetDir,
                                        target_size=(160, 160),
                                                batch_size=32,
                                        color_mode='grayscale',
                                      class_mode='categorical',
                                            subset='training')

    validation_set1 = test_datagen1.flow_from_directory(datasetDir,
                                          target_size=(160, 160),
                                                   batch_size=32,
                                        class_mode='categorical',
                                          color_mode='grayscale',
                                                  shuffle = False,
                                             subset='validation')
```

```
Found 896 images belonging to 20 classes.
Found 217 images belonging to 20 classes.
```

For the model architecture, I decided to scale down from the complexity of attempt 2 and rebuild from attempt 1's architecture because attempt 1 had the higher accuracy and less overfitting of the both. This attempt has quite a similar architecture to attempt 1's, the main differences being:

i. I changed the input shape of the model to 160px by 160px by 1 channel.
ii. I added a Batch Normalization layer, after the third convolutional layer.
iii. I included three dropout layers with values of 0.1, 0.3 and 0.5.
iv. I added a learning rate of 0.0010.
v. I added an l2 kernel regularizer with the value of 0.09, to avoid chances of overfitting.
vi. Lastly, I increased the number of epochs to 120, to give the model more training time.

```
num_classes = 20
# Define the CNN model
CNNmodel2 = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(160, 160, 1)),
    Dropout(0.1),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dropout(0.3),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax', kernel_regularizer=regularizers.l2(0.09))  # Output layer with the number of classes
])
```

```
# Compile the model
CNNmodel2.compile(optimizer=Adam(learning_rate=0.0010), loss='categorical_crossentropy', metrics=['accuracy'])
```

```
#fit the model
history2 = CNNmodel2.fit(training_set1,
                         steps_per_epoch= len(training_set1),
                         epochs=120,
                         verbose=1,
                         validation_data=validation_set1,
                         validation_steps = len(validation_set1))
```

```
Epoch 92/120
28/28 [==============================] - 58s 2s/step - loss: 1.0593 - accuracy: 0.6897 - val_loss: 1.1584 - val_accuracy: 0.6544
Epoch 93/120
28/28 [==============================] - 56s 2s/step - loss: 1.0073 - accuracy: 0.6920 - val_loss: 0.8886 - val_accuracy: 0.7281
Epoch 94/120
28/28 [==============================] - 56s 2s/step - loss: 1.0498 - accuracy: 0.6886 - val_loss: 1.2569 - val_accuracy: 0.6590
Epoch 95/120
28/28 [==============================] - 56s 2s/step - loss: 0.9987 - accuracy: 0.7054 - val_loss: 1.0211 - val_accuracy: 0.6959
Epoch 96/120
28/28 [==============================] - 57s 2s/step - loss: 1.0457 - accuracy: 0.6853 - val_loss: 1.3453 - val_accuracy: 0.6544
Epoch 97/120
```

Loss & Accuracy Results

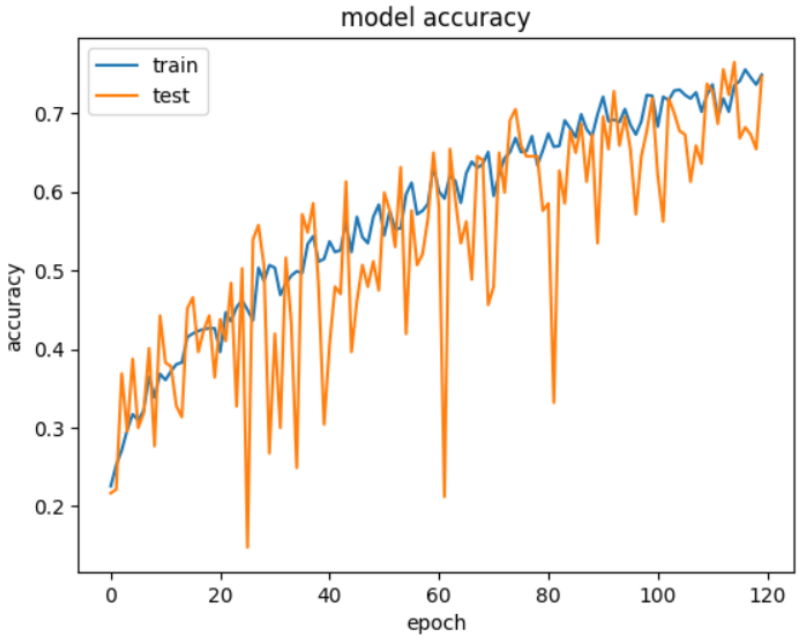
```
Epoch 116/120
28/28 [==============================] - 55s 2s/step - loss: 0.9068 - accuracy: 0.7411 - val_loss: 1.0109 - val_accuracy: 0.6682
Epoch 117/120
28/28 [==============================] - 57s 2s/step - loss: 0.8884 - accuracy: 0.7556 - val_loss: 1.0808 - val_accuracy: 0.6820
Epoch 118/120
28/28 [==============================] - 56s 2s/step - loss: 0.8399 - accuracy: 0.7455 - val_loss: 1.2195 - val_accuracy: 0.6728
Epoch 119/120
28/28 [==============================] - 56s 2s/step - loss: 0.8661 - accuracy: 0.7366 - val_loss: 1.2757 - val_accuracy: 0.6544
Epoch 120/120
28/28 [==============================] - 55s 2s/step - loss: 0.8863 - accuracy: 0.7489 - val_loss: 0.9240 - val_accuracy: 0.7465
```

```
score2 = CNNmodel2.evaluate(validation_set1, verbose=0)
print('Test Loss:', score1[0])
print('Test Accuracy:', score1[1])
```

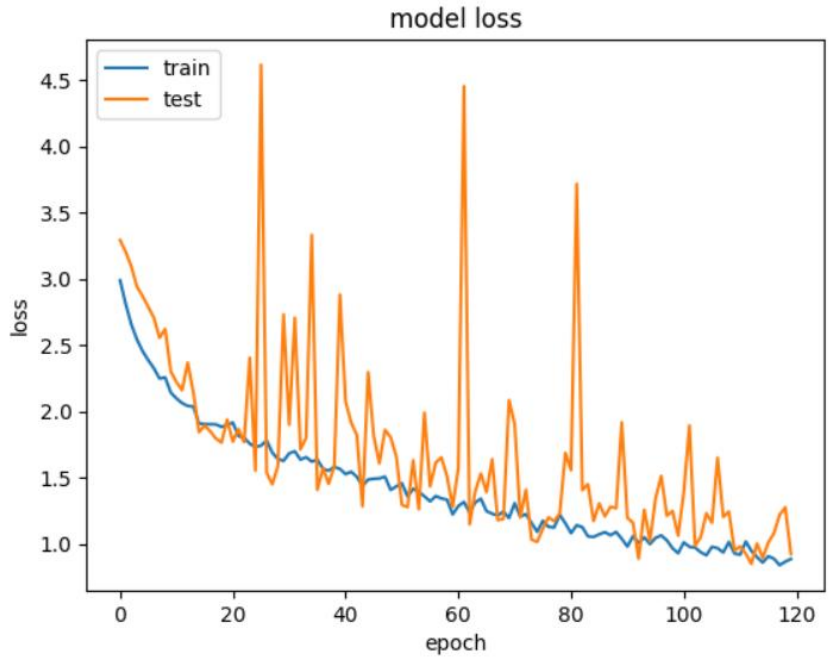
```
Test Loss: 0.9239554405212402
Test Accuracy: 0.7465437650680542
```

In this attempt, the train and test accuracy improved significantly than in the last two attempts. The train accuracy had a value of 74.9% while the test accuracy had a value of 74.7%. From the above I could see that while the model's accuracy may not necessarily be the highest it improved tremendously. From the graph below the test dataset has significant upwards and downward spikes in its accuracy, which could mean that the model is unstable.

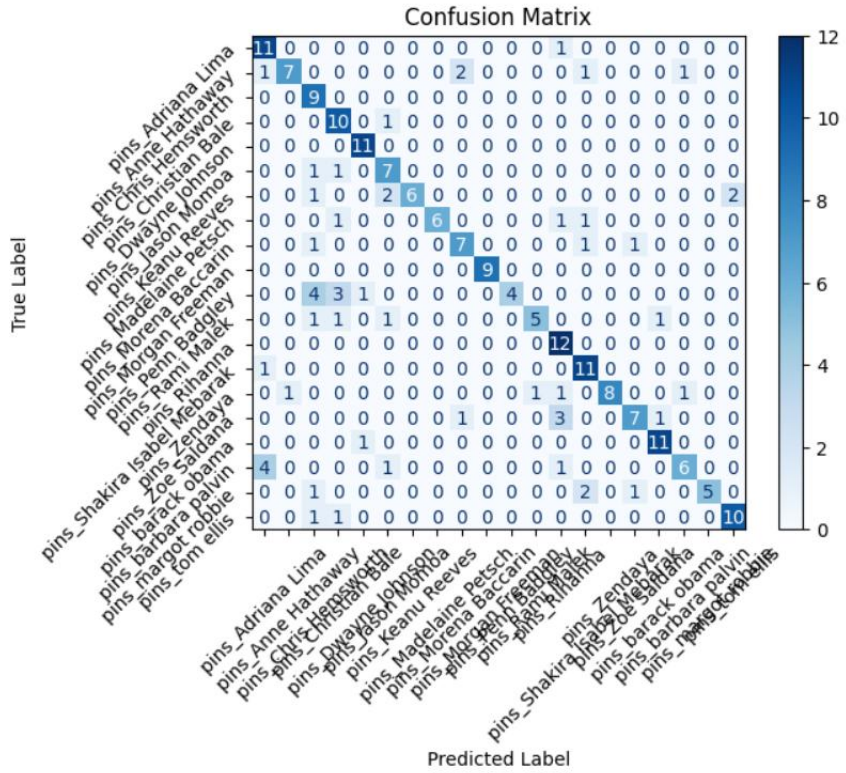
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```



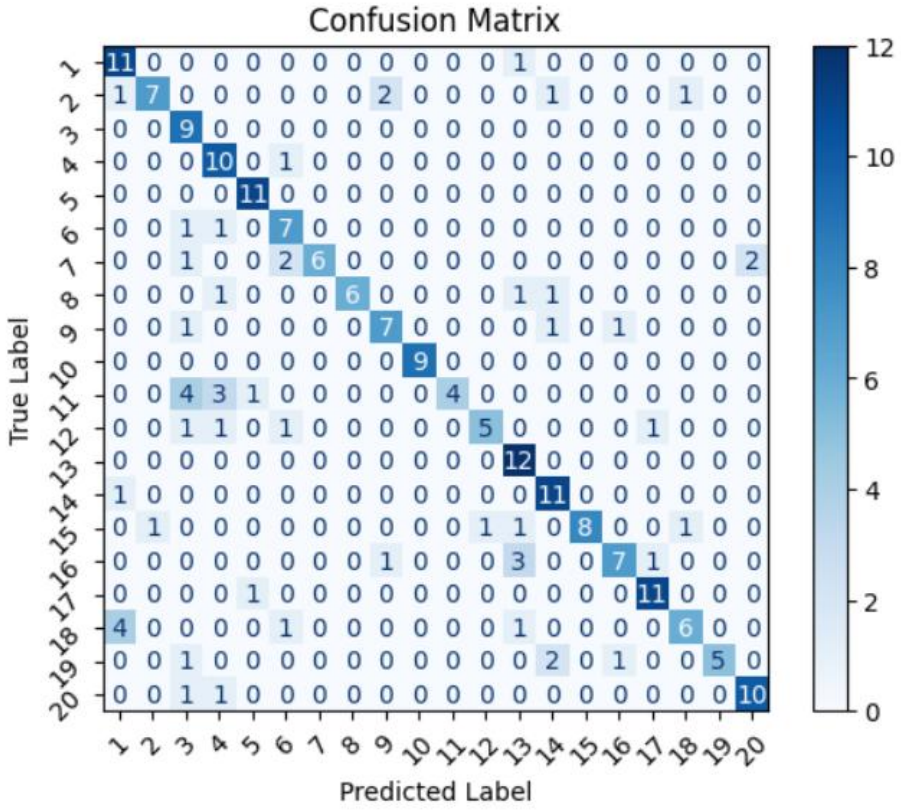
In conjunction with the higher test and training accuracy scores, the test and training loss significantly reduced. The test loss with a value of 0.92 and the training loss with a value of 0.89. Like the test accuracy, the test loss also had significant sharp upwards and downward spikes during training. Therefore, indicating that the model could still be improved further but due to time constraints I will be unable to keep fine tuning the model's parameters. However, as this attempt has produced the best results so far this will be the final code for this custom CNN sequential model.



<Figure size 1000x800 with 0 Axes>



<Figure size 1000x800 with 0 Axes>



In this last attempt, I will be fully evaluating the model by calculating its precision, recall, false positive and false negative rates, to see how well it is performing. The average precision is 80%, the average recall is 75%, the mean FPR is 1% and the mean FNR is 25%. The precision is the number of true positives divided by the number of true positives plus the number of false positives, while recall calculates the number of true positives divided by the number of true positives plus the number of false negatives. Therefore having a high precision means that an algorithm is performing properly.

The formula for precision is TP/ (TP + FP)

The formula for recall is TP/ (TP + FN)

```
Precision: [0.64705882 0.875      0.47368421 0.58823529 0.84615385 0.58333333
 1.         1.         0.7        1.         1.         0.83333333
 0.63157895 0.6875     1.         0.77777778 0.84615385 0.75
 1.         0.83333333]
Recall: [0.91666667 0.58333333 1.         0.90909091 1.         0.77777778
 0.54545455 0.66666667 0.7        1.         0.33333333 0.55555556
 1.         0.91666667 0.66666667 0.58333333 0.91666667 0.5
 0.55555556 0.83333333]
False Positive Rate (FPR): [0.02926829 0.00487805 0.04807692 0.03398058 0.00970874 0.02403846
 0.         0.         0.01449275 0.         0.         0.00480769
 0.03414634 0.02439024 0.         0.0097561  0.0097561  0.0097561
 0.         0.0097561 ]
False Negative Rate (FNR): [0.08333333 0.41666667 0.         0.09090909 0.         0.22222222
 0.45454545 0.33333333 0.3        0.         0.66666667 0.44444444
 0.         0.08333333 0.33333333 0.41666667 0.08333333 0.5
 0.44444444 0.16666667]
```

```
Average Precision: 0.8
Average Recall: 0.75
Mean False Positive Rate (FPR): 0.01
Mean False Negative Rate (FNR): 0.25
```

False positive rate (FPR) is the percentage of data that has been falsely identified as positive. The mean FPR for all 20 classes combined is 1% which is good. False negative rate (FNR), on the other hand, is the data that has been falsely classified as negative meaning that, they are positive. The mean FNR, across all classes, is 25%. This is quite high and should be improved significantly for a more reliable automated attendance system.

References

Celebrity Face Recognition - VGGFace Model ✨ (kaggle.com)

IBM (no date) *What are convolutional neural networks?*. Available at: https://www.ibm.com/topics/convolutional-neural-networks (Accessed: 29th April 2024).

Organisciak, D. (2021) *Introduction to Artificial Intelligence Lecture 5.7 Putting it all together*. 19 Feb. Available at: https://www.youtube.com/watch?v=E6yn_TxlzvI&list=PL7xLiPGAqEzSuB07HWECwY49d8mpWCRlk&index=7 (Accessed: 29 April 2021).

TensorFlow (2024) *Working with preprocessing layers: TensorFlow Core*. Available at: https://www.tensorflow.org/guide/keras/preprocessing_layers (Accessed: 20 April 2024).