# ECE 763 COMPUTER VISION

# SPRING 2022

# PROJECT 3

**SAHIL DESHPANDE**

**200375597**

**SHIVANI SHENDRE**

**200375416**

# INTRODUCTION

The objective of this project is to dive deep into application of Convolutional Neural Networks. **Convolutional neural networks (CNNs)** are a type of deep learning algorithm that has been used in a variety of real-world applications. CNNs can be trained to classify images, detect objects in an image, and even predict the next word in a sentence with incredible accuracy. We have implemented a CNN inspired by modifying the FaceNet architecture.

# DATA PREPARATION

• I have used the FDDB dataset (http://vis-www.cs.umass.edu/fddb/), which contains the annotations for 5171 faces in a set of 2845 images taken from the Faces in the Wild data set.

• The data set comes with a annotations folder which contains files with names: FDDB fold-xx.txt and FDDB-fold-xx-ellipseList.txt, where xx = {01, 02, ..., 10} represents the fold-index. Each line in the "FDDB-fold-xx.txt" file specifies a path to an image in the above-mentioned data set. The corresponding annotations are included in the file "FDDB-fold-xx-ellipseList.txt".

• Here, each face is denoted by: <major_axis_radius minor_axis_radius angle center_x center_y 1>.

• I extracted the face images from the coordinates of the rectangles created from the ellipses and resized to 20 x 20.

• I extracted the non-face images by checking the boundaries of the face rectangle coordinates for each image and considering the intersection over union criteria.

• In this way, I created 1000 training images for face and non-face each and 100 testing images for face and non-face.

# NETWORK ARCHITETCTURE

FaceNet architecture is kept as our baseline model, and we have modified the architecture to reduce the number of layers to suit our simple task of binary image classification. The FaceNet model was specifically made to detect human faces with well defined local features. This model also includes the use of embeddings, and its data is collected using the complex and convoluted MTCNN architecture. For the task of binary face classification, we do not require the depth that is provided by FaceNet. So, we changed the model by keeping just three convolutional layers instead of the nine convolutional layers.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 60, 60, 32)        896

 max_pooling2d (MaxPooling2D  (None, 30, 30, 32)       0
 )

 dropout (Dropout)           (None, 30, 30, 32)        0

 conv2d_1 (Conv2D)           (None, 30, 30, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 15, 15, 64)       0
 2D)

 dropout_1 (Dropout)         (None, 15, 15, 64)        0

 conv2d_2 (Conv2D)           (None, 15, 15, 86)        49622

 max_pooling2d_2 (MaxPooling  (None, 7, 7, 86)         0
 2D)

 dropout_2 (Dropout)         (None, 7, 7, 86)          0

 flatten (Flatten)           (None, 4214)              0

 dense (Dense)               (None, 84)                354060

 dense_1 (Dense)             (None, 2)                 170

=================================================================
Total params: 423,244
Trainable params: 423,244
Non-trainable params: 0
```

*Figure 1: Model 1 (without Batch Norm)*

# BABYSITTING THE DNN MODEL

- We first checked the loss and accuracy without preprocessing the data.
  `50/50 [==============================] - 9s 4ms/step - loss: 15.6600 - acc: 0.5144`
  We can see that the accuracy is not that great, and loss is quite high.

- Preprocessing the data by Normalization.
  We **normalized** the data to have zero mean and unit standard deviation, to improve the CNN performance. Data can be normalized by subtracting the dataset's mean from the input and dividing by the dataset's standard deviation, as shown in equation below.

$$z_{norm} = \frac{(z - \mu)}{\sqrt{\sigma^2 + \varepsilon}}$$

  This was done mathematically using the NumPy library. We used this normalized data with the above model and observed that the loss decreased when we evaluated the model **without** any **regularization**.
  `50/50 [==============================] - 0s 4ms/step - loss: 0.6468 - acc: 0.7619`

- Now we normalize the data **with regularization**. We see that the loss has increased.
  **50/50 [==============================] - 1s 4ms/step - loss: 3.5584 - acc: 0.5219**

- Keeping the regularization value constant (low - $10^{-6}$) we train the model for low learning rate and high learning rate values.
  With learning rate as $10^{-6}$ and training for 10 epochs we see that the validation loss hardly decreases and stays constant around 0.72.

```
80/80 [==============================] - 2s 8ms/step - loss: 0.7983 - acc: 0.5069 - val_loss: 0.7222 - val_acc: 0.4550
Epoch 2/10
80/80 [==============================] - 0s 5ms/step - loss: 0.7970 - acc: 0.5063 - val_loss: 0.7220 - val_acc: 0.4550
Epoch 3/10
80/80 [==============================] - 0s 5ms/step - loss: 0.8023 - acc: 0.5063 - val_loss: 0.7217 - val_acc: 0.4550
Epoch 4/10
80/80 [==============================] - 0s 5ms/step - loss: 0.7932 - acc: 0.5056 - val_loss: 0.7215 - val_acc: 0.4550
Epoch 5/10
80/80 [==============================] - 0s 5ms/step - loss: 0.7954 - acc: 0.5038 - val_loss: 0.7212 - val_acc: 0.4550
Epoch 6/10
80/80 [==============================] - 0s 5ms/step - loss: 0.7978 - acc: 0.5031 - val_loss: 0.7210 - val_acc: 0.4550
Epoch 7/10
80/80 [==============================] - 0s 5ms/step - loss: 0.7904 - acc: 0.5100 - val_loss: 0.7208 - val_acc: 0.4550
Epoch 8/10
80/80 [==============================] - 0s 5ms/step - loss: 0.7946 - acc: 0.5044 - val_loss: 0.7205 - val_acc: 0.4550
Epoch 9/10
80/80 [==============================] - 0s 5ms/step - loss: 0.7876 - acc: 0.5063 - val_loss: 0.7203 - val_acc: 0.4550
Epoch 10/10
80/80 [==============================] - 1s 9ms/step - loss: 0.7912 - acc: 0.5038 - val_loss: 0.7200 - val_acc: 0.4550
```

- With learning rate as $10^{6}$ and training for 10 epochs we see that the validation loss hardly decreases and becomes infinite as seen below –

```
80/80 [==============================] - 1s 9ms/step - loss: nan - acc: 0.4900 - val_loss: nan - val_acc: 0.5450
Epoch 2/10
80/80 [==============================] - 0s 5ms/step - loss: nan - acc: 0.4888 - val_loss: nan - val_acc: 0.5450
Epoch 3/10
80/80 [==============================] - 0s 5ms/step - loss: nan - acc: 0.4888 - val_loss: nan - val_acc: 0.5450
Epoch 4/10
80/80 [==============================] - 0s 5ms/step - loss: nan - acc: 0.4888 - val_loss: nan - val_acc: 0.5450
Epoch 5/10
80/80 [==============================] - 0s 5ms/step - loss: nan - acc: 0.4888 - val_loss: nan - val_acc: 0.5450
Epoch 6/10
80/80 [==============================] - 0s 6ms/step - loss: nan - acc: 0.4888 - val_loss: nan - val_acc: 0.5450
Epoch 7/10
80/80 [==============================] - 0s 5ms/step - loss: nan - acc: 0.4888 - val_loss: nan - val_acc: 0.5450
Epoch 8/10
80/80 [==============================] - 0s 5ms/step - loss: nan - acc: 0.4888 - val_loss: nan - val_acc: 0.5450
Epoch 9/10
80/80 [==============================] - 0s 5ms/step - loss: nan - acc: 0.4888 - val_loss: nan - val_acc: 0.5450
Epoch 10/10
80/80 [==============================] - 0s 5ms/step - loss: nan - acc: 0.4888 - val_loss: nan - val_acc: 0.5450
```

- We take a small amount of training data (20 samples) and use it to validate our model and check if its **overfitting** correctly. We train with this data for 200 epochs and get a validation accuracy of 100% which shows that the model is correctly overfitting as shown below-

```
50/50 [==============================] - 0s 6ms/step - loss: 0.0024 - acc: 1.0000 - val_loss: 0.0029 - val_acc: 1.0000
Epoch 185/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0022 - acc: 1.0000 - val_loss: 0.0032 - val_acc: 1.0000
Epoch 186/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0023 - acc: 1.0000 - val_loss: 0.0025 - val_acc: 1.0000
Epoch 187/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0034 - acc: 0.9987 - val_loss: 0.0034 - val_acc: 1.0000
Epoch 188/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0017 - acc: 1.0000 - val_loss: 0.0041 - val_acc: 1.0000
Epoch 189/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0032 - acc: 0.9994 - val_loss: 0.0045 - val_acc: 1.0000
Epoch 190/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0022 - acc: 1.0000 - val_loss: 0.0032 - val_acc: 1.0000
Epoch 191/200
50/50 [==============================] - 0s 7ms/step - loss: 0.0026 - acc: 1.0000 - val_loss: 0.0040 - val_acc: 1.0000
Epoch 192/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0031 - acc: 0.9987 - val_loss: 0.0039 - val_acc: 1.0000
Epoch 193/200
50/50 [==============================] - 0s 7ms/step - loss: 0.0046 - acc: 0.9987 - val_loss: 0.0025 - val_acc: 1.0000
Epoch 194/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0024 - acc: 0.9994 - val_loss: 0.0034 - val_acc: 1.0000
Epoch 195/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0026 - acc: 0.9994 - val_loss: 0.0032 - val_acc: 1.0000
Epoch 196/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0023 - acc: 1.0000 - val_loss: 0.0027 - val_acc: 1.0000
Epoch 197/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0028 - acc: 0.9994 - val_loss: 0.0025 - val_acc: 1.0000
Epoch 198/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0019 - acc: 1.0000 - val_loss: 0.0027 - val_acc: 1.0000
Epoch 199/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0019 - acc: 1.0000 - val_loss: 0.0022 - val_acc: 1.0000
Epoch 200/200
50/50 [==============================] - 0s 6ms/step - loss: 0.0021 - acc: 0.9994 - val_loss: 0.0034 - val_acc: 1.0000
```

- We do manual **hyperparameter tuning** by doing a course and fine search of learning rate and regularization. We train the model for 5 epochs for every value in the **coarse** and fine range and then display the top 10 hyperparameters with the best validation accuracy.
  For doing the coarse search we search for hyperparameters in the following range –
  reg = 10 ** np.random.uniform (-5,5)
  lr = 10 ** np.random.uniform (-3,-6)
  With this we observe –

```
val_acc: 0.5199999809265137 , lr: 1.4738918919127306e-05 , reg: 0.9073440893507245 ,( 68  / 100)
val_acc: 0.7724999785423279 , lr: 0.00010729969340846123 , reg: 0.00023282419833168334 ,( 69  / 100)
val_acc: 0.742500071525574 , lr: 0.0006164357760604788 , reg: 2.2988010040544093 ,( 70  / 100)
val_acc: 0.7825000286102295 , lr: 0.0002142470000234895 , reg: 1.6654168398424298e-05 ,( 71  / 100)
val_acc: 0.8050000071525574 , lr: 0.0005705687140230149 , reg: 0.029594443975503152 ,( 72  / 100)
val_acc: 0.5450000166893005 , lr: 8.407626780965233e-05 , reg: 37975.42300908583 ,( 73  / 100)
val_acc: 0.45500001311302185 , lr: 7.297706635274983e-06 , reg: 17499.55385779338 ,( 74  / 100)
val_acc: 0.45500001311302185 , lr: 7.375137297716653e-05 , reg: 8852.224444661666 ,( 75  / 100)
val_acc: 0.45500001311302185 , lr: 3.3600034458851364e-05 , reg: 17392.442985536494 ,( 76  / 100)
val_acc: 0.48750001192092896 , lr: 1.8730405941969777e-06 , reg: 3.567923273281098e-05 ,( 77  / 100)
val_acc: 0.44999998807907104 , lr: 2.449293953742685e-06 , reg: 192.25526013805344 ,( 78  / 100)
val_acc: 0.8525000214576721 , lr: 0.0008063147767040056 , reg: 0.004889055641128635 ,( 79  / 100)
val_acc: 0.47999998927116394 , lr: 1.3371087992027038e-06 , reg: 0.011605931384044824 ,( 80  / 100)
val_acc: 0.7074999809265137 , lr: 0.0002813550530487527 , reg: 0.020933470850765617 ,( 81  / 100)
val_acc: 0.5450000166893005 , lr: 0.0005590428189982627 , reg: 95777.1288440465 ,( 82  / 100)
val_acc: 0.45500001311302185 , lr: 0.0009269913429592536 , reg: 22.58044105813925 ,( 83  / 100)
val_acc: 0.45500001311302185 , lr: 3.9911821347366135e-06 , reg: 19171.895935982426 ,( 84  / 100)
val_acc: 0.7024999856948853 , lr: 8.921782911666337e-05 , reg: 4.4029836661459516e-05 ,( 85  / 100)
val_acc: 0.512499988079071 , lr: 3.0440252880376784e-06 , reg: 485.3859645522266 ,( 86  / 100)
val_acc: 0.5799999833106995 , lr: 3.5961174749896875e-05 , reg: 16.510600422152283 ,( 87  / 100)
val_acc: 0.5450000166893005 , lr: 0.000709257301308559 , reg: 95857.24242784899 ,( 88  / 100)
val_acc: 0.45500001311302185 , lr: 3.3224058875019886e-06 , reg: 4593.073508796367 ,( 89  / 100)
val_acc: 0.45500001311302185 , lr: 1.007065963520713e-06 , reg: 0.00030072709418380917 ,( 90  / 100)
val_acc: 0.5450000166893005 , lr: 1.3675738333708872e-06 , reg: 3.5381976836839145e-05 ,( 91  / 100)
val_acc: 0.4950000047683716 , lr: 1.5662230288088827e-05 , reg: 50.48129098478943 ,( 92  / 100)
val_acc: 0.637499988079071 , lr: 9.691312066990548e-05 , reg: 0.13619148232454384 ,( 93  / 100)
val_acc: 0.4925000071525574 , lr: 8.667290476309905e-06 , reg: 0.2088669907930854 ,( 94  / 100)
val_acc: 0.6825000047683716 , lr: 9.656953737327893e-05 , reg: 4.910491176622943 ,( 95  / 100)
val_acc: 0.45500001311302185 , lr: 0.0008132537627384742 , reg: 19.255621094295865 ,( 96  / 100)
val_acc: 0.5199999809265137 , lr: 8.38193543506855e-05 , reg: 1.517425167954188 ,( 97  / 100)
val_acc: 0.3075000047683716 , lr: 3.72735009977525e-06 , reg: 0.023956136592533907 ,( 98  / 100)
val_acc: 0.7825000286102295 , lr: 0.0001643502416074686 , reg: 5.8790809251771e-05 ,( 99  / 100)
```

The top 10 hyperparameters from this **coarse search** are –

```
array([[8.12500000e-01, 4.35944384e-04, 2.85211523e-04, 3.60000000e+01],
       [8.17499995e-01, 5.18892406e-04, 1.02245446e-05, 6.30000000e+01],
       [8.19999993e-01, 4.10192338e-04, 7.17655942e-04, 1.40000000e+01],
       [8.29999983e-01, 9.82801142e-04, 4.78842465e-01, 1.20000000e+01],
       [8.32499981e-01, 4.34129115e-04, 3.40424840e-01, 2.50000000e+01],
       [8.34999979e-01, 9.38471481e-04, 1.07440379e-01, 6.60000000e+01],
       [8.45000029e-01, 8.13066048e-04, 3.66316661e-05, 0.00000000e+00],
       [8.52500021e-01, 3.80907236e-04, 6.69349990e-04, 5.70000000e+01],
       [8.52500021e-01, 8.06314777e-04, 4.88905564e-03, 7.90000000e+01],
       [8.57500017e-01, 5.83857490e-04, 3.73120385e-03, 1.90000000e+01]])
```

- With the above 10 results we narrow down the range for **fine search** and the new updated range is-
  reg = 10 ** np.random.uniform (-2,-5)
  lr = 10 ** np.random.uniform (-5,-3)
  Each model is now trained for 20 epochs to do a finer search and find the optimal value.

```
val_acc: 0.795000166893005 , lr: 4.495358456038402e-05 , reg: 0.0007180743459986829 ,( 67  / 100)
val_acc: 0.41999998688697815 , lr: 2.1556052840381035e-05 , reg: 3.202368495242558e-05 ,( 68  / 100)
val_acc: 0.7599999904632568 , lr: 0.0003963467565923349 , reg: 0.0004267434700700191 ,( 69  / 100)
val_acc: 0.47999998927116394 , lr: 1.3130712562609449e-05 , reg: 0.00042521349042475205 ,( 70  / 100)
val_acc: 0.7799999713897705 , lr: 3.252482664144859e-05 , reg: 0.00022034724201214564 ,( 71  / 100)
val_acc: 0.6150000095367432 , lr: 1.1786437981811799e-05 , reg: 0.0004958666768928493 ,( 72  / 100)
val_acc: 0.6399999856948853 , lr: 5.20013209168688e-05 , reg: 3.093956369348763e-05 ,( 73  / 100)
val_acc: 0.8050000071525574 , lr: 0.0002481872182455383 , reg: 1.683032401067024e-05 ,( 74  / 100)
val_acc: 0.6899999976158142 , lr: 9.190882302590547e-05 , reg: 0.0005903473743926614 ,( 75  / 100)
val_acc: 0.8399999737739563 , lr: 0.0002920236469044793 , reg: 0.0011909416077649998 ,( 76  / 100)
val_acc: 0.550000011920929 , lr: 6.915875125986053e-05 , reg: 0.0035602187496192412 ,( 77  / 100)
val_acc: 0.5149999856948853 , lr: 1.6680815264625946e-05 , reg: 0.0001668323135873436 ,( 78  / 100)
val_acc: 0.6449999809265137 , lr: 1.983993169713841e-05 , reg: 5.56734315333063e-05 ,( 79  / 100)
val_acc: 0.5550000071525574 , lr: 1.0255575317452538e-05 , reg: 0.00021172317200224673 ,( 80  / 100)
val_acc: 0.875 , lr: 0.0008168932661990897 , reg: 0.0023611743108422744 ,( 81  / 100)
val_acc: 0.7200000286102295 , lr: 9.575593995316311e-05 , reg: 8.920949478667964e-05 ,( 82  / 100)
val_acc: 0.8199999928474426 , lr: 0.0002518411537205712 , reg: 0.005580640312498569 ,( 83  / 100)
val_acc: 0.4300000071525574 , lr: 2.858809197638533e-05 , reg: 7.747050301369705e-05 ,( 84  / 100)
val_acc: 0.7850000262260437 , lr: 5.6034438929175706e-05 , reg: 2.2318258405303068e-05 ,( 85  / 100)
val_acc: 0.6150000095367432 , lr: 3.4216475075306736e-05 , reg: 0.0006220388893062518 ,( 86  / 100)
val_acc: 0.5899999737739563 , lr: 2.9797370812948194e-05 , reg: 0.002138543906599834 ,( 87  / 100)
val_acc: 0.8299999833106995 , lr: 0.0003311811151888983 , reg: 0.00029882378732323805 ,( 88  / 100)
val_acc: 0.7699999809265137 , lr: 0.00018392590246258913 , reg: 0.0002867952587408853 ,( 89  / 100)
val_acc: 0.8299999833106995 , lr: 0.00023818324196693804 , reg: 1.3184349736961012e-05 ,( 90  / 100)
val_acc: 0.5649999976158142 , lr: 1.3770844396154703e-05 , reg: 0.0002880734351491244 ,( 91  / 100)
val_acc: 0.8799999952316284 , lr: 0.0007573187573801856 , reg: 0.0009478952468642813 ,( 92  / 100)
val_acc: 0.47999998927116394 , lr: 1.686411221367946e-05 , reg: 0.003304368053762058 ,( 93  / 100)
val_acc: 0.8700000047683716 , lr: 0.000464091303520227 , reg: 5.6518482520086546e-05 ,( 94  / 100)
val_acc: 0.6150000095367432 , lr: 3.783661162280764e-05 , reg: 0.005343451352200425 ,( 95  / 100)
val_acc: 0.8700000047683716 , lr: 0.0004650489957959484 , reg: 1.123788180699276e-05 ,( 96  / 100)
val_acc: 0.5199999809265137 , lr: 1.0041646551832493e-05 , reg: 0.00017707651961713512 ,( 97  / 100)
val_acc: 0.5699999928474426 , lr: 7.663352869392224e-05 , reg: 0.00027107105087144194 ,( 98  / 100)
val_acc: 0.8100000023841858 , lr: 0.0001250818631397665 , reg: 0.00890002697388621 ,( 99  / 100)
```

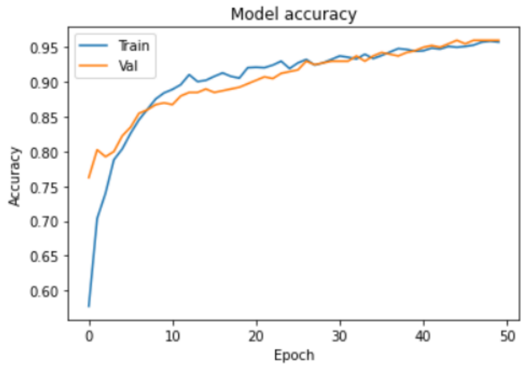
The top 10 hyperparameters from this fine search are –

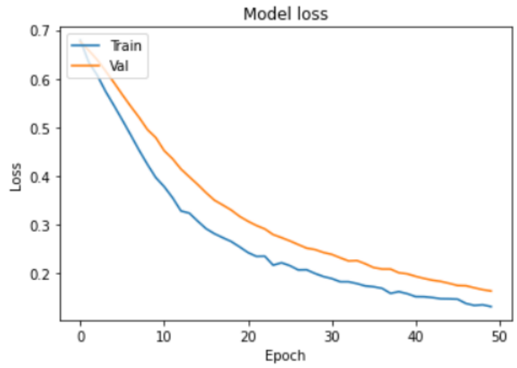
```
array([[8.70000005e-01, 7.19942596e-04, 3.53242653e-03, 4.70000000e+01],
       [8.70000005e-01, 4.65048996e-04, 1.12378818e-05, 9.60000000e+01],
       [8.75000000e-01, 5.69906394e-04, 3.87082972e-04, 3.50000000e+01],
       [8.75000000e-01, 8.16893266e-04, 2.36117431e-03, 8.10000000e+01],
       [8.79999995e-01, 7.57318757e-04, 9.47895247e-04, 9.20000000e+01],
       [8.84999990e-01, 5.87979422e-04, 2.00758238e-03, 1.50000000e+01],
       [8.84999990e-01, 9.06141970e-04, 1.75409793e-05, 3.60000000e+01],
       [8.89999986e-01, 4.86012092e-04, 7.54829106e-04, 2.50000000e+01],
       [8.89999986e-01, 3.44696623e-04, 1.45049166e-05, 4.90000000e+01],
       [9.10000026e-01, 3.71538027e-04, 3.96459921e-05, 2.30000000e+01]])
```

- With the coarse and fine search, we find the optimized hyperparameters below-
  **Learning rate - 3.71538027e-04**
  **Regularization - 3.96459921e-05**
- Now we train the model with the optimized values formed above for 50 epochs.

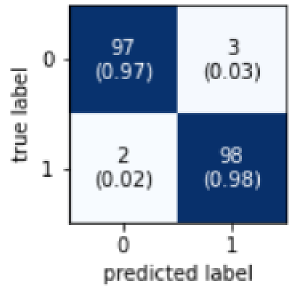
```
Epoch 35/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1747 - acc: 0.9337 - val_loss: 0.2202 - val_acc: 0.9375
Epoch 36/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1731 - acc: 0.9381 - val_loss: 0.2124 - val_acc: 0.9425
Epoch 37/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1698 - acc: 0.9431 - val_loss: 0.2095 - val_acc: 0.9400
Epoch 38/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1595 - acc: 0.9481 - val_loss: 0.2094 - val_acc: 0.9375
Epoch 39/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1630 - acc: 0.9469 - val_loss: 0.2018 - val_acc: 0.9425
Epoch 40/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1588 - acc: 0.9444 - val_loss: 0.1996 - val_acc: 0.9450
Epoch 41/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1528 - acc: 0.9450 - val_loss: 0.1945 - val_acc: 0.9500
Epoch 42/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1526 - acc: 0.9488 - val_loss: 0.1901 - val_acc: 0.9525
Epoch 43/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1509 - acc: 0.9475 - val_loss: 0.1865 - val_acc: 0.9500
Epoch 44/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1485 - acc: 0.9513 - val_loss: 0.1840 - val_acc: 0.9550
Epoch 45/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1482 - acc: 0.9500 - val_loss: 0.1801 - val_acc: 0.9600
Epoch 46/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1476 - acc: 0.9513 - val_loss: 0.1757 - val_acc: 0.9550
Epoch 47/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1388 - acc: 0.9531 - val_loss: 0.1751 - val_acc: 0.9600
Epoch 48/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1348 - acc: 0.9575 - val_loss: 0.1710 - val_acc: 0.9600
Epoch 49/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1361 - acc: 0.9588 - val_loss: 0.1670 - val_acc: 0.9600
Epoch 50/50
80/80 [==============================] - 0s 5ms/step - loss: 0.1323 - acc: 0.9575 - val_loss: 0.1644 - val_acc: 0.9600
```

- Accuracy and Loss Curves



- Accuracy on test data – **0.975**
- Confusion Matrix



- Model 2

  Now we modify the first model by adding batch normalization and Xavier Glorot initialization. Batch normalization (also known as batch norm) is a method used to make artificial neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling.
  The aim of weight initialization is to prevent layer activation outputs from exploding or vanishing during the course of a forward pass through a deep neural network. If either occurs, loss gradients will either be too large or too small to flow backwards beneficially, and the network will take longer to converge, if it is even able to do so at all.
  Xavier initialization sets a layer's weights to values chosen from a random uniform distribution that's bounded between-

$$\pm\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}$$

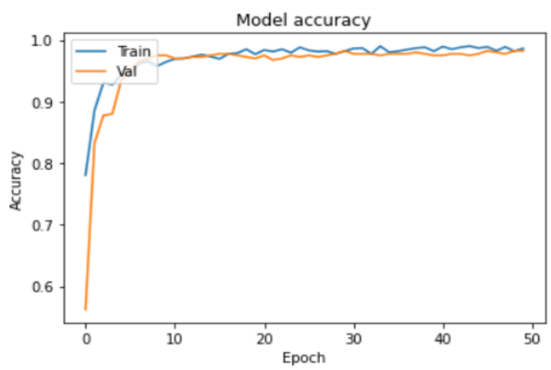
  where $n_i$ is the number of incoming network connections, or "fan-in," to the layer, and $n_{i+1}$ is the number of outgoing network connections from that layer, also known as the "fan-out."

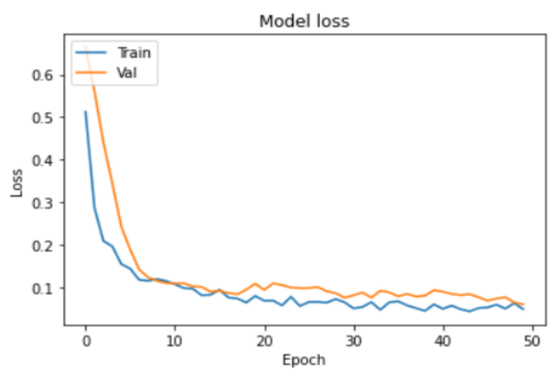
```
Model: "sequential_209"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_627 (Conv2D)         (None, 60, 60, 32)        896

 max_pooling2d_627 (MaxPooli  (None, 30, 30, 32)       0
 ng2D)

 batch_normalization (BatchN  (None, 30, 30, 32)       128
 ormalization)

 dropout_627 (Dropout)       (None, 30, 30, 32)        0

 conv2d_628 (Conv2D)         (None, 30, 30, 64)        18496

 max_pooling2d_628 (MaxPooli  (None, 15, 15, 64)       0
 ng2D)

 batch_normalization_1 (Batc  (None, 15, 15, 64)       256
 hNormalization)

 dropout_628 (Dropout)       (None, 15, 15, 64)        0

 conv2d_629 (Conv2D)         (None, 15, 15, 86)        49622

 max_pooling2d_629 (MaxPooli  (None, 7, 7, 86)         0
 ng2D)

 batch_normalization_2 (Batc  (None, 7, 7, 86)         344
 hNormalization)

 dropout_629 (Dropout)       (None, 7, 7, 86)          0

 flatten_209 (Flatten)       (None, 4214)              0

 batch_normalization_3 (Batc  (None, 4214)             16856
 hNormalization)

 dense_418 (Dense)           (None, 84)                354060

 batch_normalization_4 (Batc  (None, 84)               336
 hNormalization)

 dense_419 (Dense)           (None, 2)                 170

=================================================================
Total params: 441,164
Trainable params: 432,204
Non-trainable params: 8,960
```

- With the above model and optimal parameters found by coarse and fine search we train the model for 50 epochs.

```
Epoch 36/50
80/80 [==============================] - 1s 6ms/step - loss: 0.0669 - acc: 0.9819 - val_loss: 0.0786 - val_acc: 0.9775
Epoch 37/50
80/80 [==============================] - 0s 6ms/step - loss: 0.0571 - acc: 0.9844 - val_loss: 0.0843 - val_acc: 0.9775
Epoch 38/50
80/80 [==============================] - 0s 6ms/step - loss: 0.0508 - acc: 0.9869 - val_loss: 0.0781 - val_acc: 0.9800
Epoch 39/50
80/80 [==============================] - 0s 6ms/step - loss: 0.0441 - acc: 0.9881 - val_loss: 0.0799 - val_acc: 0.9775
Epoch 40/50
80/80 [==============================] - 0s 6ms/step - loss: 0.0597 - acc: 0.9819 - val_loss: 0.0929 - val_acc: 0.9750
Epoch 41/50
80/80 [==============================] - 0s 6ms/step - loss: 0.0492 - acc: 0.9894 - val_loss: 0.0894 - val_acc: 0.9750
Epoch 42/50
80/80 [==============================] - 1s 6ms/step - loss: 0.0567 - acc: 0.9850 - val_loss: 0.0840 - val_acc: 0.9775
Epoch 43/50
80/80 [==============================] - 0s 6ms/step - loss: 0.0482 - acc: 0.9881 - val_loss: 0.0816 - val_acc: 0.9775
Epoch 44/50
80/80 [==============================] - 1s 6ms/step - loss: 0.0432 - acc: 0.9900 - val_loss: 0.0835 - val_acc: 0.9750
Epoch 45/50
80/80 [==============================] - 0s 6ms/step - loss: 0.0509 - acc: 0.9869 - val_loss: 0.0767 - val_acc: 0.9775
Epoch 46/50
80/80 [==============================] - 0s 6ms/step - loss: 0.0524 - acc: 0.9887 - val_loss: 0.0681 - val_acc: 0.9825
Epoch 47/50
80/80 [==============================] - 0s 6ms/step - loss: 0.0590 - acc: 0.9825 - val_loss: 0.0731 - val_acc: 0.9800
Epoch 48/50
80/80 [==============================] - 1s 6ms/step - loss: 0.0496 - acc: 0.9887 - val_loss: 0.0765 - val_acc: 0.9775
Epoch 49/50
80/80 [==============================] - 0s 6ms/step - loss: 0.0624 - acc: 0.9819 - val_loss: 0.0641 - val_acc: 0.9825
Epoch 50/50
80/80 [==============================] - 0s 6ms/step - loss: 0.0482 - acc: 0.9862 - val_loss: 0.0598 - val_acc: 0.9825
```

- Loss and accuracy curve for Model 2



- Accuracy on test data – **0.975**
- Confusion Matrix