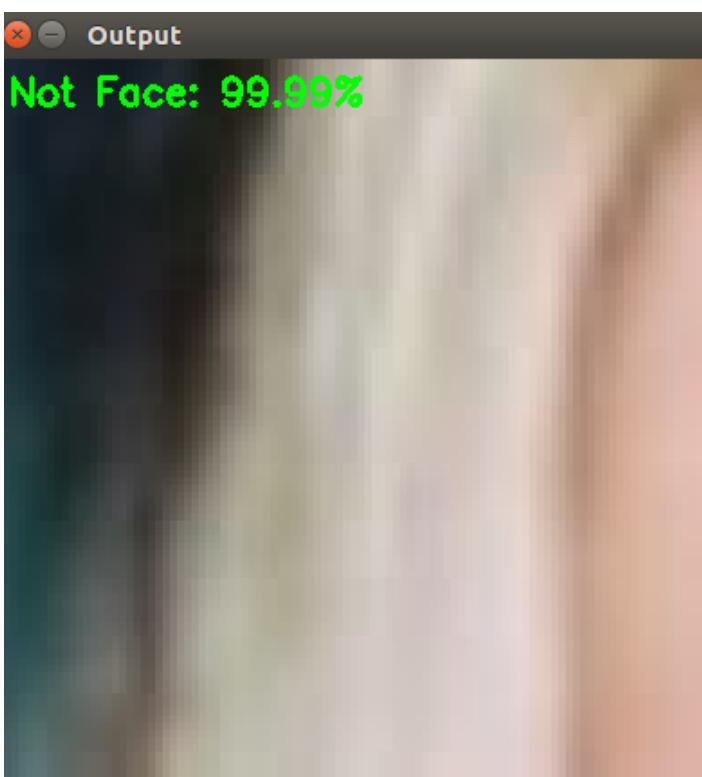
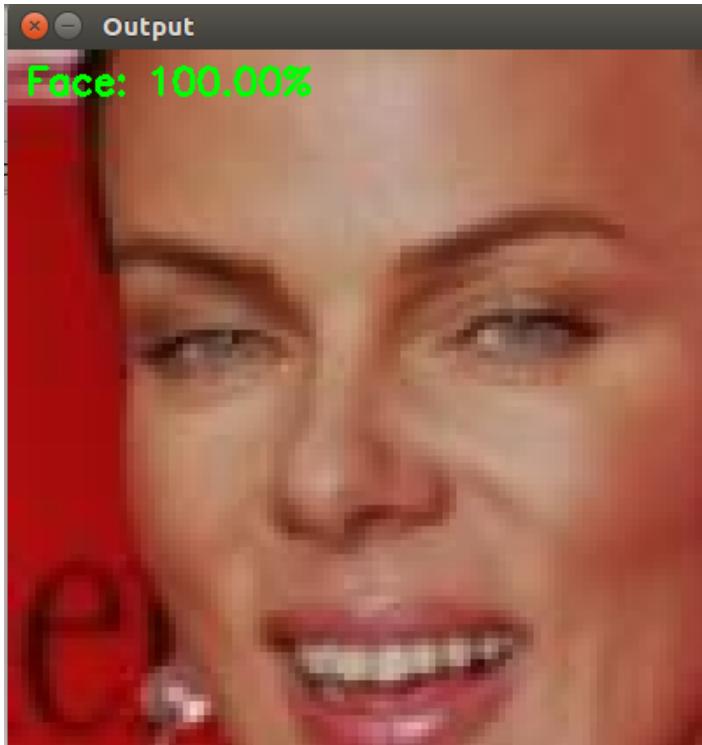


## PROJECT 2

SIDDHARTH GANESH

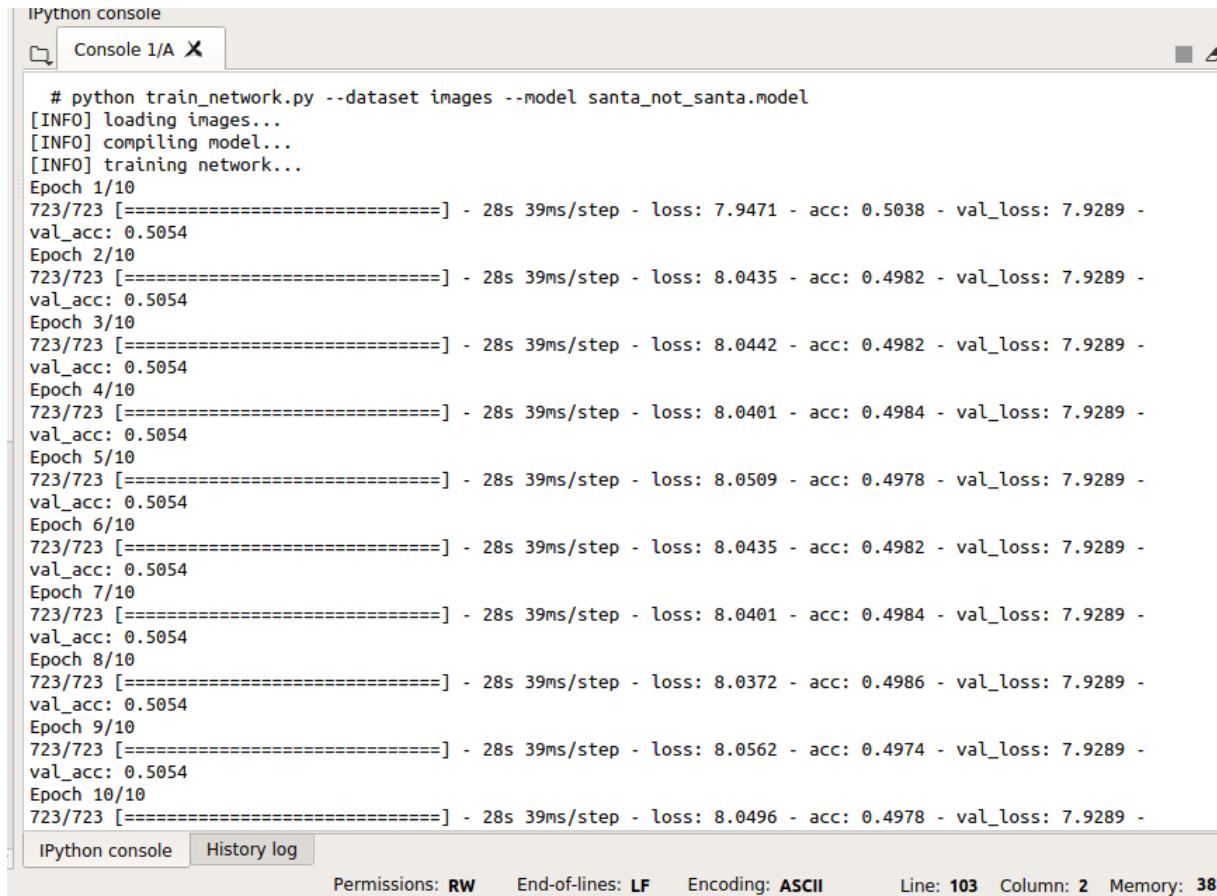
I have used a LeNet model designed using keras(backend Tensorflow) as my network architecture, with two convolution layers.

### Some Examples



## Preprocessing the data

Without Normalizing the data, the results are as shown below:



The screenshot shows an IPython console window titled "Console 1/A". The content of the console is a log of training output from a script named "train\_network.py". The log shows 10 epochs of training on 723 samples, with each epoch consisting of 723 steps. The output includes metrics like loss, accuracy, and validation loss. The accuracy remains relatively stable around 0.5054 throughout the process.

```
# python train_network.py --dataset images --model santa_not_santa.model
[INFO] loading images...
[INFO] compiling model...
[INFO] training network...
Epoch 1/10
723/723 [=====] - 28s 39ms/step - loss: 7.9471 - acc: 0.5038 - val_loss: 7.9289 -
val_acc: 0.5054
Epoch 2/10
723/723 [=====] - 28s 39ms/step - loss: 8.0435 - acc: 0.4982 - val_loss: 7.9289 -
val_acc: 0.5054
Epoch 3/10
723/723 [=====] - 28s 39ms/step - loss: 8.0442 - acc: 0.4982 - val_loss: 7.9289 -
val_acc: 0.5054
Epoch 4/10
723/723 [=====] - 28s 39ms/step - loss: 8.0401 - acc: 0.4984 - val_loss: 7.9289 -
val_acc: 0.5054
Epoch 5/10
723/723 [=====] - 28s 39ms/step - loss: 8.0509 - acc: 0.4978 - val_loss: 7.9289 -
val_acc: 0.5054
Epoch 6/10
723/723 [=====] - 28s 39ms/step - loss: 8.0435 - acc: 0.4982 - val_loss: 7.9289 -
val_acc: 0.5054
Epoch 7/10
723/723 [=====] - 28s 39ms/step - loss: 8.0401 - acc: 0.4984 - val_loss: 7.9289 -
val_acc: 0.5054
Epoch 8/10
723/723 [=====] - 28s 39ms/step - loss: 8.0372 - acc: 0.4986 - val_loss: 7.9289 -
val_acc: 0.5054
Epoch 9/10
723/723 [=====] - 28s 39ms/step - loss: 8.0562 - acc: 0.4974 - val_loss: 7.9289 -
val_acc: 0.5054
Epoch 10/10
723/723 [=====] - 28s 39ms/step - loss: 8.0496 - acc: 0.4978 - val_loss: 7.9289 -
```

After Normalizing the data, the results are as shown below:

The screenshot shows an IPython console window titled "Console 1/A". The console displays the output of a command to train a neural network. The output includes messages about loading images, compiling the model, and training the network. It provides a detailed log of 723 epochs, showing metrics like loss and accuracy at each step. The final accuracy is approximately 0.9944, and the validation loss is 0.0196.

```
# python train_network.py --dataset images --model santa_not_santa.model
[INFO] loading images...
[INFO] compiling model...
[INFO] training network...
Epoch 1/10
723/723 [=====] - 29s 40ms/step - loss: 0.0966 - acc: 0.9638 - val_loss: 0.0307 - val_acc: 0.9933
Epoch 2/10
723/723 [=====] - 29s 41ms/step - loss: 0.0465 - acc: 0.9864 - val_loss: 0.0508 - val_acc: 0.9842
Epoch 3/10
723/723 [=====] - 29s 41ms/step - loss: 0.0352 - acc: 0.9898 - val_loss: 0.0328 - val_acc: 0.9908
Epoch 4/10
723/723 [=====] - 29s 40ms/step - loss: 0.0318 - acc: 0.9899 - val_loss: 0.0579 - val_acc: 0.9848
Epoch 5/10
723/723 [=====] - 30s 41ms/step - loss: 0.0316 - acc: 0.9902 - val_loss: 0.0211 - val_acc: 0.9947
Epoch 6/10
723/723 [=====] - 30s 41ms/step - loss: 0.0282 - acc: 0.9920 - val_loss: 0.0229 - val_acc: 0.9935
Epoch 7/10
723/723 [=====] - 30s 41ms/step - loss: 0.0253 - acc: 0.9920 - val_loss: 0.0192 - val_acc: 0.9957
Epoch 8/10
723/723 [=====] - 30s 41ms/step - loss: 0.0247 - acc: 0.9920 - val_loss: 0.0257 - val_acc: 0.9930
Epoch 9/10
723/723 [=====] - 30s 41ms/step - loss: 0.0237 - acc: 0.9930 - val_loss: 0.0248 - val_acc: 0.9934
Epoch 10/10
723/723 [=====] - 30s 41ms/step - loss: 0.0186 - acc: 0.9944 - val_loss: 0.0196 -
```

As we can clearly see, normalized helped us in achieving a lower loss and we obtained a higher validation accuracy.

## BabySitting Process

Checked that the loss is reasonable, while disabling the regularization as shown below:

```
# first (and only) set of FC => RELU layers
model.add(Flatten())
model.add(Dense(500,kernel_regularizer=regularizers.l2(0.00)))
model.add(Activation("relu"))
```

The loss obtained after training the model is pretty low, as expected. But this might have a tendency to overfit the model.

```
ipython console
Console 1/A X
123/123 [=====] - 30s 41ms/step - loss: 0.0220 - acc: 0.9950 - val_loss: 0.0222 -
val_acc: 0.9944
Epoch 15/25
723/723 [=====] - 28s 39ms/step - loss: 0.0183 - acc: 0.9944 - val_loss: 0.0218 -
val_acc: 0.9957
Epoch 16/25
723/723 [=====] - 28s 39ms/step - loss: 0.0203 - acc: 0.9942 - val_loss: 0.0208 -
val_acc: 0.9949
Epoch 17/25
723/723 [=====] - 28s 39ms/step - loss: 0.0191 - acc: 0.9943 - val_loss: 0.0252 -
val_acc: 0.9925
Epoch 18/25
723/723 [=====] - 30s 42ms/step - loss: 0.0180 - acc: 0.9946 - val_loss: 0.0195 -
val_acc: 0.9948
Epoch 19/25
723/723 [=====] - 31s 43ms/step - loss: 0.0185 - acc: 0.9947 - val_loss: 0.0152 -
val_acc: 0.9956
Epoch 20/25
723/723 [=====] - 29s 40ms/step - loss: 0.0178 - acc: 0.9950 - val_loss: 0.0213 -
val_acc: 0.9948
Epoch 21/25
723/723 [=====] - 28s 39ms/step - loss: 0.0175 - acc: 0.9952 - val_loss: 0.0250 -
val_acc: 0.9942
Epoch 22/25
723/723 [=====] - 29s 40ms/step - loss: 0.0156 - acc: 0.9951 - val_loss: 0.0165 -
val_acc: 0.9952
Epoch 23/25
723/723 [=====] - 31s 43ms/step - loss: 0.0147 - acc: 0.9959 - val_loss: 0.0185 -
val_acc: 0.9957
Epoch 24/25
723/723 [=====] - 30s 42ms/step - loss: 0.0148 - acc: 0.9955 - val_loss: 0.0237 -
val_acc: 0.9938
Epoch 25/25
723/723 [=====] - 32s 44ms/step - loss: 0.0137 - acc: 0.9961 - val_loss: 0.0189 -
val_acc: 0.9961
```

So, we crank up the regularization to 1e3:

```
# initialize the model
print("[INFO] compiling model...")
model = LeNet.build(width=28, height=28, depth=3, classes=2, regular = 1e3)
opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
model.compile(loss="binary_crossentropy", optimizer=opt,
    metrics=["accuracy"])

# train the network
print("[INFO] training network...")
H = model.fit_generator(aug.flow(trainX, trainY, batch_size=BS),
    validation_data=(testX, testY), steps_per_epoch=len(trainX) // BS,
    epochs=EPOCHS, verbose=1)
```

And as expected, the loss goes up significantly, as expected.

(25th Epoch - loss 0.0137 vs loss 0.6517 )

```

IPython console
Console 1/A X
Epoch 11/25
723/723 [=====] - 34s 47ms/step - loss: 0.7719 - acc: 0.9370 - val_loss: 0.6533 -
val_acc: 0.9750
Epoch 12/25
723/723 [=====] - 34s 47ms/step - loss: 0.7844 - acc: 0.9437 - val_loss: 0.4618 -
val_acc: 0.9810
Epoch 13/25
723/723 [=====] - 34s 47ms/step - loss: 0.7901 - acc: 0.9483 - val_loss: 0.5605 -
val_acc: 0.9799
Epoch 14/25
723/723 [=====] - 34s 47ms/step - loss: 0.8790 - acc: 0.9443 - val_loss: 0.9990 -
val_acc: 0.9316
Epoch 15/25
723/723 [=====] - 34s 47ms/step - loss: 0.8018 - acc: 0.9531 - val_loss: 1.0580 -
val_acc: 0.9521
Epoch 16/25
723/723 [=====] - 34s 47ms/step - loss: 0.9402 - acc: 0.9531 - val_loss: 0.7367 -
val_acc: 0.9784
Epoch 17/25
723/723 [=====] - 34s 47ms/step - loss: 0.8192 - acc: 0.9592 - val_loss: 1.5411 -
val_acc: 0.9729
Epoch 18/25
723/723 [=====] - 34s 47ms/step - loss: 0.9359 - acc: 0.9587 - val_loss: 1.0640 -
val_acc: 0.9833
Epoch 19/25
723/723 [=====] - 34s 47ms/step - loss: 0.8251 - acc: 0.9611 - val_loss: 0.6477 -
val_acc: 0.9707
Epoch 20/25
723/723 [=====] - 36s 50ms/step - loss: 0.9164 - acc: 0.9576 - val_loss: 0.8041 -
val_acc: 0.9832
Epoch 21/25
723/723 [=====] - 35s 49ms/step - loss: 0.7127 - acc: 0.9662 - val_loss: 0.4610 -
val_acc: 0.9842
Epoch 22/25
723/723 [=====] - 35s 49ms/step - loss: 0.7941 - acc: 0.9609 - val_loss: 0.8281 -
val_acc: 0.9852
Epoch 23/25
723/723 [=====] - 35s 49ms/step - loss: 0.7179 - acc: 0.9631 - val_loss: 0.6102 -
val_acc: 0.9852
Epoch 24/25
723/723 [=====] - 35s 48ms/step - loss: 0.7975 - acc: 0.9635 - val_loss: 0.4729 -
val_acc: 0.9828
Epoch 25/25
723/723 [=====] - 35s 49ms/step - loss: 0.6517 - acc: 0.9668 - val_loss: 0.5106 -

```

IPython console History log

Permissions: RW End-of-lines: LF Encoding: ASCII Line: 36 Column: 12 Memory: 51 %

Next, we try to overfit a very small portion of the training data. I took the first 20 examples of the the face non face training data.

```

best_model = model.fit_generator(aug.flow(tiny_X, tiny_Y, batch_size=BS),
                                 validation_data=(tiny_X, tiny_Y), steps_per_epoch=len(trainX) // BS,
                                 epochs=EPOCHS, verbose=1)

```

And as expected, we overfitted the model over the data(as we can see that the accuracy is exactly 1) and the loss per epoch is very close to 0(in the magnitude of 1e-6).

IPython console

Console 1/A X

```
723/723 [=====] - 74s 102ms/step - loss: 0.0012 - acc: 0.9997 - val_loss: 8.8480e-07 -  
val_acc: 1.0000  
Epoch 15/25  
723/723 [=====] - 74s 102ms/step - loss: 1.6440e-04 - acc: 1.0000 - val_loss: 2.5518e-07  
- val_acc: 1.0000  
Epoch 16/25  
723/723 [=====] - 74s 102ms/step - loss: 3.3144e-05 - acc: 1.0000 - val_loss: 1.6308e-07  
- val_acc: 1.0000  
Epoch 17/25  
723/723 [=====] - 74s 102ms/step - loss: 1.3690e-05 - acc: 1.0000 - val_loss: 1.2293e-07  
- val_acc: 1.0000  
Epoch 18/25  
723/723 [=====] - 74s 102ms/step - loss: 1.8431e-05 - acc: 1.0000 - val_loss: 1.1097e-07  
- val_acc: 1.0000  
Epoch 19/25  
723/723 [=====] - 74s 102ms/step - loss: 8.2930e-06 - acc: 1.0000 - val_loss: 1.0967e-07  
- val_acc: 1.0000  
Epoch 20/25  
723/723 [=====] - 74s 102ms/step - loss: 5.9325e-06 - acc: 1.0000 - val_loss: 1.0960e-07  
- val_acc: 1.0000  
Epoch 21/25  
723/723 [=====] - 74s 102ms/step - loss: 0.0207 - acc: 0.9965 - val_loss: 1.3924e-04 -  
val_acc: 1.0000  
Epoch 22/25  
723/723 [=====] - 74s 102ms/step - loss: 0.0063 - acc: 0.9980 - val_loss: 1.2278e-06 -  
val_acc: 1.0000  
Epoch 23/25  
723/723 [=====] - 74s 102ms/step - loss: 0.0013 - acc: 0.9996 - val_loss: 1.0468e-06 -  
val_acc: 1.0000  
Epoch 24/25  
723/723 [=====] - 74s 102ms/step - loss: 0.0019 - acc: 0.9995 - val_loss: 1.5425e-07 -  
val_acc: 1.0000  
Epoch 25/25  
723/723 [=====] - 73s 101ms/step - loss: 1.7214e-04 - acc: 1.0000 - val_loss: 1.0960e-07
```

IPython console History log

Permissions: RW End-of-lines: LF Encoding: ASCII Line: 101 Column: 72 Memory: 45 %

Now, we start with a small learning rate and regularization that makes the loss go down. I set the value to 1e-8 as shown below:

```
36 EPOCHS = 25
37 #INIT_LR = 1e-3
38 INIT_LR = 1e-8
39 BS = 32
40
```

As we can see, the loss is barely changing from 0.6. The model improves very slowly.

IPython console

Console 1/A X

```
val_acc: 0.7000
Epoch 15/25
723/723 [=====] - 74s 102ms/step - loss: 0.6588 - acc: 0.6982 - val_loss: 0.6502 -
val_acc: 0.7000
Epoch 16/25
723/723 [=====] - 74s 102ms/step - loss: 0.6545 - acc: 0.6992 - val_loss: 0.6453 -
val_acc: 0.7000
Epoch 17/25
723/723 [=====] - 74s 102ms/step - loss: 0.6497 - acc: 0.6998 - val_loss: 0.6404 -
val_acc: 0.7000
Epoch 18/25
723/723 [=====] - 74s 102ms/step - loss: 0.6451 - acc: 0.6999 - val_loss: 0.6357 -
val_acc: 0.7000
Epoch 19/25
723/723 [=====] - 74s 102ms/step - loss: 0.6410 - acc: 0.7000 - val_loss: 0.6311 -
val_acc: 0.7000
Epoch 20/25
723/723 [=====] - 74s 102ms/step - loss: 0.6368 - acc: 0.7000 - val_loss: 0.6267 -
val_acc: 0.7000 ETA: 11s - loss: 0.6371 - acc: 0.7000
Epoch 21/25
723/723 [=====] - 74s 102ms/step - loss: 0.6325 - acc: 0.7000 - val_loss: 0.6223 -
val_acc: 0.7000 ETA: 51s - loss: 0.6339 - acc: 0.7000
21/723 [=====>.....] - ETA: 51s - loss: 0.6339 - acc: 0.7000
Epoch 22/25
723/723 [=====] - 74s 102ms/step - loss: 0.6284 - acc: 0.7000 - val_loss: 0.6181 -
val_acc: 0.7000
Epoch 23/25
723/723 [=====] - 74s 102ms/step - loss: 0.6246 - acc: 0.7000 - val_loss: 0.6139 -
val_acc: 0.7000
Epoch 24/25
723/723 [=====] - 73s 102ms/step - loss: 0.6209 - acc: 0.7000 - val_loss: 0.6099 -
val_acc: 0.7000
Epoch 25/25
723/723 [=====] - 74s 102ms/step - loss: 0.6168 - acc: 0.7000 - val_loss: 0.6060 -
```

IPython console History log

Permissions: RW End-of-lines: LF Encoding: ASCII Line: 38 Column: 15 Memory: 44 %

We then try the other extreme by setting the learning rate to 1e8 as shown below:



```
36 EPOCHS = 25
37 #INIT_LR = 1e-3
38 INIT_LR = 1e8
39 BS = 32
40
41 # initialize the data and labels
print("TFNFO! loading images...")
```

As we set the learning rate too high, we can see that the loss is exploding and we get a couple of NaNs as shown. This is an indication of high learning rate.

IPython console

Console 1/A X

```

123/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00
Epoch 15/25
723/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00
Epoch 16/25
723/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00
Epoch 17/25
723/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00
Epoch 18/25
723/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00
Epoch 19/25
723/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00
Epoch 20/25
723/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00ETA: 1s - loss: nan - acc: 0.0000e+00
Epoch 21/25
723/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00
Epoch 22/25
723/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00
Epoch 23/25
723/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00
Epoch 24/25
723/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00
Epoch 25/25
723/723 [=====] - 74s 102ms/step - loss: nan - acc: 0.0000e+00 - val_loss: nan - val_acc: 0.0000e+00

```

IPython console History log

Permissions: RW End-of-lines: LF Encoding: ASCII Line: 38 Column: 13 Memory: 36 %

## Hyperparameter Optimization

First we run a coarse search of 10 epochs each to get a rough estimate of the range in which the regularization and learning rate might be in. It is as shown below:

```

125 BS = 100
126 import random
127 max_count = 100
128 for count in xrange(max_count):
129     reg = 10**random.uniform(-5,5)
130     INIT_LR = 10**random.uniform(-3,-6)
131
132     print("Iteration:",count+1)
133     model = LeNet.build(width=28, height=28, depth=3, classes=2, regular = reg)
134     opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
135     model.compile(loss="binary_crossentropy", optimizer=opt,metrics=["accuracy"])
136
137
138     best_model = model.fit_generator(aug.flow(trainX, trainY, batch_size=BS),
139                                     validation_data=(testX, testY), steps_per_epoch=len(trainX) // BS,
140                                     epochs=EPOCHS, verbose=0)
141
142     print("lr:",INIT_LR, " , reg : ",reg, " , val_acc",best_model.history["val_acc"][-1])

```

As we can clearly see, we get a good accuracy at Iteration 9 and 22, where the learning rate is around 1e-5 - 0.0005 and the regularization is close to 0 for a good accuracy

```

IPython console
Console 1/A X
('Iteration:', 7)
('lr:', 3.975261318042757e-06, ', reg :', 12830.79223027565, ', val_acc', 0.79136969641698707)
('Iteration:', 8)
('lr:', 0.0005519761456471404, ', reg :', 41.82951745790528, ', val_acc', 0.97667488419856685)
('Iteration:', 9)
('lr:', 3.76076980288304e-05, ', reg :', 0.0007022648477261859, ', val_acc', 0.99054037290362784)
('Iteration:', 10)
('lr:', 3.1884324349395e-06, ', reg :', 1.1766277341727391, ', val_acc', 0.82596864033907025)
('Iteration:', 11)
('lr:', 2.6972208949364964e-05, ', reg :', 86.65927898385604, ', val_acc', 0.9284696133315633)
('Iteration:', 12)
('lr:', 2.0570203671240224e-06, ', reg :', 0.3841553397564044, ', val_acc', 0.77659712021440763)
('Iteration:', 13)
('lr:', 0.0003772601080123386, ', reg :', 33907.7283353357, ', val_acc', 0.85499546424209016)
('Iteration:', 14)
('lr:', 3.3022589340137414e-06, ', reg :', 0.5699166038169003, ', val_acc', 0.87300764661665109)
('Iteration:', 15)
('lr:', 9.294258928121007e-06, ', reg :', 0.0006309660395134875, ', val_acc', 0.98613450905503397)
('Iteration:', 16)
('lr:', 4.60380666621023e-06, ', reg :', 28.965903075190273, ', val_acc', 0.83724245057909907)
('Iteration:', 17)
('lr:', 2.044587304967513e-06, ', reg :', 4060.795626465704, ', val_acc', 0.68446287560280672)
('Iteration:', 18)
('lr:', 8.510077609451499e-06, ', reg :', 1397.9122388973183, ', val_acc', 0.93792924892412688)
('Iteration:', 19)
('lr:', 0.00029331542303576976, ', reg :', 7.412603159130866, ', val_acc', 0.95710768793258583)
('Iteration:', 20)
('lr:', 6.394576247456415e-06, ', reg :', 0.0004529056886041691, ', val_acc', 0.98419075737009398)
('Iteration:', 21)
('lr:', 1.2132452290274089e-06, ', reg :', 0.004938781662287656, ', val_acc', 0.89736944066284074)
('Iteration:', 22)
('lr:', 0.00013784357640294478, ', reg :', 5.282610606268209, ', val_acc', 0.97278736221430384)
('Iteration:', 23)
('lr:', 0.00016528256517021965, ', reg :', 0.14152613760069166, ', val_acc', 0.98043282304354495)

```

IPython console History log

Permissions: RW End-of-lines: LF Encoding: ASCII Line: 142 Column: 28 Memory: 57 %

So we run a fine search by changing the random initialization of the learning rates and regularizations to a tighter range, as shown below:

```

124 #%%%
125 BS = 100
126 import random
127 max_count = 100
128 for count in xrange(max_count):
129     reg = 10**random.uniform(-5,0)
130     INIT_LR = 10**random.uniform(-3,-5)
131
132     print("Iteration:",count+1)
133     model = LeNet.build(width=28, height=28, depth=3, classes=2, regular = reg)
134     opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
135     model.compile(loss="binary_crossentropy", optimizer=opt,metrics=["accuracy"])
136
137
138     best_model = model.fit_generator(aug.flow(trainX, trainY, batch_size=BS),
139                                     validation_data=(testX, testY), steps_per_epoch=len(trainX) // BS,
140                                     epochs=EPOCHS, verbose=0)
141
142     print("lr:",INIT_LR,", reg :",reg,", val_acc",best_model.history["val_acc"][9])

```

```
Console 1/A X
In [2]:
In [2]: BS = 100
...: import random
...: max_count = 100
...: for count in xrange(max_count):
...:     reg = 10*random.uniform(-5,0)
...:     INIT_LR = 10**random.uniform(-3,-5)
...:
...:     print("Iteration:",count+1)
...:     model = LeNet.build(width=28, height=28, depth=3, classes=2, regular = reg)
...:     opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
...:     model.compile(loss="binary_crossentropy", optimizer=opt,metrics=["accuracy"])
...:
...:     best_model = model.fit_generator(aug.flow(trainX, trainY, batch_size=BS),
...:         validation_data=(testX, testY), steps_per_epoch=len(trainX) // BS,
...:         epochs=EPOCHS, verbose=0)
...:
...:     print("lr:",INIT_LR," reg :",reg," val_acc",best_model.history["val_acc"][-1])
('Iteration:', 1)
('lr:', 3.500566611943586e-05, ' reg :', 0.007515148634483148, ' val_acc', 0.98432034271066482)
('Iteration:', 2)
('lr:', 5.0080212706296446e-05, ' reg :', 0.0053831379360996185, ' val_acc', 0.97900739847045803)
('Iteration:', 3)
('lr:', 0.0003440585665491032, ' reg :', 0.0100164728573973, ' val_acc', 0.99390955537932202)
('Iteration:', 4)
('lr:', 5.548466077443096e-05, ' reg :', 0.0005476873508135724, ' val_acc', 0.99196579747671476)
('Iteration:', 5)
('lr:', 0.00012818713633733626, ' reg :', 0.00043415881083411234, ' val_acc', 0.99403914006336902)
('Iteration:', 6)
('lr:', 2.2758605405276103e-05, ' reg :', 0.0004071562667332725, ' val_acc', 0.98224699935162951)
('Iteration:', 7)
('lr:', 0.0001027600099840097, ' reg :', 3.8772153394819444e-05, ' val_acc', 0.99183621356504881)
('Iteration:', 8)
('lr:', 2.4092703782676697e-05, ' reg :', 0.0012194777307564365, ' val_acc', 0.98924453378696897)
('Iteration:', 9)
('lr:', 1.2448666818678908e-05, ' reg :', 5.892312876281708e-05, ' val_acc', 0.9880782778095949)
('Iteration:', 10)
('lr:', 0.00029561935650421824, ' reg :', 0.019782043009400396, ' val_acc', 0.9923545492117124)
('Iteration:', 11)
Traceback (most recent call last):
```

IPython console History log

Permissions: RW End-of-lines: LF Encoding: ASCII Line: 133 Column: 79 Memory: 42 %

As we can see the best accuracy is obtained for:

Learning rate: 0.0001281871

Regularization: 0.0004341588

Accuracy obtained : 99.4%