Project 03 - Babysitting the training of a DNN

Mayuresh Manoj Sardesai
mmsardes@ncsu.edu

I. Introduction

Babysitting the training of a DNN is important as the final network performance depends a lot on how you set the hyper-parameters, how you initialize the weights, for how long you train the network etc. You don't want to over-train, or over-fit the model as it would lead to loss of generalization. General steps followed while training a DNN are

1) Pre-process the Data and select Data Augmentation schemes.
2) Choose architecture of the network
3) Double check if the loss is reasonable
   - Try training with a low learning rate
   - Try training with zero regularization
   - Start with small regularization and find the learning rate that makes the loss go down.
4) Hyper-parameter optimization
   - Coarse search
   - Fine Search

These methods will be explained in the section below.

II. Babysitting the training of the network

Babysitting has been done on the CIFAR data-set extensively, which is downloaded from the keras datasets. The same process is repeated for the Face data with the same model. The parameters for the face data will differ.

A. *Preprocessing the data and select data Augmentation*

A basic preprocessing operation is done by subtracting the mean and dividing by the standard deviation over the dataset. This is basically zero centering the data. This makes the model less sensitive to small changes in weights and makes it easier to optimize. This enables better weight update and easier gradient flow. It also reduces the effect of outliers to some extent as the range of the data is restricted.

```
In [53]:  1 model.compile(optimizer=optimizers.SGD(learning_rate=LEARNING_RATE),loss='sparse_categorical_crossentropy',metrics=['sparse_
          2 model.evaluate(X_train,y_train)

50000/50000 [==============================] - 9s 178us/sample - loss: 37.8876 - sparse_categorical_accuracy: 0.0812

Out[53]: [37.887600780029295, 0.08122]
```

Fig. 1: Initial Epoch loss - without preprocessing

```
37500/37500 [==============================] - 4s 110us/sample - loss: 2.3173 - sparse_categorical_accuracy: 0.1051s - loss: 2.
3194 - sparse_categorical_accuracy: 0. - ETA: 5s - loss: 2.3183 - sparse_categorical_ - ETA: 1s - loss: 2.3176 - sparse_categor
ica

Out[12]: [2.317313319498698, 0.10514667]
```

Fig. 2: Initial Epoch loss - with preprocessing

As we can see in the above figures, the initial loss is itself is bought down when we apply preprocessing. If we check for more epochs, if the data is preprocessed the loss goes down quicker as compared to when we have not. It is especially observed in case of validation loss.

B. *Choose the network architecture*

Here I have chosen a basic CNN inspired from LeNet with 3 layers. 32, 64, 128 filters in the three layers. All of them being 3x3 filters with a stride of 1. All the convolutional blocks are followed by 2x2 maxpool blocks. In the end the result is flattened and connected to a FCN with one hidden layer with 128 neurons and a softmax layer with 10 units as there are 10 classes in CIFAR. All the layers have 'relu' activations. The model is created using the Sequential API in Keras [1].

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 128)               262272
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 356,810
Trainable params: 356,810
Non-trainable params: 0
_________________________________________________________________
```

Fig. 3: Simple Convolutional Network

C. *Double checking to check if the loss is reasonable*

1) Initial loss with no regularization: The initial loss with no regularization is around 2.3 as expected for 10 classes for a categorical cross entropy loss.

```
37500/37500 [==============================] - 4s 110us/sample - loss: 2.3173 - sparse_categorical_accuracy: 0.1051s - loss: 2.
3194 - sparse_categorical_accuracy: 0. - ETA: 5s - loss: 2.3183 - sparse_categorical_ - ETA: 1s - loss: 2.3176 - sparse_categor
ica

Out[12]: [2.317313319498698, 0.10514667]
```

Fig. 4: Initial Epoch loss - with no regularization

Sanity check: If we crank up the regularization we get a higher loss as expected.

```
37500/37500 [==============================] - 3s 68us/sample - loss: 369745.2781 - sparse_categorical_accuracy: 0.1180s - los
s: 369745.2783 - sparse_catego

Out[13]: [369745.27807666664, 0.118]
```

Fig. 5: Initial Epoch loss - with no regularization

2) Try to overfit a small portion of data: We try to overfit the small portion of the data as a sanity check with no regularization and vanilla SGD optimizer. This ensures that our model is actually learning and the loss is going down. We verify this using the training accuracy (It should reach 100% theoretically). We can also check the curves as they would exhibit a huge gap between the validation and training curves. We can also see that the model reaches a very small loss as we reach the last epoch. That's a good sanity check.

```
In [15]:
1 model=create_model(num_classes,L2_REG)
2 model.compile(optimizer=optimizers.SGD(learning_rate=LEARNING_RATE),loss='sparse_categorical_crossentropy',metrics=['sparse_
3 history = model.fit(X_tiny,y_tiny,batch_size=2,epochs=EPOCHS, verbose=1)

20/20 [==============================] - 0s 1ms/sample - loss: 0.2027 - sparse_categorical_accuracy: 1.0000
Epoch 92/100
20/20 [==============================] - 0s 1ms/sample - loss: 0.1965 - sparse_categorical_accuracy: 1.0000
Epoch 93/100
20/20 [==============================] - 0s 1ms/sample - loss: 0.1868 - sparse_categorical_accuracy: 1.0000
Epoch 94/100
20/20 [==============================] - 0s 2ms/sample - loss: 0.1814 - sparse_categorical_accuracy: 1.0000
Epoch 95/100
20/20 [==============================] - 0s 1ms/sample - loss: 0.1733 - sparse_categorical_accuracy: 1.0000
Epoch 96/100
20/20 [==============================] - 0s 1ms/sample - loss: 0.1674 - sparse_categorical_accuracy: 1.0000
Epoch 97/100
20/20 [==============================] - 0s 2ms/sample - loss: 0.1617 - sparse_categorical_accuracy: 1.0000
Epoch 98/100
20/20 [==============================] - 0s 1ms/sample - loss: 0.1551 - sparse_categorical_accuracy: 1.0000
Epoch 99/100
20/20 [==============================] - 0s 1ms/sample - loss: 0.1501 - sparse_categorical_accuracy: 1.0000
Epoch 100/100
20/20 [==============================] - 0s 1ms/sample - loss: 0.1453 - sparse_categorical_accuracy: 1.0000
```

Fig. 6: Initial Epoch loss - with no regularization

3) Start with small regularization and find learning rate that makes loss go down: We start with a small regularization and try to find a learning rate that makes the loss go down. We can observe here that the loss is not going down. As we can see in Fig 7 The learning rate is so small that the model is actually not learning! The validation accuracy is still around 10% owing to the random guess from the softmax classifier.

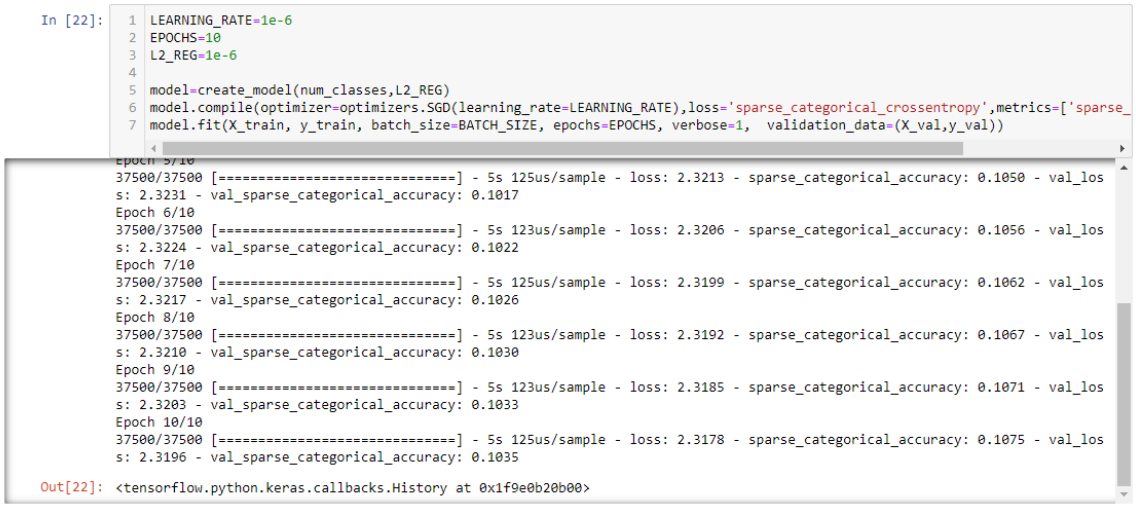


Fig. 7: Small learning rate

We can now try a higher learning rate. We set the learning rate to 1e6. As we can see in Fig. 8 The cost is so high that the loss is exploding. It is because the weights are updated by a huge number. In turn leading to the saturation of the activation units.

Fig. 8: Large learning rate

Trying to find the optimum learning rate is very important. We can keep trying values for which the loss will go down, but that is a trial and error method. Here a learning rate is tried which made the loss go down initially but as we move to the next epoch, the loss explodes as shown in Fig. 9.

```
In [32]:
LEARNING_RATE=0.4825
EPOCHS=10
L2_REG=1e-6

model=create_model(num_classes,L2_REG)
model.compile(optimizer=optimizers.SGD(learning_rate=LEARNING_RATE),loss='sparse_categorical_crossentropy',metrics=['sparse_
model.fit(X_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS, verbose=1,  validation_data=(X_val,y_val))
```

```
Train on 37500 samples, validate on 12500 samples
Epoch 1/10
37500/37500 [==============================] - 5s 146us/sample - loss: 2.1148 - sparse_categorical_accuracy: 0.2051 - val_los
s: 1.8623 - val_sparse_categorical_accuracy: 0.3003
Epoch 2/10
37500/37500 [==============================] - 5s 133us/sample - loss: nan - sparse_categorical_accuracy: 0.1827 - val_loss:
nan - val_sparse_categorical_accuracy: 0.0992
Epoch 3/10
37500/37500 [==============================] - 5s 134us/sample - loss: nan - sparse_categorical_accuracy: 0.1003 - val_loss:
nan - val_sparse_categorical_accuracy: 0.0992
Epoch 4/10
37500/37500 [==============================] - 5s 136us/sample - loss: nan - sparse_categorical_accuracy: 0.1003 - val_loss:
nan - val_sparse_categorical_accuracy: 0.0992
Epoch 5/10
37500/37500 [==============================] - 5s 131us/sample - loss: nan - sparse_categorical_accuracy: 0.1003 - val_loss:
nan - val_sparse_categorical_accuracy: 0.0992
Epoch 6/10
37500/37500 [==============================] - 5s 132us/sample - loss: nan - sparse_categorical_accuracy: 0.1003 - val_loss:
nan - val_sparse_categorical_accuracy: 0.0992
```

Fig. 9: Cost still exploding

We can adopt a cross validation strategy where we try different learning rates and regularizations in a fixed range and choose the combination that gives the best performance on the model.

*D. **Hyper-paramater tuning***

To find the optimum learning rate and regularization to apply, we first go through a few epochs to get a rough idea of how the params work and then move on to the second stage, where we have a longer run time and a finer search. These are kind of grid search techniques where we do a restricted search in a grid of the specified values for the hyper-paramaters and try to find the optimum hyper-parameters. [2]

1) Coarse search: For coarse search we take a random value of the learning rate between 1e-3 and 1e-6 and a random value for regularization between 1e-6 and 1

```
In [33]:
val_acc=[]
lrs=[]
l2_regs=[]
for i in tqdm(range(100)):
    lr = 10**np.random.uniform(-3,-6)
    l2_reg = 10**np.random.uniform(-5, 1)
    model = create_model(num_classes,l2_reg)
    model.compile(optimizer=optimizers.SGD(learning_rate=lr),loss='sparse_categorical_crossentropy',metrics=['sparse_categor
    history = model.fit(X_train,y_train,batch_size=BATCH_SIZE,epochs=5, verbose=1,validation_data=(X_val,y_val))
    val_acc.append(history.history['val_sparse_categorical_accuracy'][-1])
    lrs.append(lr)
    l2_regs.append(l2_reg)
    print(history.history['val_sparse_categorical_accuracy'][-1], lr, l2_reg)
```

```
37500/37500 [==============================] - 10s 277us/sample - loss: 2.2943 - sparse_categorical_accuracy: 0.1235 - val_lo
ss: 2.2750 - val_sparse_categorical_accuracy: 0.1352
Epoch 2/5
37500/37500 [==============================] - 8s 224us/sample - loss: 2.2587 - sparse_categorical_accuracy: 0.1588 - val_los
s: 2.2427 - val_sparse_categorical_accuracy: 0.1782
Epoch 3/5
37500/37500 [==============================] - 8s 226us/sample - loss: 2.2229 - sparse_categorical_accuracy: 0.2040 - val_los
s: 2.2032 - val_sparse_categorical_accuracy: 0.2296
Epoch 4/5
37500/37500 [==============================] - 8s 224us/sample - loss: 2.1776 - sparse_categorical_accuracy: 0.2541 - val_los
s: 2.1527 - val_sparse_categorical_accuracy: 0.2642
Epoch 5/5
37500/37500 [==============================] - 9s 227us/sample - loss: 2.1209 - sparse_categorical_accuracy: 0.2790 - val_los
s: 2.0924 - val_sparse_categorical_accuracy: 0.2890
100%|██████████| 100/100 [1:01:11<00:00, 36.71s/it]
0.28904 0.0001900990978557527 1.3054353604719474e-05
```

Fig. 10: Coarse searching strategy

```
In [36]:
logs[:,top10]
```

```
Out[36]: array([[4.37680006e-01, 4.35600013e-01, 4.16079998e-01, 3.89519989e-01,
                 3.69760007e-01, 3.60960007e-01, 3.56319994e-01, 3.55679989e-01,
                 3.52560014e-01, 3.45360011e-01],
                [9.69503562e-04, 9.64651669e-04, 7.81986128e-04, 6.58787173e-04,
                 5.93430096e-04, 4.75272040e-04, 4.26807095e-04, 4.69700905e-04,
                 3.83862909e-04, 3.84184800e-04],
                [6.37996507e-03, 3.95851376e-05, 7.01391913e-03, 7.79256949e-03,
                 2.70961933e-02, 7.53769585e-05, 1.46452043e-04, 7.70228784e-03,
                 1.19792868e-03, 8.24862764e-04]])
```

Fig. 11: Top 10 performers - coarse search

As seen in Fig. 11 we can see the top 10 performers from the coarse search strategy, we can see that the learning rates are restricted within the range 1e-3 and 1e-4 and the regularization between 1e-5 and 1e-2. Thus, we narrow down the range and perform a finer search for a longer time, i.e. increase the number of epochs.

2) Fine Search: We restrict the fine search to the smaller range and train for 15 epochs.

```
In [37]:
val_acc=[]
lrs=[]
l2_regs=[]
for i in tqdm(range(30)):
    lr = 10**np.random.uniform(-3,-4)
    l2_reg = 10**np.random.uniform(-6, -2)
    model = create_model(num_classes,l2_reg)
    model.compile(optimizer=optimizers.SGD(learning_rate=lr),loss='sparse_categorical_crossentropy',metrics=['sparse_categor
    history = model.fit(X_train,y_train,batch_size=BATCH_SIZE,epochs=15, verbose=1,validation_data=(X_val,y_val))
    val_acc.append(history.history['val_sparse_categorical_accuracy'][-1])
    lrs.append(lr)
    l2_regs.append(l2_reg)
    print(history.history['val_sparse_categorical_accuracy'][-1], lr, l2_reg)
```

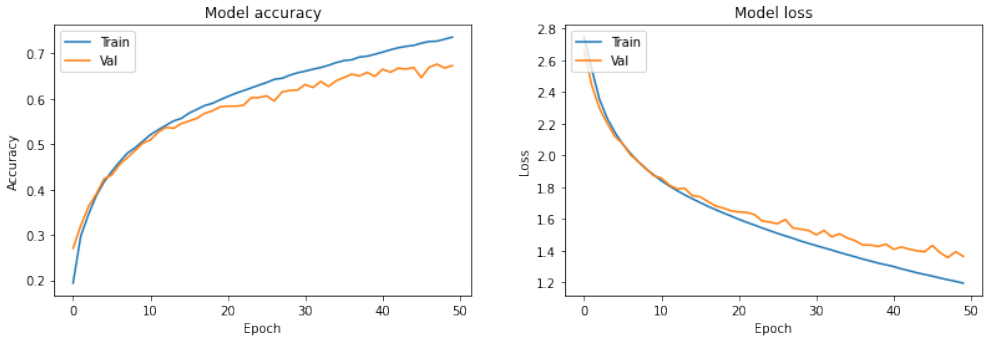
```
37500/37500 [==============================] - 10s 266us/sample - loss: 1.7446 - sparse_categorical_accuracy: 0.5032 - val_lo
ss: 1.7565 - val_sparse_categorical_accuracy: 0.4939
Epoch 12/15
37500/37500 [==============================] - 9s 250us/sample - loss: 1.7150 - sparse_categorical_accuracy: 0.5155 - val_los
s: 1.7236 - val_sparse_categorical_accuracy: 0.5026
Epoch 13/15
37500/37500 [==============================] - 9s 252us/sample - loss: 1.6875 - sparse_categorical_accuracy: 0.5244 - val_los
s: 1.6975 - val_sparse_categorical_accuracy: 0.5166
Epoch 14/15
37500/37500 [==============================] - 10s 253us/sample - loss: 1.6627 - sparse_categorical_accuracy: 0.5334 - val_lo
ss: 1.6705 - val_sparse_categorical_accuracy: 0.5238
Epoch 15/15
37500/37500 [==============================] - 9s 250us/sample - loss: 1.6378 - sparse_categorical_accuracy: 0.5423 - val_los
s: 1.6608 - val_sparse_categorical_accuracy: 0.5340
100%|██████████| 30/30 [1:09:56<00:00, 139.87s/it]

0.534 0.0007184675234522362 0.0009539288681134822
```

Fig. 12: Fine search

```
In [40]:
logs[:,top4]

Out[40]:
array([[5.47999978e-01, 5.44160008e-01, 5.39680004e-01, 5.38800001e-01],
       [8.89732394e-04, 7.88809210e-04, 7.14903532e-04, 6.92725369e-04],
       [1.35175950e-03, 4.25579137e-06, 1.74973376e-06, 3.71546369e-06]])

In [41]:
lr, l2_reg = logs[1,top4[0]],logs[2,top4[0]]
lr, l2_reg

Out[41]:
(0.0008897323938345441, 0.0013517594993773326)
```

Fig. 13: Fine search top performers

Now we will train the model using the top performer found using the top performer from the fine search. The training curves of the model can be seen below.

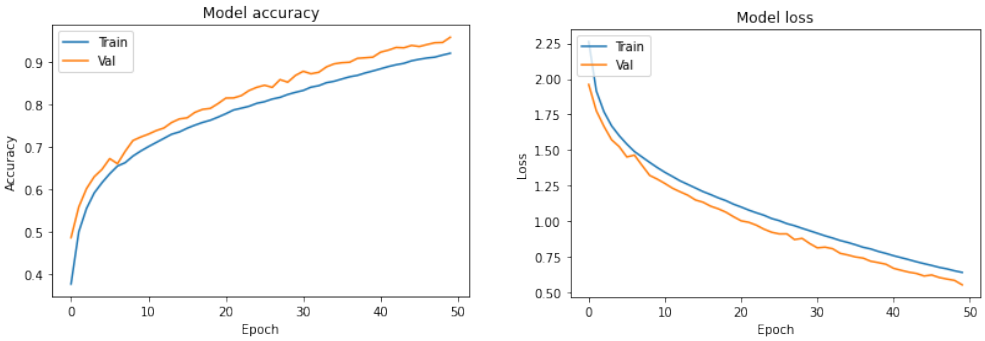


(a) Accuracy Curves (b) Loss Curves

Fig. 14: Training curves - final tuned version

E. Batch Normalization

After applying batch normalization to the model, the fine search was done again to check if the batch normalization helps us pick some different learning rate and l2 regularization. It is observed that batch norm generally helps to increase training speed as well as improve generalization.



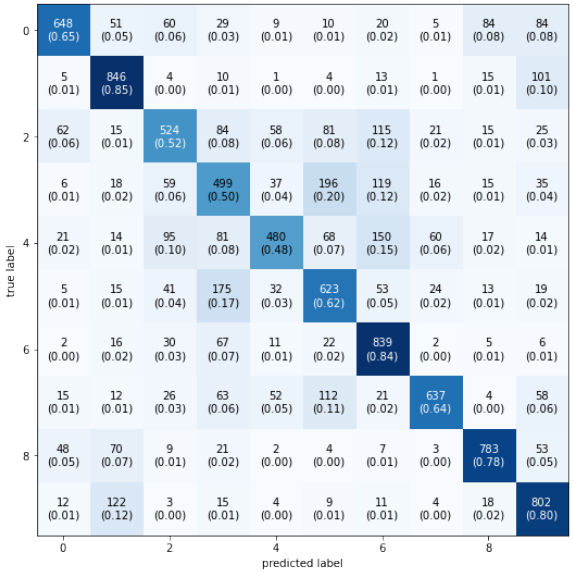
(a) Accuracy Curves (b) Loss Curves

Fig. 15: Training curves - final tuned with dropout version

It also reduces overfitting by reducing the internal covariate shift between the data. It is often found that using batchnorm also eliminates the need to use techniques like dropout, allows larger learning rates and in turn accelerates the convergence of the model. [3]

F. Predictions

The accuracy without batch normalization on test data came up to be around 66.8%. The confusion matrix and the classification report with all the metrics is as shown below.



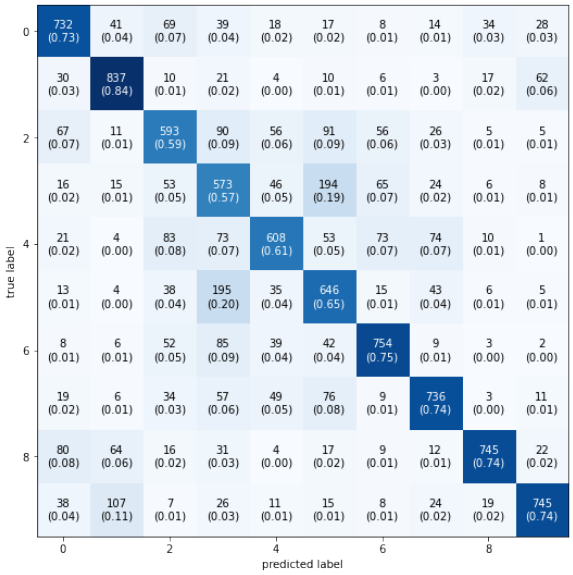
	precision	recall	f1-score	support
0	0.65	0.79	0.71	824
1	0.85	0.72	0.78	1179
2	0.52	0.62	0.57	851
3	0.50	0.48	0.49	1044
4	0.48	0.70	0.57	686
5	0.62	0.55	0.59	1129
6	0.84	0.62	0.71	1348
7	0.64	0.82	0.72	773
8	0.78	0.81	0.80	969
9	0.80	0.67	0.73	1197
accuracy			0.67	10000
macro avg	0.67	0.68	0.67	10000
weighted avg	0.69	0.67	0.67	10000

0.6681

(a) Confusion Matrix (b) Classification Report

Fig. 16: Predictions - Metrics.

If we use batchnorm the accuracy went up to about 69%. The classification report and the confusion matrix is as shown in the figure below.



(a) Confusion Matrix

	precision	recall	f1-score	support
0	0.73	0.71	0.72	1024
1	0.84	0.76	0.80	1095
2	0.59	0.62	0.61	955
3	0.57	0.48	0.52	1190
4	0.61	0.70	0.65	870
5	0.65	0.56	0.60	1161
6	0.75	0.75	0.75	1003
7	0.74	0.76	0.75	965
8	0.74	0.88	0.81	848
9	0.74	0.84	0.79	889
accuracy			0.70	10000
macro avg	0.70	0.71	0.70	10000
weighted avg	0.70	0.70	0.69	10000

0.6969

(b) Classification Report

Fig. 17: Predictions - Metrics with Batch Normalization.

III. Babysitting the training on Face Dataset from Project 1

The same process has been repeated for the Face Dataset in Project 1. Few other thinks have been tried on the Face Data, such as Batch Normalization, He Initialization, Data Augmentation.
He Initialization is normally done for layers in models with 'relu' activations. It is observed that this initialization helps the model to accelerate the training as well. We will go through the similar steps as we did for CIFAR for the Face Dataset.

A. *Preprocessing the data*

The data was organized as Face and Non Face data for project 1. There were 1000 training and 100 testing samples for each kind of data. The data was read into numpy arrays and then shuffled and split into training and validation.
The same preprocessing is applied as applied for the CIFAR dataset. And a similar observation regarding the initial loss was there for this dataset as well.

B. *Choose the network architecture*

Here I have chosen a basic CNN with 3 layers. 32, 64, 128 filters in the three layers. All of them being 3x3 filters with a stride of 1. All the convolutional blocks are followed by 2x2 maxpool blocks. In the end the result is flattened and connected to a FCN with one hidden layer with 128 neurons and a softmax layer with 2 units as there are 2 classes - Face and Non Face. All the layers have 'relu' activations. The model is created using the Sequential API in Keras [1]. We can also replace the 2 neuron dense layer with the softmax activation by a single neuron dense layer with a sigmoid or tanh activation as well. Other than that the model is same as that used for the CIFAR classification.

C. *Double checking to check if the loss is reasonable*

1) Initial loss with no regularization: The initial loss with no regularization is around 2.3 as expected for 10 classes.

```
In [34]:  1  model.compile(optimizer=optimizers.SGD(learning_rate=LEARNING_RATE),loss='sparse_categorical_crossentropy',metrics=['sparse_
In [35]:  1  model.evaluate(X_train,y_train)
          1600/1600 [==============================] - 0s 245us/sample - loss: 0.7073 - sparse_categorical_accuracy: 0.4800
Out[35]: [0.707324982881546, 0.48]
```

Fig. 18: Initial Epoch loss - with no regularization

Sanity check: If we crank up the regularization we get a higher loss as expected.

```
In [36]:
#model hyperparam
LEARNING_RATE=1e-4
EPOCHS=10
L2_REG=1e3

model=create_model(num_classes,L2_REG)
model.compile(optimizer=optimizers.SGD(learning_rate=LEARNING_RATE),loss='sparse_categorical_crossentropy',metrics=['sparse_
model.evaluate(X_train, y_train)

1600/1600 [==============================] - 0s 246us/sample - loss: 379751.1644 - sparse_categorical_accuracy: 0.5006

Out[36]: [379751.164375, 0.500625]
```

Fig. 19: Initial Epoch loss - with no regularization

2) Try to overfit a small portion of data: We try to overfit the small portion of the data as a sanity check with no regularization and vanilla SGD optimizer. This ensures that our model is actually learning and the loss is going down. We verify this using the training accuracy (It should reach 100% theoretically). We can also see that the model reaches a very small loss as we reach the last epoch. That's a good sanity check.

```
In [38]:
model=create_model(num_classes,L2_REG)
model.compile(optimizer=optimizers.SGD(learning_rate=LEARNING_RATE),loss='sparse_categorical_crossentropy',metrics=['sparse_
history = model.fit(X_tiny,y_tiny,batch_size=2,epochs=EPOCHS, verbose=1)

20/20 [==============================] - 0s 2ms/sample - loss: 0.0981 - sparse_categorical_accuracy: 1.0000
Epoch 92/100
20/20 [==============================] - 0s 2ms/sample - loss: 0.0951 - sparse_categorical_accuracy: 1.0000
Epoch 93/100
20/20 [==============================] - 0s 2ms/sample - loss: 0.1056 - sparse_categorical_accuracy: 1.0000
Epoch 94/100
20/20 [==============================] - 0s 2ms/sample - loss: 0.1004 - sparse_categorical_accuracy: 1.0000
Epoch 95/100
20/20 [==============================] - 0s 2ms/sample - loss: 0.1030 - sparse_categorical_accuracy: 1.0000
Epoch 96/100
20/20 [==============================] - 0s 2ms/sample - loss: 0.0851 - sparse_categorical_accuracy: 1.0000
Epoch 97/100
20/20 [==============================] - 0s 2ms/sample - loss: 0.0762 - sparse_categorical_accuracy: 1.0000
Epoch 98/100
20/20 [==============================] - 0s 2ms/sample - loss: 0.0846 - sparse_categorical_accuracy: 1.0000
Epoch 99/100
20/20 [==============================] - 0s 2ms/sample - loss: 0.0821 - sparse_categorical_accuracy: 1.0000
Epoch 100/100
20/20 [==============================] - 0s 2ms/sample - loss: 0.0870 - sparse_categorical_accuracy: 1.0000
```

Fig. 20: Overfitting 20 examples from the data

3) Start with small regularization and find learning rate that makes loss go down: We start with a small regularization and try to find a learning rate that makes the loss go down. We can observe here that the loss is not going down. As we can see in Fig. The learning rate is so small that the model is actually not learning! The validation accuracy is still around 10% owing to the random guess.

```
In [110]:
LEARNING_RATE=1e-6
EPOCHS=10
L2_REG=1e-6

model=create_model(num_classes,L2_REG)
model.compile(optimizer=optimizers.SGD(learning_rate=LEARNING_RATE),loss='sparse_categorical_crossentropy',metrics=['sparse_
model.fit(X_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS, verbose=1,  validation_data=(X_val,y_val))

Epoch 5/10
1600/1600 [==============================] - 1s 572us/sample - loss: 0.7236 - sparse_categorical_accuracy: 0.4631 - val_loss:
0.7170 - val_sparse_categorical_accuracy: 0.4750
Epoch 6/10
1600/1600 [==============================] - 1s 589us/sample - loss: 0.7234 - sparse_categorical_accuracy: 0.4631 - val_loss:
0.7168 - val_sparse_categorical_accuracy: 0.4750
Epoch 7/10
1600/1600 [==============================] - 1s 573us/sample - loss: 0.7232 - sparse_categorical_accuracy: 0.4631 - val_loss:
0.7167 - val_sparse_categorical_accuracy: 0.4750
Epoch 8/10
1600/1600 [==============================] - 1s 574us/sample - loss: 0.7230 - sparse_categorical_accuracy: 0.4625 - val_loss:
0.7165 - val_sparse_categorical_accuracy: 0.4750
Epoch 9/10
1600/1600 [==============================] - 1s 599us/sample - loss: 0.7228 - sparse_categorical_accuracy: 0.4619 - val_loss:
0.7163 - val_sparse_categorical_accuracy: 0.4775
Epoch 10/10
1600/1600 [==============================] - 1s 575us/sample - loss: 0.7227 - sparse_categorical_accuracy: 0.4619 - val_loss:
0.7162 - val_sparse_categorical_accuracy: 0.4775

Out[110]: <tensorflow.python.keras.callbacks.History at 0x2b0c9fded30>
```

Fig. 21: Small learning rate

We can now try a higher learning rate. We set the learning rate to 1e6. As we can see in Fig. The cost is so high that the loss is exploding. It is because the weights are updated by a huge number.

```
In [41]:
LEARNING_RATE=1e5
EPOCHS=10
L2_REG=1e-6

model=create_model(num_classes,L2_REG)
model.compile(optimizer=optimizers.SGD(learning_rate=LEARNING_RATE),loss='sparse_categorical_crossentropy',metrics=['sparse_
model.fit(X_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS, verbose=1,  validation_data=(X_val,y_val))
```

```
Epoch 5/10
1600/1600 [==============================] - 1s 317us/sample - loss: nan - sparse_categorical_accuracy: 0.5044 - val_loss: na
n - val_sparse_categorical_accuracy: 0.5075
Epoch 6/10
1600/1600 [==============================] - 0s 302us/sample - loss: nan - sparse_categorical_accuracy: 0.4856 - val_loss: na
n - val_sparse_categorical_accuracy: 0.4925
Epoch 7/10
1600/1600 [==============================] - ETA: 0s - loss: nan - sparse_categorical_accuracy: 0.51 - 0s 305us/sample - los
s: nan - sparse_categorical_accuracy: 0.5131 - val_loss: nan - val_sparse_categorical_accuracy: 0.4925
Epoch 8/10
1600/1600 [==============================] - 1s 319us/sample - loss: nan - sparse_categorical_accuracy: 0.5169 - val_loss: na
n - val_sparse_categorical_accuracy: 0.4925
Epoch 9/10
1600/1600 [==============================] - 1s 316us/sample - loss: nan - sparse_categorical_accuracy: 0.4881 - val_loss: na
n - val_sparse_categorical_accuracy: 0.4925
Epoch 10/10
1600/1600 [==============================] - 0s 302us/sample - loss: nan - sparse_categorical_accuracy: 0.4906 - val_loss: na
n - val_sparse_categorical_accuracy: 0.4925
Out[41]: <tensorflow.python.keras.callbacks.History at 0x2b02c9bffd0>
```

Fig. 22: Large learning rate

Trying to find the optimum learning rate is very important. We can keep trying values for which the loss will go down, but that is a trial and error method.

```
In [42]:
LEARNING_RATE=0.2425
EPOCHS=10
L2_REG=1e-6

model=create_model(num_classes,L2_REG)
model.compile(optimizer=optimizers.SGD(learning_rate=LEARNING_RATE),loss='sparse_categorical_crossentropy',metrics=['sparse_
model.fit(X_train, y_train, batch_size=BATCH_SIZE, epochs=EPOCHS, verbose=1,  validation_data=(X_val,y_val))
```

```
Train on 1600 samples, validate on 400 samples
Epoch 1/10
1600/1600 [==============================] - 1s 589us/sample - loss: 1.4783 - sparse_categorical_accuracy: 0.6781 - val_loss:
2.2005 - val_sparse_categorical_accuracy: 0.5325
Epoch 2/10
1600/1600 [==============================] - 1s 317us/sample - loss: nan - sparse_categorical_accuracy: 0.4925 - val_loss: na
n - val_sparse_categorical_accuracy: 0.5075
Epoch 3/10
1600/1600 [==============================] - 1s 327us/sample - loss: nan - sparse_categorical_accuracy: 0.4981 - val_loss: na
n - val_sparse_categorical_accuracy: 0.5075
Epoch 4/10
1600/1600 [==============================] - 1s 317us/sample - loss: nan - sparse_categorical_accuracy: 0.4981 - val_loss: na
n - val_sparse_categorical_accuracy: 0.5075
Epoch 5/10
1600/1600 [==============================] - 1s 325us/sample - loss: nan - sparse_categorical_accuracy: 0.4981 - val_loss: na
n - val_sparse_categorical_accuracy: 0.5075
Epoch 6/10
1600/1600 [==============================] - 0s 312us/sample - loss: nan - sparse_categorical_accuracy: 0.4981 - val_loss: na
n - val_sparse_categorical_accuracy: 0.5075
```

Fig. 23: Cost still exploding

We can adopt a cross validation strategy where we try different learning rates and regularizations in a fixed range and choose the combination that gives the best performance on the model.

*D. **Hyperparmater tuning***

To find the optimum learning rate and regularization to apply, we first go through a few epochs to get a rough idea of how the params work and then move on to the second stage, where we have a longer run time and a finer search.

1) Coarse search: For coarse search we take a random value of the learning rate between 1e-3 and 1e-6 and a random value for regularization

```
In [43]:
val_acc=[]
lrs=[]
l2_regs=[]
for i in tqdm(range(100)):
    lr = 10**np.random.uniform(-3,-6)
    l2_reg = 10**np.random.uniform(-5, 1)
    model = create_model(num_classes,l2_reg)
    model.compile(optimizer=optimizers.SGD(learning_rate=lr),loss='sparse_categorical_crossentropy',metrics=['sparse_categor
    history = model.fit(X_train,y_train,batch_size=BATCH_SIZE,epochs=5, verbose=1,validation_data=(X_val,y_val))
    val_acc.append(history.history['val_sparse_categorical_accuracy'][-1])
    lrs.append(lr)
    l2_regs.append(l2_reg)
    print(history.history['val_sparse_categorical_accuracy'][-1], lr, l2_reg)
```

```
2.0174 - val_sparse_categorical_accuracy: 0.5875
Epoch 2/5
1600/1600 [==============================] - 1s 394us/sample - loss: 2.0142 - sparse_categorical_accuracy: 0.6000 - val_loss:
2.0168 - val_sparse_categorical_accuracy: 0.5900
Epoch 3/5
1600/1600 [==============================] - 1s 384us/sample - loss: 2.0136 - sparse_categorical_accuracy: 0.6000 - val_loss:
2.0162 - val_sparse_categorical_accuracy: 0.5925
Epoch 4/5
1600/1600 [==============================] - 1s 392us/sample - loss: 2.0130 - sparse_categorical_accuracy: 0.6019 - val_loss:
2.0156 - val_sparse_categorical_accuracy: 0.5950
Epoch 5/5
1600/1600 [==============================] - 1s 397us/sample - loss: 2.0124 - sparse_categorical_accuracy: 0.6025 - val_loss:
2.0149 - val_sparse_categorical_accuracy: 0.6000

100%|██████████| 100/100 [10:53<00:00,  6.54s/it]

0.6 8.382739662675617e-06 0.003501435928545504
```

Fig. 24: Coarse searching strategy

```
In [46]:
1 logs[:,top10]

Out[46]: array([[8.72500002e-01, 8.52500021e-01, 8.52500021e-01, 8.45000029e-01,
                 8.42499971e-01, 8.39999974e-01, 8.37499976e-01, 8.32499981e-01,
                 8.17499995e-01, 8.14999998e-01],
                [7.61644656e-04, 4.75100438e-04, 8.26742014e-04, 3.54382843e-04,
                 8.86005669e-04, 9.07719034e-04, 1.88985557e-04, 2.54710752e-04,
                 9.16455451e-04, 3.47673147e-04],
                [1.07341204e-04, 7.59517082e-01, 1.06154477e-01, 5.60755486e-04,
                 1.06745960e-05, 2.81907629e-02, 1.12913460e-04, 4.05971357e-01,
                 7.02611552e-01, 1.54764117e-05]])
```

Fig. 25: Top 10 performers - coarse search

As seen in Fig. 11 we can see the top 10 performers from the coarse search strategy, we can see that the learning rates are restricted within the range 1e-3 and 1e-4 and the regularization between 1e-5 and 1. Thus, we narrow down the range and perform a finer search for a longer time, i.e. increase the number of epochs.

2) Fine Search: We restrict the fine search to the smaller range and train for 15 epochs.

```
In [48]:
val_acc=[]
lrs=[]
l2_regs=[]
for i in tqdm(range(30)):
    lr = 10**np.random.uniform(-3,-4)
    l2_reg = 10**np.random.uniform(-6, -1)
    model = create_model(num_classes,l2_reg)
    model.compile(optimizer=optimizers.SGD(learning_rate=lr),loss='sparse_categorical_crossentropy',metrics=['sparse_categor
    history = model.fit(X_train,y_train,batch_size=BATCH_SIZE,epochs=15, verbose=1,validation_data=(X_val,y_val))
    val_acc.append(history.history['val_sparse_categorical_accuracy'][-1])
    lrs.append(lr)
    l2_regs.append(l2_reg)
    print(history.history['val_sparse_categorical_accuracy'][-1], lr, l2_reg)
```

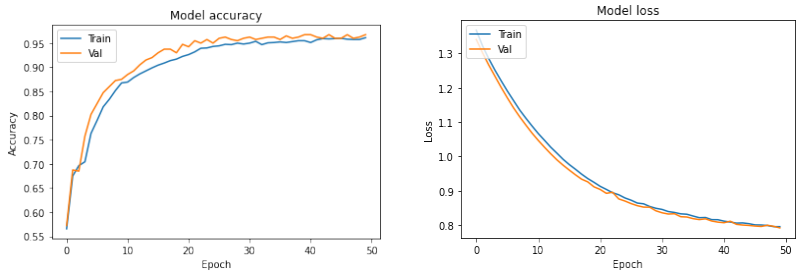
```
1600/1600 [==============================] - 1s 393us/sample - loss: 0.5858 - sparse_categorical_accuracy: 0.8131 - val_loss:
0.5776 - val_sparse_categorical_accuracy: 0.8200
Epoch 11/15
1600/1600 [==============================] - 1s 396us/sample - loss: 0.5756 - sparse_categorical_accuracy: 0.8200 - val_loss:
0.5670 - val_sparse_categorical_accuracy: 0.8250
Epoch 12/15
1600/1600 [==============================] - 1s 433us/sample - loss: 0.5656 - sparse_categorical_accuracy: 0.8206 - val_loss:
0.5567 - val_sparse_categorical_accuracy: 0.8400
Epoch 13/15
1600/1600 [==============================] - 1s 434us/sample - loss: 0.5559 - sparse_categorical_accuracy: 0.8281 - val_loss:
0.5467 - val_sparse_categorical_accuracy: 0.8425
Epoch 14/15
1600/1600 [==============================] - 1s 421us/sample - loss: 0.5464 - sparse_categorical_accuracy: 0.8313 - val_loss:
0.5370 - val_sparse_categorical_accuracy: 0.8425
Epoch 15/15
1600/1600 [==============================] - 1s 427us/sample - loss: 0.5372 - sparse_categorical_accuracy: 0.8319 - val_loss:
0.5276 - val_sparse_categorical_accuracy: 0.8475
```

Fig. 26: Fine search

```
In [51]:  1 logs[:,top4]

Out[51]: array([[9.25000012e-01, 9.04999971e-01, 8.99999976e-01, 8.92499983e-01],
                [9.05174698e-04, 7.24762937e-04, 5.04411369e-04, 4.83966941e-04],
                [1.81105723e-03, 1.13826938e-03, 4.43588249e-02, 2.37296232e-03]])
```

Fig. 27: Fine search top performers

Now we will train the model using the top performer found using the top performer from the fine search.

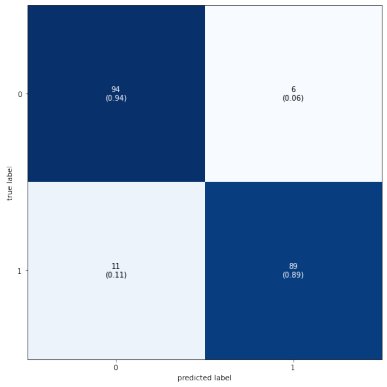


(a) Accuracy Curves (b) Loss Curves

Fig. 28: Training curves - final tuned version

*E. **Predictions***

We tried predicting With the tuned parameters. The confusion Matrix is plotted as shown below and the classification report is also printed. We can see that the tuned parameters gave a decent accuracy of 91% on the out of sample test data. We can still accelerate the convergence of the model or the generalization by using tactics such as batch norm or He Initialization. We can also generalize the dataset using data augmentation as this data set is already very small.



	precision	recall	f1-score	support
0	0.94	0.89	0.91	106
1	0.88	0.94	0.91	94
accuracy			0.91	200
macro avg	0.91	0.91	0.91	200
weighted avg	0.91	0.91	0.91	200

0.91

(a) Confusion Matrix (b) Classification Report

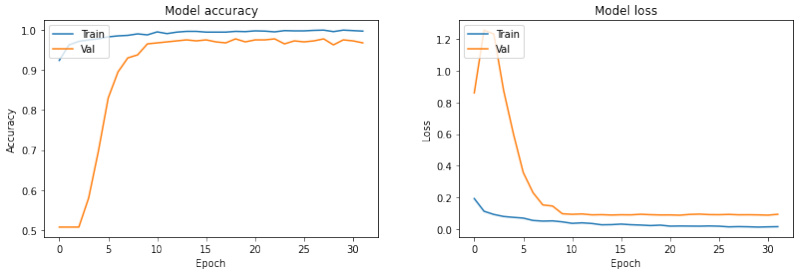
Fig. 29: Predictions - Metrics - Vanilla model.

*F. **Batch Normalization***

After applying batch normalization to the model, the fine search was done again to check if the batch normalization helps us pick some different learning rate and l2 regularization. It is observed that batch norm generally helps to increase training speed as well as improve generalization. In the figure below we can show see that the learning rate chosen from the fine search procedure for the batch norm is quite higher than that chosen in the previous fine search strategy.

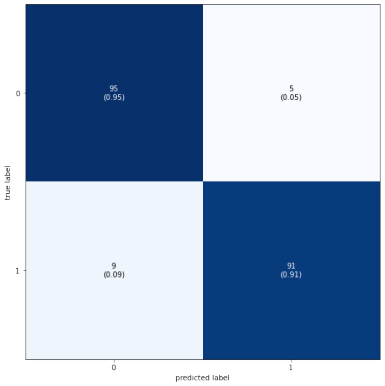
```
In [94]: 1 logs[:,top4]

Out[94]: array([[9.77500021e-01, 9.75000024e-01, 9.75000024e-01, 9.75000024e-01],
                [5.72852074e-03, 4.47587608e-03, 7.91517957e-03, 7.44720243e-03],
                [4.16020772e-06, 1.27860626e-06, 3.96499158e-06, 3.81859253e-06]])
```

Fig. 30: Fine search



(a) Accuracy Curves (b) Loss Curves

Fig. 31: Training curves - final tuned version



	precision	recall	f1-score	support
0	0.95	0.91	0.93	104
1	0.91	0.95	0.93	96
accuracy			0.93	200
macro avg	0.93	0.93	0.93	200
weighted avg	0.93	0.93	0.93	200

0.93

(a) Confusion Matrix (b) Classification Report

Fig. 32: Predictions - Metrics - With Batch Norm.

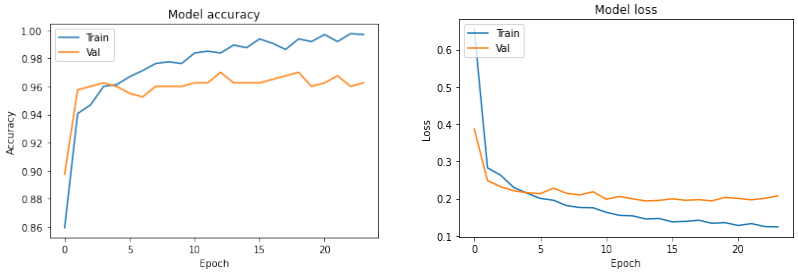
We can see that the test accuracy for the out of sample data also increased a bit due to the better generalization through batch norm along with the faster convergence.

G. He Initialization

For He initialization the fine search strategy was deployed again just as that in the Batch Norm process. We can see that in this case as well the learning rate selected is a bit higher than when He init is not applied. He init is usually used in models with relu activation units.

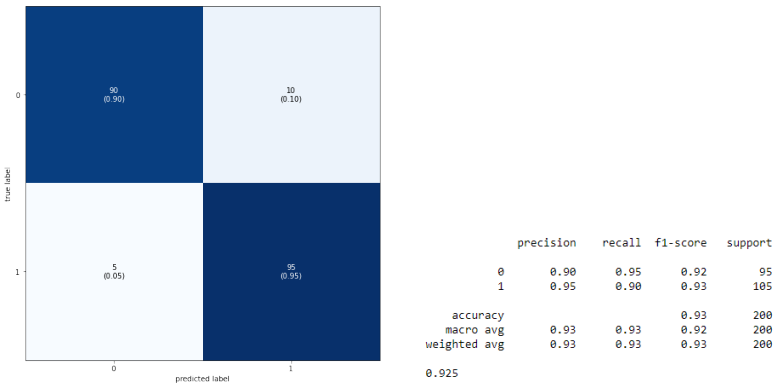
```
In [103]: 1 logs[:,top4]

Out[103]: array([[9.72500026e-01, 9.70000029e-01, 9.67499971e-01, 9.64999974e-01],
                 [5.66494724e-03, 5.77802982e-03, 3.00708412e-03, 1.84074673e-03],
                 [1.59754284e-04, 1.38677605e-04, 2.74304194e-04, 8.26537021e-05]])
```

Fig. 33: Fine search



(a) Accuracy Curves (b) Loss Curves

Fig. 34: Training curves - final tuned version



	precision	recall	f1-score	support
0	0.90	0.95	0.92	95
1	0.95	0.90	0.93	105
accuracy			0.93	200
macro avg	0.93	0.93	0.92	200
weighted avg	0.93	0.93	0.93	200

0.925

(a) Confusion Matrix (b) Classification Report

Fig. 35: Predictions - Metrics - With He Initialization.

The accuracy for out of sample test data increased when we did He initialization as well as we can see in the figure above. This is because our model has relu activation units and He initialization works better for relu activation units.

IV. Section 0 : Data Augmentation

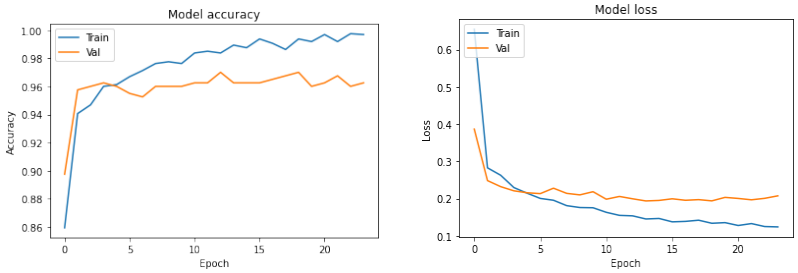
Data Augmentation is known to improve the performance especially on smaller datasets. On the face data we had we just had 1000 samples each of face and non-face dataset. These were properly cropped samples from the annotations. Some random transformations were applied(Rotation, Width shift, Height shift, horizontal flip, zoom). These augmentations help the model to generalize the data and be more robust to the out of sample data.
The best parameters for the batch normalized model were chosen. The ImageDataGenerator function was used from Keras to apply these transformations on the fly and the images were loaded in the data generator using flow().

In [113]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
tr_gen = ImageDataGenerator(rotation_range=10, width_shift_range=0.1,\
    height_shift_range=0.1, shear_range=0.1, zoom_range=0.1,\
    horizontal_flip=True, fill_mode="reflect")
val_gen = ImageDataGenerator()

train_datagen = tr_gen.flow(X_train,y_train, batch_size=BATCH_SIZE, shuffle=True)
valid_datagen = val_gen.flow(X_val,y_val,batch_size=BATCH_SIZE, shuffle=True)
tsdata = ImageDataGenerator()
test_datagen = tsdata.flow(X_test,batch_size=BATCH_SIZE, shuffle=False)
```

In [115]:
```
LR = 5.72852074e-03
L2_REG = 4.16020772e-06
model = create_model(num_classes,l2_reg, False, True)
model.compile(optimizer=optimizers.SGD(learning_rate=lr),loss='sparse_categorical_crossentropy',metrics=['sparse_categorical
```

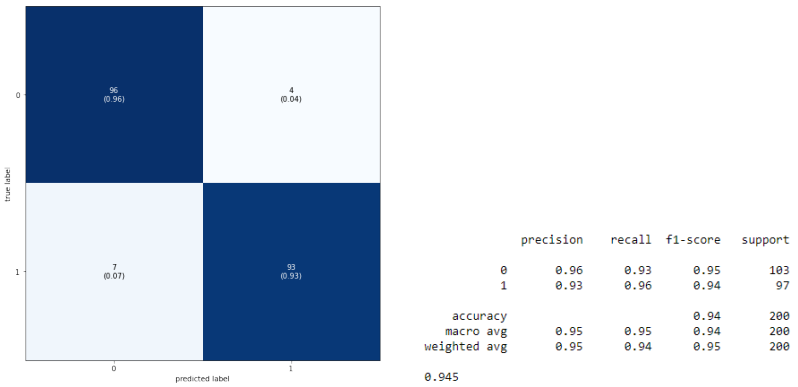
Fig. 36: Fine search

The learning curves and the prediction metrics(Confusion matrix and the classification report) are shown below.



(a) Accuracy Curves (b) Loss Curves

Fig. 37: Training curves - final tuned version



	precision	recall	f1-score	support
0	0.96	0.93	0.95	103
1	0.93	0.96	0.94	97
accuracy			0.94	200
macro avg	0.95	0.95	0.94	200
weighted avg	0.95	0.94	0.95	200

0.945

(a) Confusion Matrix (b) Classification Report

Fig. 38: Predictions - Metrics - With Data Augmentation.

We can see that the data augmentation increased the accuracy to 95% while generalizing the model in a better way.

V. Conclusion

Babysitting for a DNN was done on a simple LeNet inspired network. The network was a basic 3 layered convolutional neural network followed by fully connected network with one hidden layer. The 'relu' activation functions were used to avoid any vanishing gradient issues. Zero centering was done. If inputs to a neuron are all positive or all negative. In that case the gradient calculated during back propagation will either be positive or negative and hence parameter updates are only restricted to specific directions which in turn will make it difficult to converge. As a result, the gradient updates go too far in different directions which makes optimization harder. Many algorithms show better performances when the dataset is symmetric (with a zero-mean). Xavier initialization/ He initialization and batch normalization is done in order to improve the convergence properties of the network as well. To tune the hyper parameters, coarse and fine grid search methods are used which give us the best performing hyper parameters on the model. This is what we call the babysitting of a DNN.

References

[1] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[2] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 437–478.

[3] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.