Visual Recognition

Assignment-5

April 25th 2019.

By: Durga Yasasvi Y (IMT2016060)
Srujan Swaroop Gajula(IMT2016033)
Siddarth Reddy Desu(IMT2016037)

Hyperparameter Tuning and Optimization

- Model: Used ResNet architecture
- Dataset: CIFAR-10 data.

CONTENTS:

Following are the different hyperparameters explored:

- Layer Activation Function
  - Sigmoid, ReLU, tanh, leakyReLU, ELU.
- Data Preprocessing, normalization:
  - Subtract Mean Image Normalization.
- Dropout:
  - Strengths: 0.1, 0.3 and 0.5.
- BatchNorm:
  - With batch norm and without batch norm.
- Loss function:
  - Entropy, Hinge-loss, Mean-square.

- Learning rate:
    - Explored different learning rates (Static and Dynamic).
    - Dynamic learning rates include:
        - Step-decay, exponential decay.
- Optimizer:
    - SGD, Adam, Adadelta, Adagrad and RMSprop.
- Data Augmentation.
- Transfer Learning.
- Apart from these we also plotted different visualizations between the above hyperparameters that affect the loss and accuracies.

ACTIVATION FUNCTIONS:

Explored different activation functions.

The following are the parameters that are kept constant throughout the visualizations (in analyzing activation functions only):

- Batch_size = 128
- Epochs = 10
- Normalization = True
- Loss = Categorical Cross-entropy
- Learning Rate = Dynamic
- Only the activation function is varying.

3

```
# Training parameters
num_classes = 10
batch_size = 128
epochs = 10

activation = 'relu'                     #Activation
subtract_pixel_mean = True              #Normalization
batch_normalization_status = True       #Batch Norm

loss='categorical_crossentropy'         #Loss

learning_rate = 9e-3                    #Learning Rate
lr_scheduler_strategy = True            #Dynamic Learning Rate

conv_first=True

metrics = ['accuracy']
data_augmentation = False               #Data Augmentation
```

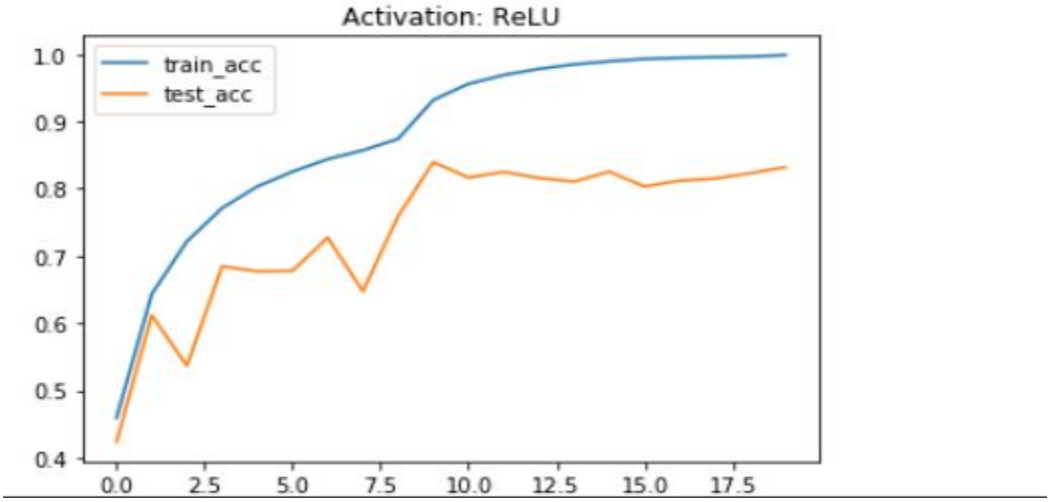
Below are the Accuracy visualizations provided for each activation function.

- **ReLU: Accuracy: ~84%**

```
Test loss: 0.924858239889145
Test accuracy: 0.832
```

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Activation: ReLU')
plt.legend(['train_acc', 'test_acc'], loc='upper left')
```

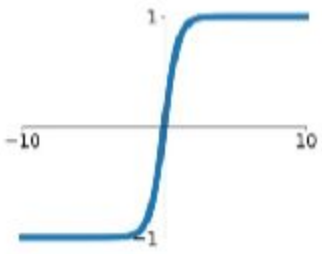
```
<matplotlib.legend.Legend at 0x7ff241ff0dd8>
```



- **Tanh: Accuracy: ~80%**
  - The activation function used here is tanh, it looks like:

tanh

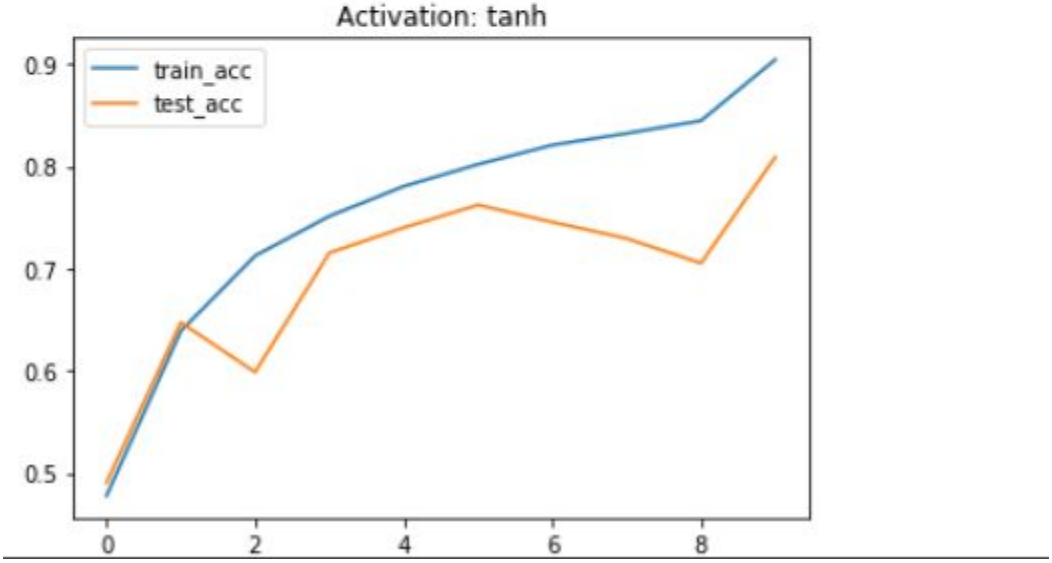
$$\tanh(x)$$



```
10000/10000 [==============================] - 2s 173us/step
Test loss: 0.7545414772510528
Test accuracy: 0.8089
```

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Activation: tanh')
plt.legend(['train_acc', 'test_acc'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7ff23c8becf8>
```

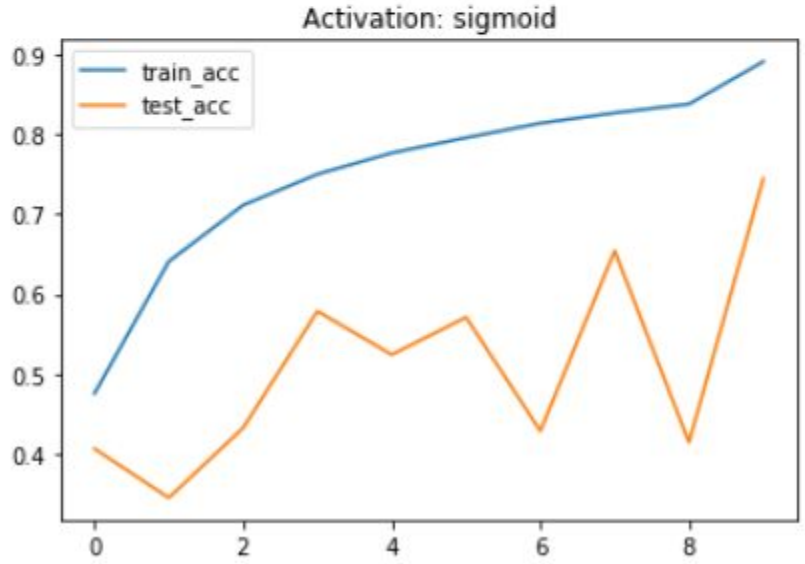


- **<u>sigmoid:</u> Accuracy: ~73%**
  - From the sigmoid activation visualization, we can clearly observe that the gap between the training accuracy and test accuracy kept on fluctuating. Hence there is overfitting here.
  - For more epochs, this gap will keep on fluctuating.

```
10000/10000 [==============================] - 2s 171us/step
Test loss: 1.005767158985138
Test accuracy: 0.7443
```

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Activation: sigmoid')
plt.legend(['train_acc', 'test_acc'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7faaaea86b00>
```



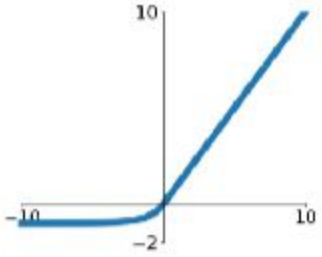
-

-

- **<u>ELU:</u> Accuracy: ~83%**
    - ELU is a modified version of ReLU. It somewhat looks like:

- **ELU**

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



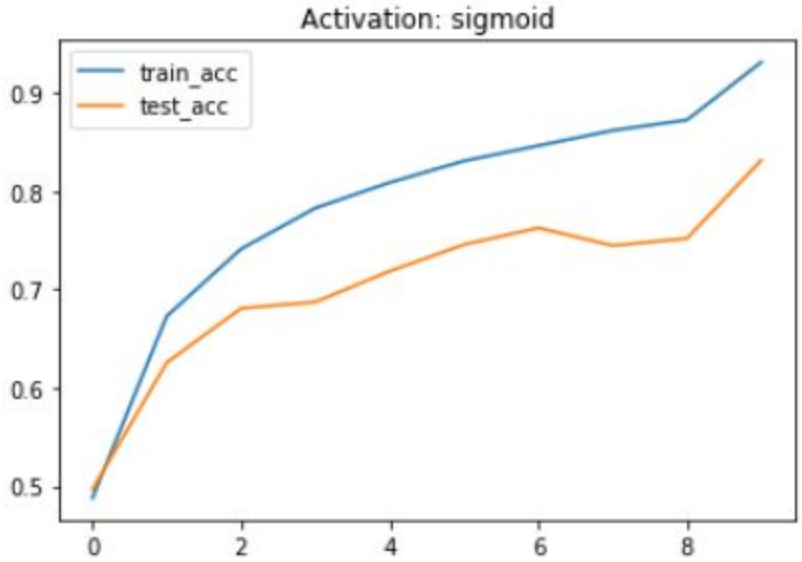
- **Accuracy:**

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 2s 166us/step
Test loss: 0.6854261425018311
Test accuracy: 0.8317
```

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Activation: ELU')
plt.legend(['train_acc', 'test_acc'], loc='upper left')
```

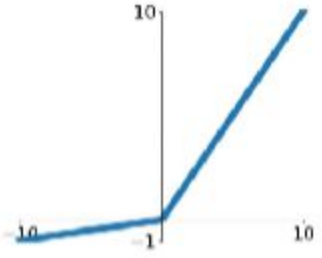
```
<matplotlib.legend.Legend at 0x7faa7186cba8>
```



- **<u>LeakyReLU:</u> Accuracy: ~80%**
    - LeakyReLU is a modified version of ReLU. It somewhat looks like:

Leaky ReLU

$$\max(0.1x, x)$$



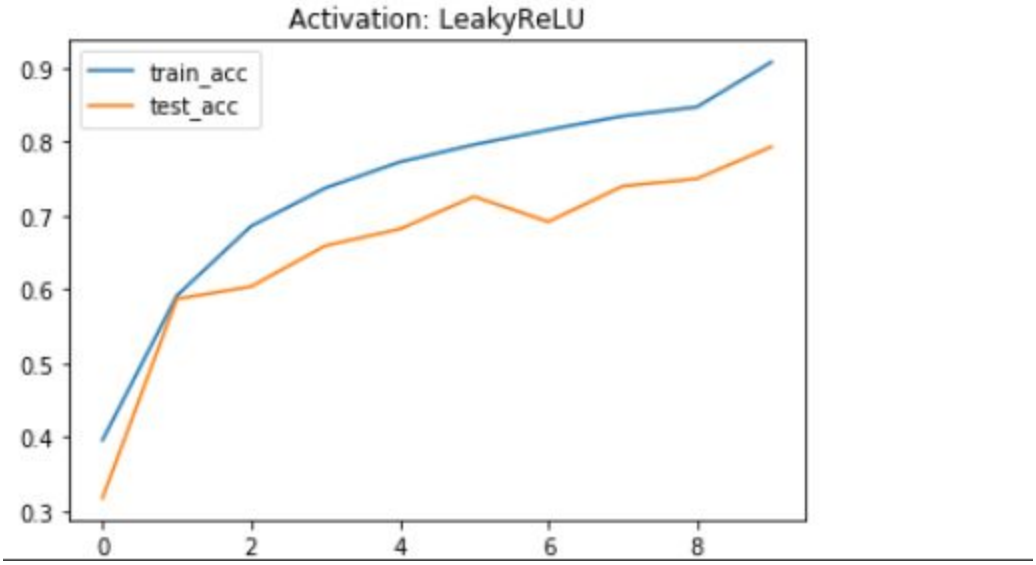
-

- **Accuracy:**

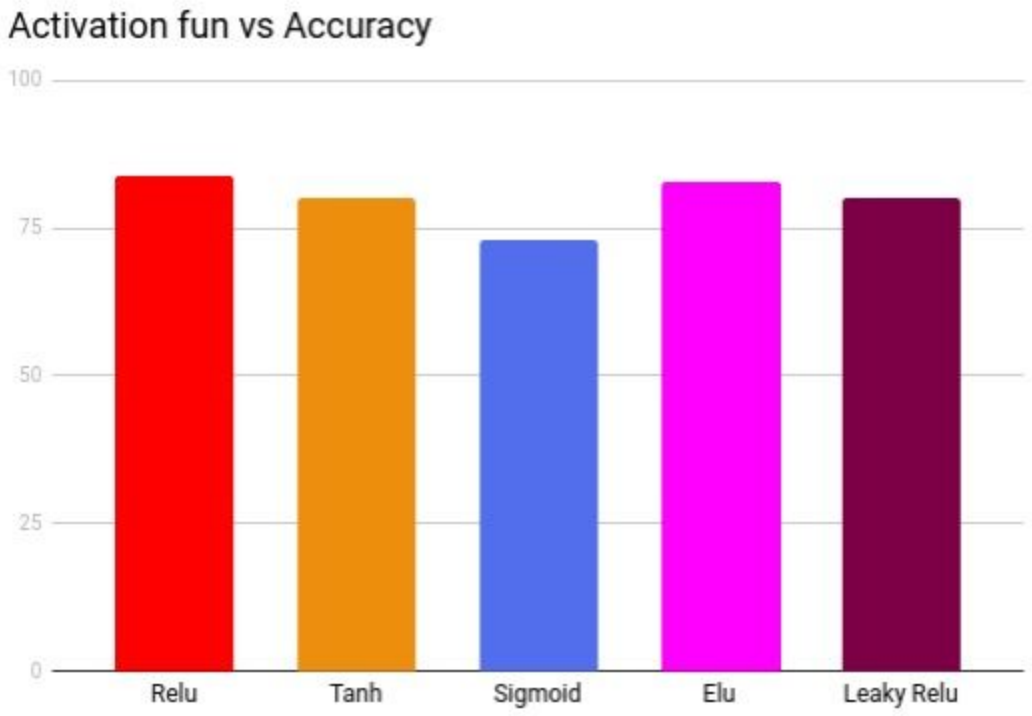
```
10000/10000 [==============================] - 5s 479us/step
Test loss: 0.7786621427536011
Test accuracy: 0.7928
```

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Activation: LeakyReLU')
plt.legend(['train_acc', 'test_acc'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7ff239afa400>
```



- Below is the Histogram for different accuracies obtained from the above-mentioned activation functions:



- **Observations:**
  - We can observe that the ‘ReLU’ activation function is the clear winner.
  - And the other activation functions like Leaky ReLU, ELU and tanh can be used to a certain extent.
  - And the sigmoid function is a bit dangerous since the gap between the training and testing accuracies kept on fluctuating.

DATA-PREPROCESSING AND NORMALIZATION:

Explored the accuracies with and without using normalization.

Normalization: Subtracting the mean of the images. Shifting all the data to the origin.

The following are the parameters that are kept constant throughout the visualizations:

- Batch_size = 128
- Epochs = 10
- Activation = ReLU
- Batch Normalization = True
- Loss = Categorical Cross-entropy
- Learning Rate = Dynamic
- Only the normalization flag is kept varying.

- **Without Normalization: Accuracy: ~79%**

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 2s 203us/step
Test loss: 0.8716356402873993
Test accuracy: 0.7928
```

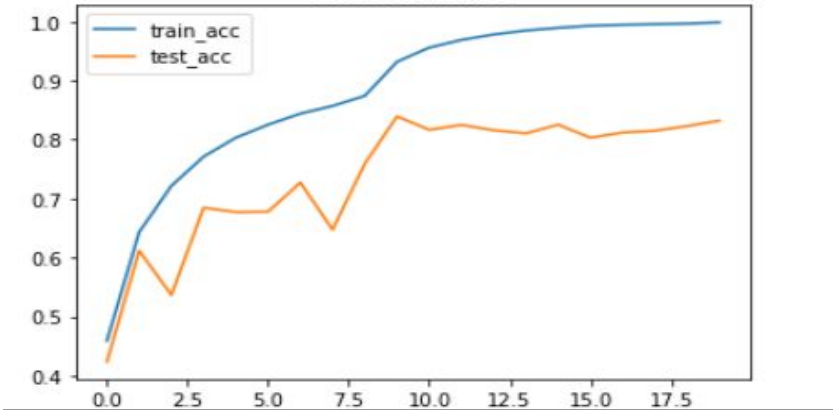
- **With Normalization: Accuracy: ~83%**

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

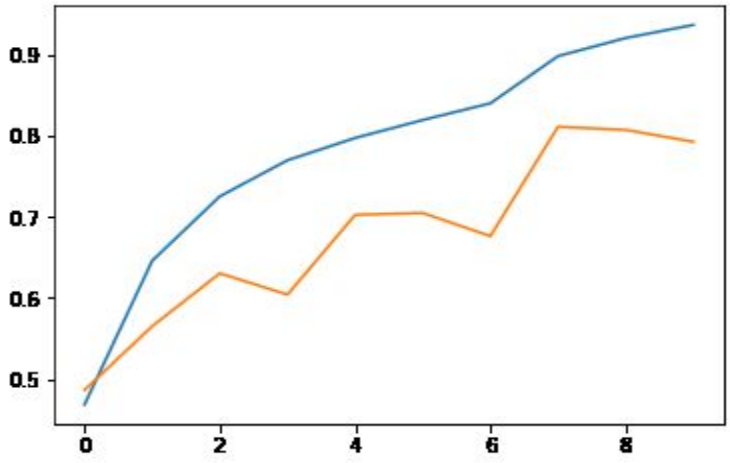
```
Test loss: 0.924858239889145
Test accuracy: 0.832
```

-

- Below are the plots between the train and test accuracies:
  - With Normalization:



  - Without Normalization:



- Observations:
    - Using Normalization definitely helps. It almost increases the accuracy by 3%.

REGULARIZATION: DROPOUT AND BATCH NORM:

DROPOUT:

Explored different dropout weights (0.1, 0.3 and 0.5).

The following are the parameters that are kept constant throughout the visualizations:

- Batch_size = 128
- Epochs = 10
- Activation = ReLU
- Normalization = True
- Loss = Categorical Cross-entropy
- Learning Rate = Dynamic

- Only the dropout strength is kept varying.

```
# Training parameters
num_classes = 10
batch_size = 128
epochs = 10

activation = 'relu'                    #Activation
subtract_pixel_mean = True             #Normalization
batch_normalization_status = False      #Batch Norm
dropout_status = True                  #True

loss = 'categorical_crossentropy'        #Loss

learning_rate = 9e-3                   #Learning Rate
lr_scheduler_strategy = True           #Dynamic Learning Rate

conv_first=True

metrics = ['accuracy']
data_augmentation = False              #Data Augmentation
```

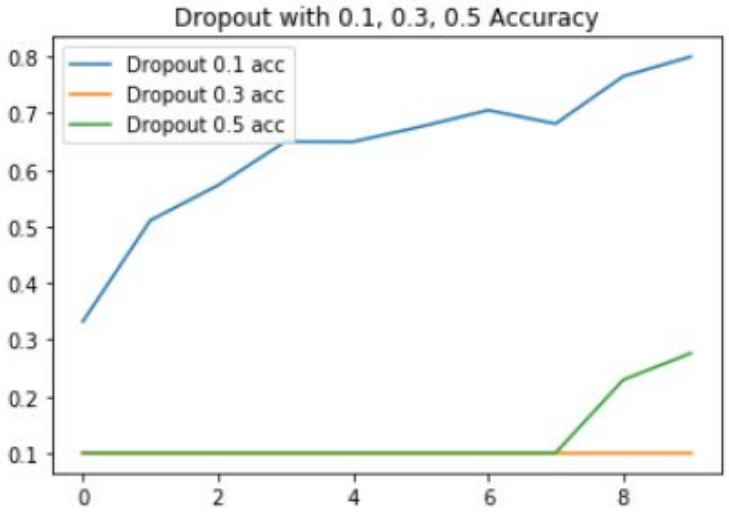
-

- **Dropout Strength**
  - **(0.1): Accuracy: ~75%**
  - **(0.3): Accuracy: ~28%**
  - **(0.5): Accuracy: ~10%**

Below are the combined accuracy and loss visualizations of the above 3 different strengths:

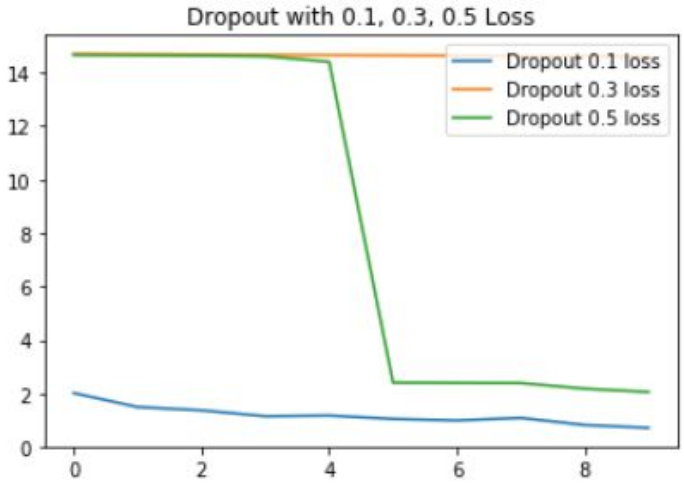
- **Accuracy visualization:**

```
<matplotlib.legend.Legend at 0x7ff22dbbecf8>
```



- **Loss function visualization:**

```
<matplotlib.legend.Legend at 0x7ff22da9d438>
```



- **Observations:**
  - **As you increase the strength of the dropout, the accuracy went down.**
  - **Also, the dropouts with strengths 0.3 and 0.5 don't do well.**

BATCH NORMALIZATION:

Explored the accuracies with and without using batch normalization.

The following are the parameters that are kept constant throughout the visualizations:

- Batch_size = 128
- Epochs = 10
- Activation = ReLU
- Normalization = True
- Loss = Categorical Cross-entropy
- Learning Rate = Dynamic
- Only the batch normalization flag is kept varying.

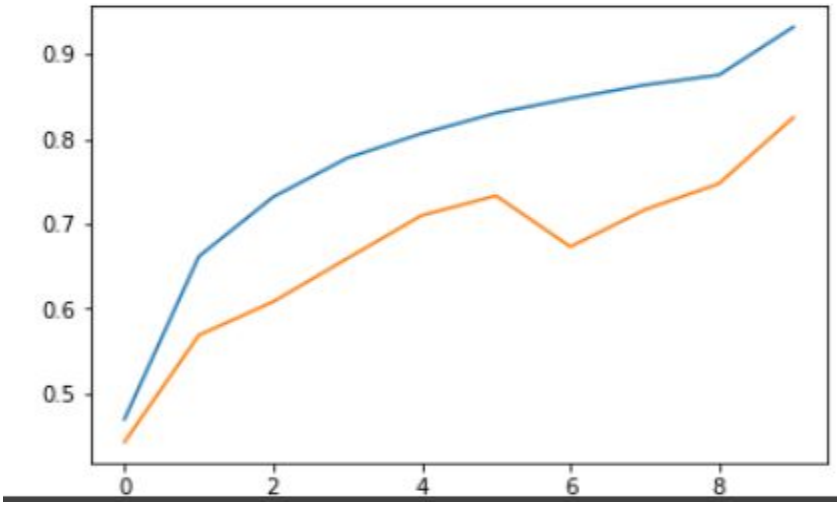
```
# Training parameters
num_classes = 10
batch_size = 128
epochs = 10

activation = 'relu'                     #Activation
subtract_pixel_mean = True              #Normalization
batch_normalization_status = True       #Batch Norm
dropout_status = True                   #True

loss = 'categorical_crossentropy'         #Loss

learning_rate = 9e-3                    #Learning Rate
lr_scheduler_strategy = True            #Dynamic Learning Rate

conv_first=True

metrics = ['accuracy']
data_augmentation = False               #Data Augmentation
```

- **With Batch Normalization: Accuracy: ~83%**

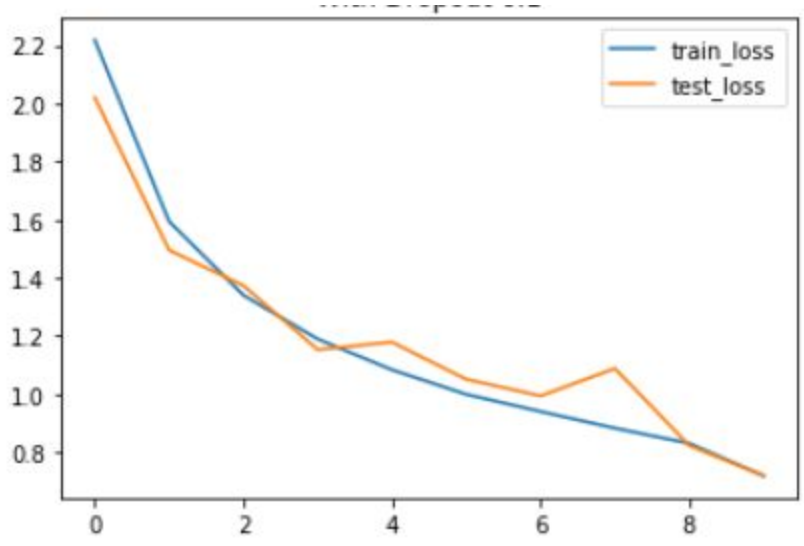
```
Test loss: 0.7182043767929077
Test accuracy: 0.8252
```

```
print(history.history.keys())
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
```

```
dict_keys(['val_loss', 'val_acc', 'loss', 'acc', 'lr'])
[<matplotlib.lines.Line2D at 0x7ff245571278>]
```



- **And the Loss obtained with Batch Norm is 0.7**

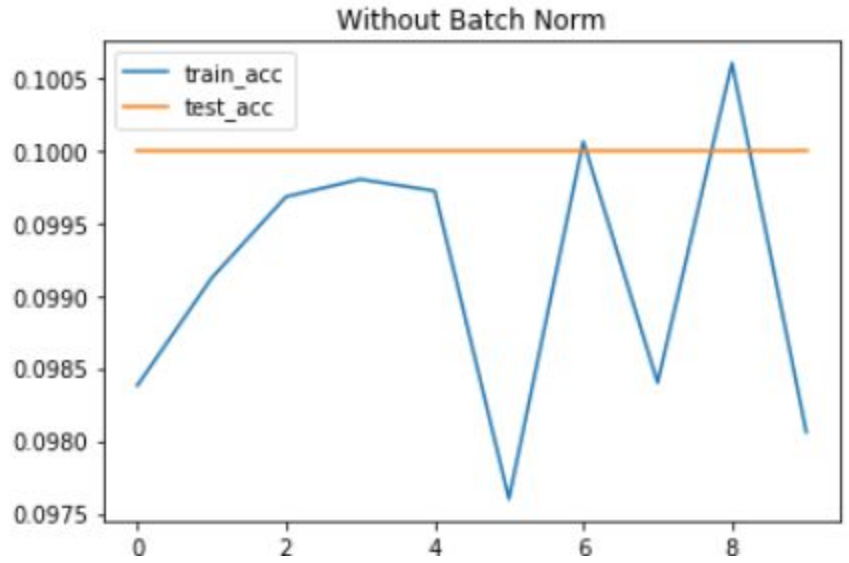


- **<u>Without Batch Normalization:</u> Accuracy: ~10%, Loss: 2.49**

```
Test loss: 2.4972865558624266
Test accuracy: 0.1
```

```
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Without Batch Norm')
plt.legend(['train_acc', 'test_acc'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7ff238a7da20>
```

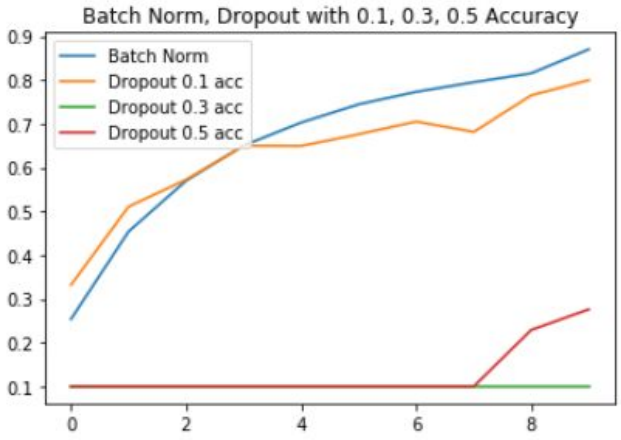


- **Observations:**
  - **We can clearly observe the change in accuracy using batch norm(83%) and without using batch normalization(10%).**
  - **Batch Normalization really helps the accuracy to stay at a high standard.**

- **Below is the accuracy and loss comparison plot between using Dropout Regularization and Batch Normalization:**
- **Accuracy comparison:**

```
# plt.plot(historyD1.history['loss'])
plt.plot(historyD1.history['val_acc'])
plt.plot(historyD3.history['val_acc'])
plt.title('Batch Norm, Dropout with 0.1, 0.3, 0.5 Accuracy')
plt.legend(['Batch Norm', 'Dropout 0.1 acc', 'Dropout 0.3 acc', 'Dropout 0.5 acc'], loc='upper left')
```

```
<matplotlib.legend.Legend at 0x7ff22a9390b8>
```



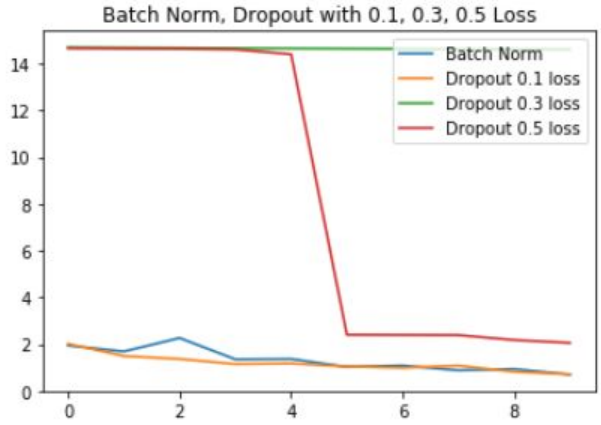
-

-

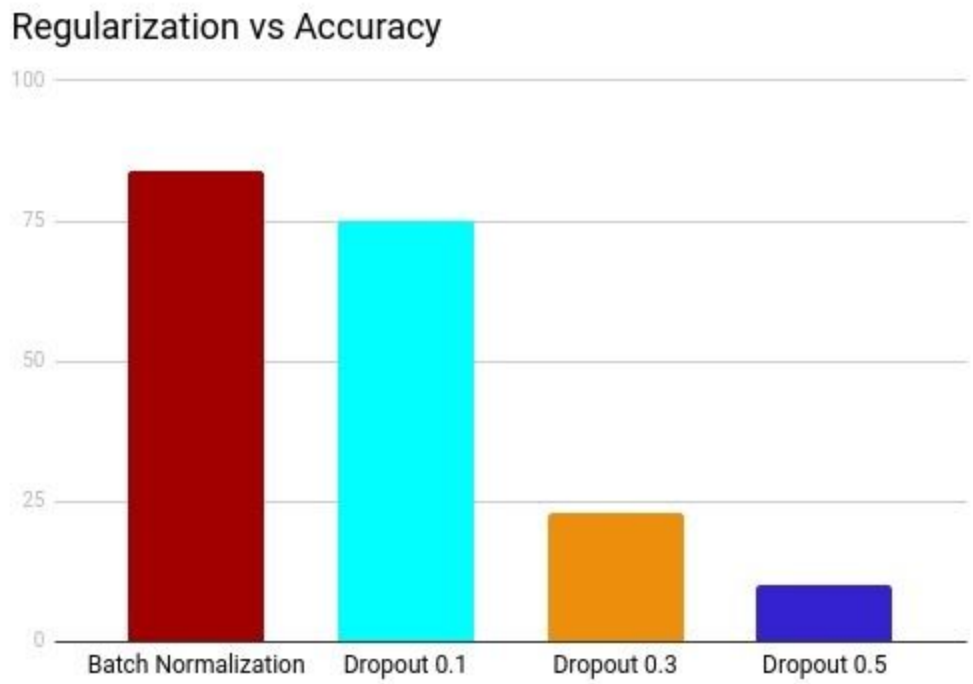
- **Loss comparison:**

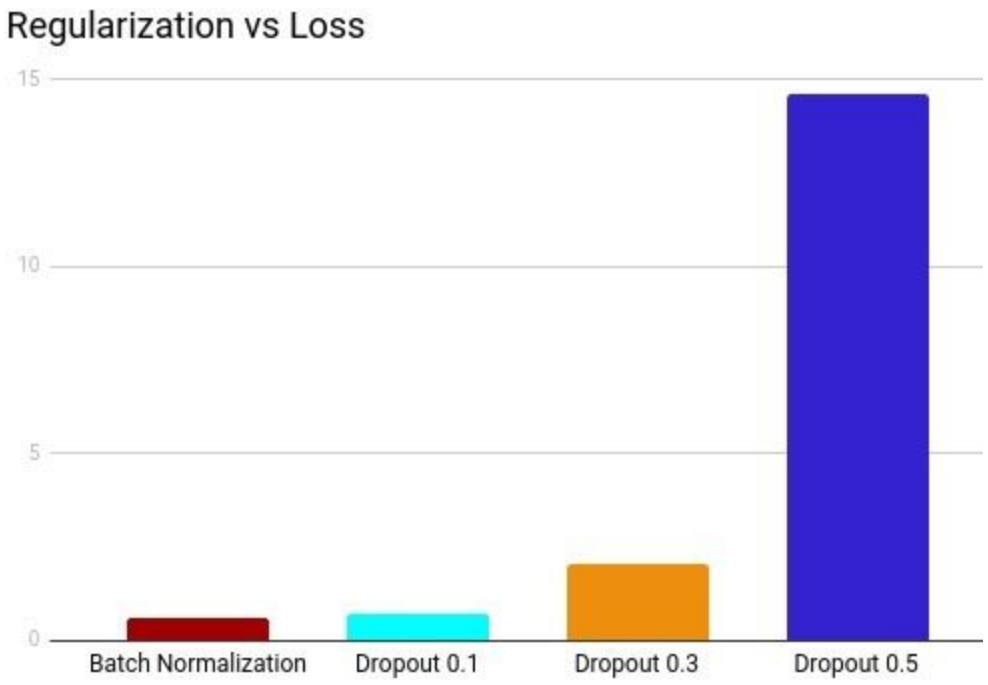
```
plt.title('Batch Norm, Dropout with 0.1, 0.3, 0.5 Loss')
plt.legend(['Batch Norm', 'Dropout 0.1 loss', 'Dropout 0.3 loss', 'Dropout 0.5 loss'], loc='upper right')
```

```
<matplotlib.legend.Legend at 0x7ff22a8b0748>
```



- **Now let's look at the Accuracy and the Loss histogram of all the regularizations used.**





-

- **Observations:**
  - **Clearly, Batch normalization is far effective than those of Dropout regularizations.**
  - **Only, Dropout with strength 0.1 does well. Others are too worse.**
  - **Hence it is preferred to use Batch Normalization over Dropout.**

LOSS FUNCTION:

Explored the accuracies by using different loss functions.

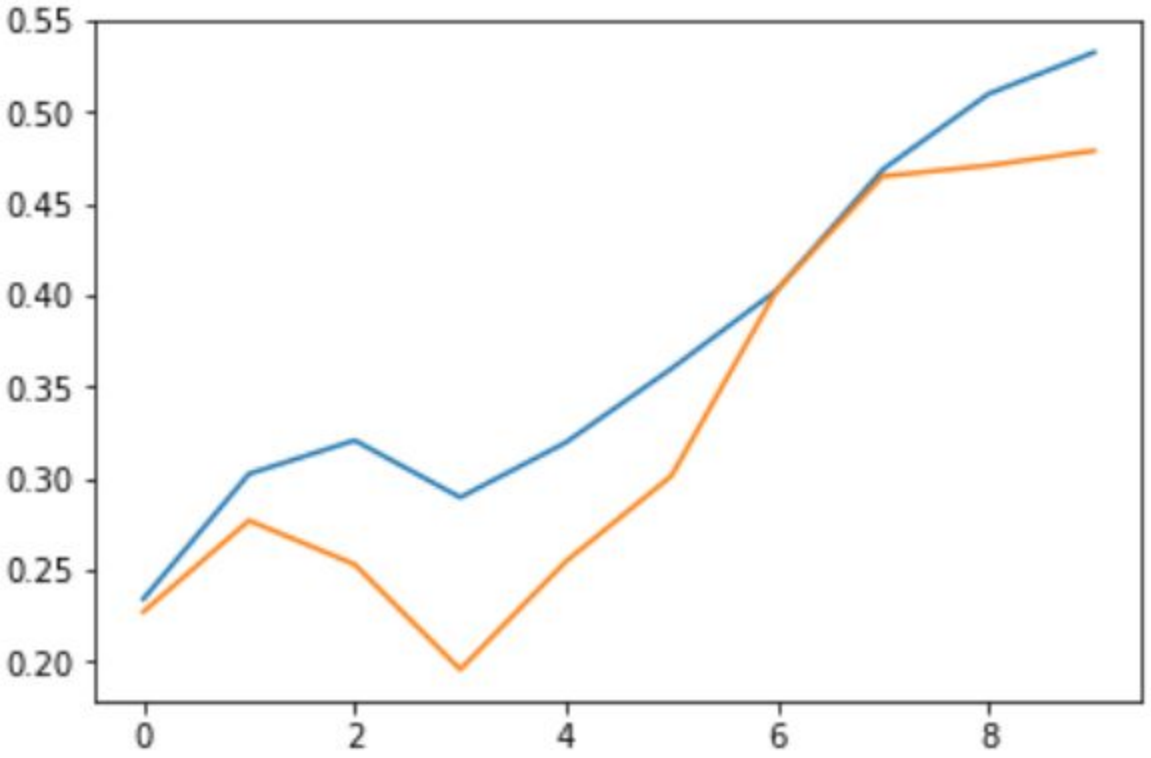
The following are the parameters that are kept constant throughout the visualizations:

- Batch_size = 128
- Epochs = 10
- Activation = ReLU
- Normalization = True
- Batch Normalization = True
- Learning Rate = Dynamic
- Only the Loss function is kept varying.

- **Loss function - Categorical Hingeloss: Accuracy: ~48%**

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 2s 211us/step
Test loss: 0.901104706954956
Test accuracy: 0.4786
```



- **<u>Loss function - Mean squared loss:</u> Accuracy: ~65%**

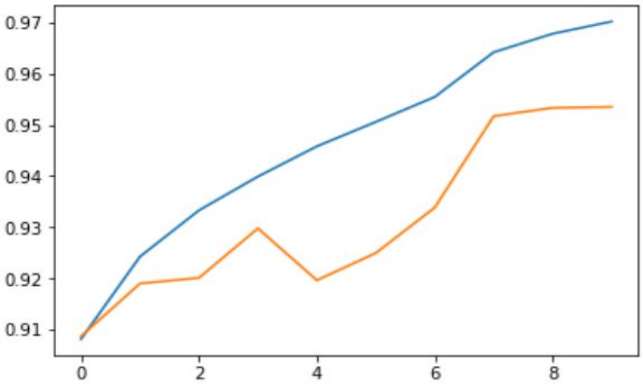
```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 2s 203us/step
Test loss: 0.13205010232925415
Test accuracy: 0.6573
```

- **Loss function - Binary Categorical: Accuracy: ~95%**

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

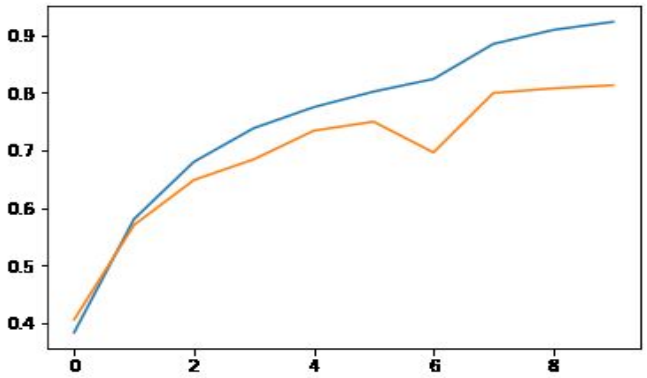
```
10000/10000 [==============================] - 2s 222us/step
Test loss: 0.2259703178882599
Test accuracy: 0.948439993095398
```



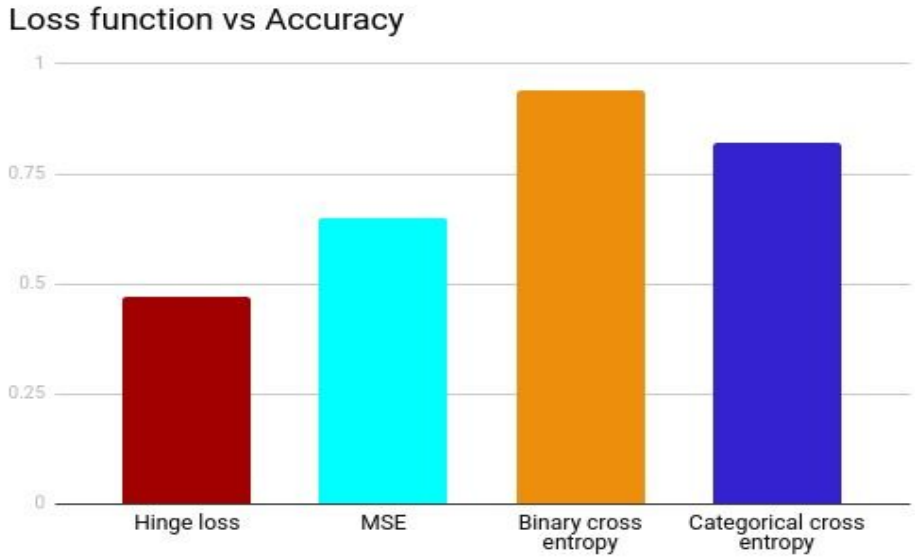
- **Loss function - Categorical entropy: Accuracy: ~83%**

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
Test loss: 0.924858239889145
Test accuracy: 0.832
```



- **Below are the different Loss functions accuracy comparison.**



  - **Observations:**
    - **Using Binary cross entropy we acquired the highest accuracy among any other combination we tried for, i.e. 95%.**
    - **The second best is using Categorical cross entropy i.e. 83%.**

LEARNING RATES AND LEARNING STRATEGIES:

Explored the accuracies by using different learning rates.

The following are the parameters that are kept constant throughout the visualizations:

- Batch_size = 128
- Epochs = 10
- Activation = ReLU
- Normalization = True
- Batch Normalization = True
- Loss function = Categorical Cross entropy.
- Only the learning rate is kept varying.

Learning strategy: We tried starting with a lower learning rate and then kept on increasing its value. We also tried for static vs dynamic learning rates. And also tried using momentum.

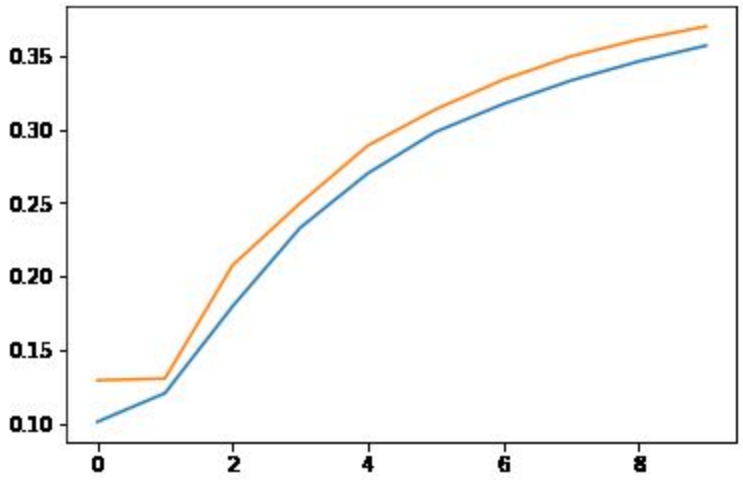
STATIC LEARNING RATES:

- **Initial learning rate 0.0001: Accuracy: ~48%**

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 1s 146us/step
Test loss: 1.5803276874542236
Test accuracy: 0.4792
```

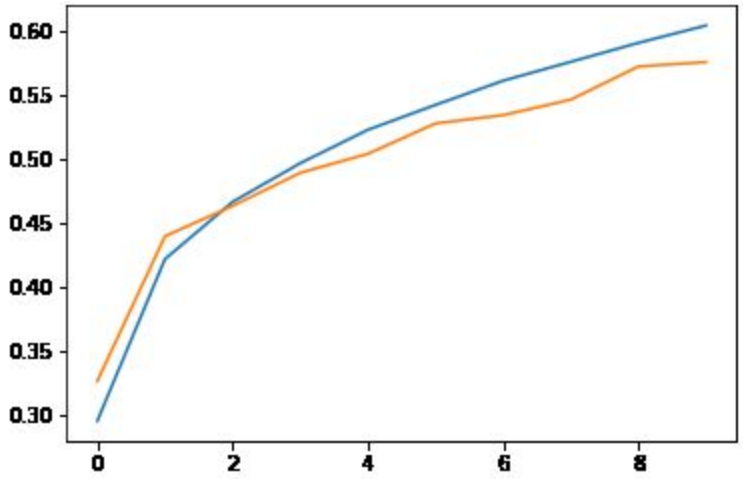
➔ Accuracy v/s Epochs



- **Initial learning rate 0.005: Accuracy: ~57%**

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 2s 159us/step
Test loss: 1.3373588306427002
Test accuracy: 0.575
```



DYNAMIC LEARNING RATES:

Here we tried different ways of dynamin learning rates like Step decay and exponential decay.

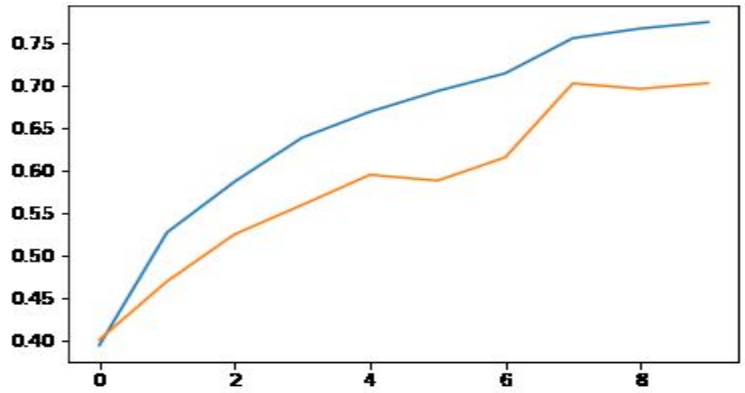
- **Step Decay with rate 0.005: Accuracy: ~70%**

```
def lr_schedule(epoch):
  initial_lrate = 0.005
  drop = 0.25
  epochs_drop = 8.0
  lrate = initial_lrate * math.pow(drop,math.floor((1+epoch)/epochs_drop))
  return lrate
```

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 1s 142us/step
Test loss: 1.0070890602111817
Test accuracy: 0.7018
```

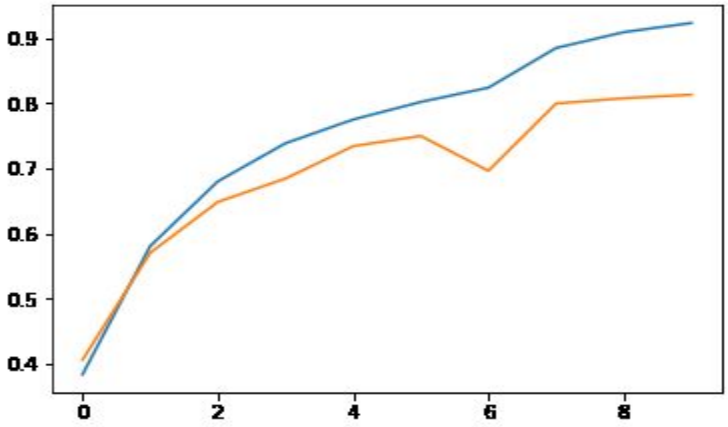
- Below is the accuracy plot between train and test:



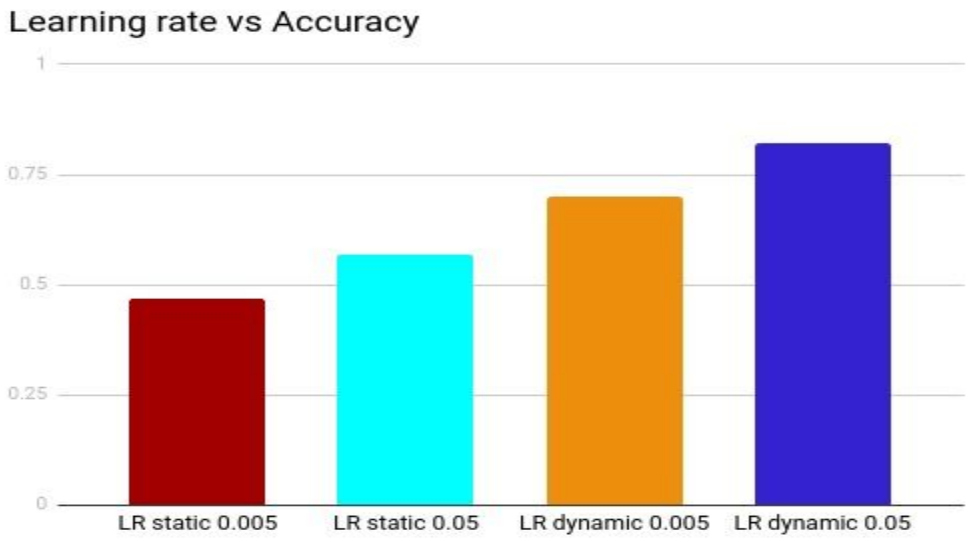
- **Step Decay with rate 0.05: Accuracy: ~83%**

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
Test loss: 0.9248582398891 45
Test accuracy: 0.832
```



- **Below are the accuracy comparisons b/w different learning rates.**



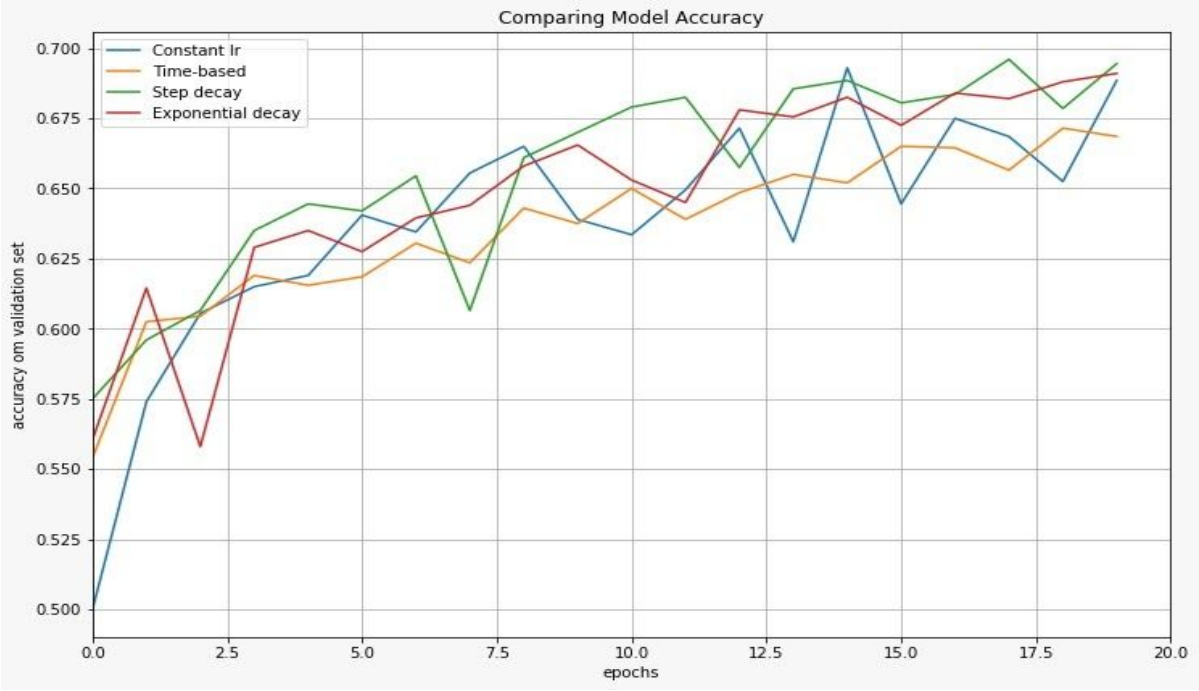
- **<u>Exponential Decay with rate 0.05:</u> Accuracy: ~68%**

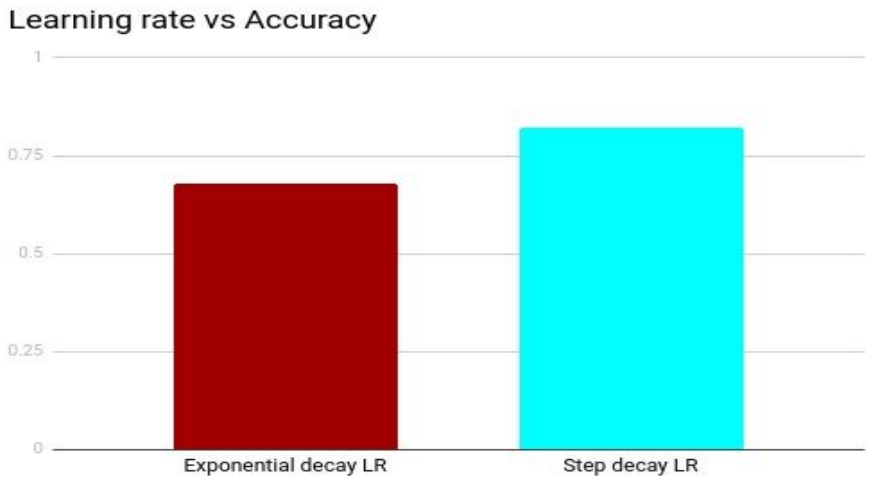
```
def lr_schedule(epoch):
  initial_lrate = 0.05
  k = 0.1
  lrate = initial_lrate *( np.exp(-k*epoch))
  return lrate
```

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 1s 144us/step
Test loss: 2.5151038160324095
Test accuracy: 0.6811
```

- Below are some of the visualizations that depict the accuracy between exponential decay, step decay and static learning rates.

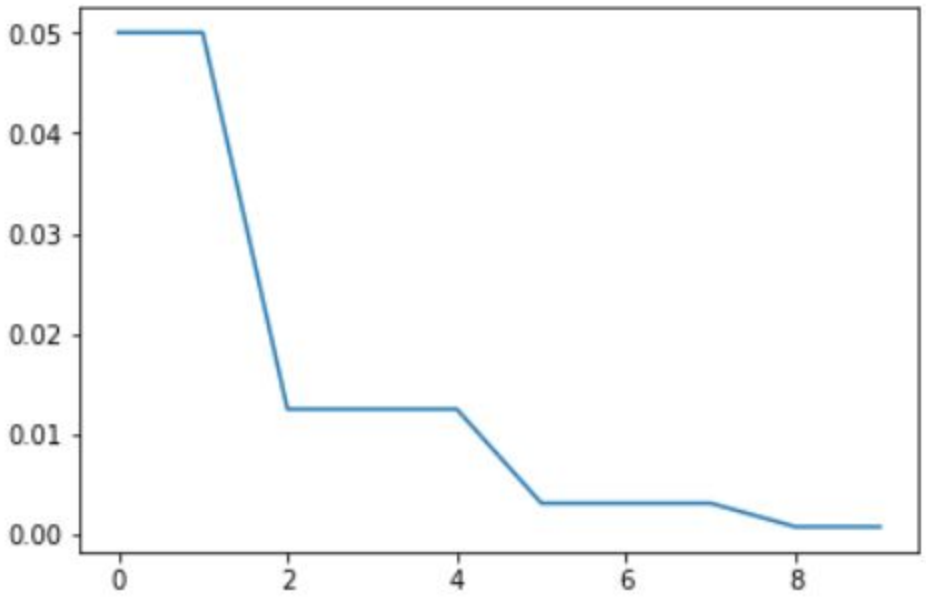




- Now let's look at the learning rates that gets updated at every epoch.
- **Learning rates in every epoch(step-decay):**

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
Learning rate:  0.05
50000/50000 [==============================] - 20s 400us/step - loss: 2.5414 - acc: 0.4303 - val_loss: 2.2301 - val_acc: 0.3799
Epoch 2/10
Learning rate:  0.05
50000/50000 [==============================] - 16s 319us/step - loss: 1.2163 - acc: 0.6173 - val_loss: 1.8898 - val_acc: 0.5145
Epoch 3/10
Learning rate:  0.0125
50000/50000 [==============================] - 16s 318us/step - loss: 0.9144 - acc: 0.7313 - val_loss: 1.1082 - val_acc: 0.6613
Epoch 4/10
Learning rate:  0.0125
50000/50000 [==============================] - 16s 321us/step - loss: 0.8121 - acc: 0.7688 - val_loss: 1.0000 - val_acc: 0.7088
Epoch 5/10
Learning rate:  0.0125
50000/50000 [==============================] - 16s 311us/step - loss: 0.7483 - acc: 0.7914 - val_loss: 0.9926 - val_acc: 0.7095
Epoch 6/10
Learning rate:  0.003125
50000/50000 [==============================] - 16s 312us/step - loss: 0.6334 - acc: 0.8346 - val_loss: 0.8500 - val_acc: 0.7576
Epoch 7/10
Learning rate:  0.003125
50000/50000 [==============================] - 16s 314us/step - loss: 0.5922 - acc: 0.8480 - val_loss: 0.8579 - val_acc: 0.7581
Epoch 8/10
Learning rate:  0.003125
50000/50000 [==============================] - 16s 318us/step - loss: 0.5646 - acc: 0.8604 - val_loss: 0.8497 - val_acc: 0.7641
Epoch 9/10
Learning rate:  0.00078125
50000/50000 [==============================] - 16s 315us/step - loss: 0.5222 - acc: 0.8758 - val_loss: 0.8312 - val_acc: 0.7683
Epoch 10/10
Learning rate:  0.00078125
50000/50000 [==============================] - 16s 311us/step - loss: 0.5092 - acc: 0.8818 - val_loss: 0.8376 - val_acc: 0.7697
```

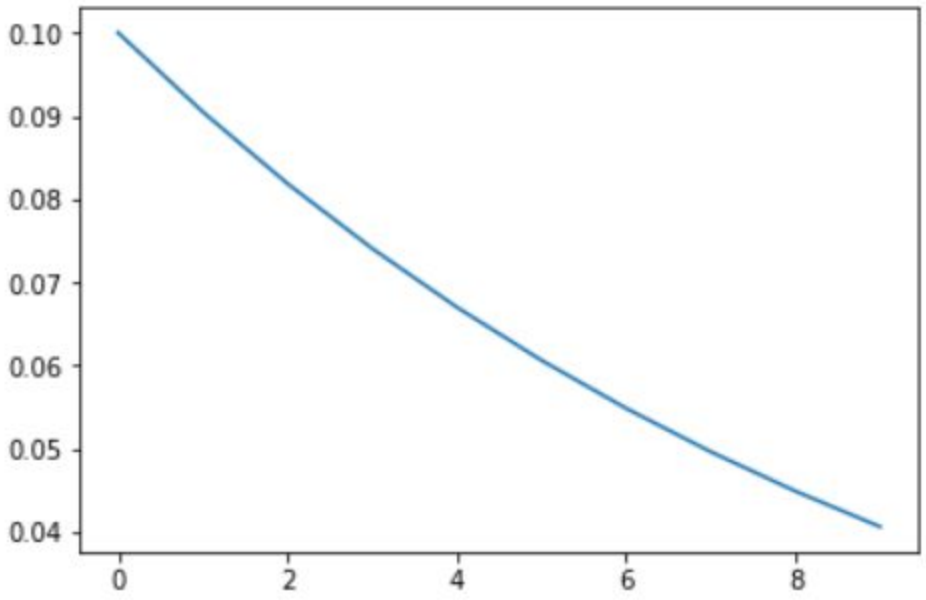
- **Learning rate variation graph(step-decay):**



- **Learning rates in every epoch(exponential-decay):**

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
Learning rate:  0.1
50000/50000 [==============================] - 19s 376us/step - loss: 3.1493 - acc: 0.3866 - val_loss: 4.6798 - val_acc: 0.2685
Epoch 2/10
Learning rate:  0.09048374180359596
50000/50000 [==============================] - 16s 317us/step - loss: 2.7195 - acc: 0.5490 - val_loss: 3.0889 - val_acc: 0.4579
Epoch 3/10
Learning rate:  0.0818730753077982
50000/50000 [==============================] - 16s 316us/step - loss: 2.5123 - acc: 0.6304 - val_loss: 2.7288 - val_acc: 0.5665
Epoch 4/10
Learning rate:  0.0740818220681718
50000/50000 [==============================] - 16s 314us/step - loss: 2.3861 - acc: 0.6780 - val_loss: 2.5813 - val_acc: 0.6263
Epoch 5/10
Learning rate:  0.06703200460356394
50000/50000 [==============================] - 16s 314us/step - loss: 2.2993 - acc: 0.7106 - val_loss: 2.5740 - val_acc: 0.6264
Epoch 6/10
Learning rate:  0.06065306597126335
50000/50000 [==============================] - 15s 309us/step - loss: 2.2269 - acc: 0.7366 - val_loss: 2.4138 - val_acc: 0.6715
Epoch 7/10
Learning rate:  0.05488116360940264
50000/50000 [==============================] - 16s 311us/step - loss: 2.1640 - acc: 0.7599 - val_loss: 2.4982 - val_acc: 0.6549
Epoch 8/10
Learning rate:  0.04965853037914095
50000/50000 [==============================] - 16s 312us/step - loss: 2.1194 - acc: 0.7737 - val_loss: 2.3805 - val_acc: 0.6938
Epoch 9/10
Learning rate:  0.044932896411722156
50000/50000 [==============================] - 16s 317us/step - loss: 2.0703 - acc: 0.7922 - val_loss: 2.6160 - val_acc: 0.6506
Epoch 10/10
Learning rate:  0.04065696597405991
50000/50000 [==============================] - 16s 319us/step - loss: 2.0301 - acc: 0.8052 - val_loss: 2.5151 - val_acc: 0.6811
```

- **Learning rate variation graph(exponential-decay):**



- **Observations:**
  - Using Smaller running rates lead to requiring many updates. And using larger running rates causes drastic updates.
  - Static learning rate might not be a good way to use, as the number of epochs might be variant with the learning rate.
  - And also from the accuracy comparisons between static and dynamic learning rates, use of dynamic learning is a good practice. Hence its always best to use dynamic learning rates.
  - Also between step decay and exponential decay learning rate, it is good to use step decay since it resulted in good accuracy.
  - Chosen the learning rate i.e. is stable in nature and not overfitting.

- For a very high learning rate, the cost would explode and the updates are very drastic here:

```
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

- For a little bit high learning rate, the loss starts at a high value and then gradually decreases later with fewer updates to reach the minimal point:

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
```

- For a good value of learning rate, the loss swiftly decreases:

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
```

INTRODUCING SGD MOMENTUM:

- **With Momentum: Accuracy: ~82%**

momentum=0.9+SGD

```
if lr_scheduler_strategy:
  sgd = optimizers.SGD(lr=0.0, momentum=0.9, decay=0.0, nesterov=False)
  optimizer=sgd
```

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 2s 179us/step
Test loss: 0.7553816204547882
Test accuracy: 0.813
```

- **<u>Without Momentum:</u> Accuracy: ~69%**

```
if lr_scheduler_strategy:
  sgd = optimizers.SGD(lr=0.0, momentum=0.0, decay=0.0, nesterov=False)
  optimizer=sgd
```

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
10000/10000 [==============================] - 2s 158us/step
Test loss: 1.0268409734725952
Test accuracy: 0.6934
```

- **Observations:**
  - Using momentum actually helps. In general, momentum speeds up the learning process.
  - Since we got better accuracy with momentum and as it also accelerates (which thus leading to faster gradients) we prefer to use it.

OPTIMIZERS:

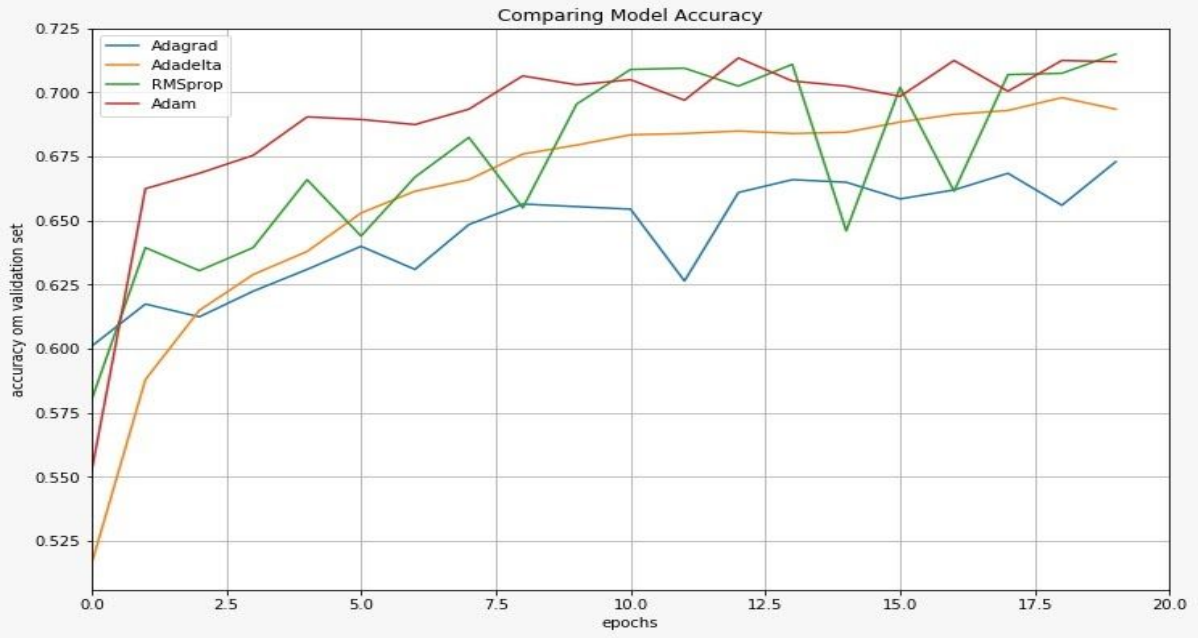
Explored the accuracies by using different optimizers.

The following are the parameters that are kept constant throughout the visualizations:

- Batch_size = 128
- Epochs = 10
- Activation = ReLU
- Normalization = True
- Batch Normalization = True
- Loss function = Categorical Cross entropy.
- Learning rate = Dynamic.
- Only optimizers are kept varying.

- Optimizers like adagrad, adadelta, RMSprop and Adam are used.
- Below is the obtained accuracy visualization between different optimizers.

-



DATA AUGMENTATION:

In general, we need lots of data to train the weights in a CNN model. But instead, we generate an extra set of data using Data augmentation.

Data augmentation can be of any image manipulation techniques like flipping, adding noise, smoothing etc…

The following are the parameters that are kept constant throughout the visualizations:

- Batch_size = 128

- Epochs = 10
- Activation = ReLU
- Normalization = True
- Batch Normalization = True
- Loss function = Categorical Cross entropy.
- Learning rate = Dynamic.
- Optimizer = Adam.

Using Data augmentation we were able to increase our accuracy by 4%.

```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
Test loss: 0.630527995300293
Test accuracy: 0.8762
```

- **With Data Augmentation: Accuracy: ~87%**
- **Without Data Augmentation: Accuracy: ~83%**

TRANSFER LEARNING:

It is a process of assigning weights of some standard pre trained model into a model that needs to be trained. In general, we assign weights of Image Net dataset model.

Since our model has got sufficient data and the model we are trying initialize (ImageNet) is of a similar dataset to CIFAR10 dataset. Hence, we just need to fine-tune the model.

We obtained an accuracy of around 86%. (This is without using Data Augmentation).

- **With Transfer Learning: Accuracy: ~86%**

BABY SITTING THE PROCESS:

- **Step-1: Preprocess the data: Normaling the data**
  - **This is done and explained under the section 'Data Preprocessing and Normalization'. (Mentioned before)**
- **Step-2: Architecture, Loss minimization.**
  - **The chosen architecture was ResNet.**
  - **All kinds of loss minimization and regularization was done and explained under the section 'Regularization'. (Mentioned Before)**
- **Step-3: Finding a good Learning Rate and regularizer**
  - **This is done and explained under the section 'Regularization', 'Learning Rates and Learning Strategies'. (Mentioned before).**
  - **All other kinds of exploring and hyperparameter tuning is explained under other sections like 'Activation Functions', 'Optimizer' etc…**

OBSERVATION SUMMARY - WHAT HYPERPARAMETERS TO CHOOSE AND CHOSEN IN THIS ASSIGNMENT:

- **Activation Function:** ReLU.
- **Data Preprocessing - Normalization:** Subtract Mean of all images.
- **Regularization:** Batch Normalization.

- **Loss functions:** Since it is CIFAR10 classification, use Cross entropy. (Using Binary cross entropy resulted in very high accuracy!!)
- **Optimizer:** Adam, use SGD + momentum.
- **Learning Rate:** Step Decay
- Use Data Augmentation.

Above are the hyperparameters chosen in this assignment and obtained the best accuracy of around ~87%.

Using Data Augmentation and Transfer learning together could have achieved higher accuracy (i.e. more than 87%).

P.S: All the observations for hyperparameter tuning are written at the end of each section (under the heading 'Observations').

Files submitted:

- All kinds of Hyperparameter tuning is done under the file name 'Hyperparameter Tuning.ipynb'.
- Highest Accuracy obtained is under the file name 'Highest Accuracy.ipynb'.
- Transfer Learning is implemented under the file name 'Transfer Learning.ipynb'.

sources:

- https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1
- https://medium.com/@GeneAshis/fast-ai-season-1-episode-7-1-performance-of-different-neural-networks-on-cifar-10-dataset-c6559595b529
- https://www.codesofinterest.com/2017/03/graph-model-training-history-keras.html
- https://kharshita.github.io/blog/2018/08/10/transfer-learning
- https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/

- https://github.com/keras-team/keras
- https://www.analyticsvidhya.com/blog/2018/10/understanding-inception-network-from-scratch/

Prepared By:

Durga Yasasvi Y, IMT2016060.

Siddharth Reddy D, IMT2016037.

Srujan Swaroop G, IMT2016033.