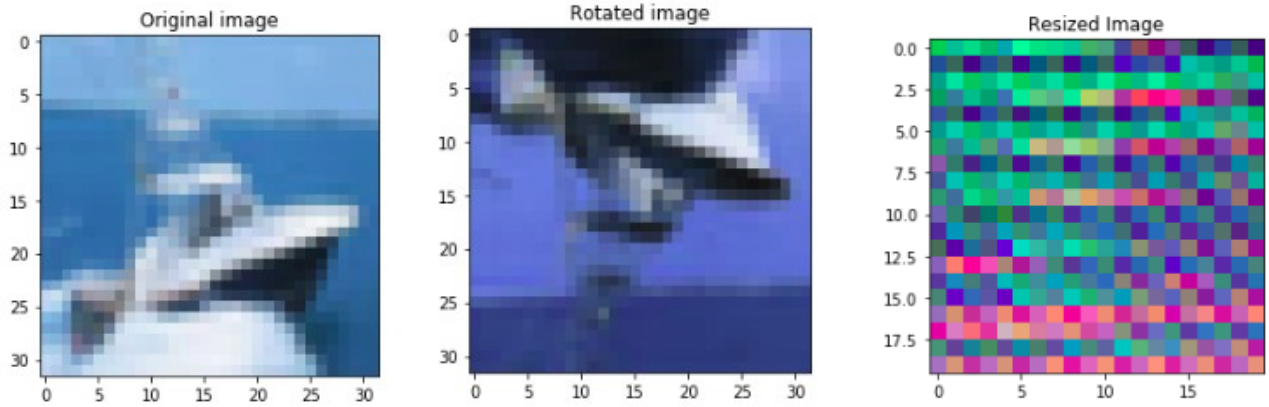
Lab Course Distributed Data Analytics
Exercise Sheet 7
Krithika Murugesan - 277537

The given dataset is CIFAR-10, with image data of ten different objects classified. It is available in five batches along with a test set. Due to computational limitations of the laptop, only one batch is used to train the network which might affect the performance.

The data is read from the folder, the predictor space is currently a 2-D array, this has to be converted to image dimensions, with a color channel, then thee output variable being categorical is one-hot encoded. The following code snippet performs these operations and gives a clean test and train data to work on.

```
#Read the data and labels
def readData(file):
    from six.moves import cPickle
    with open(file, 'rb') as fo:
        dict = cPickle.load(fo,encoding='latin1')
        data = dict['data']
        labels = dict['labels']
    return(data,labels)
```

Though we have the data, this might not be enough, to add more data to the training from that available we do data augmentation. Here two things are done, one is resizing the image to produce variations of the same data and also flipping the images around, so that the actual data it represents is same but with changes in orientation.



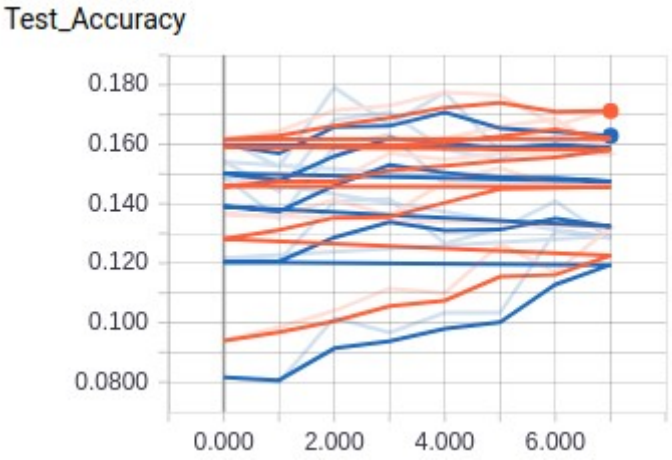
Exercise 1: Normalization Effect

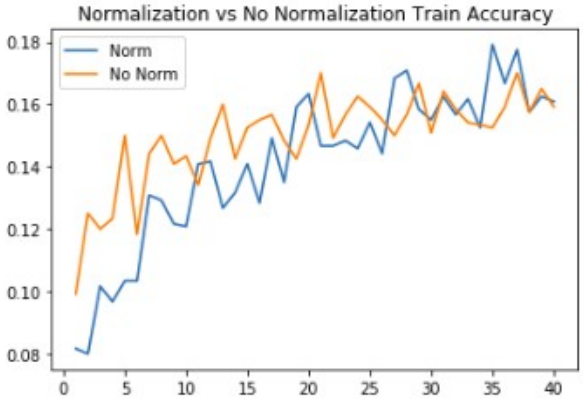
Tensorflow provides batch normalization, that is at the end of every mini-batch, he layers are whitened. It helps in faster convergence of the algorithm and augments regularization. That is it avoids over fitting of the data, that it cannot generalize on normal data. Two networks of the given architecture were built, one with normalization and one without it. The results were plotted in Tensorflow, but for easier understanding, they are also shown at each iteration and it can be seen that the both the variations also perform similarly on the train data, but the test accuracy for normalized network is higher as it generalizes the data better. The code snippet and results are given below;

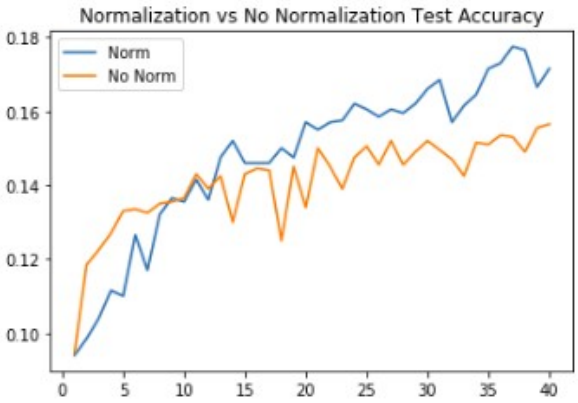
Network with Batch normalization

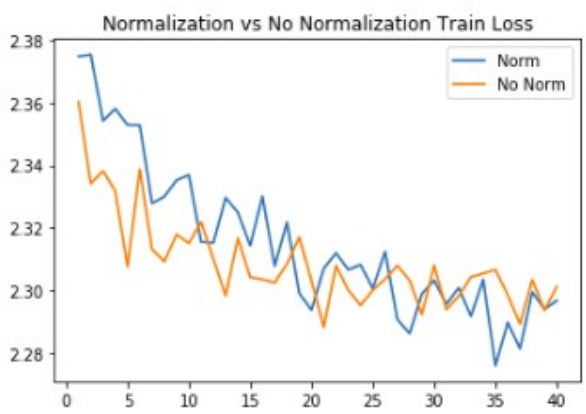
```
def conv_net(x, keep_prob):
    conv1_filter = tf.Variable(tf.truncated_normal(shape=[3, 3, 3, 64], mean=0, stddev=0.08))
    conv2_filter = tf.Variable(tf.truncated_normal(shape=[3, 3, 64, 128], mean=0, stddev=0.08))

    # 1, 2
    conv1 = tf.nn.conv2d(x, conv1_filter, strides=[1,1,1,1], padding='SAME')
    conv1 = tf.nn.relu(conv1)
    conv1_pool = tf.nn.max_pool(conv1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
    conv1_bn = tf.layers.batch_normalization(conv1_pool)

    # 3, 4
    conv2 = tf.nn.conv2d(conv1_bn, conv2_filter, strides=[1,1,1,1], padding='SAME')
    conv2 = tf.nn.relu(conv2)
    conv2_pool = tf.nn.max_pool(conv2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
    conv2_bn = tf.layers.batch_normalization(conv2_pool)

    # 9
    flat = tf.contrib.layers.flatten(conv2_bn)

    # 10
    full1 = tf.contrib.layers.fully_connected(inputs=flat, num_outputs=128, activation_fn=tf.nn.relu)
    full1 = tf.nn.dropout(full1, keep_prob)
    full1 = tf.layers.batch_normalization(full1)

    # 11
    full2 = tf.contrib.layers.fully_connected(inputs=full1, num_outputs=256, activation_fn=tf.nn.relu)
    full2 = tf.nn.dropout(full2, keep_prob)
    full2 = tf.layers.batch_normalization(full2)

    # 14
    out = tf.contrib.layers.fully_connected(inputs=full2, num_outputs=10, activation_fn=None)
    out = tf.nn.softmax(out)
    return out
```

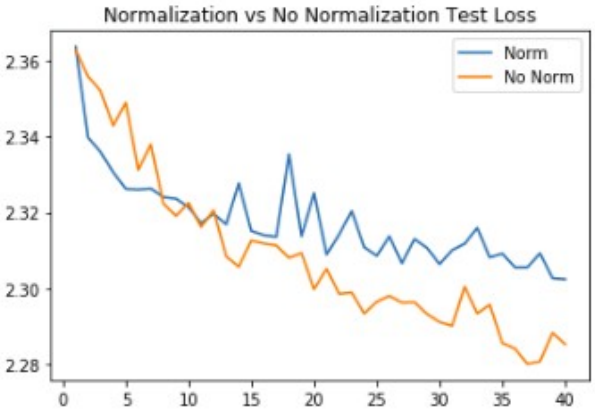
Similarly the batch normalization was removed to implement the next variation. The performance graphs are,







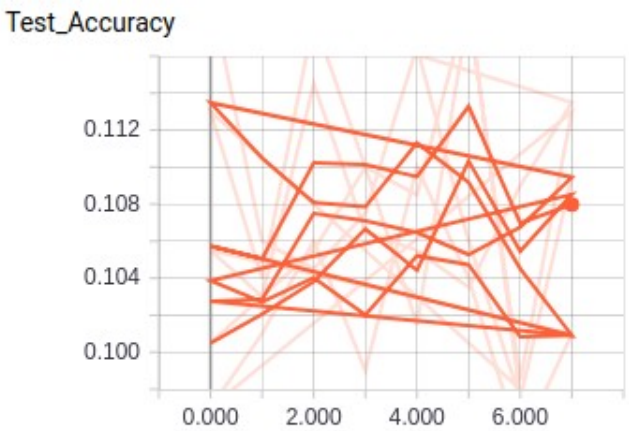


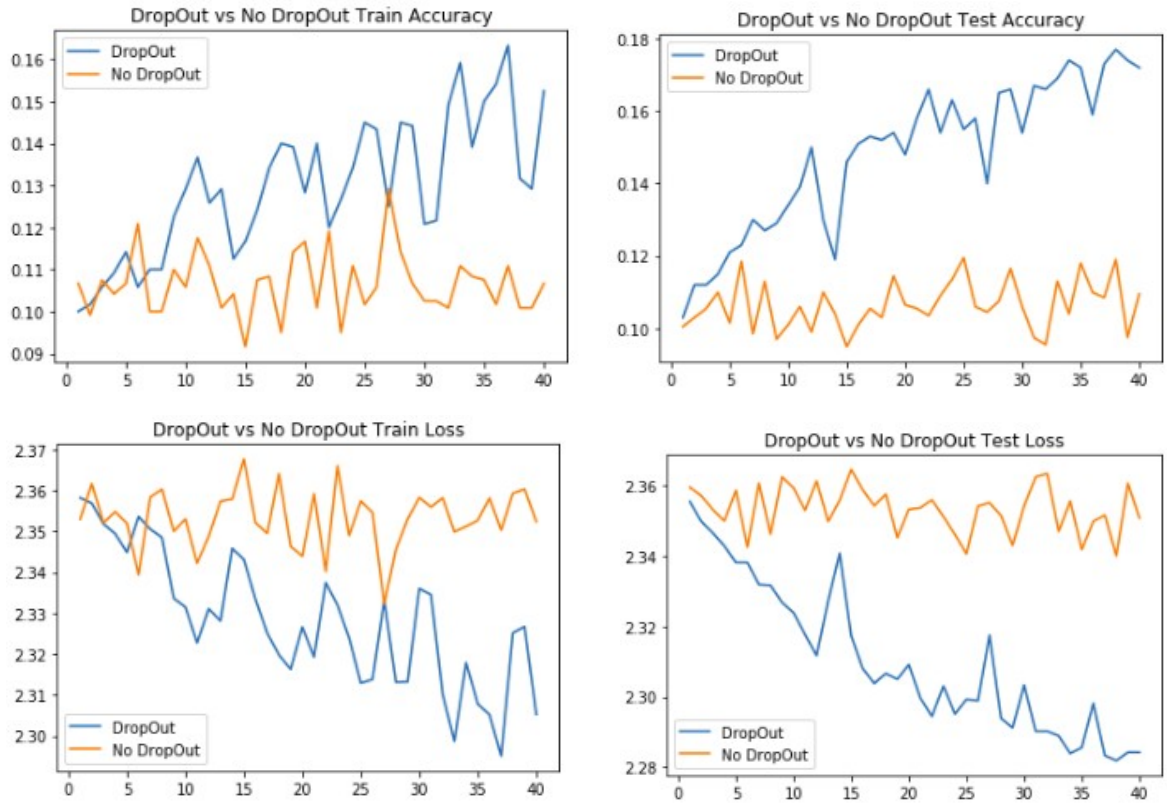


Exercise 2 : Network Regularization

Neural Networks are so good that they can learn the training data along with noises in them, making them perform bad on unseen or test data. Since, there are huge number of neurons one way to reduce over fitting is leaving off certain nodes during the forward and backward propagation. This generalizes the model, making the performance n test set better. The neurons are dropped with a probability of 0.5, the implementation also shows similar results. The network with drop out performs on accuracy, but does not converge as fast normal network without drop out.

```
def conv_net(x, keep_prob):
    conv1_filter = tf.Variable(tf.truncated_normal(shape=[3, 3, 3, 64], mean=0, stddev=0.08))
    conv2_filter = tf.Variable(tf.truncated_normal(shape=[3, 3, 64, 128], mean=0, stddev=0.08))

    # 1, 2
    conv1 = tf.nn.conv2d(x, conv1_filter, strides=[1,1,1,1], padding='SAME')
    conv1 = tf.nn.relu(conv1)
    conv1_pool = tf.nn.max_pool(conv1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
    conv1_bn = tf.layers.batch_normalization(conv1_pool)

    # 3, 4
    conv2 = tf.nn.conv2d(conv1_bn, conv2_filter, strides=[1,1,1,1], padding='SAME')
    conv2 = tf.nn.relu(conv2)
    conv2_pool = tf.nn.max_pool(conv2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
    conv2_bn = tf.layers.batch_normalization(conv2_pool)

    # 9
    flat = tf.contrib.layers.flatten(conv2_bn)

    # 10
    full1 = tf.contrib.layers.fully_connected(inputs=flat, num_outputs=128, activation_fn=tf.nn.relu)
    full1 = tf.nn.dropout(full1, 0.5)
    full1 = tf.layers.batch_normalization(full1)

    # 11
    full2 = tf.contrib.layers.fully_connected(inputs=full1, num_outputs=256, activation_fn=tf.nn.relu)
    full2 = tf.nn.dropout(full2, 0.5)
    full2 = tf.layers.batch_normalization(full2)

    # 14
    out = tf.contrib.layers.fully_connected(inputs=full2, num_outputs=10, activation_fn=None)
    out = tf.nn.softmax(out)
    return out
```





Exercise 3: Optimizers

There are several optimizers to minimize the cost function, they are used to learn the network parameters such that the results converge to global minimum. Gradient Descent being the basic of them. In this exercise two optimizers have to be compared RMSProp and Adam namely. Root Mean Square Propagation (RMSProp) maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight based on how quickly it is changes. This means the algorithm does well on online and non-stationary problems. Adam is an optimization algorithm that can used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data. Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance)

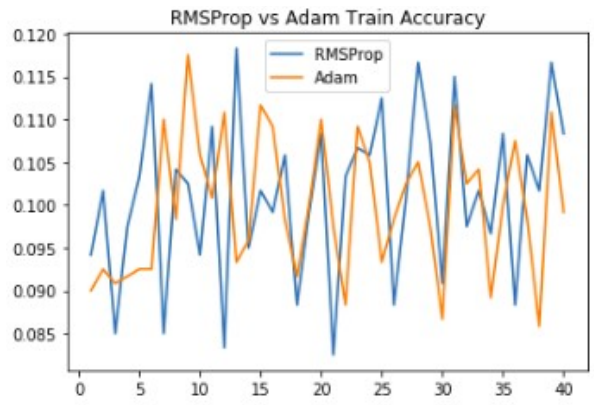
Both the optimizers are implemented with the same network structure, since it is trained only for a few epochs, no huge differences can be seen.

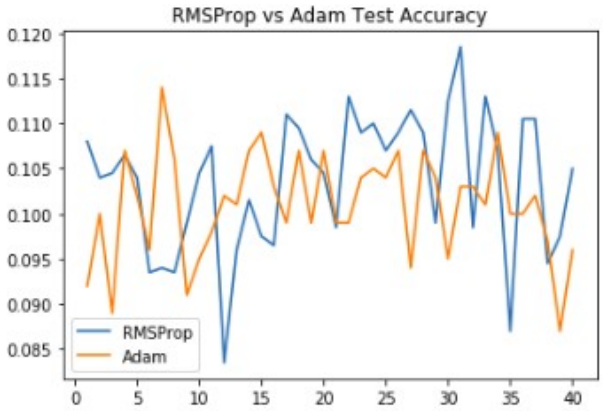
```
# Loss and Optimizer
with tf.name_scope("cost"):
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))

tf.summary.histogram("cost", cost)
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate).minimize(cost)
```

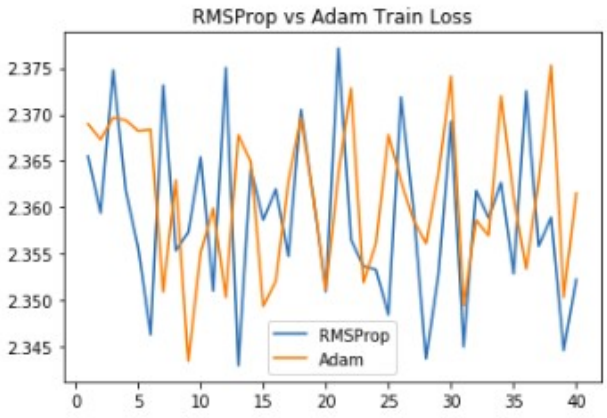
```
# Loss and Optimizer
with tf.name_scope("cost"):
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))

tf.summary.histogram("cost", cost)
optimizer =   tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
```

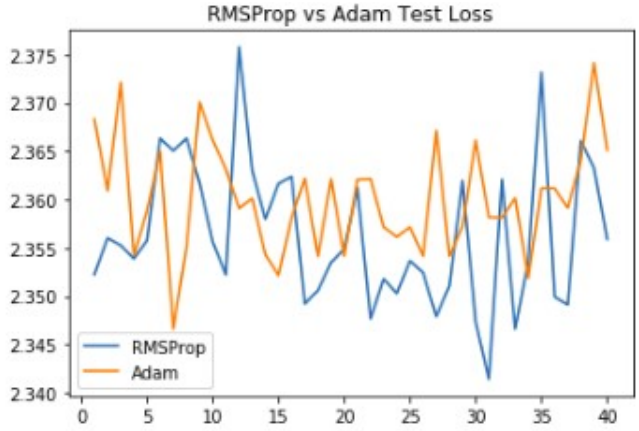
```
conv1_filter = tf.Variable(tf.truncated_normal(shape=[3, 3, 3, 64], mean=0, stddev=0.08))
conv2_filter = tf.Variable(tf.truncated_normal(shape=[3, 3, 64, 128], mean=0, stddev=0.08))

# 1, 2
conv1 = tf.nn.conv2d(x, conv1_filter, strides=[1,1,1,1], padding='SAME')
conv1 = tf.nn.relu(conv1)
conv1_pool = tf.nn.max_pool(conv1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
conv1_bn = tf.layers.batch_normalization(conv1_pool)

# 3, 4
conv2 = tf.nn.conv2d(conv1_bn, conv2_filter, strides=[1,1,1,1], padding='SAME')
conv2 = tf.nn.relu(conv2)
conv2_pool = tf.nn.max_pool(conv2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
conv2_bn = tf.layers.batch_normalization(conv2_pool)

# 9
flat = tf.contrib.layers.flatten(conv2_bn)

# 10
full1 = tf.contrib.layers.fully_connected(inputs=flat, num_outputs=128, activation_fn=tf.nn.relu)
full1 = tf.nn.dropout(full1, 0.5)
full1 = tf.layers.batch_normalization(full1)

# 11
full2 = tf.contrib.layers.fully_connected(inputs=full1, num_outputs=256, activation_fn=tf.nn.relu)
full2 = tf.nn.dropout(full2, 0.5)
full2 = tf.layers.batch_normalization(full2)

# 14
out = tf.contrib.layers.fully_connected(inputs=full2, num_outputs=10, activation_fn=None)
out = tf.nn.softmax(out)
```

The performance graphs are as follows,

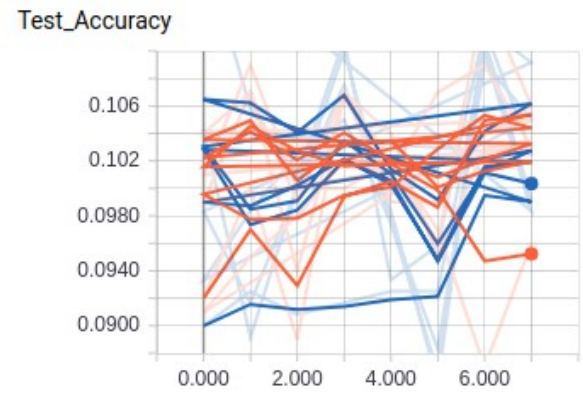




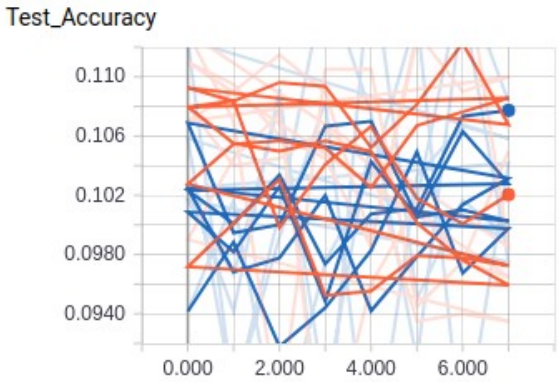




All the scalars and graphs were implemented in Tensor board for all exercises as well, for clarity purpose they are not illustrated. Some sample screen shots of network for this exercise are given below,



Adam Optimizer



RMSProp Optimizer