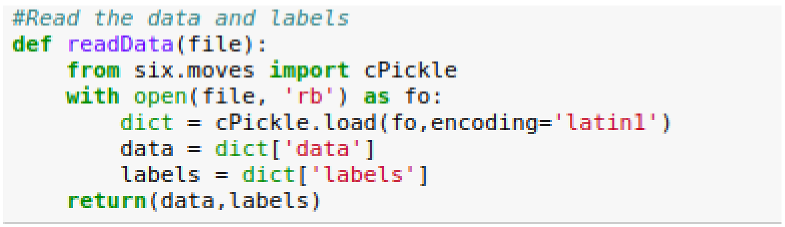
Distributed Data Analytics Lab - Exercise 06
Krithika Murugesan - 277537

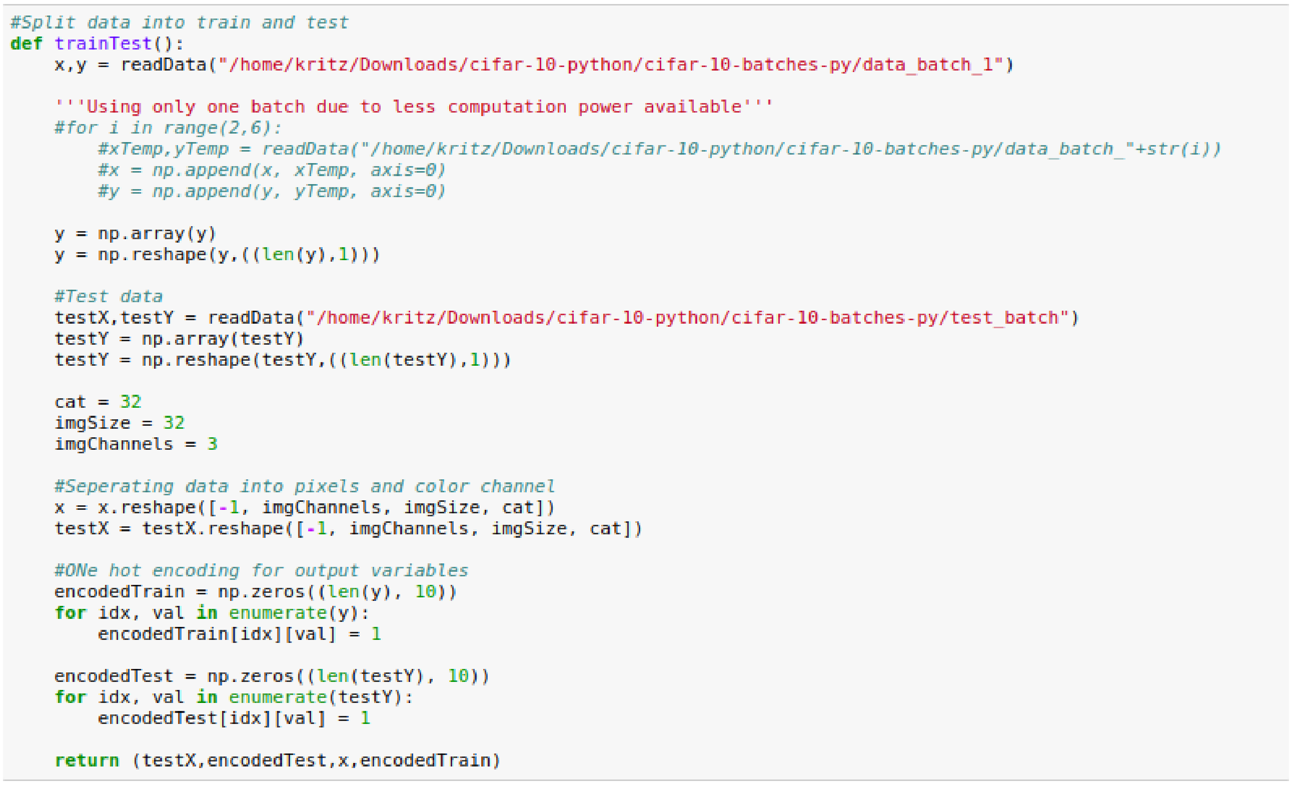
Exercise 1 : Convolutional Neural Network

CNNs are very powerful neural networks, they are capable of computing convolutions in data with fewer number of calculations compared to fully-connected layers. 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.



```python
#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)
```



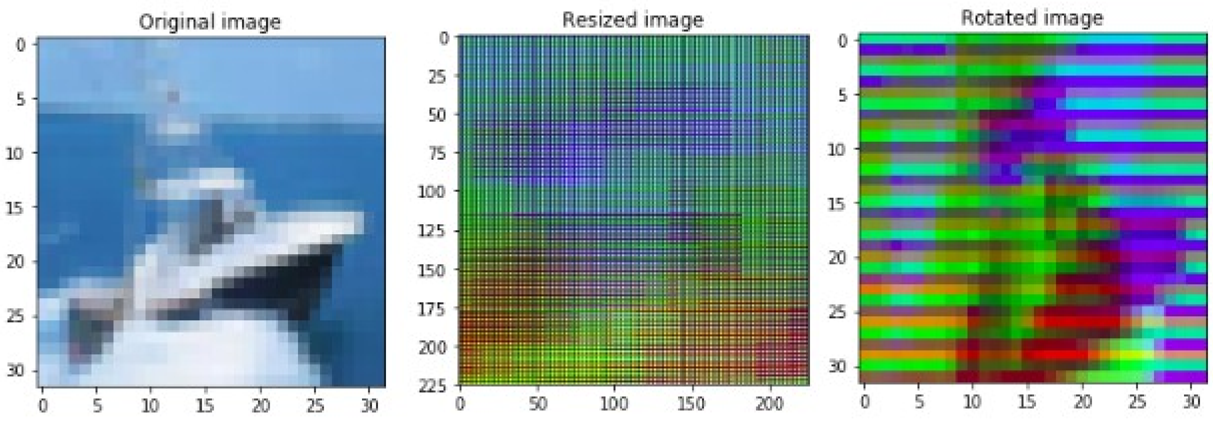
```python
#Split data into train and test
def trainTest():
    x,y = readData("/home/kritz/Downloads/cifar-10-python/cifar-10-batches-py/data_batch_1")

    '''Using only one batch due to less computation power available'''
    #for i in range(2,6):
        #xTemp,yTemp = readData("/home/kritz/Downloads/cifar-10-python/cifar-10-batches-py/data_batch_"+str(i))
        #x = np.append(x, xTemp, axis=0)
        #y = np.append(y, yTemp, axis=0)

    y = np.array(y)
    y = np.reshape(y,((len(y),1)))

    #Test data
    testX,testY = readData("/home/kritz/Downloads/cifar-10-python/cifar-10-batches-py/test_batch")
    testY = np.array(testY)
    testY = np.reshape(testY,((len(testY),1)))

    cat = 32
    imgSize = 32
    imgChannels = 3

    #Seperating data into pixels and color channel
    x = x.reshape([-1, imgChannels, imgSize, cat])
    testX = testX.reshape([-1, imgChannels, imgSize, cat])

    #ONe hot encoding for output variables
    encodedTrain = np.zeros((len(y), 10))
    for idx, val in enumerate(y):
        encodedTrain[idx][val] = 1

    encodedTest = np.zeros((len(testY), 10))
    for idx, val in enumerate(testY):
        encodedTest[idx][val] = 1

    return (testX,encodedTest,x,encodedTrain)
```

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. If this code is used on the complete batch, the laptop becomes inactive, therefore it's implementation is shown here, while it is not used in the actual code.

```
def tfResize(each):
    IMAGE_SIZE = 225
    X_data = []
    tf.reset_default_graph()
    X = tf.placeholder(tf.float32, (None, None, 3))
    tf_img = tf.image.resize_images(X, (IMAGE_SIZE, IMAGE_SIZE), tf.image.ResizeMethod.NEAREST_NEIGHBOR)
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())

        # Each image is resized individually as different image may be of different size.
        #for each in trainX:
        resized_img = sess.run(tf_img, feed_dict = {X: each})
        #X_data.append(resized_img)

    X_data = np.array(resized_img, dtype = np.float32) # Convert to numpy
    return X_data
```

```
def tfRotate(img):
    X_rotate = []
    tf.reset_default_graph()
    X = tf.placeholder(tf.float32, shape = (3,32,32))
    k = tf.placeholder(tf.int32)
    tf_img = tf.image.rot90(X, k = k)
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        #for img in X_imgs:
        for i in range(3):  # Rotation at 90, 180 and 270 degrees
            rotated_img = sess.run(tf_img, feed_dict = {X: img, k: i + 1})
            #X_rotate.append(rotated_img)

    X_rotate = np.array(rotated_img, dtype = np.float32)
    return X_rotate
```

The output image of these functions quite depict their actual goal. The original image the resized one and the rotated resized image are as follows respectively,

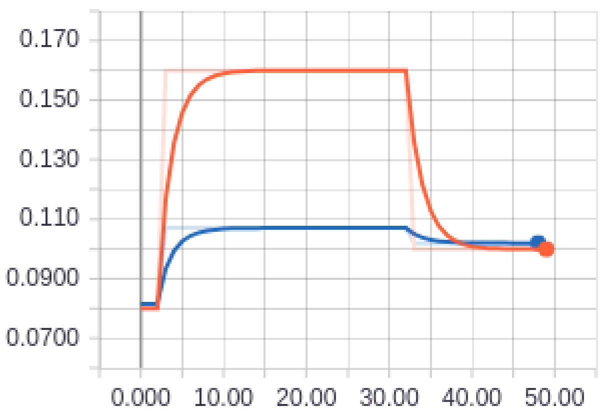


Now, the pre-processing is done, the next things left to do is to build our network. The given network structure is implemented, first a convolution layer with ReLU activation function. This layer is connected to a max pool layer which is connected fully-connected layer with ReLU function and an output layer with the softmax function. The network structure is as follows,

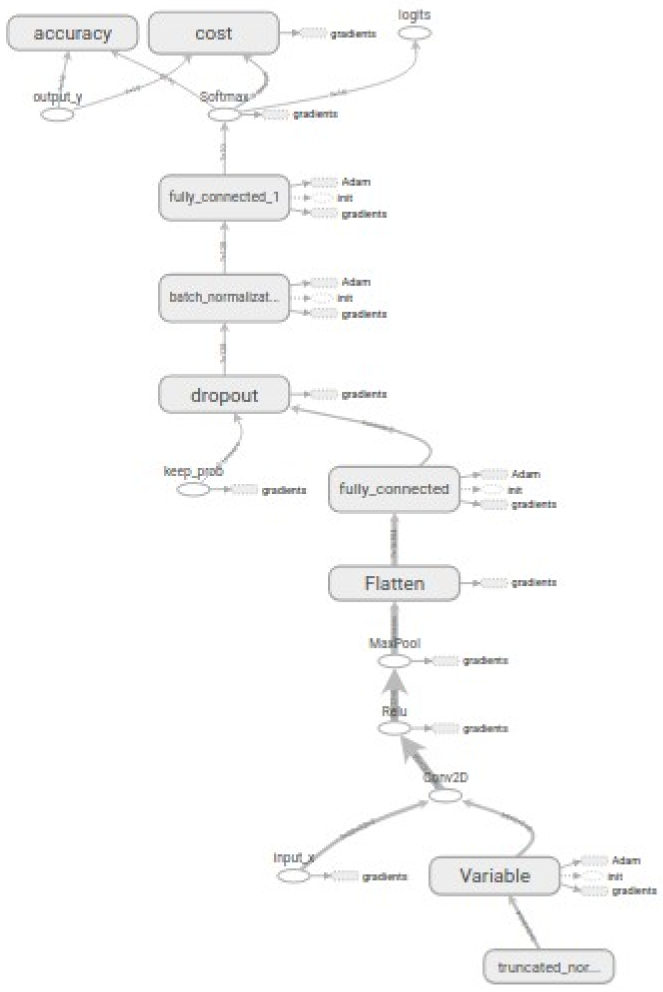
```python
def convNet(x, keep_prob):
    convFilter = tf.Variable(tf.truncated_normal(shape=[3, 3, 3, 64], mean=0, stddev=0.08))

    #Convolution layer with ReLU
    conv = tf.nn.conv2d(x, convFilter, strides=[1,1,1,1], padding='SAME')
    conv = tf.nn.relu(conv)
    #tf.summary.histogram("convolution layer", conv)

    #Max pool layer
    convPool = tf.nn.max_pool(conv, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
    #tf.summary.histogram("Max pool layer", convPool)

    #Flatten to feed as input to fully connnected layer with ReLU
    flat = tf.contrib.layers.flatten(convPool)
    full = tf.contrib.layers.fully_connected(inputs=flat, num_outputs=128, activation_fn=tf.nn.relu)
    full = tf.nn.dropout(full, keep_prob)
    full = tf.layers.batch_normalization(full)
    #tf.summary.histogram("Fully connected layer", full)

    #Output layer with softmax activation
    out = tf.contrib.layers.fully_connected(inputs=full, num_outputs=10, activation_fn=None)
    out = tf.nn.softmax(out)
    #tf.summary.histogram("Output activation", out)
    return out
```

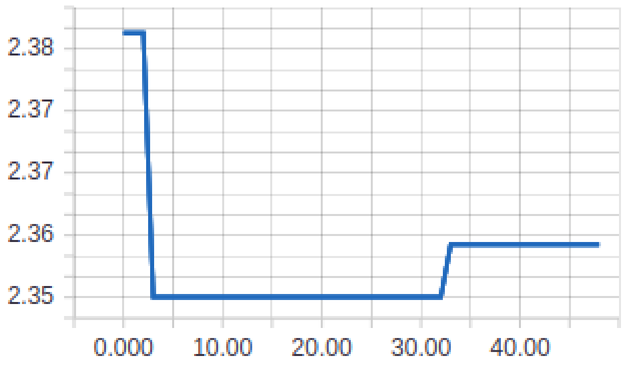
The cost function and predictions are defined as usual. Cross-entropy loss is used as the cost function an an Adam Optimizer is used. A significantly larger mini-batch size is used due to hardware and time limitations. The graphs are plotted in Tensor board.

```python
# 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)

# Accuracy
with tf.name_scope("accuracy"):
    correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(y, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32), name='accuracy')
tf.summary.histogram("accuracy", accuracy)

accuracy_summary = tf.summary.scalar("Test Accuracy", accuracy)

summary_op = tf.summary.merge_all()
with tf.Session() as sess:

    aug = trainX

    # Initializing the variables
    sess.run(tf.global_variables_initializer())

    # log writer object
    writer = tf.summary.FileWriter(logs_path, graph=tf.get_default_graph())
    test_writer = tf.summary.FileWriter(logs_path + '/test', graph=tf.get_default_graph())

    batches = int(np.ceil(len(aug) / batchSize))
    for each in range(epochs):
        for i in range(batches):
            batchX = aug[i*batchSize: (i+1)*batchSize]
            batchY = trainY[i*batchSize: (i+1)*batchSize]

            trainNn(sess, optimizer, keep_probability, batchX, batchY)
        summary,_,c,accu = sess.run([summary_op,optimizer,cost,accuracy], feed_dict={x: batchX,y: batchY,keep_prob: 1
        summ,_,acc = sess.run([accuracy_summary,optimizer,accuracy],feed_dict={x: testX[:100,],y: testY[:100,],keep_p
        test_writer.add_summary(summary, each)
        writer.add_summary(summ,each)
        writer.flush()

        print("Epoch : ",each+1)
```



The blue line represents the test accuracy, it seems a little off though. While the orange line represents the accuracy in training set. The graph of the architecture and cost distribution are shown below.



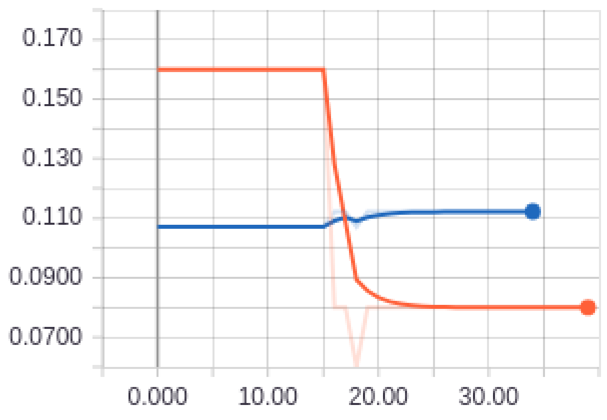


Exercise 2

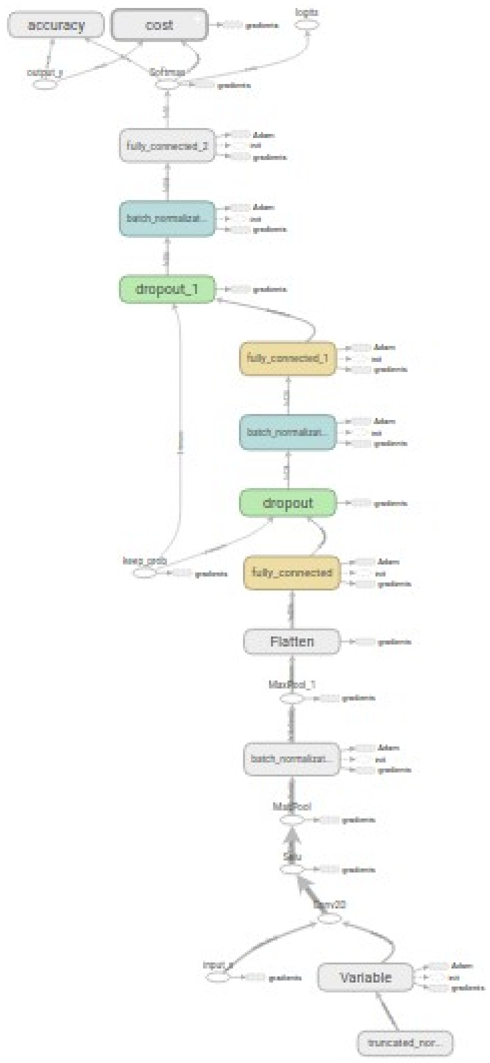
This is very similar to the first exercise except for the network structure, the network has a self-normalized activation convolution layer at the beginning, which is aided by a max pooling layer, a batch normalization layer another max pooling layer, two fully-connected layers with selu activation and a softmax layer as the output layer. The structure is implemented as follows in python

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

    #concolution layer with SeLU
    conv1 = tf.nn.conv2d(x, conv1_filter, strides=[1,1,1,1], padding='SAME')
    conv1 = tf.nn.selu(conv1)

    #Max Pooling
    conv1_pool = tf.nn.max_pool(conv1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')

    #Normalization
    conv1_bn = tf.layers.batch_normalization(conv1_pool)

    #Max pooling
    conv1_pool2 = tf.nn.max_pool(conv1_bn, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')

    #Flatten
    flat = tf.contrib.layers.flatten(conv1_pool2)

    #Fully connected layer 1 with seLU
    full1 = tf.contrib.layers.fully_connected(inputs=flat, num_outputs=128, activation_fn=tf.nn.selu)
    full1 = tf.nn.dropout(full1, keep_prob)
    full1 = tf.layers.batch_normalization(full1)

    #Fully connected layer 2 with seLU
    full2 = tf.contrib.layers.fully_connected(inputs=full1, num_outputs=256, activation_fn=tf.nn.selu)
    full2 = tf.nn.dropout(full2, keep_prob)
    full2 = tf.layers.batch_normalization(full2)

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

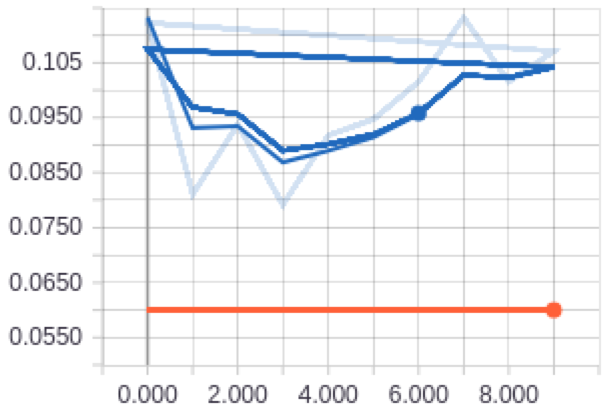
Only one batch of data is used and this being a more complex networks takes more time to execute. The other function implementations are same as in the previous case. The accuracy are as follows, where blue represents the test and orange the train.



The graph structure being,



The writer function writes the accuracy to the file at the end of every epoch, but at each iteration the accuracy is less than 0.1 and values are flushed after the epoch loss causing anomalies in the accuracy graph. The following accuracy graph for exercise 1 is same as the previous ones but for one epoch, it can be seen that the train accuracy(orange line) is fine and the overshoot previously was due to writing summaries in the outer loop.


0.105
0.0950
0.0850
0.0750
0.0650
0.0550
0.000
2.000
4.000
6.000
8.000