NAME-ISTIAQUE MANNAFEE SHAIKAT

MATRICULATION NUMBER-303527

Exercise 1: Convolutional Neural Networks for Image Classification (CNN) (10 points)

Explaining the code:

- **The functions:**

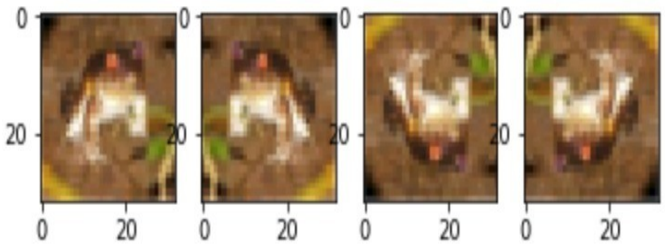
1. First function **variable_summaries** is used to create the tensorboard scalar and histogram.

2. **Load_label_names** and **load_cfar10_batch** is used to load the data of features and labels to train the model.

```
def variable_summaries(var,name):
  """Attach a lot of summaries to a Tensor (for TensorBoard visualization)."""
  with tf.name_scope('summaries_'+name):
    mean = tf.reduce_mean(var)
    tf.summary.scalar('mean_'+name, mean)
    tf.summary.scalar(name+'_value',var)
    tf.summary.histogram('histogram_'+name, var)

def load_label_names():
    return ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

def load_cfar10_batch(cifar10_dataset_folder_path, batch_id):
    with open(cifar10_dataset_folder_path + '/data_batch_' + str(batch_id), mode='rb') as file:
        # note the encoding type is 'latin1'
        batch = pickle.load(file, encoding='latin1')

    features = batch['data'].reshape((len(batch['data']), 3, 32, 32)).transpose(0, 2, 3, 1)
    labels = batch['labels']
    return features, labels
```

3. After collecting all the data it preprocessed through several layers they are normalization(min-max feature scaling),One-hot-encoding and **data augmentation**. The data is deformed to different shape or structure using 3 variant(rotation, horizontal_flip and vertical_flip) and an augmented shape of 5000 is made to create 5000 more data like this with the same label. Then this data is added to the training set to train the neural network

BEFORE AFTER



```
def normalize(x):
    min_val = np.min(x)
    max_val = np.max(x)
    x = (x-min_val) / (max_val-min_val)
    return x

def one_hot_encode(x):
    encoded = np.zeros((len(x), 10))
    for idx, val in enumerate(x):
        encoded[idx][val] = 1
    return encoded

def data_aug(features,labels,augment_size=5000):
    image_generator = ImageDataGenerator(rotation_range=90,horizontal_flip=True,vertical_flip=True)
    train_size=features.shape[0]
#   get transformed images
    randidx = np.random.randint(train_size, size=augment_size)

    labels=np.array(labels)
    x_augmented = features[randidx].copy()
    y_augmented = labels[randidx].copy()
    x_augmented = image_generator.flow(x_augmented, np.zeros(augment_size),
                                       batch_size=augment_size, shuffle=False).next()[0]
#   append augmented data to trainset
    trainx = np.concatenate((features, x_augmented))
    trainy= np.concatenate((labels, y_augmented))
    return trainx,trainy
```

4. This **_preprocess_and_save** is used to augument,normalize and one hot code the labels then save a dump file of it for later use.

```
def _preprocess_and_save(normalize, one_hot_encode,data_aug,features, labels, filename):
    features,labels=data_aug(features,labels)
    features = normalize(features)
    labels = one_hot_encode(labels)

    pickle.dump((features, labels), open(filename, 'wb'))
```

5. In mini batch optimization function the validation data(1%) is separated from training data then preprocess the whole data set(90%) which is normalize the features, one hot encoding and augmentation then save the file. In case if validation and test data same process is followed.

```
def minibatchoptimization(cifar10_dataset_folder_path, normalize, one_hot_encode,data_aug):
    n_batches = 5
    valid_features = []
    valid_labels = []

    for batch_i in range(1, n_batches + 1):
        features, labels = load_cfar10_batch(cifar10_dataset_folder_path, batch_i)

        # find index to be the point as validation data in the whole dataset of the batch (10%)
        index_of_validation = int(len(features) * 0.1)

        # preprocess the 90% of the whole dataset of the batch
        # - normalize the features
        # - one_hot_encode the lables
        # - save in a new file named, "preprocess_batch_" + batch_number
        # - each file for each batch
        _preprocess_and_save(normalize, one_hot_encode,data_aug,
                             features[:-index_of_validation], labels[:-index_of_validation],
                             'preprocess_batch_' + str(batch_i) + '.p')

        # unlike the training dataset, validation dataset will be added through all batch dataset
        # - take 10% of the whold dataset of the batch
        # - add them into a list of
        #   - valid_features
        #   - valid_labels
        valid_features.extend(features[-index_of_validation:])
        valid_labels.extend(labels[-index_of_validation:])

    # preprocess the all stacked validation dataset
    _preprocess_and_save(normalize, one_hot_encode,data_aug,
                         np.array(valid_features), np.array(valid_labels),
                         'preprocess_validation.p')

    # load the test dataset
    with open(cifar10_dataset_folder_path + '/test_batch', mode='rb') as file:
        batch = pickle.load(file, encoding='latin1')

    # preprocess the testing data
    test_features = batch['data'].reshape((len(batch['data']), 3, 32, 32)).transpose(0, 2, 3, 1)
    test_labels = batch['labels']

    # Preprocess and Save all testing data
    _preprocess_and_save(normalize, one_hot_encode,data_aug,
                         np.array(test_features), np.array(test_labels),
                         'preprocess_testing.p')
```

6. This is the architecture which I followed according to the question in CNN

    1. conv1: convolution and rectified linear activation (RELU)
    2. pool1: max pooling
    3. Flatten the data
    4. FC1: fully connected layer with rectified linear activation (RELU)

```
tf.reset_default_graph()

# Inputs
x = tf.placeholder(tf.float32, shape=(None, 32, 32, 3), name='input_x')
y =  tf.placeholder(tf.float32, shape=(None, 10), name='output_y')
keep_prob = tf.placeholder(tf.float32, name='keep_prob')

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


    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)
    variable_summaries(conv1,'conv1')
    variable_summaries(conv1_pool,'conv1_pool')
    variable_summaries(conv1,'conv1_bn')



    flat = tf.contrib.layers.flatten(conv1)
    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)
    variable_summaries(full1,'full1')



    out = tf.contrib.layers.fully_connected(inputs=full1, num_outputs=10, activation_fn=None)
    return out
```

7. These two functions are used to create batches of features and labels from the saved file **preprocess_batch_** to train the data batch by batch in every epoch

```
def batch_features_labels(features, labels, batch_size):
    """
    Split features and labels into batches
    """
    for start in range(0, len(features), batch_size):
        end = min(start + batch_size, len(features))
        yield features[start:end], labels[start:end]

def load_preprocess_training_batch(batch_id, batch_size):
    """
    Load the Preprocessed Training data and return them in batches of <batch_size> or less
    """
    filename = 'preprocess_batch_' + str(batch_id) + '.p'
    features, labels = pickle.load(open(filename, mode='rb'))

    # Return the training data in batches of size <batch_size> or less
    return batch_features_labels(features, labels, batch_size)
```

8. We used L2 regularization in loss function.then this loss is minimized in adam optimizer.

   REGULARIZATION-Regularization is a technique to discourage the complexity of the model. It does this by penalizing the loss function. This helps to solve the over fitting problem.

L2 & L1 regularization

L1 and L2 are the most common types of regularization. These update the general cost function by adding another term known as the regularization term.

Cost function = Loss (say, binary cross entropy) + Regularization term

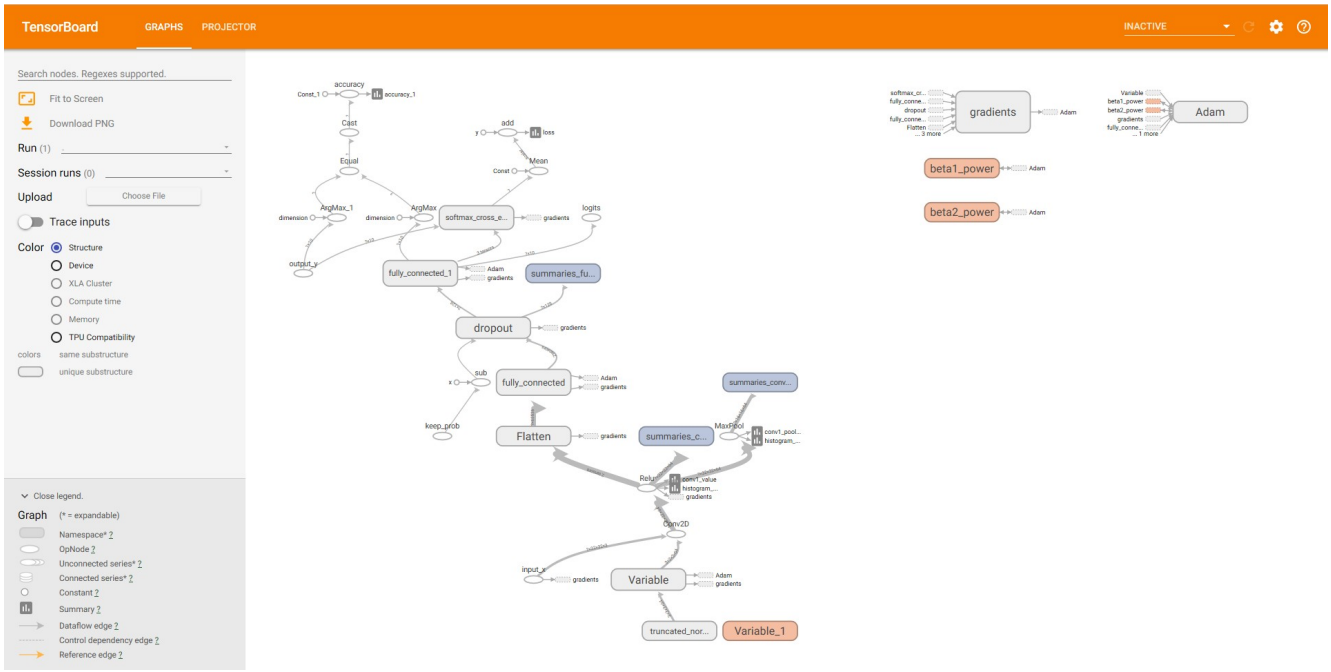
Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent.

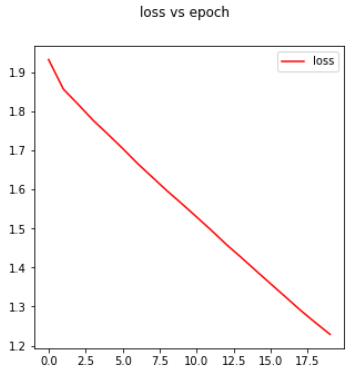
```
epochs = 4
batch_size = 128
keep_probability= 0.7
learning_rate = 0.001

logits = conv_net(x, keep_prob)
model = tf.identity(logits, name='logits') # Name logits Tensor, so that can be loaded from disk after training

# Loss and Optimizer
my_normal_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,labels = y))
reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
reg_constant = 0.01  # Choose an appropriate one.
loss = my_normal_loss + reg_constant * sum(reg_losses)

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)

# 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')
variable_summaries(loss,'loss')
variable_summaries(accuracy, 'accuracy')
```

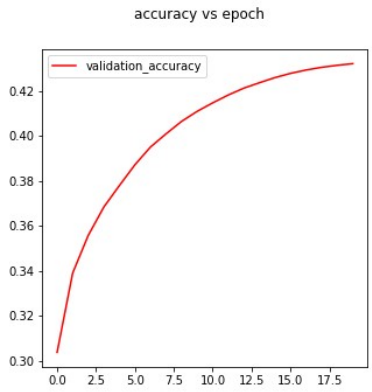
Then we run the session by training the data and collecting accuracy of validation data. At the end we find the test data accuracy.

```
save_model_path = r'C:\Users\manna\Documents\GitHub\CIFAR10-img-classification-tensorflow\graph'

print('Training...')
with tf.Session() as sess:
    merged_summary_op = tf.summary.merge_all()
    summary_writer = tf.summary.FileWriter(r"C:\Users\manna\Documents\GitHub\CIFAR10-img-classification-tensorflow\graph
    # Initializing the variables
    sess.run(tf.global_variables_initializer())

    # Training cycle
    for epoch in range(epochs):
        # Loop over all batches
        n_batches = 5
        for batch_i in range(1, n_batches + 1):
            for batch_features, batch_labels in load_preprocess_training_batch(batch_i, batch_size):
                train_neural_network(sess, optimizer, keep_probability, batch_features, batch_labels)

            print('Epoch {:>2}, CIFAR-10 Batch {}:  '.format(epoch + 1, batch_i), end='')
            print_stats(sess, batch_features, batch_labels, loss, accuracy)
    print ("Testing Accuracy:", sess.run(accuracy, feed_dict={x: test_features,y:test_labels,keep_prob: 1.}))
    # Save Model
    saver = tf.train.Saver()
    save_path = saver.save(sess, save_model_path)
```

RESULT (TENSOR Graph)







test data accuracy= 44.023%

Exercise 2: Convolutional Neural Networks for Image Classification (CNN) (10 points)

Model code :

1.conv1: convolution and self normalization activation (selu)

2. pool1: max pooling

3. norm: batch normalization

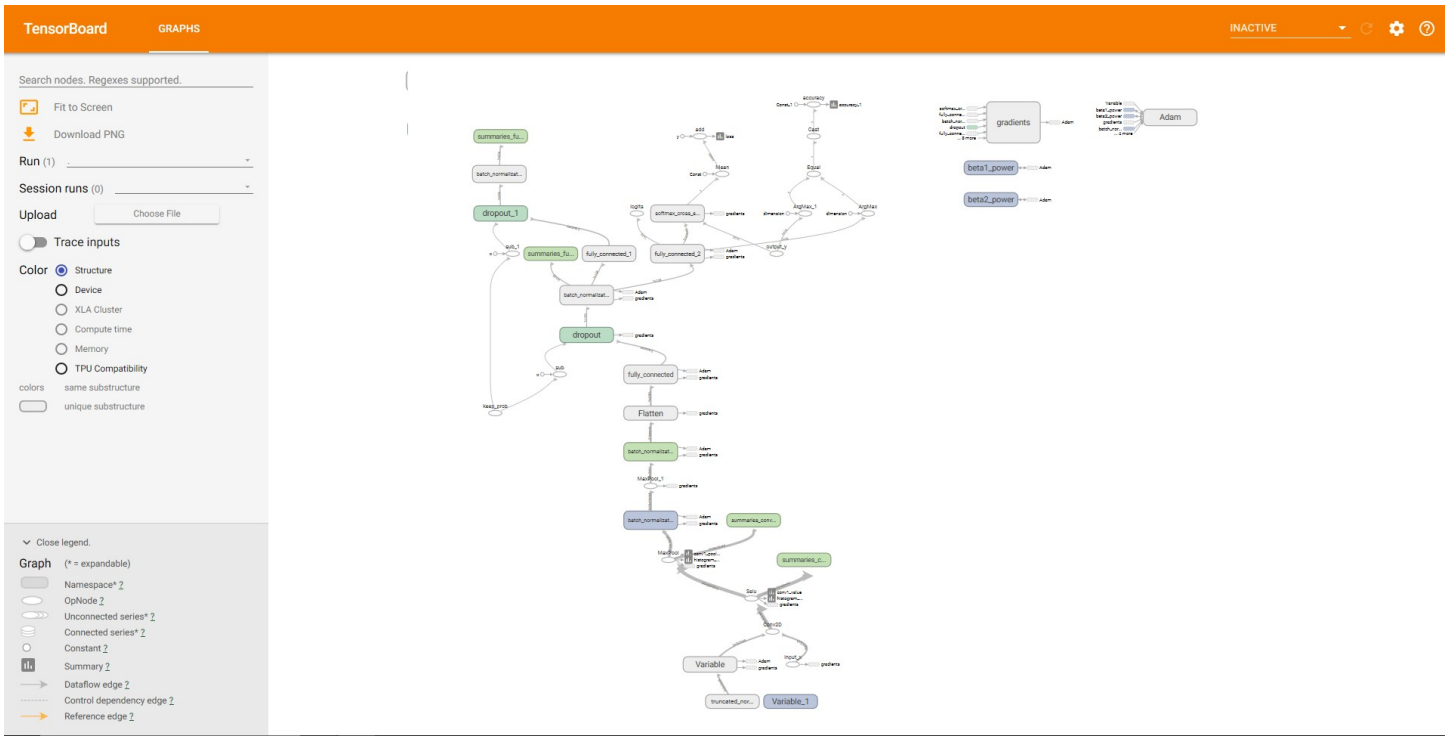
4. pool2: max pooling

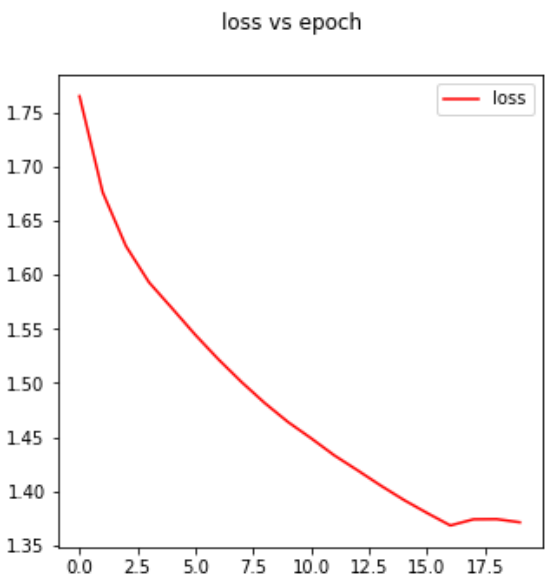
5. FC1: fully connected layer with self normalization activation (selu)

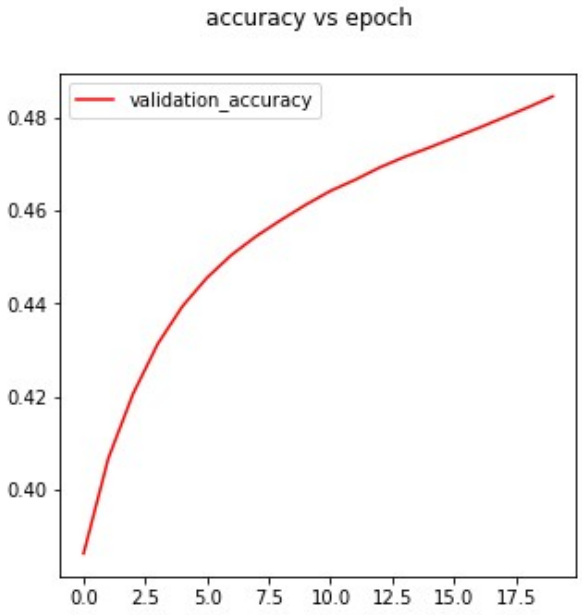
6. FC2: fully connected layer with self normalization activation (selu)

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


    conv1 = tf.nn.conv2d(x, conv1_filter, strides=[1,1,1,1], padding='SAME')
    conv1 = tf.nn.selu(conv1)
    conv1_pool_1 = 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_1)
    conv1_pool_2 = tf.nn.max_pool(conv1_bn, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
    conv1_bn = tf.layers.batch_normalization(conv1_pool_2)
    variable_summaries(conv1,'conv1')
    variable_summaries(conv1_pool_1,'conv1_pool')
#     variable_summaries(conv1,'conv1_bn')



    flat = tf.contrib.layers.flatten(conv1_bn)
    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)
    variable_summaries(full1,'full1')

    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)
    variable_summaries(full2,'full2')



    out = tf.contrib.layers.fully_connected(inputs=full1, num_outputs=10, activation_fn=None)
    return out
```

RESULT (TENSOR Graph)







Test data accuracy= 53.23%