In [ ]:

```
#*********************************************************************************
#
#                     Project Assignment on Convolution neural network
#
#*********************************************************************************
# Problem statement: To Classify the images from CIFAR-10 DataSet Using CNN


# loading the Data ,by unzipping the tar file and creating batches of images
# Install tqdm file

from urllib.request import urlretrieve
from os.path import isfile, isdir

from tqdm import tqdm
import tarfile


cifar10_dataset_folder_path = 'cifar-10-batches-py'

class DLProgress(tqdm):
    last_block = 0

    def hook(self, block_num=1, block_size=1, total_size=None):
        self.total = total_size
        self.update((block_num - self.last_block) * block_size)
        self.last_block = block_num

# if the file cifar tar is not downloaded , then download
if not isfile('cifar-10-python.tar.gz'):
    with DLProgress(unit='B', unit_scale=True, miniters=1, desc='CIFAR-10 Dataset'):
        urlretrieve('https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz',
                    'cifar-10-python.tar.gz',pbar.hook)

# open the file and extract all the batches from the tar file
if not isdir(cifar10_dataset_folder_path):
    with tarfile.open('cifar-10-python.tar.gz') as tar:
        tar.extractall()
        tar.close()
```

In [ ]:

```
#Each batch had many images , so there are 4 dimension , first value is images , 2nd and 3rd size and colour
# in transpose we mention the index number of the dimensions
# The input features are normalised to save on memory , for accuracy and within a particular scale
# pooling can be differnet types, one of them is max pooling

import pickle
#pickle is for serializing and unserializing the image data into bits/bytes
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelBinarizer


# Defining function to load the names of the labels of whose images in the DataSet
def load_label_names():
    """    Load the label names from file    """
    return ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse','ship','truck']


# Function to convert the input image data into the way tensor for Convolution accepts i.e.width X height X chan
# and returns the features and labels containing the data from the batch

def load_cfar10_batch(cifar10_dataset_folder_path, batch_id):
    """    Load a batch of the dataset    """
    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


# Function to show the image selected . It internally loads the image from the batch mentioned in
# the path given then reshapes the image

def display_stats(cifar10_dataset_folder_path, batch_id, sample_id):
    """    Display Stats of the the dataset    """
    batch_ids = list(range(1, 6))
    if batch_id not in batch_ids:
        print('Batch Id out of Range. Possible Batch Ids: {}'.format(batch_ids))
```

```
        return None

    features, labels = load_cfar10_batch(cifar10_dataset_folder_path, batch_id)

    if not (0 <= sample_id < len(features)):
        print('{} samples in batch {}.  {} is out of range.'.format(len(features), batch_id, sample_id))
        return None

    print('\nStats of batch #{}:'.format(batch_id))
    print('# of Samples: {}\n'.format(len(features)))

    label_names = load_label_names()
    label_counts = dict(zip(*np.unique(labels, return_counts=True)))
    for key, value in label_counts.items():
        print('Label Counts of [{}]({}) : {}'.format(key, label_names[key].upper(), value))

    sample_image = features[sample_id]
    sample_label = labels[sample_id]

    print('\nExample of Image {}:'.format(sample_id))
    print('Image - Min Value: {} Max Value: {}'.format(sample_image.min(), sample_image.max()))
    print('Image - Shape: {}'.format(sample_image.shape))
    print('Label - Label Id: {} Name: {}'.format(sample_label, label_names[sample_label]))

    plt.imshow(sample_image)

```

In [ ]:

```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import numpy as np

# Explore the dataset
batch_id = 2
sample_id = 2700

cifar10_dataset_folder_path = "C:/Users/Maximus/pythonsessions/session 39Project_CNN/cifar-10-batches-py"
display_stats(cifar10_dataset_folder_path,batch_id,sample_id)
```

In [ ]:

```
# Normalisation : Function to normalize the Data , so that it lies within a range and we do not
# come across exploding gradient, here the value taken by input is in range of 0-255
# , this is also called squashing of the input value

from sklearn import preprocessing
lb = preprocessing.LabelBinarizer()
lb.fit(range(10))

def normalize(x):
    """
        argument
            - x: input image data in numpy array [32, 32, 3]
        return
            - normalized x
    """
    # change 4
    x = np.array(x)
    min_val = np.min(x)
    max_val = np.max(x)
    x = (x-min_val) / (max_val-min_val)
    return x

# One-Hot Encoding : Since we will be classifying the images as one of the output 10 categories
# and that is identified as highest probability of one of these 10 values. WE use one hot encoding
# to identify the output category
def one_hot_encode(x):
    """
        argument
            - x: a list of labels
        return
            - one hot encoding matrix (number of labels, number of class)
    """
    return lb.transform(x)


```

In [ ]:

```
# Preprocess the Data , i.e normalize the input values of image and one hot encode the output
# values and write the new values to the file

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

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


# In this function , the image data for all the batches is preprocessed i.e normalized and one hot
# encoded. This data is then split into training and Validation Data , 10% of for validation
# Even Testing Data is preprocessed and dumped into the file
def preprocess_and_save_data(cifar10_dataset_folder_path, normalize, one_hot_encode):
    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,
                             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,
                         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,
                         np.array(test_features), np.array(test_labels),
                         'preprocess_training.p')
```

In [ ]:

```
# calling the function to preprocess the data and normalize and one hot encode the output
preprocess_and_save_data(cifar10_dataset_folder_path, normalize, one_hot_encode)


# loading the data from the validation file
valid_features, valid_labels = pickle.load(open('preprocess_validation.p', mode='rb'))
```

In [ ]:

```
valid_features[:10]  , valid_labels[:10]
```

In [ ]:

```
# Defining Convoltion Layer

def conv2d_maxpool(x_tensor, conv_num_outputs, conv_ksize, conv_strides, pool_ksize, pool_strides):
    """
    Apply convolution then max pooling to x_tensor
    :param x_tensor: TensorFlow Tensor
    :param conv_num_outputs: Number of outputs for the convolutional layer
    :param conv_ksize: kernal size 2-D Tuple for the convolutional layer
    :param conv_strides: Stride 2-D Tuple for convolution
    :param pool_ksize: kernal size 2-D Tuple for pool
    :param pool_strides: Stride 2-D Tuple for pool
    : return: A tensor that represents convolution and max pooling of x_tensor
    """
    # Weights
    shape_of_x_tensor = x_tensor.get_shape().as_list()

    F_W = tf.Variable(tf.truncated_normal([conv_ksize[0],conv_ksize[1],shape_of_x_tensor[-1],conv_num_outputs]
                                          ,dtype=tf.float32,stddev=0.2))

    F_b = tf.Variable(tf.zeros([conv_num_outputs],dtype=tf.float32))

    strides_conv = [1,conv_strides[0],conv_strides[1],1]
    padding = 'SAME'

    output = tf.nn.conv2d(x_tensor, F_W, strides_conv, padding )
    output = tf.nn.bias_add(output, F_b)

    # Nonlinear activation (ReLU)
    output = tf.nn.relu(output)

    # Max pooling
    ksize_maxpool = [1,pool_ksize[0],pool_ksize[1],1]
    strides_maxpool = [1,pool_strides[0],pool_strides[1],1]
    output = tf.nn.max_pool(output, ksize_maxpool ,strides_maxpool,padding)

    return output


```

In [ ]:

```python
#Defining the flatten layer , after the convolution layer

def flatten(x_tensor):
    """
    Flatten x_tensor to (Batch Size, Flattened Image Size)
    : x_tensor: A tensor of size (Batch Size, ...), where ... are the image dimensions.
    : return: A tensor of size (Batch Size, Flattened Image Size).
    """
    shape   = x_tensor.get_shape().as_list()

    flat_dim = shape[1] * shape[2] * shape[3]

    output = tf.reshape(x_tensor,[-1,flat_dim])
    return output

# defining the fully connected Layer

def fully_conn(x_tensor, num_outputs):
    """
    Apply a fully connected layer to x_tensor using weight and bias
    : x_tensor: A 2-D tensor where the first dimension is batch size.
    : num_outputs: The number of output that the new tensor should be.
    : return: A 2-D tensor where the second dimension is num_outputs.
    """
    # Weights and bias
    shape = x_tensor.get_shape().as_list()

    W = tf.Variable(tf.truncated_normal([shape[1], num_outputs],dtype=tf.float32, stddev=0.2))
    b = tf.Variable(tf.zeros([num_outputs],dtype = tf.float32))

    # The fully connected layer
    x = tf.add(tf.matmul(x_tensor, W), b)

    # ReLU activation function
    out = tf.nn.relu(x)
    return out


def output(x_tensor, num_outputs):
    """
    Apply a output layer to x_tensor using weight and bias
```

```
    : x_tensor: A 2-D tensor where the first dimension is batch size.
    : num_outputs: The number of output that the new tensor should be.
    : return: A 2-D tensor where the second dimension is num_outputs.
    """
    shape = x_tensor.get_shape().as_list()


    # Weights and bias
    W = tf.Variable(tf.truncated_normal([shape[1], num_outputs],dtype=tf.float32, stddev=.2))
    b = tf.Variable(tf.zeros([num_outputs],dtype=tf.float32))


    # The output layer
    out = tf.add(tf.matmul(x_tensor, W), b)
    return out
```

In [ ]:

```
# building the CNN network , consisting of 3 convolution layers , 3 fully connected layers


def conv_net(x, keep_prob):
    """
    Create a convolutional neural network model
    : x: Placeholder tensor that holds image data.
    : keep_prob: Placeholder tensor that hold dropout keep probability.
    : return: Tensor that represents logits
    """
    # 3 convolution layers with max pooling
    # All layers with same kernel, stride and maxpooling params

    out = conv2d_maxpool(x,conv_num_outputs=16,conv_ksize=(3,3),conv_strides=(1,1),pool_ksize=(2,2),pool_strides
    out = conv2d_maxpool(out,conv_num_outputs=32,conv_ksize=(3,3),conv_strides=(1,1),pool_ksize=(2,2),pool_strid
    out = conv2d_maxpool(out,conv_num_outputs=64,conv_ksize=(3,3),conv_strides=(1,1),pool_ksize=(2,2),pool_strid


    out = flatten(out)

    # Fully connected Layer

    out   = fully_conn(out, num_outputs = 64)
    out   = tf.nn.dropout(out, keep_prob)

    out   = fully_conn(out, num_outputs = 32)
    out   = tf.nn.dropout(out, keep_prob)

    out   = fully_conn(out, num_outputs = 16)

#change 2no 3rd drop out done
#    out   = tf.nn.dropout(out, keep_prob)


    out = output(out,num_outputs=10)

    return out


```

In [ ]:
```
# initializing the hyperparameters for CNN

epochs = 100
batch_size = 128
keep_probability = 0.5
learning_rate = 0.001
```

In [ ]:
```
keep_probability
```

In [ ]:

```
# the output of the last Fully connected layer ,i.e logits is given to conv_net( this internally
# calls the softmax function . )

# Resetting the default graph , so that all the previous weights, bias, inputs, etc.. are reset
import tensorflow as tf

tf.reset_default_graph()

# Placeholders for Inputs-- features and Labels , first dimension is the no of images
x =  tf.placeholder(tf.float32, shape=(None, 32, 32, 3), name='input_x')
y =  tf.placeholder(tf.float32, shape=(None, 10), name='output_y')

# no of neurons to be ignored in the Fully connected network, so that overfitting doesnt occur
keep_prob = tf.placeholder(tf.float32, name='keep_prob')


logits = conv_net(x, keep_prob)

#change 6 retrun variable name
#model = tf.identity(logits, name='logits') # Name logits Tensor, so that can be loaded from disk after training

logits = tf.identity(logits, name='logits')
# Loss and Optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y))

# change 3 learning rate not mentioned
#optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
optimizer = tf.train.AdamOptimizer().minimize(cost)

# 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')
```

In [ ]:

```
# training the neural network

def train_neural_network(session, optimizer, keep_probability, feature_batch, label_batch):
    session.run(optimizer,
                feed_dict={
                    x: feature_batch,
                    y: label_batch,
                    keep_prob: keep_probability
                })
```

In [ ]:

```
# printing the stats for each batch

def print_stats(session, feature_batch, label_batch, cost, accuracy):
    """
    Print information about loss and validation accuracy
    : session: Current TensorFlow session
    : feature_batch: Batch of Numpy image data
    : label_batch: Batch of Numpy label data
    : cost: TensorFlow cost function
    : accuracy: TensorFlow accuracy function
    """
    valid_acc = sess.run(accuracy,
                         feed_dict={
                             x: valid_features,
                             y: valid_labels,
                             keep_prob: 1.
                         })
    train_acc = sess.run(accuracy,
                         feed_dict={
                             x: feature_batch,
                             y: label_batch,
                             keep_prob: 1.
                         })


    train_cost = sess.run(cost,
                     feed_dict={
                         x: feature_batch,
                         y: label_batch,
                         keep_prob: 1.
                     })


    print('Cost: {::>8.5f} Accuracy on training {:.4f} '
          'Validation Accuracy: {:.4f}'.format(train_cost,train_acc, valid_acc))
```

In [ ]:

```
# function to split the features list into batches , each batch of batch_size

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]

# loading the data from the file and get batches
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)
```

In [ ]:

```
# Training for a single batch
# This function trains the neural network for each batch from the features list
#save_model_path = './image_classification'
#keep_probability = 0.6

print('Training...on a single batch')
with tf.Session() as sess:
    # Initializing the variables
    sess.run(tf.global_variables_initializer())


    # Training cycle
    for epoch in range(epochs):
        # for a single batch
        batch_i = 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, cost, accuracy)

```

In [ ]:

```
# This function trains the neural network for each batch from the features list
import timeit



save_model_path = './image_classification'

start = timeit.timeit()
print('Training...')
with tf.Session() as sess:
    # Initializing the variables
    sess.run(tf.global_variables_initializer())
    #epochs = 10
    # 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, cost, accuracy)

    end = timeit.timeit()
    tm = (end-start)*1000


    # Save Model
    saver = tf.train.Saver()
    save_path = saver.save(sess, save_model_path)
```

In [ ]:

```
# time to train  the model for all the batches and epochs
print("The Total time to train the model is : " ,tm," ms")
```

In [ ]:

```
# function to display the images
from sklearn.preprocessing import LabelBinarizer
import matplotlib.pyplot as plt
import random

n_samples = 4
top_n_predictions = 5
model_path = save_model_path

# Defining function to load the names of the labels of whose images in the DataSet
def load_label_names():
    """    Load the label names from file    """
    return ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse','ship','truck']


def display_image_predictions(features, labels, predictions):
    n_classes = 10

    label_names = load_label_names()
    label_binarizer = LabelBinarizer()
    label_binarizer.fit(range(n_classes))
    label_ids = label_binarizer.inverse_transform(np.array(labels))

    fig, axies = plt.subplots(nrows=4, ncols=2)
    fig.tight_layout()
    fig.suptitle('Softmax Predictions', fontsize=20, y=1.1)

    n_predictions = 5
    margin = 0.05
    ind = np.arange(n_predictions)
    width = (1. - 2. * margin) / n_predictions
#    print(width)


    for image_i, (feature, label_id, pred_indicies, pred_values) in enumerate(zip(features, label_ids, predictio
        pred_names = [label_names[pred_i] for pred_i in pred_indicies]
        correct_name = label_names[label_id]

#        axies[image_i][0].imshow(feature)
#change1
        axies[image_i][0].imshow(feature)
```

```
        axies[image_i][0].set_title(correct_name)
        axies[image_i][0].set_axis_off()

#         print(ind + margin)
        axies[image_i][1].barh(ind + margin, pred_values[::-1], width)
        axies[image_i][1].set_yticks(ind + margin)
        axies[image_i][1].set_yticklabels(pred_names[::-1])
        axies[image_i][1].set_xticks([0, 0.5, 1.0])
```

In [ ]:

```
# Testing the Model, after executing the model
import random

def test_model():
    ''' To Test the model after the model is been trained '''

    test_features, test_labels = pickle.load(open('preprocess_training.p', mode='rb'))
    loaded_graph = tf.Graph()


    with tf.Session(graph=loaded_graph) as sess:
        # Load model
        loader = tf.train.import_meta_graph(model_path + '.meta')
        loader.restore(sess, model_path)


        # Get Tensors from loaded model
        loaded_x = loaded_graph.get_tensor_by_name('input_x:0')
        loaded_y = loaded_graph.get_tensor_by_name('output_y:0')
        loaded_keep_prob = loaded_graph.get_tensor_by_name('keep_prob:0')
        loaded_logits = loaded_graph.get_tensor_by_name('logits:0')
        loaded_acc = loaded_graph.get_tensor_by_name('accuracy:0')

        # Get accuracy in batches for memory limitations
        test_batch_acc_total = 0
        test_batch_count = 0

        for test_feature_batch, test_label_batch in batch_features_labels(test_features, test_labels, batch_size
            test_batch_acc_total += sess.run(loaded_acc, feed_dict={loaded_x: test_feature_batch,
                                                                    loaded_y: test_label_batch,
                                                                    loaded_keep_prob: 1.0
                                                                     })
            test_batch_count += 1


        print('Testing Accuracy: {}\n'.format(test_batch_acc_total/test_batch_count))

        # Print Random Samples
        random_test_features, random_test_labels = tuple(zip(*random.sample(list(zip(test_features, test_labels)
        random_test_predictions = sess.run( tf.nn.top_k(tf.nn.softmax(loaded_logits), top_n_predictions),
                                           feed_dict={loaded_x: random_test_features,loaded_y: random_test_label
```

```
                                                    loaded_keep_prob: 1.0 } )

        display_image_predictions(random_test_features, random_test_labels, random_test_predictions)
```

In [ ]:

```
# testing the model

test_model()
```