In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import scipy.misc
import glob
import sys


class NeuralNetwork(object):

    """
    Abstraction of neural network.
    Stores parameters, activations, cached values.
    Provides necessary functions for training and prediction.
    """
    def __init__(self, layer_dimensions, drop_prob=0.0, reg_lambda=0.0,
momentum_beta=0, rms_beta=0, adam_beta1=0, adam_beta2=0, data_augmentati
on=False):
        """
        Initializes the weights and biases for each layer
        :param layer_dimensions: (list) number of nodes in each layer
        :param drop_prob: drop probability for dropout layers. Only requ
ired in part 2 of the assignment
        :param reg_lambda: regularization parameter. Only required in pa
rt 2 of the assignment
        """
        np.random.seed(1)

        self.parameters = {}
        self.num_layers = len(layer_dimensions) - 1

        # init parameters
        for layer in range(1, self.num_layers + 1):
            w = np.divide(np.random.normal(0, 1, (layer_dimensions[layer
], layer_dimensions[layer - 1])), np.sqrt(layer_dimensions[layer - 1]))
            b = np.zeros(layer_dimensions[layer])

            self.parameters[layer] = [w, b]

    def affineForward(self, A, W, b):
        """
        Forward pass for the affine layer.
        :param A: input matrix, shape (L, S), where L is the number of h
idden units in the previous layer and S is
        the number of samples
        :returns: the affine product WA + b, along with the cache requir
ed for the backward pass
        """
        Z = np.dot(W, A)
        for i in range(len(Z)):
            Z[i] = Z[i] + b[i]
        return Z, [W, A, b, Z]

    def activationForward(self, A, activation="relu"):
        """
        Common interface to access all activation functions.
```

```
        :param A: input to the activation function
        :param prob: activation funciton to apply to A. Just "relu" for
 this assignment.
        :returns: activation(A)
        """
        if(activation == "relu"):
            ret = self.relu(A)
        return ret

    def relu(self, X):
        A = np.maximum(0, X)
        return A

    def forwardPropagation(self, X):
        """
        Runs an input X through the neural network to compute activation
s
        for all layers. Returns the output computed at the last layer al
ong
        with the cache required for backpropagation.
        :returns: (tuple) AL, cache
            WHERE
            AL is activation of last layer
            cache is cached values for each layer that
                      are needed in further steps
        """
        cache = []
        cache.append([]) # Empty cache for layer 1
        Z, cacheLayer = self.affineForward(X, self.parameters[1][0], sel
f.parameters[1][1])
        A = self.activationForward(Z)

        cache.append(cacheLayer)
        for layer in range(2, self.num_layers):
            Z, cacheLayer = self.affineForward(A, self.parameters[layer]
[0], self.parameters[layer][1])
            A = self.activationForward(Z)
            cache.append(cacheLayer)
        Z, cacheLayer = self.affineForward(A, self.parameters[self.num_l
ayers][0], self.parameters[self.num_layers][1])
        AL = self.softmax(Z)
        cache.append(cacheLayer)

        return AL, cache

    def costFunction(self, AL, y):
        """
        :param AL: Activation of last layer, shape (num_classes, S)
        :param y: labels, shape (S)
        :param alpha: regularization parameter
        :returns cost, dAL: A scalar denoting cost and the gradient of c
ost
        """
        # compute loss
        m = y.shape[0]
        correct_label_prob = AL[y, range(m)]
        cost = -np.sum(np.log(correct_label_prob)) / m
```

```
        dAL = AL
        dAL[y, range(AL.shape[1])] = dAL[y, range(AL.shape[1])] - 1
        return cost, dAL

    def softmax(self, X):
        return np.exp(X) / np.sum(np.exp(X), axis = 0)

    def affineBackward(self, dA_prev, cache):
        """
        Backward pass for the affine layer.
        :param dA_prev: gradient from the next layer.
        :param cache: cache returned in affineForward
        :returns dA: gradient on the input to this layer
                 dW: gradient on the weights
                 db: gradient on the bias
        """
        W = cache[0]
        A = cache[1]
        b = cache[2]
        Z = cache[3]

        dZ_prev = np.multiply(dA_prev, self.relu_derivative(Z))
        dA = np.dot(W.transpose(), dZ_prev)
        dW = np.dot(dZ_prev, A.transpose())
        db = np.mean(dZ_prev, axis = 1) # Aggregate samples

        return dA, dW, db

    def affineBackwardLastLayer(self, dA_prev, Y, cache):
        W = cache[0]
        A = cache[1]
        b = cache[2]
        Z = cache[3]
        dZ_prev = dA_prev
        dA = np.dot(W.transpose(), dZ_prev)
        dW = np.dot(dZ_prev, A.transpose())
        db = np.mean(dZ_prev, axis = 1) # Aggregate samples

        return dA, dW, db


    def relu_derivative(self, cached_x):
        relu_d = 1 * (cached_x > 0)
        return relu_d


    def backPropagation(self, dAL, Y, cache):
        """
        Run backpropagation to compute gradients on all paramters in the
model
        :param dAL: gradient on the last layer of the network. Returned
by the cost function.
        :param Y: labels
        :param cache: cached values during forwardprop
        :returns gradients: dW and db for each weight/bias
```

```
        """
        gradients = {}
        dA, dW, db = self.affineBackwardLastLayer(dAL, Y, cache[self.num
_layers])
        gradients[self.num_layers] = [dW, db]

        for i in range(self.num_layers - 1):
            layer = self.num_layers - 1 - i
            dA, dW, db = self.affineBackward(dA, cache[layer])
            gradients[layer] = [dW, db]
        return gradients

    def updateParameters(self, gradients, alpha):
        """
        :param gradients: gradients for each weight/bias
        :param alpha: step size for gradient descent
        """
        for layer in range(1, self.num_layers + 1):
            self.parameters[layer][0] = self.parameters[layer][0] - alph
a * gradients[layer][0]
            self.parameters[layer][1] = self.parameters[layer][1] - alph
a * gradients[layer][1]

    def calculateAccuracy(self, y_actual, y_prediction):
        correct = 0
        for i in range(len(y_actual)):
            if y_prediction[i] == y_actual[i]:
                correct = correct + 1
        accuracy = correct / len(y_actual) * 100
        return accuracy



    def train(self, X, y, iters=1000, alpha=0.0001, batch_size=100, prin
t_every=100):
        """
        :param X: input samples, each column is a sample
        :param y: labels for input samples, y.shape[0] must equal X.shap
e[1]
        :param iters: number of training iterations
        :param alpha: step size for gradient descent
        :param batch_size: number of samples in a minibatch
        :param print_every: no. of iterations to print debug info after
        """
        no_of_examples = X.shape[1]
        no_of_batches = int(no_of_examples / batch_size)

        # Split into training and validation sets
        splitIndex = int(.9 * len(X[0])) # 90% train, 10% validation
        X_train = X[:, :splitIndex]
        y_train = y[:splitIndex]
        X_validation = X[:, splitIndex:]
        y_validation = y[splitIndex:]


        for iteration in range(0, iters):
            self.parameters['batch_index'] = 0
```

```
            for batches in range(no_of_batches):
                # get minibatch
                X_batch, Y_batch = self.get_batch(X, y, batch_size)

                # forward prop
                AL, cache = self.forwardPropagation(X_batch)

                # compute loss
                cost, dAL = self.costFunction(AL, Y_batch)

                # compute gradients
                gradients = self.backPropagation(dAL, Y_batch, cache)

                # update weights and biases based on gradient
                self.updateParameters(gradients, alpha)

            if iteration % print_every == 0:
                # print cost, train and validation set accuracies
                print("Metrics for Iteration " + str(iteration))

                # cost
                print("     Cost: " + str(cost))

                # train accuracy
                y_train_prediction = self.predict(X_train)
                train_accuracy = self.calculateAccuracy(y_train, y_train
_prediction)
                print("     Training accuracy: " + "{0:.3f}".format(trai
n_accuracy) + " percent")

                # validation accuracy
                y_validation_prediction = self.predict(X_validation)
                validation_accuracy = self.calculateAccuracy(y_validatio
n, y_validation_prediction)
                print("     Validation accuracy: " + "{0:.3f}".format(va
lidation_accuracy) + " percent")




    def predict(self, X):
        """
        Make predictions for each sample
        """
        AL, cache = self.forwardPropagation(X)
        y_pred = np.argmax(AL, axis = 0)

        return y_pred

    def get_batch(self, X, y, batch_size):

        current_index=self.parameters["batch_index"]
        self.parameters["batch_index"]=self.parameters["batch_index"]+ba
tch_size
        X_batch,y_batch = X[:,current_index:current_index+batch_size], y
[current_index:current_index+batch_size]
```

```
        return X_batch, y_batch

def one_hot(y, num_classes=10):

    y_one_hot = np.zeros((num_classes, y.shape[0]))
    y_one_hot[y, range(y.shape[0])] = 1
    return y_one_hot

def save_predictions(filename, y):
    """
    Dumps y into .npy file
    """
    np.save(filename, y)

def get_train_data_1(data_root_path):
    images = []
    train_data_path = data_root_path + 'train'
    y = []
    for i in range(1,6):
        #print(i)
        file_path = train_data_path + '/data_batch_' + str(i)
        dictionary = unpickle(file_path)
        X_temp = dictionary[b'data']
        X_temp = X_temp.T
        X_temp = np.true_divide(X_temp,255)
        #print(X_temp.shape)
        y_temp = dictionary[b'labels']
        #print(len(y_temp))
        images.append(X_temp)
        y = y + y_temp
    X = np.column_stack(images)
    y = np.asarray(y)
#     print("============")
#     print(X.shape)
#     print(len(y))

    return X, y

def get_label_mapping_1(label_file):

    dictionary = unpickle(label_file)
    label2id = {}
    id2label = dictionary[b'label_names']
    count = 0

    for label in id2label:
        label2id[label] = count
        count += 1
    return id2label, label2id

def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict

'''
```

```
NOTE: DATA LOADING:
Directory structure: data/cifar-10-batches-py/

TRAIN: data/cifar-10-batches-py/train
This folder contains all the data_batch_1 to data_batch_5 files

TEST: data/cifar-10-batches-py/test
This folder contains the test_batch file

REST: data/cifar-10-batches-py/rest
This folder conatins the rest of the batches.meta and Readme file.
'''

data_root_path_1 = 'data/cifar-10-batches-py/'
X_train, y_train = get_train_data_1(data_root_path_1)
print("X_train shape: ", X_train)
print("y_train shape: ", y_train.shape)
id2label_1, label2id_1 = get_label_mapping_1('data/cifar-10-batches-py/r
est/batches.meta')
print(label2id_1)

#################################################################
layer_dimensions = [X_train.shape[0], 400, 250, 10]  # including the inp
ut and output layers
NN = NeuralNetwork(layer_dimensions)
NN.train(X_train, y_train, iters=800, alpha=0.001, batch_size=100, print
_every=100)
```

```
X_train shape:  [[0.23137255 0.60392157 1.         ... 0.1372549  0.741
17647 0.89803922]
 [0.16862745 0.49411765 0.99215686 ... 0.15686275 0.72941176 0.9254902
]
 [0.19607843 0.41176471 0.99215686 ... 0.16470588 0.7254902  0.9176470
6]
 ...
 [0.54901961 0.54509804 0.3254902  ... 0.30196078 0.6627451  0.6784313
7]
 [0.32941176 0.55686275 0.3254902  ... 0.25882353 0.67058824 0.6352941
2]
 [0.28235294 0.56470588 0.32941176 ... 0.19607843 0.67058824 0.6313725
5]]
y_train shape:  (50000,)
{b'airplane': 0, b'automobile': 1, b'bird': 2, b'cat': 3, b'deer': 4,
b'dog': 5, b'frog': 6, b'horse': 7, b'ship': 8, b'truck': 9}
Metrics for Iteration 0
     Cost: 1.9213975227274473
     Training accuracy: 36.713 percent
     Validation accuracy: 36.580 percent
Metrics for Iteration 100
     Cost: 0.4259693272242243
     Training accuracy: 71.727 percent
     Validation accuracy: 84.280 percent
Metrics for Iteration 200
     Cost: 0.1174498812679115
     Training accuracy: 78.240 percent
     Validation accuracy: 91.680 percent
Metrics for Iteration 300
     Cost: 0.11863453237614012
     Training accuracy: 82.807 percent
     Validation accuracy: 94.680 percent
Metrics for Iteration 400
     Cost: 0.027576873825546827
     Training accuracy: 87.569 percent
     Validation accuracy: 96.460 percent
Metrics for Iteration 500
     Cost: 0.08330369204333093
     Training accuracy: 87.351 percent
     Validation accuracy: 96.300 percent
Metrics for Iteration 600
     Cost: 0.06092175652809502
     Training accuracy: 93.011 percent
     Validation accuracy: 99.040 percent
Metrics for Iteration 700
     Cost: 0.07067867473129953
     Training accuracy: 90.473 percent
     Validation accuracy: 97.140 percent
```

We know the above Training and Validation Accuracies are confusing.(Please READ)

The reason this happened was, in our "train" function while we created X_train and y_train which had 45000 samples after removing samples for validation set. Inside the inner for loop we forgot to update the variable names from X,y to X_train, y_train. So our model got trained on entire 50000 training samples while we printed Accuracies based on validation set size numbers.

But the accuracy that we got for the test set was accurate: 51.18%

So we ended up having a traditional "train test model" and not "train, validation and test model".

We were not able to re-run the code as this itself took us 5hrs to run and we were on a time crunch.

We hope you will consider this as an honest mistake for not following the train validation and test set format.

And we will make sure to not repeat this next time.

In [2]:
```
dictionary = unpickle('data/cifar-10-batches-py/test/test_batch')
```

We got a Test Accuracy of 49.14%

In [3]:

```
X_test = dictionary[b'data']

X_test = X_test.T
X_test = np.true_divide(X_test, 255)
#print(X_temp.shape)
y_test = dictionary[b'labels']
print(X_test.shape)

y_predicted = NN.predict(X_test)
save_predictions('ans1-ung200-avs431', y_predicted)

# test if your numpy file has been saved correctly
loaded_y = np.load('ans1-ung200-avs431.npy')
print(loaded_y.shape)
loaded_y[:10]

#y_validate2 = NN2.predict(X_test)
from sklearn.metrics import accuracy_score
print("Accuracy=",accuracy_score(y_test,y_predicted)*100)
```

```
(3072, 10000)
(10000,)
Accuracy= 49.14
```