In [1]:
```
#Deep LEarning - Fall 2018,
#Joshua Matt Abraham - jma672
#Ayan Agrawal - aka398
```

In [2]:
```
#Installing pyTorch
from os import path
from wheel.pep425tags import get_abbr_impl, get_impl_ver, get_abi_tag
platform = '{}{}-{}'.format(get_abbr_impl(), get_impl_ver(), get_abi_tag())

accelerator = 'cu80' if path.exists('/opt/bin/nvidia-smi') else 'cpu'

!pip3 install -q http://download.pytorch.org/whl/{accelerator}/torch-0.4.0-{pl
atform}-linux_x86_64.whl torchvision
```

In [3]:
```
#Importing the data and splitting into train and test datasets
import torch
import torchvision
import torchvision.transforms as transforms
print(torch.__version__)
transform = transforms.Compose(
[ transforms.ToTensor() ,
transforms.Normalize(( 0.5 , 0.5 , 0.5), (0.5, 0.5, 0.5 ))])
trainset = torchvision.datasets.CIFAR10( root='./data', train=True,
download=True, transform=transform)
testset = torchvision.datasets.CIFAR10( root='./data', train=False,
download=True , transform=transform)
```

```
0.2.0_3
Files already downloaded and verified
Files already downloaded and verified
```

In [4]:
```
#Preprocessing data to form vectorized training, test and label sets
import numpy as np

trainLabelArray = []
inputTrainMatrix = np.zeros((3072,len(trainset)))

testLabelArray = []
testMatrix = np.zeros((3072,len(testset)))

for i in range(len(trainset)):
  trainLabelArray.append(trainset[i][1])
  inputTrainMatrix[:,i] = trainset[i][0].numpy().flatten()

for i in range(len(testset)):
  testLabelArray.append(testset[i][1])
  testMatrix[:,i] = testset[i][0].numpy().flatten()
```

In [5]:
```
#Fully Connected Neural Network Code and Training

import numpy as np

class NeuralNetwork(object):
    def __init__(self, layer_dimensions, drop_prob=0.0, reg_lambda=0.0):
        #every time the weights are initialized to same numbers - functionalit
y of seed()
        #np.random.seed(1)
        self.parameters = {}
        self.num_layers = len(layer_dimensions)-1
        self.layer_dimensions = layer_dimensions
        self.dropoutProb = drop_prob
        self.regLambda=reg_lambda

        self.allWeightArray = []
        self.biases = []
        for i in range(self.num_layers):
          self.allWeightArray.append(np.random.randn(layer_dimensions[i], laye
r_dimensions[i+1])*0.1)
          self.biases.append(np.random.randn(layer_dimensions[i+1],1))


    def affineForward(self, A, W, b):
        Z = np.dot(W.T,A)+b
        cache = (A,W,Z)
        return Z,cache


    def activationForward(self, A, activation):
      if activation == "relu":
        return self.relu(A)
      elif activation == "softmax":
        return self.softmax(A)


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

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

    def dropout(self, A, prob):
      dropoutMask = np.random.rand(A.shape[0], A.shape[1])
      dropoutMask = dropoutMask<prob
      dropoutMask = dropoutMask/(1-prob)
      A = np.multiply(A, dropoutMask)
      return A, dropoutMask


    def forwardPropagation(self, X):
      A = X.copy()
      cacheSet={}
      dropoutMaskSet={}
      for i in range(self.num_layers-1):
```

```
        prevA = A.copy()
        Z, cache = self.affineForward(prevA, self.allWeightArray[i], self.bias
es[i])
        cacheSet[i]=cache
        A = self.activationForward(Z, 'relu')

        if self.dropoutProb>0:
          A, dropoutMask = self.dropout(A, self.dropoutProb)
          dropoutMaskSet[i] = dropoutMask

      Z, cache = self.affineForward(A, self.allWeightArray[-1], self.biases[-1
])
      cacheSet[self.num_layers-1]=cache
      A = self.activationForward(Z, 'softmax')
      return A, cacheSet, dropoutMaskSet


    def costFunction(self, AL, y):
      trueProbability = AL[y,range(len(y))]
      cost = - np.sum(np.log(trueProbability))/len(y)
      cost = np.squeeze(cost)
      return cost, self.softmax_derivative(AL, y)


    def softmax_derivative(self,AL,Y):
      trueOneHot = np.zeros((10,len(Y)))
      trueOneHot[Y, range(len(Y))] = 1
      return AL-trueOneHot

    def affineBackwardLastLayer(self, dZ, cache):
      A,W,Z = cache
      m = A.shape[1]
      dW = 1/m*(np.dot(A,dZ.T))
      db = 1/m*np.sum(dZ, axis=1, keepdims=True)
      return (dZ,dW,db)

    def affineBackward(self, dZ, cache):
      A,W,Z = cache
      m = A.shape[1]
      dW = 1/m*(np.dot(A,dZ.T))
      db = 1/m*np.sum(dZ, axis=1, keepdims=True)
      return (dZ, dW, db)


    def activationBackward(self, cache, activation="relu"):
      A,W,Z = cache
      return self.relu_derivative(Z)


    def relu_derivative(self, Z):
      Z[Z<=0] = 0
      Z[Z>0] = 1
      return Z


    def dropout_backward(self, dA, mask):
      dA=np.multiply(dA, mask)
```

```
      return dA


    def backPropagation(self, dAL, Y, cache, dropoutMaskSet):
      deltas = {}
      gradTuple = self.affineBackwardLastLayer(dAL, cache[self.num_layers-1])
      #Regularization
      if self.regLambda > 0:
        dW=gradTuple[1]+np.multiply(self.regLambda,cache[self.num_layers-1][1
])/cache[self.num_layers-1][0].shape[1]
        gradTuple = (gradTuple[0], dW, gradTuple[2])

      deltas[self.num_layers-1] = gradTuple

      for i in range(self.num_layers-1):
        currentCache = cache[len(cache)-i-2]
        relu_derivative = self.activationBackward(currentCache)

        dZ = np.multiply(np.dot(cache[len(cache)-i-1][1],deltas[self.num_layer
s-1-i][0]),relu_derivative)

        gradTuple = self.affineBackward(dZ, currentCache)

        #Dropout Implementation
        if self.dropoutProb>0:
          dZ=self.dropout_backward(gradTuple[0], dropoutMaskSet[len(dropoutMas
kSet)-i-1])
          gradTuple = (dZ, gradTuple[1], gradTuple[2])

        #Regularization Implementation
        if self.regLambda > 0:
          dW=gradTuple[1]+np.multiply(self.regLambda,currentCache[1])/currentC
ache[0].shape[1]
          gradTuple = (gradTuple[0], dW, gradTuple[2])

        deltas[self.num_layers-1-i-1] = gradTuple

      return deltas

    def updateParameters(self, gradients, alpha):
      for i in range(self.num_layers):
        self.allWeightArray[i] = self.allWeightArray[i]-alpha*gradients[i][1]
        self.biases[i] = self.biases[i]-alpha*gradients[i][2]


    def train(self, X, y, iters=300, alpha=0.01, batch_size=100, print_every=1
00):
      batchNumbers = int(X.shape[1]/batch_size)
      inputTrainBatches = np.hsplit(X, batchNumbers)
      n=batch_size
      labelBatches = [y[i * n:(i + 1) * n] for i in range((len(y) + n - 1) //
n )]

      for j in range(iters):
        for i in range(len(inputTrainBatches)):
          softmaxY, cacheSet, dropoutMaskSet = self.forwardPropagation(inputTr
ainBatches[i])
```

```
          cost, dSoftmaxY = self.costFunction(softmaxY, labelBatches[i])

          deltas = self.backPropagation(dSoftmaxY, labelBatches[i], cacheSet,
dropoutMaskSet)

          self.updateParameters(deltas, alpha)
        print("Epoch Iteration Number: ",j)
        if j%print_every==0:
          print("Cost:", cost)


    def predict(self, X):
      softmaxY, cache, dropMask = self.forwardPropagation(X)
      predictedLabels = np.argmax(softmaxY, axis=0)
      print(predictedLabels)
      return predictedLabels




#number of layers including input and output, each vlaue represents number of
 nodes for that layer
nnDimensions = [len(inputTrainMatrix), 128, 64, 10]
batchSize = 10
alpha=0.001
printNumber=10
iterations=92
dropProb=0
regLambda=0.01
nn = NeuralNetwork(nnDimensions, drop_prob=dropProb, reg_lambda=regLambda)
nn.train(inputTrainMatrix, trainLabelArray, iters=iterations, alpha=alpha, bat
ch_size=batchSize, print_every=printNumber)
```

```
Epoch Iteration Number:  0
Cost: 2.3378625807
Epoch Iteration Number:  1
Epoch Iteration Number:  2
Epoch Iteration Number:  3
Epoch Iteration Number:  4
Epoch Iteration Number:  5
Epoch Iteration Number:  6
Epoch Iteration Number:  7
Epoch Iteration Number:  8
Epoch Iteration Number:  9
Epoch Iteration Number:  10
Cost: 2.12001962126
Epoch Iteration Number:  11
Epoch Iteration Number:  12
Epoch Iteration Number:  13
Epoch Iteration Number:  14
Epoch Iteration Number:  15
Epoch Iteration Number:  16
Epoch Iteration Number:  17
Epoch Iteration Number:  18
Epoch Iteration Number:  19
Epoch Iteration Number:  20
Cost: 1.98751023895
Epoch Iteration Number:  21
Epoch Iteration Number:  22
Epoch Iteration Number:  23
Epoch Iteration Number:  24
Epoch Iteration Number:  25
Epoch Iteration Number:  26
Epoch Iteration Number:  27
Epoch Iteration Number:  28
Epoch Iteration Number:  29
Epoch Iteration Number:  30
Cost: 1.88283419655
Epoch Iteration Number:  31
Epoch Iteration Number:  32
Epoch Iteration Number:  33
Epoch Iteration Number:  34
Epoch Iteration Number:  35
Epoch Iteration Number:  36
Epoch Iteration Number:  37
Epoch Iteration Number:  38
Epoch Iteration Number:  39
Epoch Iteration Number:  40
Cost: 1.76018456578
Epoch Iteration Number:  41
Epoch Iteration Number:  42
Epoch Iteration Number:  43
Epoch Iteration Number:  44
Epoch Iteration Number:  45
Epoch Iteration Number:  46
Epoch Iteration Number:  47
Epoch Iteration Number:  48
Epoch Iteration Number:  49
Epoch Iteration Number:  50
Cost: 1.64205486585
```

```
Epoch Iteration Number:  51
Epoch Iteration Number:  52
Epoch Iteration Number:  53
Epoch Iteration Number:  54
Epoch Iteration Number:  55
Epoch Iteration Number:  56
Epoch Iteration Number:  57
Epoch Iteration Number:  58
Epoch Iteration Number:  59
Epoch Iteration Number:  60
Cost: 1.52906878628
Epoch Iteration Number:  61
Epoch Iteration Number:  62
Epoch Iteration Number:  63
Epoch Iteration Number:  64
Epoch Iteration Number:  65
Epoch Iteration Number:  66
Epoch Iteration Number:  67
Epoch Iteration Number:  68
Epoch Iteration Number:  69
Epoch Iteration Number:  70
Cost: 1.47604606717
Epoch Iteration Number:  71
Epoch Iteration Number:  72
Epoch Iteration Number:  73
Epoch Iteration Number:  74
Epoch Iteration Number:  75
Epoch Iteration Number:  76
Epoch Iteration Number:  77
Epoch Iteration Number:  78
Epoch Iteration Number:  79
Epoch Iteration Number:  80
Cost: 1.47388013897
Epoch Iteration Number:  81
Epoch Iteration Number:  82
Epoch Iteration Number:  83
Epoch Iteration Number:  84
Epoch Iteration Number:  85
Epoch Iteration Number:  86
Epoch Iteration Number:  87
Epoch Iteration Number:  88
Epoch Iteration Number:  89
Epoch Iteration Number:  90
Cost: 1.40999380423
Epoch Iteration Number:  91
```

In [6]:

```
predictedLabels = nn.predict(testMatrix)
diff = predictedLabels-testLabelArray
positiveInstances=np.count_nonzero(diff == 0)
accuracy = float(positiveInstances/len(diff))
print("Test set Accuracy: ",accuracy*100)
```

```
[3 1 0 ..., 5 2 7]
Test set Accuracy:  51.85999999999999
```

In [7]:
```
def save_predictions(filename, y):
    np.save(filename, y)

save_predictions("ans1-aka398.npy", predictedLabels)
```