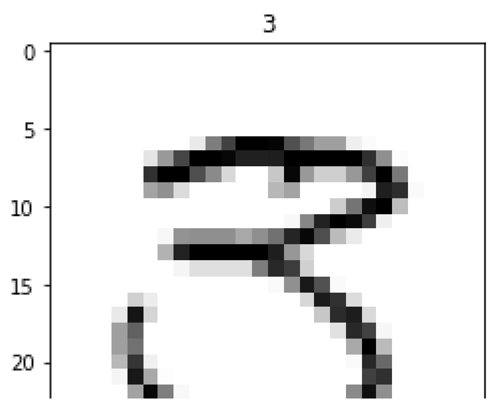
```
import numpy as np
import pandas as pd
import math
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import os
```

```
train_data = pd.read_csv("/content/train.csv")
test_data= pd.read_csv("/content/test.csv")
train_labels=np.array(train_data.loc[:,'label'])
train_data=np.array(train_data.loc[:,train_data.columns!='label'])
```

```
index=7;
plt.title((train_labels[index]))
plt.imshow(train_data[index].reshape(28,28), cmap=cm.binary)
```

```
<matplotlib.image.AxesImage at 0x7f3b52007850>
```




```
print("train data")
y_value=np.zeros((1,10))
for i in range (10):
```

```
    print("occurance of ",i,"=",np.count_nonzero(train_labels==i))
    y_value[0,i-1]= np.count_nonzero(train_labels==i)
```

```
train data
occurance of  0 = 4132
occurance of  1 = 4684
occurance of  2 = 4177
occurance of  3 = 4351
occurance of  4 = 4072
occurance of  5 = 3795
occurance of  6 = 4137
occurance of  7 = 4401
occurance of  8 = 4063
occurance of  9 = 4188
```

```
train_data=np.reshape(train_data,[784,42000])
train_label=np.zeros((10,42000))
for col in range (42000):
    val=train_labels[col]
    for row in range (10):
        if (val==row):
            train_label[val,col]=1
print("train_data shape="+str(np.shape(train_data)))
print("train_label shape="+str(np.shape(train_label)))
```

```
train_data shape=(784, 42000)
train_label shape=(10, 42000)
```

```
def sigmoid(Z):
    A = 1/(1+np.exp(-Z))
    cache = Z
    return A, cache

def relu(Z):
```

```
        A = np.maximum(0,Z)
        cache = Z
        return A, cache

    def softmax(Z):
        e_x = np.exp(Z)
        A= e_x / np.sum(np.exp(Z))
        cache=Z
        return A,cache


    def relu_backward(dA, cache):
        Z = cache
        dZ = np.array(dA, copy=True)
        dZ[Z <= 0] = 0
        assert (dZ.shape == Z.shape)
        return dZ

    def sigmoid_backward(dA, cache):
        Z = cache
        s = 1/(1+np.exp(-Z))
        dZ = dA * s * (1-s)
        assert (dZ.shape == Z.shape)
        return dZ

    def softmax_backward(Z,cache):
        Z=cache
        length=10
        dZ=np.zeros((42000,10))
        Z=np.transpose(Z)
        for row in range (0,42000):
                den=(np.sum(np.exp(Z[row,:])))*(np.sum(np.exp(Z[row,:])))
                for col in range (0,10):
                    sums=0
                    for j in range (0,10):
                        if (j!=col):
                            sums=sums+(math.exp(Z[row,j]))
```

```
                dZ[row,col]=(math.exp(Z[row,col])*sums)/den
        dZ=np.transpose(dZ)
        Z=np.transpose(Z)

        assert (dZ.shape == Z.shape)
        return dZ

    def initialize_parameters_deep(layer_dims):
        #np.random.seed(1)
        parameters = {}
        L = len(layer_dims)            # number of layers in the network

        for l in range(1, L):
            parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) / np.sqrt(layer_dims[l-1]) #*0.01
            parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        return parameters


    def linear_forward(A, W, b):
        Z = np.dot(W,A) +b
        cache = (A, W, b)
        assert(Z.shape == (W.shape[0], A.shape[1]))
        return Z, cache

    def linear_activation_forward(A_prev, W, b, activation):
        if activation == "sigmoid":
            # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
            Z, linear_cache = linear_forward(A_prev, W, b)
            A, activation_cache = sigmoid(Z)
        elif activation == "relu":
            # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
            Z, linear_cache = linear_forward(A_prev, W, b)
            #print("Z="+str(Z))
            A, activation_cache = relu(Z)
        elif activation == "softmax":
            # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
            Z, linear_cache = linear_forward(A_prev, W, b)
            A, activation_cache = softmax(Z)
```

```
    cache = (linear_cache, activation_cache)
    return A, cache

def L_model_forward(X, parameters):
    caches = []
    A = X
    L = len(parameters) // 2                  # number of layers in the neural network
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters['W' + str(l)], parameters['b' + str(l)], activation = "relu"
        caches.append(cache)
    AL, cache = linear_activation_forward(A, parameters['W' + str(L)], parameters['b' + str(L)], activation = "softmax")
    caches.append(cache)
    return AL, caches



def compute_cost(AL, Y):

    m = Y.shape[1]
    cost = (-1 / m) * np.sum(np.multiply(Y, np.log(AL)) + np.multiply(1 - Y, np.log(1 - AL)))
    return cost


def linear_backward(dZ, cache):
    A_prev, W, b = cache
    m = A_prev.shape[1]
    dW = 1./m * np.dot(dZ,A_prev.T)
    db = (1/m)*np.sum(dZ, axis=1, keepdims=True);
    dA_prev = np.dot(W.T,dZ)
    return dA_prev, dW, db

def linear_activation_backward(dA, cache, activation):
    linear_cache, activation_cache = cache
    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
```

```
            dA_prev, dW, db = linear_backward(dZ, linear_cache)
        elif activation == "softmax":
            dZ = softmax_backward(dA, activation_cache)
            dA_prev, dW, db = linear_backward(dZ, linear_cache)
        return dA_prev, dW, db

    def L_model_backward(AL, Y, caches):
        grads = {}
        L = len(caches)
        dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))
        M=len(layers_dims)
        current_cache = caches[M-2]
        grads["dA"+str(M-1)], grads["dW"+str(M-1)], grads["db"+str(M-1)] = linear_activation_backward(dAL, current_cache, activa
        for l in reversed(range(L-1)):
            current_cache = caches[l]
            dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA" + str(l + 2)], current_cache, activation = "r
            grads["dA" + str(l + 1)] = dA_prev_temp
            grads["dW" + str(l + 1)] = dW_temp
            grads["db" + str(l + 1)] = db_temp

        return grads



    def update_parameters(parameters, grads, learning_rate):
        for l in range(len_update-1):
            parameters["W" + str(l+1)] =parameters["W" + str(l+1)] - (learning_rate*grads["dW" + str(l+1)])
            parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - (learning_rate*grads["db" + str(l+1)])
        return parameters


    def plot_graph(cost_plot):

        x_value=list(range(1,len(cost_plot)+1))
        plt.xlabel('iteration')
        plt.ylabel('cost')
        plt.plot(x_value,cost_plot,0.,color='g')



    layers_dims = [784,500,400,300,100,10]
```

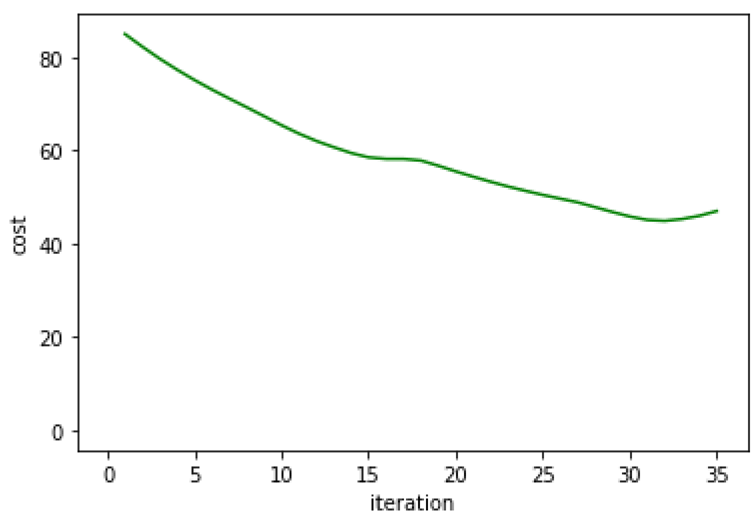
```
layers_dims = [784,500,400,300,100,10]
len_update=len(layers_dims)



def L_layer_model(X, Y, layers_dims, learning_rate , num_iterations , print_cost=False):#lr was 0.009
    costs = []
    cost_plot=np.zeros(num_iterations)
    parameters = initialize_parameters_deep(layers_dims)
    for i in range(0, num_iterations):
        AL, caches = L_model_forward(X, parameters)
        cost =compute_cost(AL, Y)
        grads = L_model_backward(AL, Y, caches)
        parameters = update_parameters(parameters, grads, learning_rate)
        cost_plot[i]=cost;

    plot_graph(cost_plot)
    return parameters


parameters = L_layer_model(train_data, train_label, layers_dims,learning_rate = 0.0005, num_iterations =35 , print_cost = Tr
```



✓ 5m 47s completed at 12:00 AM