```
import numpy as np
from numpy import random as rnd
import csv
import matplotlib.pyplot as plt
from collections import defaultdict
#from keras.datasets import cifar10 # Data from keras package

def unpickle(file):
    "Returns an a dictionary of a single batch, if filename is correct"
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict


def calculate_mean(x_train):
    "Calculate the mean of a set of training data"
    mean = np.mean(x_train,axis=0)
    return mean


def center_data(x_train):
    #TODO make sure that this works as intended
    #x_train = x_train.astype('float32')
    mean = calculate_mean(x_train)
    centered_data = x_train - mean
    centered_data /= 255
    #centered_data = centered_data.astype(np.uint8)
    return centered_data

def sigmoid_activation_function(b):


    return np.power((1+np.exp(-b)), -1)


def shuffle(X, y):
    '''
    Shuffle two corresponding arrays
    '''
    assert len(X) == len(y)
    random = np.arange(len(X))
    np.random.shuffle(random)
    X = X[random]
    y = y[random]
    return X, y


def g(b):
    #return np.tanh(b)    inner_sum = inner_sum.reshape((len(inner_sum),1))
    #return np.tanh(b)
    return sigmoid_activation_function(b)

def dg(b):
    #return (1-np.power(np.tanh(b),2))

    o_b = sigmoid_activation_function(b)
    #o_b= (1-np.power(np.tanh(b),2))
    #return o_b
    return o_b*(1-o_b)
```

```python

def create_layer(inputsize, layersize):
    neurons = np.zeros((layersize,1))
    neurons = neurons.reshape(layersize,1)
    w_ij = rnd.normal(size=(layersize, inputsize),scale=(1/(np.sqrt(inputsize))))
    return (neurons, w_ij)

def esign(i):
    if i == 0:
        return 1
    return np.sign(i)

def calculate_output(threshold, weight_matrix, x):
    if weight_matrix.shape[1]==1:
        weight_matrix = weight_matrix.T
    inner_sum = weight_matrix@x
    #inner_sum = inner_sum.reshape((len(inner_sum),x.shape[1]))
    inner_sum = inner_sum - threshold
    ret = g(inner_sum)
    assert(ret.shape == threshold.shape)
    return ret

def calculate_output_batch(threshold, weight_matrix, V, batchsize):


    inner_sum = weight_matrix@V
    #inner_sum = inner_sum.reshape((len(inner_sum),x.shape[1]))
    inner_sum = inner_sum.reshape((threshold.shape[0],batchsize))
    inner_sum = inner_sum - threshold
    ret = g(inner_sum)
    assert(ret.shape == (threshold.shape[0],batchsize))
    return ret



def propagate_forward(V : list, O : list, w : list, batchsize : int) -> None:

    #TODO make sure that it works as intended

    assert(len(V)-1==len(O) and len(O)==len(w))

    for l in range(1,len(V)):
        V[l]= calculate_output_batch(O[l-1],w[l-1],V[l-1], batchsize)


def calculate_b(w, x, O):
    inner_sum = np.dot(w,x)
    inner_sum = inner_sum - O
    return inner_sum

def calculate_d_L(b, t, V):

    #TODO make sure it works properly
    return dg(b)(t-V)

def calculate_single_d(d_l, w, b):

    #TODO make sure that the order of multiplications is right
    dgb = dg(b)
    inner_product = (d_l.T@w).T
    #ret = np.multiply(inner_product.T, dgb)
    ret = np.multiply(inner_product,dgb)
    #ret = d_l*w.T*dgb
    #print(ret)
    return ret


def calculate_all_d(V, O, w, t):

    #TODO make sure that it works as intended
    L = len(O)-1
    b_L = calculate_b(w[L],V[L], O[L])
    b_L = dg(b_L)
    temp = (t-V[-1])
    delta_start = b_L*temp
    d = [[]]*(L+1)
    d[L] = delta_start
    for i in range(L, 0, -1):
        #TODO make sure the range function works as intended
        if(w[i].shape[1]==1):
            w[i] = w[i].T

        b =  calculate_b(w[i-1],V[i-1],O[i-1])
        delta = calculate_single_d(d[i],w[i],b)
        d[i-1] = delta
    return d



def propagate_backwards(V : list, O : list, w : list, t : int, learning_rate: float, batchsize : int):
    #TODO include support for batchsize
    L = len(w)
    d = calculate_all_d(V,O,w,t)
    for l in range(L):
        if(w[l].shape[1]==1):
            d_w = learning_rate*(d[l]@V[l])
        else:
            d_w = learning_rate*(d[l]@V[l].T)
        d_O = learning_rate* (d[l]@np.ones((batchsize,1)))
        w[l] +=  d_w
        O[l] -= d_O


def create_multible_layers(*argv):
    w = []
    V = []
    O = []
    inputlayer = np.zeros((argv[0],1))
    V.append(inputlayer)
    old_layersize = argv[0]
    for layersize in argv[1:]:
        O_i = np.zeros((layersize,1))
        V_i, w_i = create_layer(old_layersize,layersize)
        old_layersize = layersize
        w.append(w_i)
        V.append(V_i)
        O.append(O_i)
    return (w,V,O)
```

```python


def learn(learning_rate:float, data: list, labels: list,test_data : list, test_labels : list, batchsize: int, *argv):

    w,V,O = create_multible_layers(*argv)
    #merged = np.concatenate([data, labels], axis=1)
    ret = defaultdict(list)
    #TODO create a dictionary
    return_w = []
    return_c = []
    return_O = []
    return_H = []
    return_u = []
    for T in range(100):
        #Permutate the data in the beginning of each epoch
        data,labels = shuffle(data,labels)
        for i in range(0, len(data), batchsize):
        # Add test after each epoch (also easy)
            #print("test")
            my = i
            bs = batchsize
            if my + batchsize >= len(data):
                bs =  len(data) - my
            V[0] = np.array(data[my:my+bs]).T
            t = np.array(labels[my:my+bs]).T
            #t = t.reshape(bs,t.shape[1])

            propagate_forward(V,O,w, bs)
            propagate_backwards(V,O,w,t,learning_rate, bs)
        #print(w[0])
        #C = calculate_classification_error(V,w,O,test_data, test_labels)
        #print(w[0])
        newH, new_u = calculate_H_and_U(V,w,O,data, labels)
        return_H.append(newH)
        return_u.append(new_u)
        #return_c.append(C)
        return_w.append(w)
        return_O.append(O)
    print("Finished learning!")
    ret["H"] = return_H
    ret["u"] = return_u
    ret["w"] = return_w
    ret["O"] = return_O
    ret["C"] = return_c
    return ret

def calculate_H_and_U(V,w,O,training_set, label_set):
    H = 0
    #initialize u(l)
    u = []
    for O_l in O:
        u.append(np.zeros(O_l.shape))
    batchsize = 100
    for i in range(0,len(training_set),batchsize):
        my = i
        bs = batchsize
        if my + batchsize >= len(data):
            bs =  len(training_set) - my
        V[0] = training_set[my:my+bs].T
        t = label_set[my:my+bs].T
        propagate_forward(V,O,w,bs)
        #TODO add to U(l) and H
        #Update H
        y_i = np.argmax(V[-1],axis=0)
        y_i = y_i.tolist()
        transformlabels(y_i,10)
        y_i = np.array(y_i).T
        #y = np.zeros((len(V[-1]),bs))
        #print(y_i)
        #y[:,y_i] = 1
        y_i = y_i.reshape(10,bs)
        temp = y_i - t
        H += np.sum(np.abs(y_i-t))
        #update u
        d = calculate_all_d(V,O,w,t)
        for l in range(len(u)):
            u[l] += (d[l]@np.ones((batchsize,1)))
    for l in range(len(u)):
        u[l] = np.linalg.norm(u[l])
    return H/2, u




def calculate_classification_error(V,w,O, test_data, test_labels):
    inner_sum = 0
    pval = len(test_data)
    for my in range(pval):
        V[0] = np.array(test_data[my])
        t_my = test_labels[my]
        #Propagate Forward
        propagate_forward(V,O,w,1)
        #inner_sum += np.abs(esign(V[-1][0])-t_my)
        y_i = np.argmax(V[-1],axis=0)
        y = np.zeros((len(V[-1]),1))
        #print(y_i)
        y[y_i] = 1
        inner_sum += np.sum(np.abs(y-t_my))
        #print("error")
    return inner_sum/(2*pval)



def transformlabels(labels, n):
    for i,label in enumerate(labels):
        temp = np.zeros((n,1))
        temp[label] = 1
        labels[i] = temp


folder_name = "/home/edin/uni/ANN/ex-3/cifar-10-batches-py/"
batch_1 = unpickle(folder_name+"data_batch_1")
batch_2 = unpickle(folder_name+"data_batch_2")
batch_3 = unpickle(folder_name+"data_batch_3")
batch_4 = unpickle(folder_name+"data_batch_4")
batch_5 = unpickle(folder_name+"data_batch_5")

data = batch_1[b'data']
data = np.append(data,batch_2[b'data'], axis=0)
data = np.append(data,batch_3[b'data'], axis=0)
data = np.append(data,batch_4[b'data'], axis=0)
data = np.append(data,batch_5[b'data'], axis=0)
labels = batch_1[b'labels']
#Since labels are only a normal array
labels += batch_2[b'labels']
labels += batch_3[b'labels']
labels += batch_4[b'labels']
labels += batch_5[b'labels']
data = center_data(data)
transformlabels(labels,10)
labels = np.array(labels)
labels = labels.astype(np.uint8)
labels = np.array(labels).reshape(50000,10)

test_batch = unpickle(folder_name+"test_batch")
test_data = batch_5[b'data']
test_labels = batch_5[b'labels']
test_data = center_data(test_data)
transformlabels(test_labels,10)
test_labels = np.array(test_labels)

test_labels = test_labels.astype(np.uint8)

"""
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# Samples in trainset
samples_train = X_train.shape[0]

# Samples in testset
samples_test = X_test.shape[0]

# Input units (3072)
input_units = X_train.shape[1]*X_train.shape[2]*X_train.shape[3]

X_train = X_train.reshape(-1, input_units)

#transformlabels(y_train,10)
X_train_cent = center_data(X_train)


def one_hot_enc(y):
    z = 10
    return np.eye(z)[y].reshape((len(y), z))

y_train = one_hot_enc(y_test)
"""
learning_rate = 0.01
batchsize = 100
#IT DOES NOT WORK
#POSSIBLE PROBLEMS:
# Permutation is not correct
# -> Add assert statements or smth
# Centering Function is fucked up
# Batch processing is fucked up
# -> Test with Aufgabe 2.1 and see if it works
ret_dic = learn(learning_rate, data,labels,[],[],batchsize,3072,20,20,20,20,10)

```