```python
import pickle
from random import uniform
from tqdm import tqdm

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
pd.options.display.width = 0


class Dataset():
    """ class representing a dataset """

    def __init__(self, data=None, K=None, X=None, Y=None, y=None, name=None):
        if data is not None:
            self.X, self.Y, self.y = self.separate(data, K)
        else:
            self.X, self.Y, self.y = X, Y, y
        self.name = name


    def __str__(self):
        return "Dataset: " + self.name

    def separate(self, data, K):

        def loadBatch(filename):
            """ Copied from the dataset website, given for lab """
            with open('../Dataset/'+filename, 'rb') as fo:
                dict = pickle.load(fo, encoding='bytes')
            return dict

        def sep(data):
            """ does the separation into datapoints, one-hot matrix and labels """
            X = np.array(data.get(b'data'), dtype=float).T
            labels = np.array([data.get(b'labels')])
            Y = np.zeros((K, X.shape[1]))
            Y[labels, np.arange(labels.size)] = 1
            return X, Y, labels

        d = loadBatch(data[0])
        X, Y, y = sep(d)
        if len(data) > 1:
            for i in range(1, len(data)):
```

```
            d = loadBatch(data[i])
            Xc, Yc, y_c = sep(d)
            X = np.concatenate((X, Xc), axis=1)
            Y = np.concatenate((Y, Yc), axis=1)
            y = np.concatenate((y, y_c), axis=1)
        return X, Y, y

    def split(self, Lsplit, Usplit, name):
        """ split existing object data and creates new """
        return Dataset(X=self.X[:, Lsplit:Usplit], Y=self.Y[:, Lsplit:Usplit], y=self.y[:, Lsplit:Usplit], name=name)

    def normalize(self, mean, std):
        self.X = (self.X - mean) / std


class Network():
    """ class representing a neural network """

    def __init__(self, layers, lmbda=0.001, eta=0.001, batchNorm=False, init_method='he', sig_value=None):
        # initialization
        self.K = layers[-1]
        self.lmbda = lmbda
        self.eta = eta
        self.W = []
        self.b = []
        self.x = []
        self.layStruct = layers
        self.endEpoch = None
        self.batchNorm = batchNorm
        self.init_method = init_method
        if self.init_method != 'he':
            self.sig_value = sig_value
        if self.batchNorm:
            self.muAvg = []
            self.varAvg = []
            self.alpha = 0.9
            self.gamma = []
            self.beta = []
            self.sHats = []
            self.s = []
            self.mu = []
            self.var = []
        self.setWeightsBiases()
        # metrics
```

```
        self.trainAcc = []
        self.valAcc = []
        self.testAcc = []
        self.trainCost = []
        self.valCost = []
        self.testCost = []
        self.trainLoss = []
        self.valLoss = []
        self.testLoss = []

    def __str__(self):
        toStr = {
            "layers": [len(self.layStruct)-2],
            "lambda": [self.lmbda],
            "eta": [self.eta],
            "training accuracy (max)": [max(self.trainAcc)],
            "training accuracy (max) epoch": [np.argmax(self.trainAcc)],
            "training loss (min)": [min(self.trainLoss)],
            "training loss (min) epoch": [np.argmin(self.trainLoss)],
            "validation accuracy (max)": [max(self.valAcc)],
            "validation accuracy (max) epoch": [np.argmax(self.valAcc)],
            "validation loss (min)": [min(self.valLoss)],
            "validation loss (min) epoch": [np.argmin(self.valLoss)],
            "test accuracy": [self.testAcc[0]]
        }
        return str(pd.DataFrame(toStr))

    def setWeightsBiases(self, mu=0.0):
        """ initialize weights and biases """
        np.random.seed(400)
        self.W.clear()
        self.b.clear()
        if self.batchNorm:
            self.gamma.clear()
            self.beta.clear()
            self.muAvg.clear()
            self.varAvg.clear()
        for i, currentLayer in enumerate(self.layStruct[:-1]):
            nextLayer = self.layStruct[i+1]
            if self.init_method == 'he':
                self.W.append(np.random.normal(mu, (np.sqrt(2/currentLayer)), (nextLayer, currentLayer)))
            else:
                self.W.append(np.random.normal(mu, self.sig_value, (nextLayer, currentLayer)))
            self.b.append(np.zeros((nextLayer, 1)))
```

```python
        if self.batchNorm and i < len(self.layStruct) -2:
            self.gamma.append(np.ones((nextLayer, 1)))
            self.beta.append(np.zeros((nextLayer, 1)))
            self.muAvg.append(np.zeros((nextLayer, 1)))
            self.varAvg.append(np.zeros((nextLayer, 1)))

def evaluateClassifier(self, X, W, b, gamma=None, beta=None):
    """
    Outputs P = softmax(Wx + b) as KxDim-matrix,
    where each column is sums to 1
    """
    def relu(x):
        return np.maximum(0, x)

    def softmax(x):
        """ Standard definition of the softmax function, given for lab """
        return np.exp(x) / np.sum(np.exp(x), axis=0)

    def batchNorm(s, i, gamma, beta):
        """ computes the batch normalization step """
        # calculate current values
        mu_c = np.mean(s, axis=1, keepdims=True)
        var_c = np.var(s, axis=1, keepdims=True)
        sHat_c = (s - mu_c) / np.sqrt(var_c + np.finfo(float).eps)
        # append for backward pass
        self.mu.append(mu_c)
        self.var.append(var_c)
        self.sHats.append(sHat_c)
        self.muAvg[i] = self.alpha * self.muAvg[i] + (1-self.alpha) * mu_c
        self.varAvg[i] = self.alpha * self.varAvg[i] + (1-self.alpha) * var_c
        if gamma == beta == None:
            return np.multiply(self.gamma[i], sHat_c) + self.beta[i]
        if gamma == None and beta != None:
            return np.multiply(self.gamma[i], sHat_c) + beta[i]
        if gamma != None and beta == None:
            return np.multiply(gamma[i], sHat_c) + self.beta[i]

    self.x.clear()
    self.x.append(X)
    if self.batchNorm:
        self.mu.clear()
        self.var.clear()
        self.s.clear()
        self.sHats.clear()
```

```python
        for i in range(len(W) - 1):
            s = np.matmul(W[i], X) + b[i]
            if self.batchNorm:
                self.s.append(s)
                s = batchNorm(s, i, gamma, beta)
            X = relu(s)
            self.x.append(X)
        return softmax(np.matmul(W[-1], X) + b[-1])

    def computeCost(self, X, Y, W, b, gamma=None, beta=None):
        """ computes cost of loss for the network """
        P = self.evaluateClassifier(X, W, b, gamma, beta)
        L = ((1 / np.size(X, 1)) * -np.sum(Y*np.log(P)))
        reg = sum([(self.lmbda * np.sum(np.square(w))) for w in W])
        J = L + reg
        return J, P, L

    def computeAccuracy(self, P, y):
        """ Accuracy defined as correctly classified of total datapoints """
        P_max = np.array([np.argmax(P, axis=0)])
        return np.array(np.where(P_max == np.array(y))).shape[1] / np.size(y)

    def computeGradients(self, P, Y, bsize):
        """ Computes gradients using chain rule """

        def gradWeightsBiases(G, bsize, i):
            """ computes gradient of w_i and b_i """
            gradW.insert(0, ((1 / bsize) * np.matmul(G, np.array(self.x[i]).T) + 2*self.lmbda*self.W[i]))
            gradB.insert(0, (np.array((1 / bsize) * np.matmul(G, np.ones(bsize))).reshape(np.size(self.W[i], 0), 1)))
            G = np.matmul(self.W[i].T, G)
            indH = np.where(self.x[i] > 0, 1, 0)
            return np.multiply(G, indH > 0)

        def gradGammaBeta(G, bsize, i):
            """ computes gradient of gamma_i and beta_i """

            def batchNormBackPass(G, i):
                """ back propogation for batch normalization """
                n = G.shape[1]
                sigma1 = np.power(self.var[i] + np.finfo(float).eps, -0.5)
                sigma2 = np.power(self.var[i] + np.finfo(float).eps, -1.5)
                G1 = np.multiply(G, sigma1)
                G2 = np.multiply(G, sigma2)
                D = self.s[i] - self.mu[i]
```

```python
            c = np.sum(np.multiply(G2, D), axis=1, keepdims=True)
            G = G1 - (1/n) * np.sum(G1, axis=1, keepdims=True) - (1/n) * np.multiply(D, c)
            return G

        i = i - 1
        gI = np.array((1 / bsize) * np.matmul(np.multiply(G, self.sHats[i]), np.ones(bsize).reshape(bsize, 1)))
        bI = np.array((1 / bsize) * np.matmul(G, np.ones(bsize).reshape(bsize, 1)))
        gradGamma.insert(0, gI)
        gradBeta.insert(0, bI)
        G = np.multiply(G, self.gamma[i])
        G = batchNormBackPass(G, i)
        return G

    gradW = []
    gradB = []
    gradGamma = []
    gradBeta = []
    G_out = -(Y - P)
    for i in reversed(range(len(self.x))):
        G_out = gradWeightsBiases(G_out, bsize, i)
        if self.batchNorm and i > 0:
            G_out = gradGammaBeta(G_out, bsize, i)
    return [gradW, gradB, gradGamma, gradBeta]

def computeGradsNumSlow(self, X, Y, h):
    """ Converted from matlab code, modifed for k layers """

    gradW = [np.zeros(w.shape) for w in self.W]
    gradB = [np.zeros(b.shape) for b in self.b]
    gradGamma = [np.ones(gl.shape) for gl in self.gamma]
    gradBeta = [np.zeros(bl.shape) for bl in self.beta]

    W = self.W.copy()
    B = self.b.copy()
    gamma = self.gamma.copy()
    beta = self.beta.copy()


    for i, b in enumerate(B):
        for j in range(len(b)):
            bTry = np.array(b)
            bTry[j] -= h
            B[i] = bTry
            c1, _, _ = self.computeCost(X, Y, self.W, B)
```

```
            bTry = np.array(b)
            bTry[j] += h
            B[i] = bTry
            c2, _, _ = self.computeCost(X, Y, self.W, B)

            gradB[i][j] = (c2-c1) / (2*h)

    for k, w in enumerate(W):
        for i in range(w.shape[0]):
            for j in range(w.shape[1]):
                wTry = np.array(w)
                wTry[i, j] -= h
                W[k] = wTry
                c1, _, _ = self.computeCost(X, Y, W, self.b)

                wTry = np.array(w)
                wTry[i, j] += h
                W[k] = wTry
                c2, _, _ = self.computeCost(X, Y, W, self.b)

                gradW[k][i, j] = (c2-c1) / (2*h)

    for i, g in enumerate(gamma):
        for j in range(len(g)):
            gTry = np.array(g)
            gTry[j] -= h
            gamma[i] = gTry
            c1, _, _ = self.computeCost(X, Y, self.W, self.b, gamma=gamma)

            gTry = np.array(g)
            gTry[j] += h
            gamma[i] = gTry
            c2, _, _ = self.computeCost(X, Y, self.W, self.b, gamma=gamma)

            gradGamma[i][j] = (c2-c1) / (2*h)

    for i, bt in enumerate(beta):
        for j in range(len(bt)):
            btTry = np.array(bt)
            btTry[j] -= h
            beta[i] = btTry
            c1, _, _ = self.computeCost(X, Y, self.W, self.b, gamma=None, beta=beta)
```

```python
            btTry = np.array(bt)
            btTry[j] += h
            beta[i] = btTry
            c2, _, _ = self.computeCost(X, Y, self.W, self.b, gamma=None, beta=beta)

            gradBeta[i][j] = (c2-c1) / (2*h)

    return [gradW, gradB, gradGamma, gradBeta]

def gradientCheck(self, gradA, gradN, eps):
    """ computes the relative error between analytical and numerical gradient """

    def check(gA, gN, eps):
        diff = np.absolute(np.subtract(gA, gN))
        thresh = np.full(diff.shape, eps)
        summ = np.add(np.absolute(gA), np.absolute(gN))
        denom = np.maximum(thresh, summ)
        return np.divide(diff, denom)

    gradRes = []
    for i in range(len(gradA)):
        gradRes.append(check(gradA[i], gradN[i], eps))
    return gradRes

def updateParameters(self, gradW, gradB):
    for i in range(len(self.W)):
        self.W[i] = self.W[i] - self.eta * gradW[i]
        self.b[i] = self.b[i] - self.eta * gradB[i]

def updateEta(self, eta):
    self.eta = eta

def miniBatch(self, train, bsize, cycEtaData=None, cyclicalEta=False, shuffle=False):
    """ bsize'ed batches evaluated """

    if cycEtaData is not None:
        etaMin, etaMax, ns, t, l, cyclicalEta = cycEtaData
        diff = etaMax-etaMin

    randI = np.random.permutation(train.X.shape[1])
    for i in range(int(np.size(train.X, 1)/bsize)):
        if shuffle:
            randBatchRange = range(i * bsize, ((i + 1) * bsize))
            batchRange = randI[randBatchRange]
```

```
            else:
                batchRange = range(i * bsize, ((i + 1) * bsize))
            P = self.evaluateClassifier(train.X[:, batchRange], self.W, self.b)
            grad = self.computeGradients(P, train.Y[:, batchRange], bsize)
            self.updateParameters(grad[0], grad[1])

            if cyclicalEta:
                tmin, tmax = 2*l*ns, (2*l+1)*ns
                if (tmin <= t <= tmax):
                    self.updateEta(etaMin + ((t - tmin) / ns) * diff)
                else:
                    self.updateEta(etaMax - ((t - tmax) / ns) * diff)
                t += 1
                if (t % (2*ns)) == 0:
                    print("cycle complete")
                    l += 1
        return [etaMin, etaMax, ns, t, l, cyclicalEta] if cyclicalEta else None

    def fit(self, train, val, nepochs=200, bsize=100, cyclicalEta=False, numCycles=3, nsAq=500, shuffle=False, cycEtaData=None,
lmbda=None, lmbdaSearch=False):

        if cyclicalEta:
            if cycEtaData is None:
                etaMin, etaMax = 10**-5, 10**-1
                cycEtaData = [etaMin, etaMax]
            ns = nsAq
            cycEtaData.extend([ns, 0, 0, True])
            self.updateEta(cycEtaData[0])

        if lmbda is not None:
            self.lmbda = lmbda

        for epoch in tqdm(range(nepochs)):
            cycEtaData = self.miniBatch(train, bsize=bsize, cycEtaData=cycEtaData, shuffle=shuffle)
            if not lmbdaSearch:
                trainAcc, trainLoss, trainCost = self.computeAccLoss(train)
                self.trainAcc.append(trainAcc)
                self.trainLoss.append(trainLoss)
                self.trainCost.append(trainCost)
                valAcc, valLoss, valCost = self.computeAccLoss(val)
                self.valAcc.append(valAcc)
                self.valLoss.append(valLoss)
                self.valCost.append(valCost)
            if cyclicalEta and cycEtaData[4] == numCycles:
```

```python
                break
        self.endEpoch = epoch

    def evaluate(self, data):
        testAcc, testLoss, testCost = self.computeAccLoss(data)
        self.testAcc.append(testAcc)
        self.testLoss.append(testLoss)
        self.testCost.append(testCost)

    def computeAccLoss(self, data):
        J, P, L = self.computeCost(data.X, data.Y, self.W, self.b)
        acc = self.computeAccuracy(P, data.y)
        return acc, L, J


def plotGraph(lst1, lst2, rangeX, yLabel, xLabel, lst1Label, lst2Label, yLim):
    fig, ax = plt.subplots()
    ax.plot(rangeX, lst1, label=lst1Label)
    ax.plot(rangeX, lst2, label=lst2Label)
    ax.legend()
    ax.set(xlabel=xLabel, ylabel=yLabel, ylim=(0, yLim), xlim=(0, len(rangeX)))
    ax.grid()
    ax.margins(0)


def trainBatches(size):
    """ choose to train with 1-5 batches """
    trainVal = ['data_batch_1', 'data_batch_2', 'data_batch_3', 'data_batch_4', 'data_batch_5'][:size]
    test = ['test_batch']
    return [trainVal, test]

def getData(batches, trSize, K):
    allData = Dataset(batches[0], K)
    splitNr = int(trSize*allData.X.shape[1])
    train = allData.split(0, splitNr, name="training data")
    val = allData.split(splitNr, -1, name="validation data")
    test = Dataset(batches[-1], K, name="test data")
    # normalize datapoints to training data
    mean = np.array([np.mean(train.X, 1)]).T
    std = np.array([np.std(train.X, 1)]).T
    train.normalize(mean, std)
    val.normalize(mean, std)
    test.normalize(mean, std)
    return train, val, test
```

```python
def lambdaSearch(sgd, train, val, cycles=2, lMin=0.001, lMax=0.05, n=20, t=2, eps=0.0001):
    narrow = np.ceil(n/t)
    res = np.full((n, 3), -1.0)
    i = 0
    l = 1
    while i < n:
        print("coarse-search", i+1)
        lmbda = uniform(lMin, lMax)
        sgd.fit(train, val, nepochs=100, cyclicalEta=True, numCycles=2, nsAq=2250, lmbda=0.005, lmbdaSearch=True, shuffle=True)
        _, P, _ = sgd.computeCost(val.X, val.Y, sgd.W, sgd.b)
        acc_val = sgd.computeAccuracy(P, val.y)
        res[i][0], res[i][1], res[i][2] = lmbda, acc_val, i+1
        i += 1
        sgd.setWeightsBiases()
        print("val_acc={}".format(acc_val), "lmbda={}".format(sgd.lmbda))
        if i == l*narrow:
            print("fine-search")
            res = res[res[:, 1].argsort()[::-1]]
            lMin, lMax = sorted([res[0][0], res[1][0]])
            lMin -= eps
            lMax += eps
            l += 1
            print("lMin", lMin, "lMax", lMax)
    res = res[res[:, 1].argsort()[::-1]]
    df = pd.DataFrame(data=res, columns=["lambda", "val_accuracy", "iteration"])
    tfile = open('lambda_search2_lab3.txt', 'a')
    tfile.write(df.to_string(index=False))
    tfile.close()
    return res[0][0]


def main():
    K = 10
    batches = trainBatches(4)
    trainData, valData, testData = getData(batches=batches, trSize=0.9, K=K)


    """ # EXERCISE 2
    # 3-LAYER 0.004881282007886553
    sgd = Network(layers=[trainData.X.shape[0], 50, 50, K], lmbda=None, eta=0.001, batchNorm=True)
    sgd.fit(trainData, valData, nepochs=200, bsize=100, cyclicalEta=True, numCycles=2, nsAq=2250, lmbda=0.005, shuffle=True)
    sgd.evaluate(testData) """
```

```
    """ # 9-LAYER
    sgd = Network(layers=[trainData.X.shape[0], 50, 30, 20, 20, 10, 10, 10, 10, K], lmbda=0.005, eta=0.001, batchNorm=True)
    sgd.fit(trainData, valData, nepochs=200, bsize=100, cyclicalEta=True, numCycles=2, nsAq=2250, lmbda=0.005, shuffle=True)
    sgd.evaluate(testData)
     """

    """ # EXERCISE 3
    # LAMBDA SEARCH 0.0035267718972283686
    sgd = Network(layers=[trainData.X.shape[0], 50, 50, K], lmbda=None, eta=0.001, batchNorm=True)
    lmbda = lambdaSearch(sgd, trainData, valData)
    print(lmbda) """

    """ # EXERCISE 3
    # training of best network using lambda-value from lambda search
    # 3-LAYER
    sgd = Network(layers=[trainData.X.shape[0], 50, 50, K], lmbda=None, eta=0.001, batchNorm=True)
    sgd.fit(trainData, valData, nepochs=200, bsize=100, cyclicalEta=True, numCycles=3, nsAq=2250, lmbda=0.0035267718972283686,
shuffle=True)
    sgd.evaluate(testData)  """

    """ # 9-LAYER
    sgd = Network(layers=[trainData.X.shape[0], 50, 30, 20, 20, 10, 10, 10, 10, K], lmbda=None, eta=0.001, batchNorm=True)
    sgd.fit(trainData, valData, nepochs=200, bsize=100, cyclicalEta=True, numCycles=2, nsAq=2250, lmbda=0.0035267718972283686,
shuffle=True)
    sgd.evaluate(testData) """

    """ # EXERCISE 4
    # Sensitivity to initialization
    res = np.full((6, 4), -1.0)
    i = 0
    for bn in [True, False]:
        for sig in [0.1, 0.001, 0.0001]:
            sgd = Network(layers=[trainData.X.shape[0], 50, 50, K], lmbda=None, eta=0.001, batchNorm=bn, init_method='sig',
sig_value=sig)
            sgd.fit(trainData, valData, nepochs=200, bsize=100, cyclicalEta=True, numCycles=2, nsAq=2250, lmbda=0.005, shuffle=True)
            sgd.evaluate(testData)
            bn_true = 1.0 if bn else 0.0
            res[i][0], res[i][1], res[i][2], res[i][3] = bn_true, sig, max(sgd.valAcc), sgd.testAcc[0]
            i+=1
    print(res)
    df = pd.DataFrame(data=res, columns=["batch_norm", "sig_value", "val_accuracy", "test_accuracy"])
    tfile = open('sense_init_lab3.txt', 'a')
    tfile.write(df.to_string(index=False))
    tfile.close() """
```

```python
    """ sgd = Network(layers=[trainData.X.shape[0], 50, 50, K], lmbda=None, eta=0.001, batchNorm=True, init_method='sig', 
sig_value=0.0001)
    sgd.fit(trainData, valData, nepochs=200, bsize=100, cyclicalEta=True, numCycles=2, nsAq=2250, lmbda=0.005, shuffle=True) """

    """ sgd = Network(layers=[trainData.X.shape[0], 50, 50, K], lmbda=None, eta=0.001, batchNorm=False, init_method='sig', 
sig_value=0.0001)
    sgd.fit(trainData, valData, nepochs=200, bsize=100, cyclicalEta=True, numCycles=2, nsAq=2250, lmbda=0.005, shuffle=True) """

    """ # plot graphs for cost, loss and accuracy of the trained network
    plotGraph(sgd.trainCost, sgd.valCost, range(sgd.endEpoch+1),
              "Cost", "Epochs", "Training cost", "Validation cost", 4)
    plotGraph(sgd.trainLoss, sgd.valLoss, range(sgd.endEpoch+1),
              "Loss", "Epochs", "Training loss", "Validation loss", 3)
    plotGraph(sgd.trainAcc, sgd.valAcc, range(sgd.endEpoch+1), "Accuracy",
              "Epochs", "Training accuracy", "Validation accuracy", 1)
    #print(sgd)
    plt.show() """


if __name__ == "__main__":
    main()


####### UNIT TEST - EXTRACT TO NEW SCRIPT IF USING ########

import unittest

import numpy as np

from code.assignment3 import *


class TestModel(unittest.TestCase):

    def setUp(self):
        dim = 20
        dp = 5
        train_data, _, _ = getData([['data_batch_1']], 0.5, 10)
        self.net = Network(layers=[dim, 50, 50, 10], lmbda=0, eta=0.001, batchNorm=True)
        self.eps = 1e-5
        P = self.net.evaluateClassifier(train_data.X[:dim, :dp], self.net.W, self.net.b)
        self.gradW_a, self.gradB_a, self.gradGamma_a, self.gradBeta_a = self.net.computeGradients(P[:dim, :dp], train_data.Y[:dim, 
:dp], dp)
```

```python
        self.gradW_n, self.gradB_n, self.gradGamma_n, self.gradBeta_n = self.net.computeGradsNumSlow(train_data.X[:dim, :dp],
train_data.Y[:dim, :dp], 1e-5)
        self.checkW = self.net.gradientCheck(self.gradW_a, self.gradW_n, self.eps)
        self.checkB = self.net.gradientCheck(self.gradB_a, self.gradB_n, self.eps)
        self.checkGamma = self.net.gradientCheck(self.gradGamma_a, self.gradGamma_n, self.eps)
        self.checkBeta = self.net.gradientCheck(self.gradBeta_a, self.gradBeta_n, self.eps)

    def test_gradientMean(self):
        for i in range(len(self.gradW_a)):
            self.assertAlmostEqual(self.gradW_n[i].mean(), self.gradW_a[i].mean(), places=7)
            self.assertAlmostEqual(self.gradB_n[i].mean(), self.gradB_a[i].mean(), places=7)

        for i in range(len(self.gradGamma_a)):
            self.assertAlmostEqual(self.gradGamma_n[i].mean(), self.gradGamma_a[i].mean(), places=7)
            self.assertAlmostEqual(self.gradBeta_n[i].mean(), self.gradBeta_a[i].mean(), places=7)

    def test_relError(self):
        for i in range(len(self.checkW)):
            self.assertLessEqual(np.max(self.checkW[i]), self.eps)
            self.assertLessEqual(self.checkW[i].mean(), self.eps)
            self.assertLessEqual(np.max(self.checkB[i]), self.eps)
            self.assertLessEqual(self.checkB[i].mean(), self.eps)

        for i in range(len(self.checkGamma)):
            self.assertLessEqual(np.max(self.checkGamma[i]), self.eps)
            self.assertLessEqual(self.checkGamma[i].mean(), self.eps)
            self.assertLessEqual(np.max(self.checkBeta[i]), self.eps)
            self.assertLessEqual(self.checkBeta[i].mean(), self.eps)

if __name__=='__main__':
    unittest.main()
```