```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
        self.acc = []
        self.cost = []
        self.loss = []

    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

    def randomize(self):
        """ randomize datapoints """
        pass
```

```python
    def setAccuracy(self, acc):
        self.acc.append(acc)

    def setLoss(self, loss):
        self.loss.append(loss)

    def setCost(self, cost):
        self.cost.append(cost)


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

    def __init__(self, data, trSize, layers, lmbda=0.001, eta=0.003):
        # initialization
        self.K = layers[-1]
        self.lmbda = lmbda
        self.eta = eta
        self.W = []
        self.b = []
        self.h = []
        allData = Dataset(data[0], self.K)
        splitNr = int(trSize*allData.X.shape[1])
        self.train = allData.split(0, splitNr, "training data")
        self.val = allData.split(splitNr, -1, "validation data")
        self.test = Dataset(data[-1], self.K, name="test data")
        if layers[0] is None:
            layers[0] = self.train.X.shape[0]
        self.layStruct = layers
        self.endEpoch = None
        self.setWeightsBiases()
        print(self.train.X.shape)

        # normalize datapoints to training data
        mean = np.array([np.mean(self.train.X, 1)]).T
        std = np.array([np.std(self.train.X, 1)]).T
        self.train.normalize(mean, std)
        self.val.normalize(mean, std)
        self.test.normalize(mean, std)

    def __str__(self):
        toStr = {
            "layers": [len(self.layStruct)-2],
            "lambda": [self.lmbda],
            "eta": [self.eta],
            "training accuracy (max)": [max(self.train.acc)],
            "training loss (min)": [min(self.train.loss)],
            "validation accuracy (max)": [max(self.val.acc)],
            "validation loss (min)": [min(self.val.loss)],
            "test accuracy": [self.test.acc[0]]
        }
        return str(pd.DataFrame(toStr))

    def setWeightsBiases(self, mu=0):
        """ initialize weights and biases """
        # np.random.seed(400)
        self.W.clear()
        self.b.clear()
        for i, l in enumerate(self.layStruct[:-1]):
            self.W.append(np.random.normal(mu, (1/np.sqrt(l)), (self.layStruct[i+1], l)))
            self.b.append(np.zeros((self.layStruct[i+1], 1)))

    def evaluateClassifier(self, X, W, b):
        """
        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)

    self.h.clear()
    self.h.append(X)
    for i in range(len(W) - 1):
        X = relu(np.matmul(W[i], X) + b[i])
        self.h.append(X)
    return softmax(np.matmul(W[-1], X) + b[-1])

def computeCost(self, X, Y, W, b):
    """ computes cost of loss for the network """
    P = self.evaluateClassifier(X, W, b)
    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 """
    gradW = []
    gradB = []
    G = -(Y - P)
    for i in reversed(range(len(self.h))):
        gradW.insert(0, ((1 / bsize) * np.matmul(G, np.array(self.h[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.h[i] > 0, 1, 0)
        G = np.multiply(G, indH > 0)
    return [gradW, gradB]

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]

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

        return [gradW, gradB]

    def gradientCheck(self, gradW_a, gradW_n, gradB_a, gradB_n, eps):
        """ computes the relative error between analytical and numerical gradient calcs """

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

        resW = []
        resB = []
        for i in range(len(gradW_a)):
            resW.append(check(gradW_a[i], gradW_n[i], eps))
            resB.append(check(gradB_a[i], gradB_n[i], eps))
        return resW, resB

    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, bsize, cycEtaData=None, cyclicalEta=False):
        """ bsize'ed batches evaluated """

        if cycEtaData is not None:
            etaMin, etaMax, ns, t, l, cyclicalEta = cycEtaData
            diff = etaMax-etaMin
        for i in range(int(np.size(self.train.X, 1)/bsize)):
            n = i*bsize
            P = self.evaluateClassifier(self.train.X[:, n:n+bsize], self.W, self.b)
            grad = self.computeGradients(P, self.train.Y[:, n:n+bsize], 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, nepochs=40, bsize=100, cyclicalEta=False, numCycles=3, nsAq=500, cycEtaData=None, lmbda=None,
lmbdaSearch=False):
```

```python
        def computeAccLoss(data):
            J, P, L = self.computeCost(data.X, data.Y, self.W, self.b)
            acc = self.computeAccuracy(P, data.y)
            data.setAccuracy(acc)
            data.setLoss(L)
            data.setCost(J)

        def calcK(nsAq, bsize):
            return (nsAq*bsize)/self.train.X.shape[1]

        if cyclicalEta:
            if cycEtaData is None:
                etaMin, etaMax = 10**-5, 10**-1
                cycEtaData = [etaMin, etaMax]
            ns = calcK(nsAq, bsize) * np.floor(self.train.X.shape[1] / bsize)
            cycEtaData.extend([ns, 0, 0, True])
            self.updateEta(cycEtaData[0])
        print("ns", ns)
        if lmbda is not None:
            self.lmbda = lmbda
        for epoch in tqdm(range(nepochs)):
            cycEtaData = self.miniBatch(bsize=bsize, cycEtaData=cycEtaData)
            if not lmbdaSearch:
                computeAccLoss(self.train)
                computeAccLoss(self.val)
            if cyclicalEta and cycEtaData[4] == numCycles:
                break
        computeAccLoss(self.test)
        self.endEpoch = epoch


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 trainSize(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 lambdaSearch(sgd, cycles=2, lMin=0.001, lMax=0.005, 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:
        lmbda = uniform(lMin, lMax)
        _ = sgd.fit(nepochs=20, cyclicalEta=True, numCycles=cycles, nsAq=900, lmbda=lmbda, lmbdaSearch=True)
        _, P, _ = sgd.computeCost(sgd.val.X, sgd.val.Y, sgd.W, sgd.b)
        acc = sgd.computeAccuracy(P, sgd.val.y)
        res[i][0], res[i][1], res[i][2] = lmbda, acc, i+1
        i += 1
        print(i)
        sgd.setWeightsBiases()
        if i == l*narrow:
            res = res[res[:, 1].argsort()[::-1]]
            lMin, lMax = sorted([res[0][0], res[1][0]])
```

```
            lMin -= eps
            lMax += eps
            cycles = 2*cycles
            l += 1
            print("lMin", lMin, "lMax", lMax)
    res = res[res[:, 1].argsort()[::-1]]
    df = pd.DataFrame(data=res, columns=["lambda", "accuracy, validation", "iteration"])
    tfile = open('lambda_search2.txt', 'a')
    tfile.write(df.to_string(index=False))
    tfile.close()
    return res[0][0]


def main():
    # num of classes
    K = 10
    data = trainSize(5)
    # split training/validation, e.g 0.6 -> 60% used for training
    trainValSplit = 0.98
    # num epochs
    nepochs = 200
    # batch size
    bsize = 100
    # aquired ns-value
    nsAq = 1200
    # num of cycles when using cyclic eta
    numCycles = 3
    # initialize vanilla network
    sgd = Network(data, trainValSplit, [None, 50, K])

    # EXERCISE 4 lambda searching
    # lmbda = lambdaSearch(sgd)

    # EXERCISE 2
    # sgd.fit(nepochs=nepochs, bsize=bsize, lmbda=0)

    # EXERCISE 3
    # sgd.fit(nepochs=nepochs, cyclicalEta=True, numCycles=1, nsAq=500, lmbda=0.01)
    # sgd.fit(nepochs=nepochs, cyclicalEta=True, numCycles=3, nsAq=800, lmbda=0.01)

    # EXERCISE 5 training of best network using lambda-value from lambda search
    sgd.fit(nepochs=nepochs, cyclicalEta=True, numCycles=numCycles, nsAq=nsAq, lmbda=0.003038)

    # plot graphs for cost, loss and accuracy of the trained network
    plotGraph(sgd.train.cost, sgd.val.cost, range(sgd.endEpoch+1),
              "Cost", "Epochs", "Training cost", "Validation cost", 4)
    plotGraph(sgd.train.loss, sgd.val.loss, range(sgd.endEpoch+1),
              "Loss", "Epochs", "Training loss", "Validation loss", 3)
    plotGraph(sgd.train.acc, sgd.val.acc, range(sgd.endEpoch+1), "Accuracy",
              "Epochs", "Training accuracy", "Validation accuracy", 1)
    print(sgd)
    plt.show()


if __name__ == "__main__":
    main()
```

```python
import unittest

import numpy as np

from assign2.assignment2 import *


class TestModel(unittest.TestCase):

    def setUp(self):
        dim = 20
        dp = 2
        self.net = Network([['data_batch_1']], 0.5, [dim, 50, 10], 0, 0.001)
        self.eps = 10**-6
        P = self.net.evaluateClassifier(self.net.train.X[:dim, :dp], self.net.W, self.net.b)
        self.gradW_a, self.gradB_a = self.net.computeGradients(P[:dim, :dp], self.net.train.Y[:dim, :dp], dp)
        self.gradW_n, self.gradB_n = self.net.computeGradsNumSlow(
            self.net.train.X[:dim, :dp], self.net.train.Y[:dim, :dp], 10**-5)
        self.checkW, self.checkB = self.net.gradientCheck(
            self.gradW_a, self.gradW_n, self.gradB_a, self.gradB_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)

    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)
```