```python
import pickle

import matplotlib.pyplot as plt
import numpy as np

from functions import *  # functions provided by the course for the lab


# ------ test class for gradient testing (EXTRACT TO SEPARATE FILE!)
import unittest


class TestModel(unittest.TestCase):

    def setUp(self):
        self.train = np.array(separate(loadBatch('data_batch_1'), 10, 10000))
        self.W = np.random.normal(0, 0.01, (10, 3072))
        self.b = np.random.normal(0, 0.01, (10, 1))
        self.lmda = 0
        self.P = evaluateClassifier(self.train[0], self.W, self.b)
        dp = 10
        self.gradW_n, self.gradB_n = computeGradsNumSlow(
            self.train[0][:, :dp], self.train[1][:, :dp], self.W, self.b, self.lmda, 10**-6)
        self.gradW_a, self.gradB_a = computeGradients(
            self.P[:, :dp], self.train[0][:, :dp], self.train[1][:, :dp], self.W, self.lmda, dp)
        self.checkW, self.checkB = gradientCheck(self.gradW_a, self.gradW_n, self.gradB_a, self.gradB_n, 10**-6)

    def test_gradientMean(self):
        self.assertAlmostEqual(self.gradW_n.mean(), self.gradW_a.mean(), places=7)
        self.assertAlmostEqual(self.gradB_n.mean(), self.gradB_a.mean(), places=7)

    def test_relError(self):
        self.assertLessEqual(np.max(self.checkW), 10**-6)
        self.assertLessEqual(self.checkW.mean(), 10**-6)
        self.assertLessEqual(np.max(self.checkB), 10**-6)
        self.assertLessEqual(self.checkB.mean(), 10**-6)

# ------ END TEST CLASS


def plotGraph(lst1, lst2, rangeX, yLabel, xLabel, lst1Label, lst2Label):
    plt.figure()
    plt.plot(rangeX, lst1, label=lst1Label)
    plt.plot(rangeX, lst2, label=lst2Label)
    plt.xlabel(xLabel)
    plt.ylabel(yLabel)
    plt.legend()


def separate(data, K):
    """
    Takes in dataset and seperates
    returns: data X (dim x N), one-hot label matrix Y (KxN), labels(1xN)
    """
    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


def normalize(X, mean, std):
    return (X - mean) / std


def evaluateClassifier(X, W, b):
    """
    Outputs P = softmax(Wx + b) as KxDim-matrix,
    where each column is sums to 1
    """
    return softmax(np.matmul(W, X) + b)


def computeCost(X, Y, W, b, lmda):
    """ computes cost of loss for the network """
    P = evaluateClassifier(X, W, b)
    J = ((1 / np.size(X, 1)) * -np.sum(Y*np.log(P))) + (lmda * np.sum(np.square(W)))
    return J, P


def computeAccuracy(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(P, X, Y, W, lmda, bsize):
    """ Computes gradients using chain rule """
```

```python
    G = -(Y - P)
    grad_W = (1 / bsize) * np.matmul(G, np.array(X).T) + 2*lmda*W
    grad_b = np.array((1 / bsize) * np.matmul(G, np.ones(bsize))).reshape(np.size(W, 0), 1
    return [grad_W, grad_b]


def gradientCheck(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 = check(gradW_a, gradW_n, eps)
    resB = check(gradB_a, gradB_n, eps)
    return resW, resB


def updateParameters(W, b, grad_W, grad_b, eta):
    W = W - eta * grad_W
    b = b - eta * grad_b
    return W, b


def miniBatch(X, Y, y, W, b, lmda, bsize, eta):
    """ bsize'ed batches evaluated """
    for i in range(int(np.size(X, 1)/bsize)):
        n = i*bsize
        P = evaluateClassifier(X[:, n:n+bsize], W, b)
        grad = computeGradients(P, X[:, n:n+bsize], Y[:, n:n+bsize], W, lmda, bsize)
        W, b = updateParameters(W, b, grad[0], grad[1], eta)
    return W, b


def main():
    """ loading of data, initilisation of parameters and main script """

    K = 10  # num of classes

    # load data
    train = loadBatch('data_batch_1')
    validation = loadBatch('data_batch_2')
    test = loadBatch('data_batch_3')

    # separate data
    trainX, trainY, train_y = separate(train, K)
    valX, valY, val_y = separate(validation, K)
    testX, testY, test_y = separate(test, K)

    # pre-process data
    trainXmean = np.array([np.mean(trainX, 1)]).T
    trainXstd = np.array([np.std(trainX, 1)]).T
    trainX = normalize(trainX, trainXmean, trainXstd)
    valX = normalize(valX, trainXmean, trainXstd)
    testX = normalize(testX, trainXmean, trainXstd)

    # initialize parameters
    W_start = np.random.normal(0, 0.01, (K, np.size(trainX, 0)))
    b_start = np.random.normal(0, 0.01, (K, 1))
    lmda = [0, 0, 0.1, 1]
    bsize = 100
    eta = [0.1, 0.001, 0.001, 0.001]
    epochs = 40
    accuracy = []
    loss = []
    weight_layers = []

    for i in range(4):

        W = W_start
        b = b_start

        accEpochsTrain = []
        accEpochsVal = []
        lossTrain = []
        lossVal = []

        # training the network
        for epoch in range(epochs):
            # minibatch returning W_star, b_star
            W, b = miniBatch(trainX, trainY, train_y, W, b, lmda[i], bsize, eta[i])

            # compute training loss and accuracy for each epoch
            J_train, P_train = computeCost(trainX, trainY, W, b, lmda[i])
            accTrain = computeAccuracy(P_train, train_y)
            accEpochsTrain.append(accTrain)
```

```python
            lossTrain.append(J_train)

            # compute validation loss and accuracy for each epoch
            J_val, P_val = computeCost(valX, valY, W, b, lmda[i])
            accVal = computeAccuracy(P_val, val_y)
            accEpochsVal.append(accVal)
            lossVal.append(J_val)

        # compute test loss and accuracy
        J_test, P_test = computeCost(testX, testY, W, b, lmda[i])
        accTest = computeAccuracy(P_test, test_y)

        # collect results
        accuracy.append([accEpochsTrain, accEpochsVal, accTest])
        loss.append([lossTrain, lossVal, J_test])
        weight_layers.append(W)

    # plotting
    for j in range(4):
        print("Test accuracy:", accuracy[j][2], "Test loss:", loss[j][2], "lamda:", lmda[j], "eta:", eta[j])
        plotGraph(accuracy[j][0], accuracy[j][1], range(epochs), "Accuracy",
                  "Epochs", "Training accuracy", "Validation accuracy")
        plotGraph(loss[j][0], loss[j][1], range(epochs), "Loss", "Epochs", "Training loss", "Validation loss")
        montage(weight_layers[j])
    plt.show()


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