In [ ]:

```python
from functions import *
import matplotlib.pyplot as plt
import numpy as np
import pickle
```

In [ ]:

```python
# Functions
def LoadBatch(file): # Excercise 1.1

    with open("data/"+file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')

    pixelDat = dict[b'data']
    labels = dict[b'labels']
    labelsOneHot = np.zeros((len(labels),10))

    for index in range(len(labels)): # Not efficient :)
        labelsOneHot[index][labels[index]] = 1

    return pixelDat, labelsOneHot, labels

def EvaluateClassifier(X, W, b): # Excercise 1.4, no need to transpose x!
    s = W @ X + b # @ is matrix multiplication
    return softmax(s)

def ComputeCost(X, Y, W, b, lamb): # Excercise 1.5
    J = 0
    P = EvaluateClassifier(X, W, b)
    P_t = P.T
    for i in range(len(P_t)):
        J += -np.dot(Y[i], np.log(P_t[i]))
    J /= len(X[0]) # Divide by dimensionality
    loss = J # For documentation
    J += lamb * np.sum(np.power(W,2)) # WTerm

    return J, P, loss

def ComputeAccuracy(X, y, W, b): # Excercise 1.6
    nCorr = 0
    P = EvaluateClassifier(X, W, b)
    for index in range(X.T.shape[0]):
        p = P.T[index]
        predClass = np.argmax(p)
        if predClass == y[index]:
            nCorr += 1

    acc = nCorr/X.T.shape[0]
    return acc

def ComputeGradients(X, Y, P, W, lamb, b_start=0, b_size=None): # Excercise 1.7
    if b_size is None:
        b_size = len(Y)
    # Get random subset of X and Y (not implemented as not required)
    X_batch = X.T[b_start:b_start+b_size].T # Because Python
    Y_batch = Y[b_start:b_start+b_size].T
    P_batch = P.T[b_start:b_start+b_size].T
    G_vec = - (Y_batch-P_batch) # G_vec is nxK
    dldw = np.dot(G_vec, X_batch.T)/b_size # dldw is KxD
    dldb = np.sum(G_vec, axis=1)/b_size # dldb is Kx1
    grad_W = dldw + 2*lamb*W # gradW is KxD
    grad_b = dldb # gradB is Kx1
    return grad_W, grad_b

def init_variables():
    # Excercise 1.2
    X_train, Y_train, X_val, Y_val, X_test, Y_test = None, None, None, None, None, None
    for file in ["data_batch_1", "data_batch_2", "test_batch"]:
        X, Y, y = LoadBatch(file)
        mean_X = np.mean(X, axis=0)
        std_X = np.std(X, axis=0)
        X = X - mean_X
        X = X / std_X
        X = X.T # Make x stored in columns
        if file == "data_batch_1":
            X_train,Y_train, y_train = X, Y, y
        elif file == "data_batch_2":
            X_val,Y_val, y_val = X, Y, y
        else:
            X_test,Y_test, y_test = X, Y, y

    # Excercise 1.3
    np.random.seed(111)
    K = 10 # Number of labels
    d = len(X.T[0]) # dimensionality
    W = np.random.normal(0, 0.01, (K, d)) # Wierd, check here
    b = np.random.normal(0, 0.01, (K,1))

    return X_train, Y_train, y_train, X_val, Y_val, y_val, X_test, Y_test, y_test, W, b
```

In [ ]:

```python
def check_gradients():
    X, Y, y, _, _, _, _, _, _, W, b = init_variables()
    lamb = 0
    P = EvaluateClassifier(X, W, b)
    J,P,_ = ComputeCost(X, Y, W, b, lamb) # P is now nxK for easier handling
    acc = ComputeAccuracy(X, y, W, b)
    c, d = ComputeGradsNumSlow(X.T[0:20].T, Y[0:20], P.T[0:20].T, W, b, lamb, 10**-8)
    a, b = ComputeGradients(X, Y, P, W, lamb,20)
    print("Gradient check:")
    print("W:")
    print("Derived:", a)
    print("Check:", c)
    print("b:")
    print("Derived:", b)
    print("Check:", d)
    maxDiff = 0
    for val in enumerate(np.nditer(a-c)):
        diff = np.abs(val[1])
        if diff > maxDiff:
            maxDiff = diff
    print("Max gradient difference:", maxDiff)
    maxDiff = 0
    for val in enumerate(np.nditer(b-d)):
        diff = np.abs(val[1])
        if diff > maxDiff:
            maxDiff = diff
    print("Max gradient difference:", maxDiff)

if False:
    check_gradients()
```

In [ ]:

```python
def MiniBatchGD(X, Y, y, W, b, lamb, n_epochs, n_batch, eta, X_val, Y_val, y_val): # Excercise 1.8
    acc_hist,cost_hist, loss_hist, acc_hist_val, cost_hist_val, loss_hist_val,loss_hist_val = [] ,[], [], [],
    # Train, initial val
    acc = ComputeAccuracy(X, y, W, b)
    cost, _, loss = ComputeCost(X, Y, W, b, lamb)
    acc_hist.append(acc), cost_hist.append(cost), loss_hist.append(loss)
    # Validation, initial val
    acc = ComputeAccuracy(X_val, y_val, W, b)
    cost, _, loss = ComputeCost(X_val, Y_val, W, b, lamb)
    acc_hist_val.append(acc), cost_hist_val.append(cost), loss_hist_val.append(loss)

    for epoch in range(n_epochs): # Main loop
        for batch in range(int(len(Y)/n_batch)):
            P = EvaluateClassifier(X, W, b)
            grad_W, grad_b = ComputeGradients(X, Y, P, W, lamb, b_start=batch*n_batch, b_size=n_batch)
            W = W - grad_W*eta
            grad_b = grad_b.reshape(b.shape)
            b = b - grad_b*eta
        # Train
        acc = ComputeAccuracy(X, y, W, b)
        cost, _, loss = ComputeCost(X, Y, W, b, lamb)
        acc_hist.append(acc), cost_hist.append(cost), loss_hist.append(loss)
        # Validation
        acc = ComputeAccuracy(X_val, y_val, W, b)
        cost, _, loss = ComputeCost(X_val, Y_val, W, b, lamb)
        acc_hist_val.append(acc), cost_hist_val.append(cost), loss_hist_val.append(loss)
        print("Epoch:", epoch, "Accuracy:", acc_hist[-1])

    return W, b, cost_hist, acc_hist, loss_hist, cost_hist_val, acc_hist_val, loss_hist_val

X_train, Y_train, y_train, X_val, Y_val, y_val, X_test, Y_test, y_test, W, b = init_variables()
lamb = 1
n_epochs = 2
n_batch = 100
eta = 0.001
W, b, cost_hist, acc_hist, loss_hist, cost_hist_val, acc_hist_val, loss_hist_val = MiniBatchGD(X=X_train, Y=Y_t
```

In [ ]:

```python
x = [i for i in range(n_epochs+1)]
plt.clf()
plt.title("Cost graph")
plt.plot(x, cost_hist, label = "Training")
plt.plot(x, cost_hist_val, label = "Valuation")
plt.legend()
plt.show()
plt.clf()
plt.title("Loss graph")
plt.plot(x, loss_hist, label = "Training")
plt.plot(x, loss_hist_val, label = "Valuation")
plt.legend()
plt.show()
plt.clf()
plt.title("Accuracy graph")
plt.plot(x, acc_hist, label = "Training")
plt.plot(x, acc_hist_val, label = "Valuation")
plt.legend()
plt.show()
```

In [ ]:

```python
print("Final test accuracy:", ComputeAccuracy(X_test, y_test, W, b))
```

In [ ]:

```python
montage(W)
```