Deep Learning: Assignment 1

Anton Stråhle

April 6, 2020

Introduction

The functions needed for the base part of this assignment were successfully implemented and the corresponding gradients obtained through `computeGradients` in Appendix 1 did not deviate from those computed numerically. All the functions created for, and used in, the base assignment can be found in Appendix 1 (with the exception of the function `montage` present in functions.py on Canvas as it has not been altered as well as certain functions used to generate figures and plots).

Gradient Calculation

In order to test my `computeGradients` function i used the function `numpy.testing.assert_array_almost_equal` which compares to numpy arrays for a set number of digits. If all cells are equal the function returns `None` and if not it returns the errors in all cells as well as some additional information like the fraction of deviating cells as well as the max error and max relative error. In the case of my analytical gradients and those computed numerically the function returned `None` when using $1e-6$, meaning that the arrays were identical up to 6 decimal places.

Evaluation

In the following section we present both the validation loss and the training loss for the different set of parameters as well as the final accuracy on the test set.

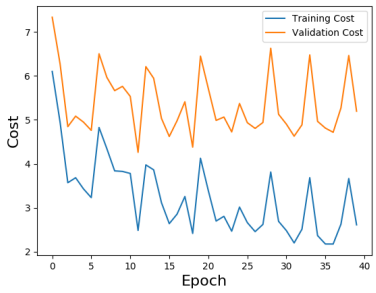

Figure 1: With $\lambda = 0$, $\eta = 0.1$
Final Accuracy: 27.83%

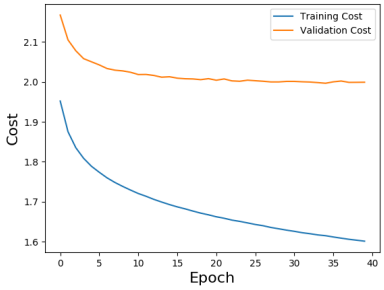

Figure 2: With $\lambda = 0$, $\eta = 0.001$
Final Accuracy: 38.07%

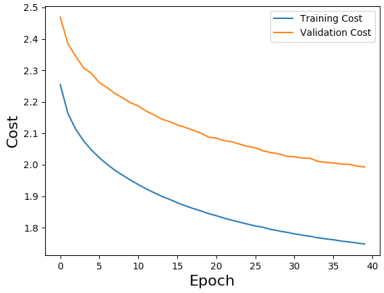

Figure 3: With $\lambda = 0.1$, $\eta = 0.001$
Final Accuracy: 38.34%

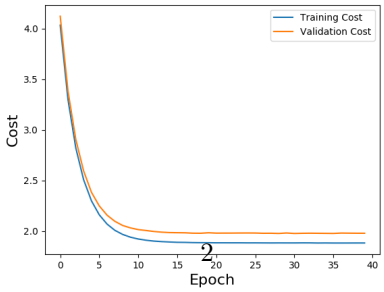

Figure 4: With $\lambda = 1$, $\eta = 0.001$
Final Accuracy: 36.49%

Graphic representation of W

The number of epochs and the sizes of the batches are 40 and 100 respectively in all four cases.

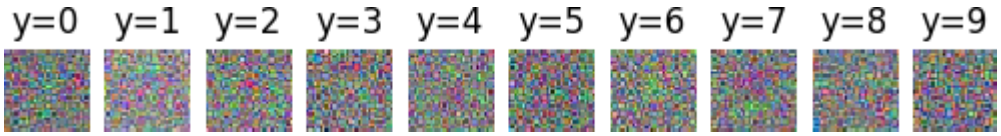

Figure 5: With $\lambda = 0$, $\eta = 0.1$

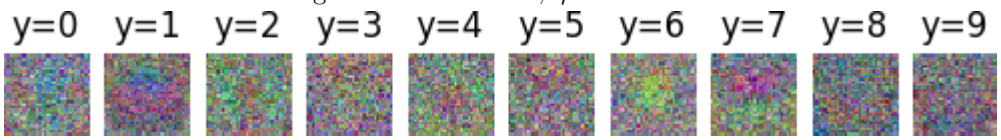

Figure 6: With $\lambda = 0$, $\eta = 0.001$

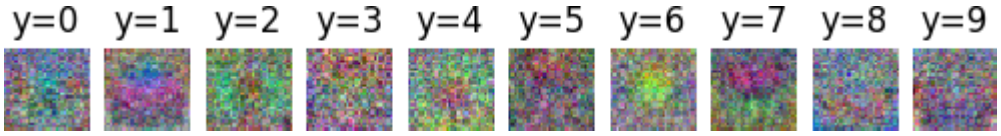

Figure 7: With $\lambda = 0.1$, $\eta = 0.001$

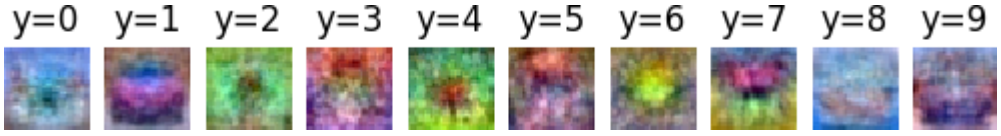

Figure 8: With $\lambda = 1$, $\eta = 0.001$

As we can from Figure 5 through 8 more regularization, i.e. an increase in λ, leads to smoother images. In Figure 8 where $y = 1$ one can for example notice the outline of a car.

Comments

As can be seen in Figure 1 through 4 the increase of the regularization parameter λ increases the difference between the training and validation cost as the model generalizes better. This can also be seen in the visualization of the weight matrices in Figure 5 through 8 as more regularization (i.e. a larger value of λ) leads to a cleared image.

Choosing the correct learning rate is critical as a learning rate which is too high might lead to the overshooting of a minima which leads to a more fluctuating decrease in the cost function (as can be seen in Figure 1) as the gradient jumps back and forth over the minima but never reaches it (very problematic for ravines). On the other hand a learning rate which is too small can require extreme training times in order to achieve adequate classification (which of course is not to be desired). As such we wish to find a learning rate that achieves adequate results in a viable time frame.

Appendix 1

```
#Softmax, taken from functions.py on the Canvas

def softMax(x):
    return np.exp(x) / np.sum(np.exp(x), axis=0)

#Load batches, also partially taken from functions.py

def loadBatch(filename):
    with open('Datasets/'+filename, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')

        y = np.asarray(dict[b'labels'])

        Y = np.zeros((10000,10))
        Y[np.arange(y.size), y] = 1

        X = np.asarray(dict[b'data'])/255

    return X.transpose(),Y.transpose(), y

#Data

xTrain,yTrain,lTrain = loadBatch("data_batch_1")

xVal,yVal,lVal = loadBatch("data_batch_2")

xTest,yTest,lTest = loadBatch("data_batch_3")

#Normalization

def transformData(xOut, xTrain = xTrain):
    return (xOut-np.mean(xTrain, axis = 0))/np.std(xTrain, axis = 0)

#Normalized Data

xTrain = transformData(xTrain)
xVal = transformData(xVal)
xTest = transformData(xTest)

#Initializing parameters

W = np.random.normal(0, 0.01, (10,3072))
b = np.random.normal(0, 0.01, (10,1))
```

```
#Evaluation

def evaluateClassifier(X, W, b):
    s = np.matmul(W,X) + b
    return softMax(s)

#Compute Cost

def computeCost(X, Y, W, b, lmda):

    p = evaluateClassifier(X, W, b)

    return np.mean(-np.log(np.diag(np.matmul(Y.transpose(), p)))) +
        lmda*np.sum(W**2)


#Compute Accuracy

def computeAccuracy(X, y, W, b):
    p = evaluateClassifier(X, W, b)
    opt = np.argmax(p, axis = 0)
    return np.mean(np.equal(opt, y))

#Computing Batch Gradients

def computeGradients(X, Y, W, b, lmda):

    nb = X.shape[1]

    p = evaluateClassifier(X, W, b)
    g = -(Y-p)

    gradW = np.matmul(g, X.transpose())/nb + 2*lmda*W
    gradB = np.matmul(g, np.ones((nb, 1)))/nb

    return gradW, gradB
```

```
#Mini Batch Gradient Descent

def miniBatchGD(X, Y, GDParams, intW, intb, lmda):

    W = intW
    b = intb

    nBatch = GDParams[0]
    eta = GDParams[1]
    nEpochs = GDParams[2]
    N = X.shape[1]

    for i in range(nEpochs):

        p = np.random.permutation(N)

        permX = X[:, p]
        permY = Y[:, p]

        for j in range(N//nBatch):
            jStart = (j-1)*nBatch
            jEnd = j*nBatch - 1
            XBatch = permX[:, jStart:jEnd]
            YBatch = permY[:, jStart:jEnd]

            gradW, gradb = computeGradients(XBatch, YBatch, W, b, lmda)

            W -= eta*gradW
            b -= eta*gradb

    return W, b
```