Deep Learning: Bonus Assignment 1

Anton Stråhle

March 28, 2020

Bonus 1

I have chosen to implement the follwing three improvements to the basic model.

(i) Training for longer whilst using the validation set in order to avoid overfitting

(ii) Shuffling the order of the training example at the beginning of each epoch

(iii) Training several networks with different initializations and combining aggregating their classifications through majority vote.

(i)

In order to do this we calculate the accuracy on the validation set after each epoch whilst storing the parameters that generate the highest accuracy on the validation set. By doing this the eventual overfitting by increasing the number of epochs will be avoided as we will simply disregard the changes made to the parameters once the network starts to generalize poorly.

(ii)

We simply permute the data before each epoch in order to get different batches during each epoch.

(iii)

I chose to somewhat lazily reimplement the mini batch gradient descent in an object oriented manner in order to be able to manage multiple different classifiers. The classifications are then aggregated through a majority vote in order to achieve a final classification.

The object oritented approach which differs slightly can be found in Appendix 1 along with the notes (i), (ii) and (iii) to signal the implementations of the three improvements.

Improvements in accuracy

When comparing the changes to the accuracy for the different improvements we use a batch size of 100, 40 epochs, $\eta = 0.001$ and a $\lambda = 0.1$. When including (ii) we use 100 epochs instead and for (iii) we use 5 separate networks. We note that all improvements applied seems to affect the final accuracy quite minimally, whilst generally slightly positive. Although it is expected the usage of ensembling produces a significantly more stable accuracy as the standard deviation is a lot smaller.

Table 1: Errors

Improvements	Error
	0.3773 ± 0.0011
(i)	0.3754 ± 0.0022
(ii)	0.3751 ± 0.0024
(iii)	0.3888 ± 0.0012
(i) & (ii)	0.388 ± 0.002
(i) & (iii)	0.3884 ± 0.0008
(ii) & (iii)	0.3864 ± 0.0026
(i), (ii) & (iii)	0.3901 ± 0.0005

Bonus 2

In this bonus assignment we abandon the cross-entropy loss in favor of the so called hinge loss, or multiclass SVM loss, which is formulated as follows. The relevant functions can be found in Appendix 2.

We once again run through the same sets of parameters are in the base assignment whilst still letting the number of epochs equal 40 and the batch size equal 100. This leaves us with the following training and validation costs over the epochs.

$$l_h = \frac{1}{\text{sample size}} \sum_i \sum_{j \neq y_i} \max(0, \mathbf{W}'_j \mathbf{X}_i - \mathbf{W}'_{y_i} \mathbf{X}_i + \Delta) + \lambda \sum_{i,j} W_{i,j}^2$$

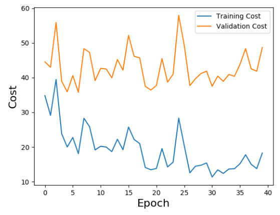

Figure 1: With $\lambda = 0$, $\eta = 0.001$
Final Accuracy: 26.15%

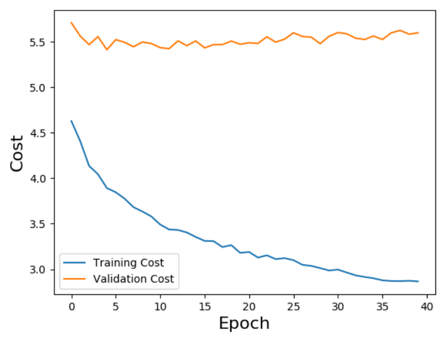

Figure 2: With $\lambda = 0$, $\eta = 0.001$
Final Accuracy: 32.06%

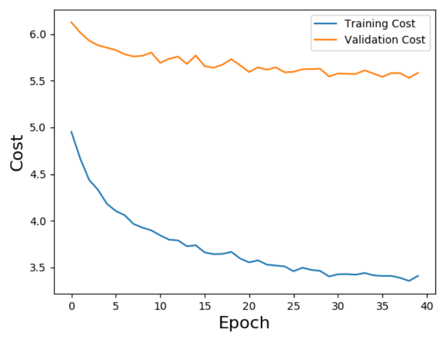

Figure 3: With $\lambda = 0.1$, $\eta = 0.001$
Final Accuracy: 34.01%

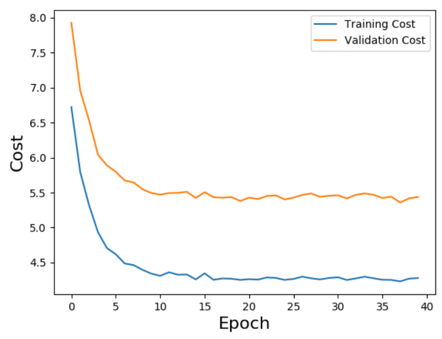

Figure 4: With $\lambda = 1$, $\eta = 0.001$
Final Accuracy: 34.64%

We can also visualize the final weight matrices as was done in the base assignments.

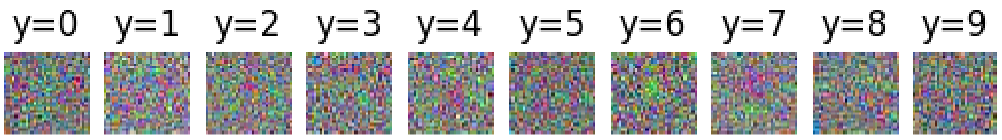

Figure 5: With $\lambda = 0$, $\eta = 0.1$

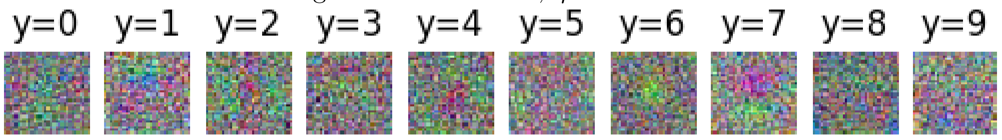

Figure 6: With $\lambda = 0$, $\eta = 0.001$

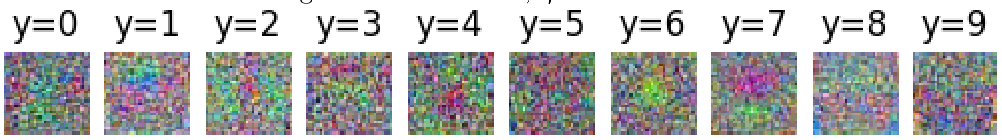

Figure 7: With $\lambda = 0.1$, $\eta = 0.001$

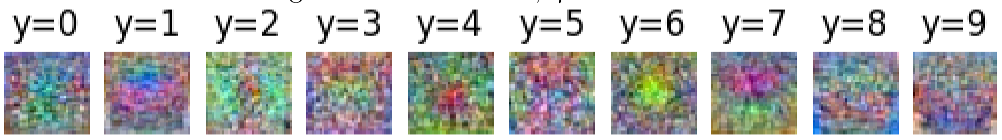

Figure 8: With $\lambda = 1$, $\eta = 0.001$

Comments

When comparing the errors obtained by using hinge loss for a variety of different parameters (presented under Figure 1 through 4) with those in the base assignment, for a set of parameters also presented in Table 1 above, we note that it seems to fall short compared to the cross-entropy loss. This could of course be due to a variety of reasons, for example suboptimal initialization or parameter selection. In conclussion however is seems that for the sets of parameters observed and the initialization used, cross-entropy loss seems to generate a higher accuracy (and also clearer visual representations of the classes as can be seen when comparing the visualizations in the base assignment and in this bonus assignment).

It could be of interest to attempt the application of the above improvements in bonus assignment 1 to the network using hinge loss (or any of the improvements mentioned in the assignment) in order to observe if the hinge loss is more sensitive to these adjustments as they had very minor effects on the network using cross-entropy loss.

Appendix 1

```
class mbGD(object):

    def __init__(self, intW, intB, xTrain = xTrain, yTrain = yTrain,
         xVal = xVal, lVal = lVal):
        self.W = intW
        self.b = intB
        self.xt = xTrain
        self.yt = yTrain
        self.xv = xVal
        self.lv = lVal

    def train(self, GDParams, lmda):

        bestW = self.W
        bestb = self.b

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

        bestAcc = computeAccuracy(self.xv, self.lv, bestW, bestb)

        for i in range(nEpochs):

            p = np.random.permutation(N) (i)

            permX = self.xt[:, p]
            permY = self.yt[:, 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, self.W,
                    self.b, lmda)

                self.W -= eta*gradW
                self.b -= eta*gradb
            acc = computeAccuracy(self.xv, self.lv, bestW, bestb) (ii)

            if acc > bestAcc:

                bestW = self.W
                bestb = self.b

        self.W = bestW
        self.b = bestb

    def classify(self, X):

        p = evaluateClassifier(X, self.W, self.b)

        return np.argmax(p, axis = 0)

#Ensemble Classifier

class ensembleMBGD(object):  (iii)

    def __init__(self, nNetworks):
        self.n = nNetworks
        self.classifiers = []

    def initialize(self, GDParams, lmda):

        for i in range(self.n):
            """ Other options? """
            W = np.random.normal(0, 0.01, (10,3072))
            b = np.random.normal(0, 0.01, (10,1))

            new = mbGD(W, b)
            new.train(GDParams, lmda)

            self.classifiers.append(new)

    def classify(self, X):

        ensembleOut = np.zeros((self.n, X.shape[1]))

        for i in range(self.n):
            out = self.classifiers[i].classify(X)
            ensembleOut[i,] = out

        return sc.mode(ensembleOut, axis = 0)[0]
```

Appendix 2

```python
def computeSVMLoss(X, Y, W, b, lmda):
    s = np.matmul(W,X) + b
    y = np.argmax(Y, axis = 0)

    l = 0

    for i in range(X.shape[1]):
        for j in range(Y.shape[0]):
            if j != y[i]:
                l += np.maximum(s[j,i]-s[y[i],i] + 1, 0)

    return l/X.shape[1] + lmda*np.sum(W**2)


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

    gradW = np.zeros(W.shape)
    gradb = np.zeros(b.shape)

    s = np.matmul(W,X) + b
    y = np.argmax(Y, axis = 0)

    for i in range(X.shape[1]):
        for j in range(Y.shape[0]):
            if j != y[i]:
                if s[j,i]-s[y[i],i] > -1:

                    gradW[j] += X[:,i]
                    gradb[j] += 1
                    gradW[y[i]] -= X[:,i]
                    gradb[y[i]] -= 1


    gradW = gradW/X.shape[1] + 2*lmda*W
    gradb /= X.shape[1]


    return gradW, gradb
```

```
def miniBatchSVMGD(X, Y, GDParams, intW, intb, lmda, xVal, yVal):

    W = intW
    b = intb

    trainingCost = []
    validationCost = []

    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 = computeSVMGrads(XBatch, YBatch, W, b, lmda)

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

        tc = computeSVMLoss(X, Y, W, b, lmda)
        vc = computeSVMLoss(xVal, yVal, W, b, lmda)

        trainingCost.append(tc)
        validationCost.append(vc)

    return W, b, trainingCost, validationCost
```