Deep Learning: Assignment 3

2022-05-06

Lucy McCarren ▌ mccarren@kth.se

1. Introduction

In this assignment I constructed, trained and tested k-layer networks with multiple outputs to classify images from CIFAR-10. The networks were trained and outputs observed both with and without batch normalization of the k-layer network.

Gradient checking

In order to check that the analytic gradient computations were correct, I used the given function `computeGradientsNum`, and compared my gradients using the numpy function `numpy.assert_almost_equal` to 6 decimal accuracy. When computing the gradients for the 9-layer network without batch normalization the gradients started to disappear meaning that we cannot learn, and therefore need batch normalization.

2. Evolution of the loss function for a 3-layer network

Without batch normalization

Figures 1-3: Cost, loss and accuracy curves without batch normalization The hyper-parameters are `eta_min = 1e-5, eta_max = 1e-1, alpha = 0.8, lambda = 0.005, sig = 1e-1`. Accuracy of **48%** was achieved.

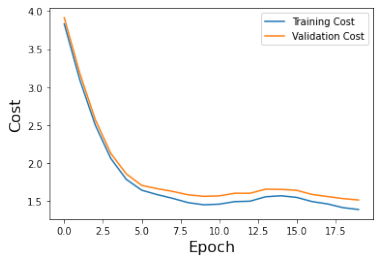


Figure 1: Cost

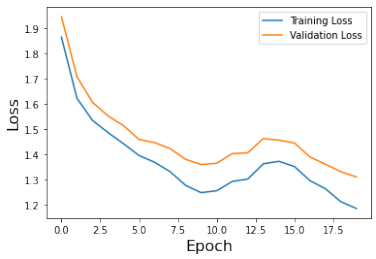


Figure 2: Loss

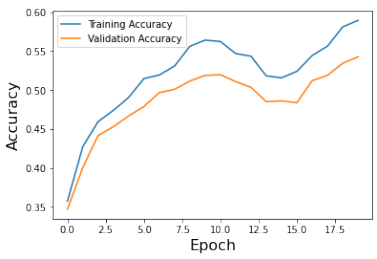


Figure 3: Accuracy

With batch normalization

Figures 4-6: Cost, loss and accuracy curves with batch normalization The hyper-parameters are `eta_min = 1e-5, eta_max = 1e-1, alpha = 0.8, lambda = 0.005, sig = 1e-1`. Ac-

curacy of **52%** was achieved.

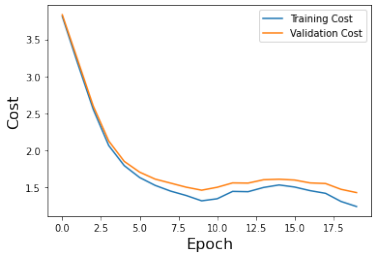


Figure 4: Cost

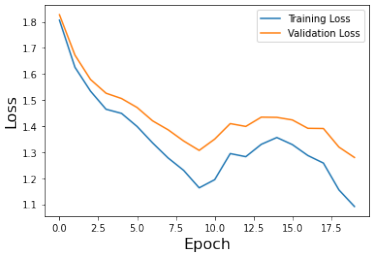


Figure 5: Loss

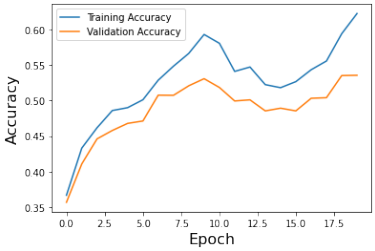


Figure 6: Accuracy

3. Evolution of the loss function for a 9-layer network

Without batch normalization

Figures 7-9: Cost, loss and accuracy curves without batch normalization The hyper-parameters are `eta_min = 1e-5, eta_max = 1e-1, alpha = 0.8, lambda = 0.005, sig = 1e-1`. Accuracy of **48.6%** was achieved.

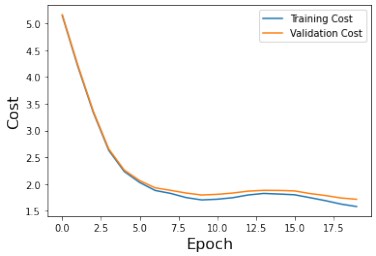


Figure 7: Cost

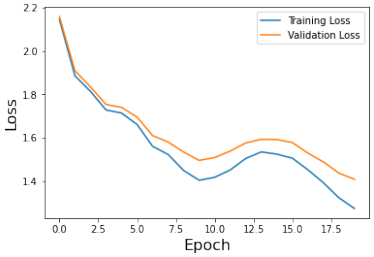


Figure 8: Loss

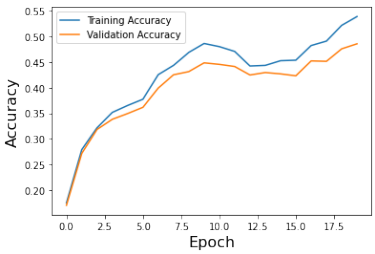


Figure 9: Accuracy

With batch normalization

Figures 10-12: Cost, loss and accuracy curves with batch normalization The hyper-parameters are `eta_min = 1e-5, eta_max = 1e-1, alpha = 0.8, lambda = 0.005, sig = 1e-1`. Accuracy of **53.6%** was achieved.

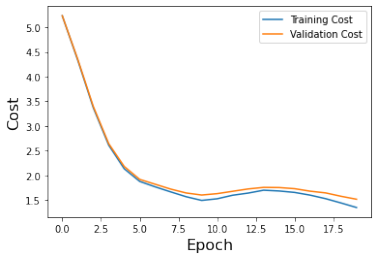


Figure 10: Cost

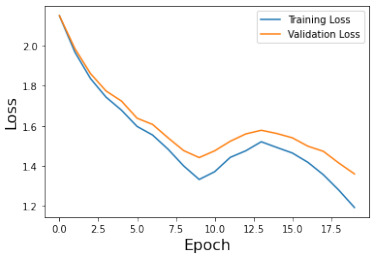


Figure 11: Loss

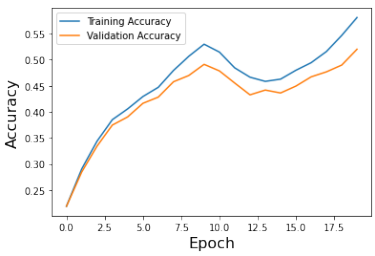


Figure 12: Accuracy

4. Search for lambda

The next task of the assignment was to perform a coarse-to-fine search to optimize the regularization term `lambda`. To do this I searched 10 random values in the range $U \sim Uniform(-4,-1)$ where $\lambda = 10^U$.

During this search, the highest validation accuracy was with $\lambda = 0.002$ or $\lambda = 0.007$

λ	Validation Accuracy
0.00013	51.8%
0.00017	52.02%
0.00053	53.2%
0.00267	54.6%
0.00234	**55.1%**
0.00268	**55.5%**
0.00323	54.6%
0.00649	**55.0%**
0.00813	**55.0%**
0.01210	54.2%
0.02814	52.7%
0.04962	51.7%
0.04965	50.78%
0.08485	49.36%

Training the network with batch regularization using $\lambda = 0.002$ resulted in accuracy of 53.1% on the 3-layer network and 50.1% on the 9-layer network.

Training the network with batch regularization using $\lambda = 0.007$ resulted in accuracy of 54% on the 3-layer network and 51.4% on the 9-layer network.

5. Sensitivity to initialization

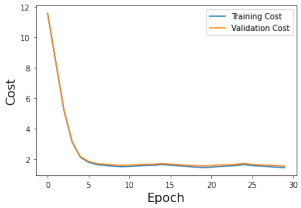
The final part of the assignment was to analyse the effect that batch normalization has on the networks sensitivity to regularization parameters. Specifically, instead of using He-initialization, each weight parameter was initialized to be normally distributed with sigmas equal to the same value sig at each layer for $[\sigma = 1e-1, 1e-3, 1e-4]$. The network was trained both with and without batch normalization to see the effect on the final test accuracy.

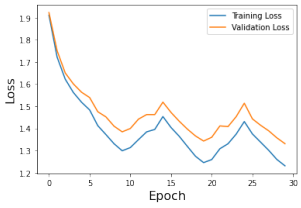
3-layer network

The 3-layer network was trained using $[\sigma = 1e-1, 1e-3, 1e-4]$ respectively instead of He-initialization both with and without batch normalization.

Firstly lets look at the results **without batch normalization**. We see that the network trains effectively with $[\sigma = 1e-1]$ but not for other values of sigma.

Figures 13-15: $[\sigma = 1e-1]$, **Final Accuracy: 53%**





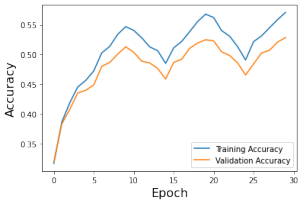
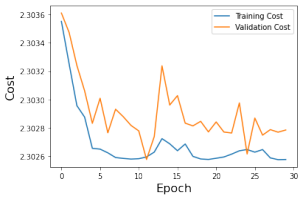
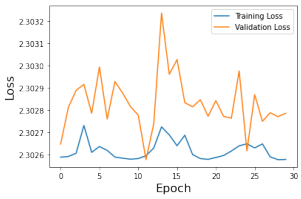


Figure 13: Cost $[\sigma = 1e-1]$ Figure 14: Loss $[\sigma = 1e-1]$ Figure 15: Acc $[\sigma = 1e-1]$

Figures 16-18: $[\sigma = 1e-3]$, **Final Accuracy: 10%**





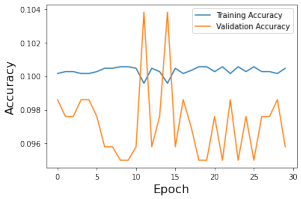
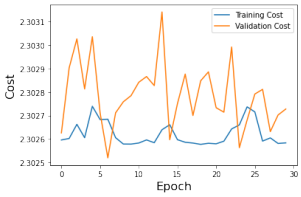
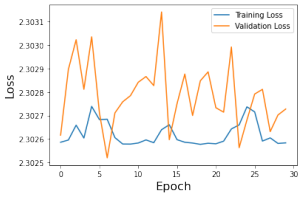


Figure 16: Cost$[\sigma = 1e-3]$ Figure 17: Loss$[\sigma = 1e-3]$ Figure 18: Acc$[\sigma = 1e-3]$

Figures 19-21: $[\sigma = 1e-4]$, **Final Accuracy: 10%**





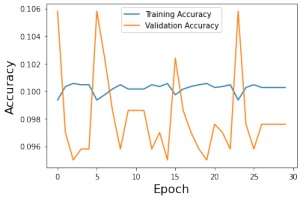
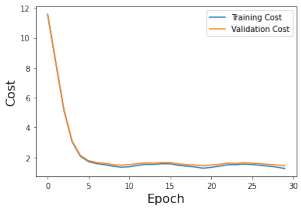
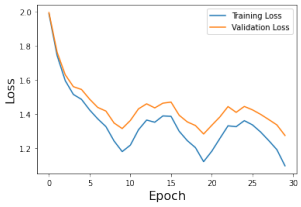


Figure 19: Cost$[\sigma = 1e-4]$ Figure 20: Loss$[\sigma = 1e-4]$ Figure 21: Acc$[\sigma = 1e-4]$

Next lets look at the results on the 3-layer network **with batch normalization**.

Figures 22-24: $[\sigma = 1e-1]$, **Final Accuracy: 53.8%**





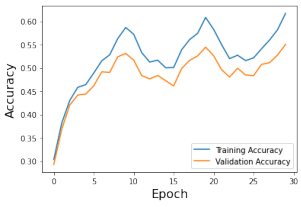
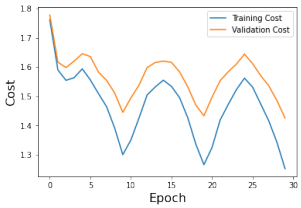
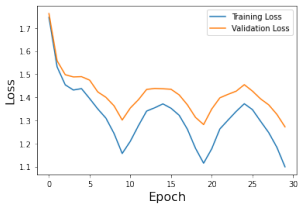


Figure 22: Cost $[\sigma = 1e-1]$ Figure 23: Loss $[\sigma = 1e-1]$ Figure 24: Acc $[\sigma = 1e-1]$

Figures 25-27: $[\sigma = 1e-3]$, **Final Accuracy: 53.6%**





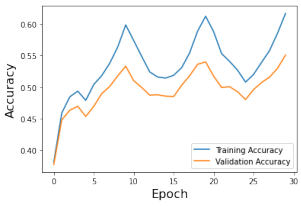
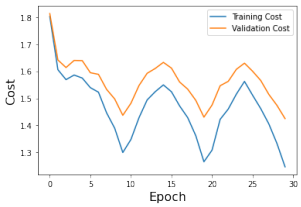
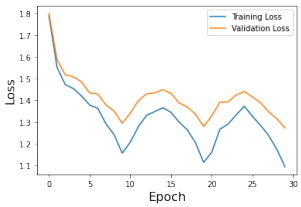


Figure 25: Cost$[\sigma = 1e-3]$ Figure 26: Loss$[\sigma = 1e-3]$ Figure 27: Acc$[\sigma = 1e-3]$

Figures 28-30: $[\sigma = 1e-4]$, **Final Accuracy: 53.8%**





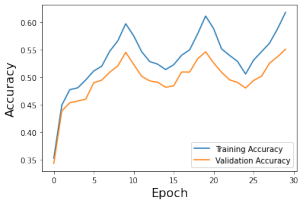


Figure 28: Cost$[\sigma = 1e-4]$ Figure 29: Loss$[\sigma = 1e-4]$ Figure 30: Acc$[\sigma = 1e-4]$

9-layer network

One would assume to see similar results on a deeper network, that lack of batch normalization makes the network more sensitive to initialization parameters. Let's look at a 9-layer network to verify this, using $[\sigma = 1e-1, 1e-3, 1e-4]$ respectively and both with and without batch normalization.

Firstly lets look at the results **without batch normalization**.

Figures 31-33: $[\sigma = 1e-1]$, **Final Accuracy: 10%**

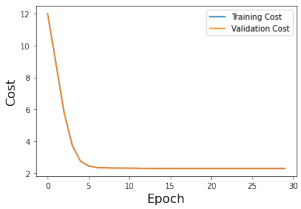


Figure 31: Cost $[\sigma = 1e-1]$

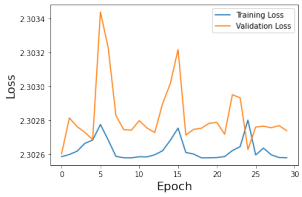


Figure 32: Loss $[\sigma = 1e-1]$

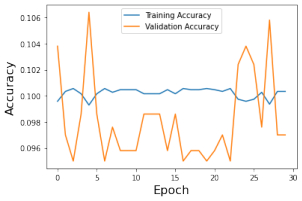


Figure 33: Acc $[\sigma = 1e-1]$

Figures 34-36: $[\sigma = 1e-3]$, **Final Accuracy: 10%**

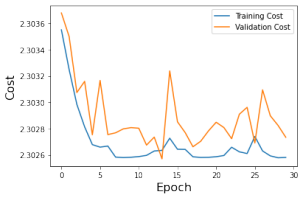


Figure 34: Cost$[\sigma = 1e-3]$

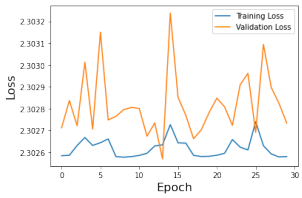


Figure 35: Loss$[\sigma = 1e-3]$

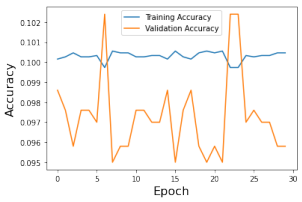


Figure 36: Acc$[\sigma = 1e-3]$

Figures 37-39: $[\sigma = 1e-4]$, **Final Accuracy: 10%**

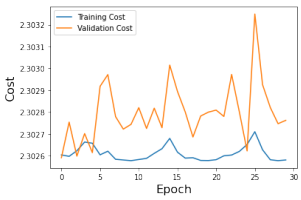


Figure 37: Cost$[\sigma = 1e-4]$

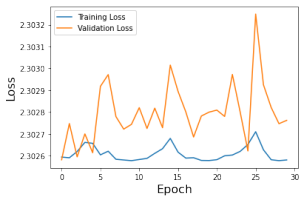


Figure 38: Loss$[\sigma = 1e-4]$

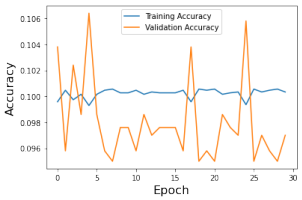


Figure 39: Acc$[\sigma = 1e-4]$

Next lets look at the results on the 9-layer network **with batch normalization**.

Figures 40-42: $[\sigma = 1e-1]$, **Final Accuracy: 53%**

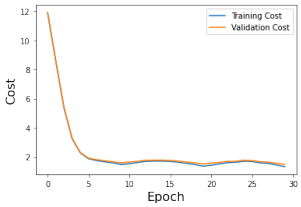


Figure 40: Cost$[\sigma = 1e-1]$

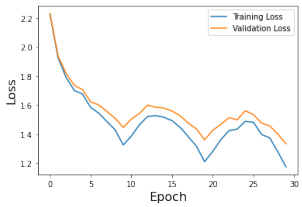


Figure 41: Loss$[\sigma = 1e-1]$

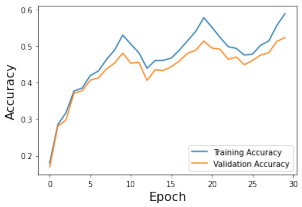


Figure 42: Acc$[\sigma = 1e-1]$

Figures 43-45: $[\sigma = 1e-3]$, **Final Accuracy: 51%**

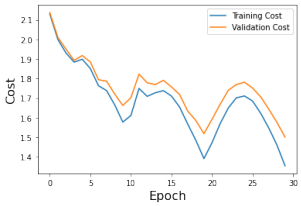


Figure 43: Cost$[\sigma = 1e-3]$

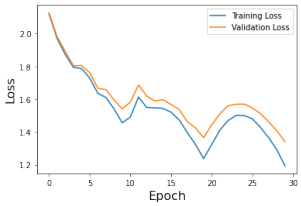


Figure 44: Loss$[\sigma = 1e-3]$

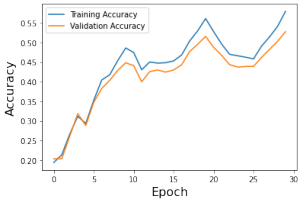


Figure 45: Acc$[\sigma = 1e-3]$

Figures 46-48: $[\sigma = 1e-4]$, **Final Accuracy: 50.5%**

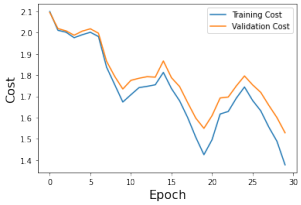


Figure 46: Cost$[\sigma = 1e-4]$

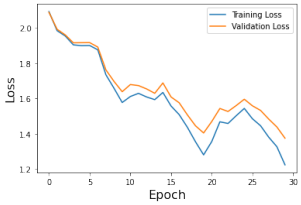


Figure 47: Loss$[\sigma = 1e-4]$

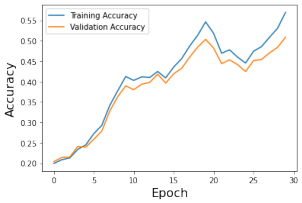


Figure 48: Acc$[\sigma = 1e-4]$

Conclusion

To conclude, we can see from these experiments that batch normalization decreases the dependency on good initialization. This means that the network can still learn efficiently even though it was initialized poorly, while a network without batch normalization will not.

Code Appendix

```
#Initializing parameters
class BNN(object):

    def __init__(self, layers, lmda = 0, alpha = 0.8, batchNorm = True, he = True, sig = 1e
        self.layers = layers
        self.nl = len(layers)   1
        self.W, self.b = [], []
        self.batchNorm = batchNorm
        self.he = he
        self.sig = sig
        self.lmda = lmda

        if batchNorm:
            self.gamma, self.beta, self.muMA, self.vaMA = [],[],[],[]
            self.alpha = alpha
        for layer in self.layers:
            if batchNorm:
                W, b, gamma, beta, mu, v = initializeParam(layer, he, batchNorm, sig)
                self.W.append(W)
                self.b.append(b)
                self.gamma.append(gamma)
                self.beta.append(beta)
                self.muMA.append(mu)
                self.vaMA.append(v)
            else:
                W, b = initializeParam(layer, he, batchNorm, sig)
                self.W.append(W)
                self.b.append(b)

    def zeros(self):
        wl, bl = [], []
        for i in range(len(self.W)):
            W = np.zeros_like(self.W[i])
            b = np.zeros_like(self.b[i])
            wl.append(W)
            bl.append(b)
        if self.batchNorm:
            gl, betal = [], []
            for i in range(len(self.gamma)):
                gamma =  np.zeros_like(self.gamma[i])
                beta =  np.zeros_like(self.beta[i])
                gl.append(gamma)
                betal.append(beta)
```

```
        return wl, bl, gl, betal
    else:
        return wl, bl

def evaluateClassifier(self, X, training = True):
    nb = X.shape[1]
    s = np.copy(X)
    if self.batchNorm:
        sl, shl, hl, mul, vl = [], [], [], [], []
        for i in range(len(self.W)):
            hl.append(s)
            s = np.matmul(self.W[i],s) + self.b[i]
            if i < self.nl:
                sl.append(s)
                if training: #Training
                    mu = np.mean(s, axis = 1, keepdims = True)
                    v = np.var(s, axis = 1, keepdims = True)*(nb 1)/nb
                    mul.append(mu)
                    vl.append(v)
                    s = (s   mu)/np.sqrt(v + np.finfo(np.float).eps)
                    self.muMA[i] = self.alpha*self.muMA[i] + (1 self.alpha)*mu
                    self.vaMA[i] = self.alpha*self.vaMA[i] + (1 self.alpha)*v
                else: #Testing
                    s = (s   self.muMA[i])/np.sqrt(self.vaMA[i] + np.finfo(np.float).ep
                shl.append(s)
                s = relu(np.multiply(s,self.gamma[i]) + self.beta[i])
            else:
                p = softMax(s)
        return p, hl, sl, shl, mul, vl
    else:
        hl = [s]
        for i in range(len(self.W) 1):
            s = relu(np.matmul(self.W[i], s) + self.b[i])
            hl.append(s)
        p = softMax(np.matmul(self.W[ 1] , s)+ self.b[ 1])
        return p, hl

def computeGradients(self, X, Y):
    nb = X.shape[1]
    if self.batchNorm:
        gradW, gradB, gradG, gradBeta = self.zeros()
        p, h, s, sh, mu, v = self.evaluateClassifier(X, training = True)
        g =  (Y p)
        gradW[self.nl] = np.matmul(g,h[self.nl].transpose())/nb + 2*self.lmda*self.W[s
        gradB[self.nl] = np.matmul(g,np.ones((nb, 1)))/nb
```

```
            g = np.matmul(self.W[self.nl].transpose(), g)
            g = np.multiply(g, h[self.nl] > 0)
            for i in range(self.nl   1,  1,  1):
                gradG[i] = np.matmul(np.multiply(g, sh[i]),np.ones((nb,1)))/nb
                gradBeta[i] = np.matmul(g,np.ones((nb,1)))/nb
                g = np.multiply(g, self.gamma[i])
                g = self.batchNormBackPass(g, s[i], mu[i], v[i])
                gradW[i] = np.matmul(g,h[i].transpose())/nb + 2*self.lmda*self.W[i]
                gradB[i] = np.matmul(g,np.ones((nb, 1)))/nb
                if i > 0:
                    g = np.matmul(self.W[i].transpose(),g)
                    g = np.multiply(g, h[i] > 0)
            return gradW, gradB, gradG, gradBeta
        else:
            gradW, gradB = self.zeros()
            p, h = self.evaluateClassifier(X)
            g =  (Y p)
            for i in range(self.nl,  1,  1):
                gradW[i] = np.matmul(g, h[i].transpose())/nb + 2*self.lmda*self.W[i]
                gradB[i] = np.matmul(g, np.ones((nb,1)))/nb
                g = np.matmul(self.W[i].transpose(), g)
                g = np.multiply(g, h[i] > 0)
            return gradW, gradB

    def batchNormBackPass(self, g, s, mu, v, eps = np.finfo(float).eps):
        N = g.shape[1]
        sigma1 = np.power(v + eps,  0.5)
        sigma2 = np.power(v + eps,  1.5)
        g1 = np.multiply(g, sigma1)
        g2 = np.multiply(g, sigma2)
        d = s mu
        c = np.sum(np.multiply(g2, d), axis = 1, keepdims = True)
        gRet = g1   np.sum(g1, axis = 1, keepdims = True)/N   np.multiply(d,c)/N
        return gRet

    def computeCost(self, X, Y):
        N = X.shape[1]
        if self.batchNorm:
            p,_,_,_,_,_ = self.evaluateClassifier(X)
        else:
            p,_ = self.evaluateClassifier(X)
        lab = np.argmax(Y, axis = 0)
        l = 0
        r = 0
        for i in range(N):
```

```
            l  = np.log(p[lab[i], i])
        for i in range(len(self.W)):
            r += np.sum(np.square(self.W[i]))
        j = l/N + self.lmda*r
        return j, l/N

#Compute Accuracynp.matmul(W[i], hl[i-1]) + b[i]
    def computeAccuracy(self, X, y):
        if self.batchNorm:
            p,_,_,_,_,_ = self.evaluateClassifier(X, training = False)
        else:
            p,_ = self.evaluateClassifier(X, training = False)
        opt = np.argmax(p, axis = 0)
        return np.mean(np.equal(opt, y))

#Numerical Gradients
    def computeGradientsNum(self, X, Y, h = 1e-6):
        if self.batchNorm:
            gradW, gradB, gradG, gradBeta = self.zeros()
            #b
            print("b")
            for j in range(len(gradB)):
                for i in range(gradB[j].shape[0]):
                    old = self.b
                    self.b[j][i]  = h
                    c1 = self.computeCost(X, Y)
                    self.b = old
                    old = self.b
                    self.b[j][i] += h
                    c2 = self.computeCost(X, Y)
                    self.b = old
                    gradB[j][i] = (c2 - c1)/(2*h)
            #W
            print("W")
            for j in range(len(gradW)):
                for i in range(len(gradW[j].flatten())):
                        old = self.W[j].flat[i]
                        self.W[j].flat[i]  = h
                        c1 = self.computeCost(X, Y)
                        self.W[j].flat[i] = old
                        self.W[j].flat[i] += h
                        c2 = self.computeCost(X, Y)
                        self.W[j].flat[i] = old
                        gradW[j].flat[i] = (c2 - c1)/(2*h)
            #Gamma
```

```
        print("Gamma")
        for j in range(len(gradG)):
            for i in range(gradG[j].shape[0]):
                old = self.gamma
                self.gamma[j][i]  = h
                c1 = self.computeCost(X, Y)
                self.gamma = old
                self.gamma[j][i] += h
                c2 = self.computeCost(X, Y)
                self.gamma = old
                gradG[j][i] = (c2 c1)/(2*h)
        #Beta
        print("Beta")
        for j in range(len(gradBeta)):
            for i in range(gradBeta[j].shape[0]):
                old = self.beta
                self.beta[j][i]  = h
                c1 = self.computeCost(X, Y)
                self.beta = old
                self.beta[j][i] += h
                c2 = self.computeCost(X, Y)
                self.beta = old
                gradBeta[j][i] = (c2 c1)/(2*h)
        return gradW, gradB, gradG, gradBeta
    else:
        gradW, gradB = self.zeros()

        #b
        print("b")
        for j in range(len(gradB)):
            for i in range(gradB[j].shape[0]):
                old = self.b
                self.b[j][i]  = h
                c1 = self.computeCost(X, Y)
                self.b = old
                self.b[j][i] += h
                c2 = self.computeCost(X, Y)
                self.b = old
                gradB[j][i] = (c2 c1)/(2*h)

        #W
        print("W")
        for j in range(len(gradW)):
            for i in range(len(gradW[j].flatten())):
                    old = self.W[j].flat[i]
```

```
                    self.W[j].flat[i]  = h
                    c1 = self.computeCost(X, Y)
                    self.W[j].flat[i] = old
                    self.W[j].flat[i] += h
                    c2 = self.computeCost(X, Y)
                    self.W[j].flat[i] = old
                    gradW[j].flat[i] = (c2 c1)/(2*h)
        return gradW, gradB

def miniBatchGD(self, X, Y, GDParams, xVal, yVal):
    trainingl = np.argmax(Y, axis = 0)
    validationl = np.argmax(yVal, axis = 0)
    nBatch = GDParams[0]
    etaMin = GDParams[1]
    etaMax = GDParams[2]
    nEpochs = GDParams[3]
    nS = GDParams[4]
    N = X.shape[1]
    eta = etaMin
    tll, vll, tcl, vcl, tacl, vacl = [], [], [], [], [], []
    t = 0
    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]
            if self.batchNorm:
                gradW, gradB, gradG, gradBeta = self.computeGradients(XBatch, YBatch)
                for i in range(len(self.gamma)):
                    self.gamma[i]  = eta*gradG[i]
                    self.beta[i]  = eta*gradBeta[i]
            else:
                gradW, gradB = self.computeGradients(XBatch, YBatch)
            for i in range(len(self.W)):
                self.W[i]  = eta*gradW[i]
                self.b[i]  = eta*gradB[i]
            if t <= nS:
                t +=1
                eta = etaMin + t * (etaMax   etaMin)/nS
            elif t < 2*nS:
                t +=1
```

```
                eta = etaMax   (t nS) * (etaMax   etaMin)/nS
            else:
                eta = etaMin
                t = 0
        tc, tl = self.computeCost(X, Y)
        vc, vl = self.computeCost(xVal, yVal)
        tcl.append(tc)
        vcl.append(vc)
        tll.append(tl)
        vll.append(vl)
        tacl.append(self.computeAccuracy(X, trainingl))
        vacl.append(self.computeAccuracy(xVal, validationl))
    return tcl, vcl, tll, vll, tacl, vacl
```