Felicia Rosell
28 June 2017

Report Lab 3

Search Engines DD2424

In this assignment, mini-batch gradient descent was used to classify images from CIFAR-10 into 10 classes. The network had three layers and L2 regularization, momentum and moving averages were used. The training set was of size 10 000 and had 3072 features. The hidden layers of the network had 50 and 20 nodes.

Gradients

The gradients of W and b were calculated in the backward pass. These gradients were checked for correctness against gradients computed with the central difference formula.
If the result of the following two equations held, gradients were considered as being correct.

$$|\max(g_a - g_n)| < 10^{-6} \quad (1)$$

$$\frac{|g_a - g_n|}{\max(\varepsilon, |g_a| + |g_n|)} < 10^{-4} \quad (2)$$

Where g_a is the analytically computed gradient, g_n is the numerically computed gradient and $\varepsilon = 0.001$. In the initialization, the standard deviation for W was increased to 0.1 to avoid numerical precision issues.

Table 1. Resulting difference between analytical and numerical gradients, using only 10 input samples with 100 dimensions and 2 layers (50 hidden nodes.)

	Equation 1, Max difference	Equation 2, Relative error
W1	7.8949e-11	3.5374e-11
W2	6.4048e-11	3.7979e-07
b1	6.1062e-17	1.4212e-13
b2	4.5816e-11	7.3634e-08

Table 2. Resulting difference between analytical and numerical gradients, using only 10 input samples with 100 dimensions and 3 layers (50 and 30 hidden nodes respectively).

	Equation 1, Max difference	Equation 2, Relative error
W1	8.2539e-11	3.5721e-10
W2	6 6.5115e-11	2.8481e-09
W3	5.9699e-11	2.7936e-07
b1	4.0246e-17	9.9937e-14
b2	2.3592e-17	5.4542e-14
b3	4.1835e-11	6.3697e-08

Table 3. Resulting difference between analytical and numerical gradients, using only 10 input samples with 100 dimensions and 4 layers (50, 30 and 30 hidden nodes respectively).

	Equation 1, Max difference	Equation 2, Relative error
W1	1.324486e-09	2.845198e-10
W2	2.991323e-10	1.454697e-06

W3	1.460155e-10	3.680406e-09
W4	5.256399e-11	2.954331e-07
b1	4.440893e-11	1.087792e-07
b2	2.220448e-11	5.874749e-08
b3	2.220444e-11	2.220444e-08
b4	2.807528e-11	5.071322e-08

When looking at tables 1-3, you can draw the conclusion that the computed gradients are correct.

Evolution of the loss function for a 3-layer network

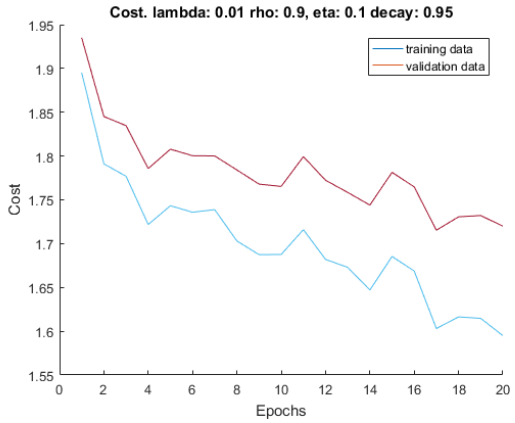

Figure 1. Cost for a 3-layer network using batch normalization

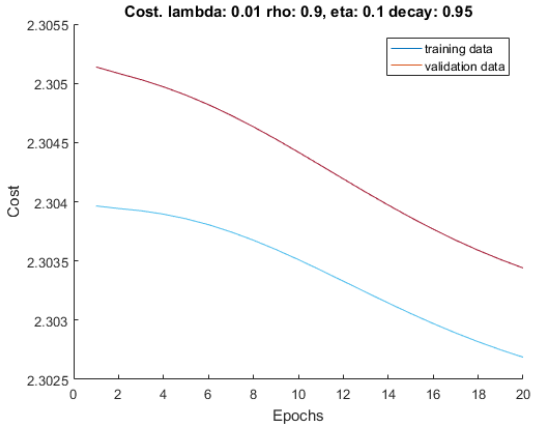

Figure 2. Cost for a 3-layer network, not using batch normalization

Coarse search for λ and η

A coarse random search was performed for values of η between 10^0 and 10^{-3}, and values for λ between 10^{-1} and 10^{-6}. The network ran for 5 epochs and the best hyper-parameters can be seen below.

Table 4. Accuracy on the validation set for different hyper-parameters when training for only 5 epochs.

η	λ	Accuracy
0.069957	0.0000069414	0.431400
0.032812	0.0000079982	0.429900
0.015888	0.0001508533	0.426800

Felicia Rosell
28 June 2017

Fine search for λ and η

In the fine search, the values of η searched were in between 10^{0} and 10^{-3}, and values of λ were between 10^{-2} and 10^{-6}. The network ran for 7 epochs and the best hyper-parameters can be seen below.

Table 5. Accuracy on the validation set for different hyper-parameters when training for 7 epochs.

η	λ	Accuracy
0.010190	0.0000015673	0.431400
0.010417	0.0012047585	0.431300
0.007105	0.0009954781	0.431100

2nd Fine search for λ and η

In the fine search, the values of η searched were in between 10^{1} and 10^{-3}, and values of λ were between 10^{-2} and 10^{-6}. The network ran for 10 epochs and the best hyper-parameters can be seen below.

Table 6. Accuracy on the validation set for different hyper-parameters when training for 10 epochs.

η	λ	Accuracy
0.042633	0.0000051760	0.428500
0.003087	0.0000032683	0.427400
0.017479	0.0005219310	0.427100

Final training

The best parameters found were $\eta = 0.0426$ and $\lambda = 0.00000518$

The network was trained using these hyper-parameters for 20 epochs. The training and validation cost can be seen in table 6, and the networks performance on the test data was 41.60 %.

Felicia Rosell
28 June 2017

Comparing networks with and without batch normalization

When using a 2-layer network, with 50 hidden nodes, the following results were found.

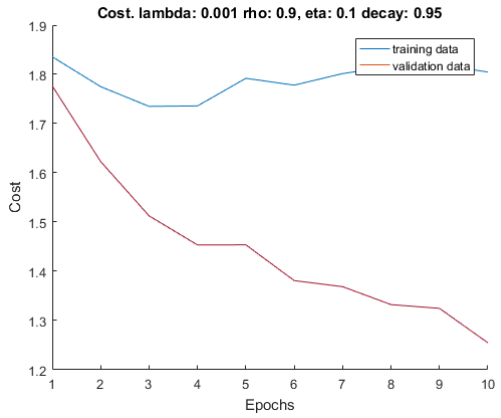

Figure 3. Cost for a network with batch normalization

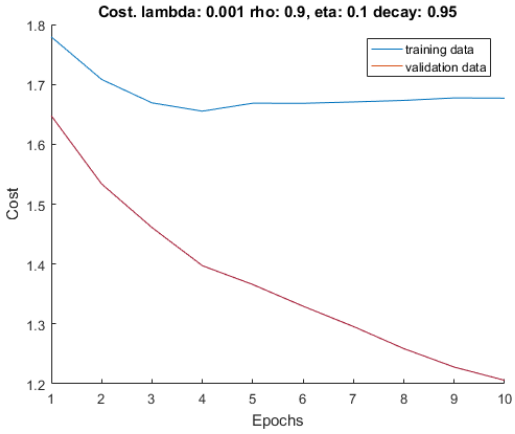

Figure 4. Cost for a network without batch normalization

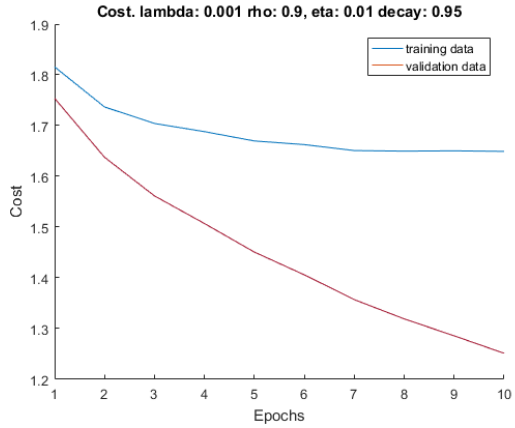

Figure 5. Cost for a network with batch normalization

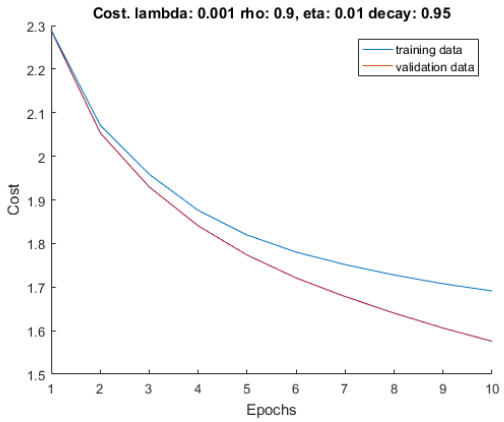

Figure 6. Cost for a network without batch normalization

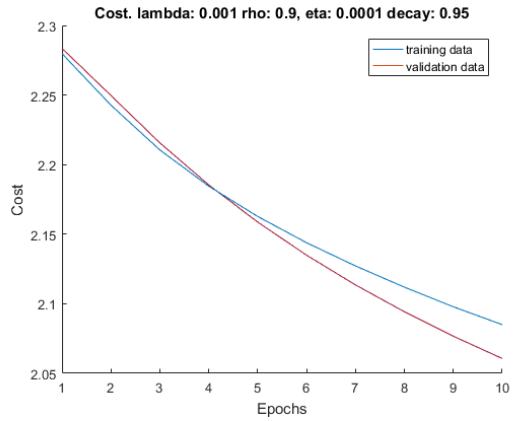

Figure 7. Cost for a network with batch normalization

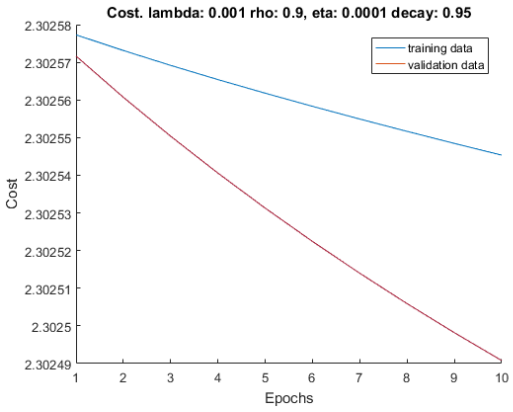

Figure 8. Cost for a network without batch normalization

The question is, is it true that higher learning rates can be used when the network has batch normalization? From figures 3-8 you can draw the conclusion that in the case of a 2-layer network, it

does not seem to be true. From figures 3-4, almost the opposite seems to be true. Batch normalization does however seem to increase the performance overall.

Felicia Rosell
28 June 2017

Code

```
[X,Y,y] = LoadBatch('data_batch_1.mat');

subset = size(X,2);
featureSubset = size(X,1);

X = X(1:featureSubset,1:subset);
Y = Y(:,1:subset);

[val_X,val_Y,val_y] = LoadBatch('data_batch_2.mat');
val_X = val_X(1:featureSubset,:);

[test_X,test_Y,test_y] = LoadBatch('test_batch.mat');
test_X = test_X(1:featureSubset,:);

% Subtract the mean of the training input
% on the training, validation and test input set
mean_X = mean(X, 2);

X = X - repmat(mean_X, [1, size(X, 2)]);
val_X = val_X - repmat(mean_X, [1, size(val_X, 2)]);
test_X = test_X - repmat(mean_X, [1, size(test_X, 2)]);

K = size(Y,1);
d = size(X,1);
N = size(X,2);

n_epochs = 10;
n_batch = 100;
hiddenNodes = [50];
[W,b] = InitializeParameters(d, K, hiddenNodes);

lambda = 0.001;
eta = 1;
decayRate = 0.95;
rho = 0.9;

global BATCH_NORMALIZATION;
BATCH_NORMALIZATION = 1;

[Wstar, bstar] = MiniBatchGD(X, Y, val_X, val_Y, val_y, n_batch, eta,
n_epochs, W, b, lambda, rho, decayRate);

%correct = CheckGradients()
%FindParameters(X, Y, val_X, val_Y, val_y);

function [s] = BatchNormalize(s, mean, variance)
% s, mean and variance are all of the same size: mx1
% So s is the score for one input point and variance and mean is
% calculated from one batch
    eta = 0.00001;
    s = (diag(variance + eta))^(-1/2)*(s-mean);
    if any(isnan(s)) || any(isinf(s))
        disp('s was NaN in BatchNormalize!')
    end
```

```
end

function g = BatchNormBackPass(g, s, mean, variance) %Verified
% g is of size n * m and is the cost gradients for all entries in the layer
(with respect to score)
% s is the scores for the entire layer, and is of size m * n
% Mean and variance are of size m x 1
    eta = 0.00001;
    Vb = diag(variance + eta);
    gradJs = g;
    n = size(s,2);
    v = Vb^(-3/2);
    if any(isnan(v(:))) || any(isinf(v(:)))
        disp('Value was not real number! BatchNormBackPass')
    end

    summ = 0;
    for i=1:n
        summ = summ + (gradJs(i,:)*Vb^(-3/2)*diag(s(:,i)-mean));
    end
    gradJvar = -1/2*summ;

    gradJmean = -sum(gradJs*Vb^(-1/2));

    for i=1:n
        g(i,:) = gradJs(i,:)*Vb^(-1/2) + 2/n * gradJvar * diag(s(:,i) -
mean) + gradJmean/n;
    end
end

function correct = CheckGradients()

    [X,Y,~] = LoadBatch('data_batch_1.mat');

    N = 10;
    d = 100;
    K = 10;

    X = X(1:d,1:N);
    Y = Y(1:K,1:N);

    mean_X = mean(X, 2);

    X = X - repmat(mean_X, [1, size(X, 2)]);

    hiddenNodes = [50, 30, 30];

    [W,b] = InitializeParameters(d, K, hiddenNodes);
    [s1, H, P, m, variance, scoresNorm] = EvaluateClassifier(X, W, b);
    correct = 1;

    lambda = 0;
    [gradW, gradb] = ComputeGradientsBatchNorm(X, H, s1, Y, P, W, lambda,
m, variance, scoresNorm);
    disp('Computed gradients');


    %Checking gradients
```

```
    [gradb_num, gradW_num] = ComputeGradsNumSlow(X, Y, W, b, lambda, 1e-5);

    for i=1:size(gradW,1)
        disp(['W', num2str(i), ' grad: ']);
        ga = gradW{i};
        gn = gradW_num{i};

        relativeError = sqrt(sum(sum((ga - gn).^2))) / max(0.001,
sum(sum(ga)) + sum(sum(gn)));
        maxDiff = max(max(abs(ga - gn)));
        disp('max difference,    Relative error ');
        sprintf('%e \t %e', [maxDiff, relativeError])


        disp(['max value: ', num2str(max(max(ga)))]);
        if relativeError > 10E-4
            correct = 0;
        end
        if maxDiff > 10E-6
            correct = 0;
        end
    end

    for i=1:size(gradb,1)
        disp(['b', num2str(i), ' grad: ']);
        ga = gradb{i};
        gn = gradb_num{i};

        relativeError = sqrt(sum(sum((ga - gn).^2))) / max(0.001,
sum(sum(ga)) + sum(sum(gn)));
        maxDiff = max(max(abs(ga - gn)));
        disp('max difference,    Relative error ');
        sprintf('%e \t %e', [maxDiff, relativeError])
        if relativeError > 10E-4
            correct = 0;
        end
        if maxDiff > 10E-6
            correct = 0;
        end
    end

end

function acc = ComputeAccuracy(X, y, W, b)
%Calculate the accuracy scalar
%   that is the percentage of correctly classified
%   samples

    [~, ~, P, ~, ~, ~] = EvaluateClassifier(X, W, b);
    sumCorrect = 0;

    for sample=1:size(P,2)
        [~, class] = max(P(:,sample));

        if class == y(sample)
            sumCorrect = sumCorrect + 1;
        end
    end
    acc = sumCorrect / sample;
```

```
end

function [J] = ComputeCost(X, Y, W, b, lambda, varargin)
%Computes the cost
%   J is a scalar with the sum of the loss of the network's
%       predictions for the images in X relative
%       to the labels and regularization term on W

    if size(varargin,1) == 0
        [~, ~, P, ~, ~, ~] = EvaluateClassifier(X, W, b);
    else
        [~, ~, P, ~, ~, ~] = EvaluateClassifier(X, W, b, varargin{1},
varargin{2});
    end
    s = 0;

    N = size(X,2);

    for i=1:N
        cross = -log(dot(Y(:,i)',P(:,i)));
        s = s + cross;
    end
    s = s / N;

    sumR = 0;
    for l=1:size(W,1)
        sumR = sumR + sum(diag(W{l}.^2));
    end

    J = s + lambda*sumR;

end

function [gradW, gradb] = ComputeGradientsBatchNorm(X, H, s1, Y, P, W,
lambda, mean, variance, sNorm)
%• each column of X corresponds to an image and it has size d×n.
%• each column of Y (K×n) is the one-hot ground truth label for the
corresponding
%   column of X.
%• each column of P contains the probability for each label for the image
%   in the corresponding column of X. P has size K×n.
%• grad_W1 has size m x d
%• grad_W2 has size k x m
%• grad_b1 has size m x 1
%• grad_b2 has size k x 1
    global BATCH_NORMALIZATION;
    n = size(X,2);

    layers = size(W,1);
    gradb = cell(layers, 1);
    gradW = cell(layers, 1);

    for j = 1:layers
        gradW{j} = zeros(size(W{j}));
        gradb{j} = zeros(size(W{j}, 1), 1);
    end

    k = size(Y,1);
    g = zeros(n,k);
```

```
    for i=1:n
        y = Y(:,i);
        p = P(:,i);
        g(i,:) = - (y'/(y'*p))*(diag(p)-p*p');
    end

    gradb{layers} = sum(g)'/n;
    gradW{layers} = (g'*H{layers-1}')/n + 2*lambda*W{layers};

    if BATCH_NORMALIZATION
        s = sNorm{layers-1};
    else
        s = s1{layers-1};
    end
    ind = s > 0;
    g = g*W{layers};

    for i=1:n
        g(i,:) = g(i,:)*diag(ind(:,i));
    end

    for j = layers-1:-1:1
        if BATCH_NORMALIZATION
            g = BatchNormBackPass(g, s1{j}, mean{j}, variance{j});
        end
        if j == 1
            x = X;
        else
            x = H{j-1};
        end

        gradb{j} = sum(g)'/n;

        gradW{j} = (g'*x')/n + 2*lambda*W{j};

        if j > 1
            if BATCH_NORMALIZATION
                s = sNorm{j-1};
            else
                s = s1{j-1};
            end
            ind = s > 0;
            g = g*W{j};
            for i=1:n
                g(i,:) = g(i,:)*diag(ind(:,i));
            end
        end
    end
end

function [scores, H, P, mean, variance, scoresNorm] = EvaluateClassifier(X,
W, b, varargin)
%Evaluates the classifier for a mini-batch
%   by calculating the score
%   and softmax
%   each column of P contains the probability of each label
%       for the image. P has size K*N
%   Sending in mean and variance causes the function to use
%   those values instead of the computed ones.
% Return:
```

```
%   H contains X[2...l] (has size l-1)
%   scores contains the unnormalized scores of size lx1 x mxn
%   scoresNorm only contains the normalized scores for layers 1..l-1
%   the same goes for mean  and variance
    global BATCH_NORMALIZATION;
    layers = size(W,1);

    scores = cell(layers, 1);
    scoresNorm = cell(layers-1, 1);

    mean = cell(layers-1, 1);
    variance = cell(layers-1, 1);

    H = cell(layers-1, 1);

    for j = 1 : layers-1

        M = size(W{j},1);
        K = size(W{j+1},1);

        if j == 1
            h = X;
        else
            h = H{j-1};
        end
        N = size(h,2);
        P = zeros(K,N);

        scores{j} = zeros(M, N);
        mean{j} = zeros(M, 1);

        % Calculate scores for the entire batch, one input at a time
        for i=1:N
            scores{j}(:,i) = W{j}*h(:,i) + b{j};
            % mean is mean of all inputs, a column vector where each entry
            % is the average input for that feature
            mean{j} = mean{j} + scores{j}(:,i);
        end

        mean{j} = mean{j}/N;
        variance{j} = var(scores{j}, 0, 2) * (N-1)/N;

        if size(varargin) > 0
            mean{j} = varargin{1}{j};
            variance{j} = varargin{2}{j};
        end

        %disp(['Scores: ', num2str(size(scores{j})), '. mean: ',
num2str(size(mean{j})) , '. var: ', num2str(size(variance{j}))]);

        % Batch normalize each input in the batch, one at a time
        if BATCH_NORMALIZATION
            for i=1:N
                scoresNorm{j}(:,i) = BatchNormalize(scores{j}(:,i),
mean{j}, variance{j}); %Verfied
            end
        end

        % Calculate activation function for the entire batch
```

```
        if BATCH_NORMALIZATION
            H{j} = max(scoresNorm{j}, 0);
        else
            H{j} = max(scores{j}, 0);
        end

    end

    N = size(H{layers-1},2);

    for i=1:N
        scores{layers}(:,i) = W{layers}*H{layers-1}(:,i) + b{layers};
        s = scores{layers}(:,i);
        P(:,i) = exp(s)/dot(ones(K,1),exp(s));
    end
end

function y = FindParameters(X, Y, val_X, val_Y, val_y)

    m = [50, 30];
    K = size(Y,1);
    d = size(X,1);

    n_epochs = 10;
    n_batch = 100;
    decayRate = 0.95;
    rho = 0.9;

    e_min = -3;
    e_max = -1;

    el_min = -6;
    el_max = -2;

    fileID = fopen('test.txt','a');
    fprintf(fileID,'%8s\t%11s\t%8s\t%8s\n','eta', 'lambda', 'accuracy',
'average acc');

    tries = 100;
    el = el_min + (el_max - el_min) * rand(tries,1);
    lambdas = 10.^el;


    e = e_min + (e_max - e_min) * rand(tries,1);
    etas = 10.^e;

    for i=1:tries
       bestAcc = 0;
       averageAcc = 0;
       iterations = 1;
       for j=1:iterations
            [W,b] = InitializeParameters(d, K, m);
            lambda = lambdas(i,1);
            eta = etas(i,1);
            [Wstar, bstar] = MiniBatchGD(X, Y, val_X, val_Y, val_y,
n_batch, eta, n_epochs, W, b, lambda, rho, decayRate);
            acc = ComputeAccuracy(val_X, val_y, Wstar, bstar);

            if acc > bestAcc
```

```
                bestAcc = acc;
            end
            averageAcc = averageAcc + acc;
        end
        averageAcc = averageAcc / iterations;
        disp(['i: ', num2str(i)]);
        A = [eta, lambda, bestAcc, averageAcc]

        fprintf(fileID,'%0.6f\t%0.10f\t%0.6f\t%1.6f\n',A);
        format shortg
        disp(clock)

    end

    fclose(fileID);

    y = 1;
end

function [W,b] = InitializeParameters(dim, numClasses, hiddenNodes)

    numLayers = size(hiddenNodes,2) + 1;
    W = cell(numLayers, 1);
    b = cell(numLayers, 1);


    W{1} = randn(hiddenNodes(1), dim)*0.001;
    for i=2:numLayers-1
        W{i} = randn(hiddenNodes(i), hiddenNodes(i-1))*0.001;
    end

    W{numLayers} = randn(numClasses, hiddenNodes(numLayers-1))*0.001;

    for i=1:numLayers
        b{i} = zeros(size(W{i},1),1);

    end
end

function [X, Y, y] = LoadBatch(filename)
%Function that reads the data from the file
%   X is a matrix containing image pixel data.
%       it has size d*N, N is number of
%       images = 10000, and d is dimensionality = 32*32*2=3072,
%       each column represents one image
%   Y contains on each column the one-hot represention of the label
%       for each image
%       and is the size N*K where K is #labels = 10
%   y is a row vector containing the label for each image, between 1 and 10
    batch = load(filename);
    X = double(batch.data')/255;
    y = batch.labels' + 1;
    N = size(X,2);
    K = 10;
    Y = zeros(K,N);
    for i=1:N
        Y(y(i),i) = 1;
    end
end
```

```
function [Wstar, bstar] = MiniBatchGD(X, Y, Xval, Yval, yval, n_batch, eta,
n_epochs, W, b, lambda, rho, decayRate)
%Mini-batch learning function of W and b, with gradient descent
%   X training images
%   Y labels for training images
%   W and b initial values
%   lambda regularization factor in the cost function
%   GDparams contains n_batch, eta and n_epochs

    N = size(X,2);

    costTrain = zeros(1, n_epochs);
    costVal = zeros(1, n_epochs);

    layers = size(W,1);

    mom_W = cell(layers, 1);
    mom_b = cell(layers, 1);

    meanAv = cell(layers,1);
    varianceAv = cell(layers,1);
    alpha = 0.99;

    for i=1:layers
        mom_W{i} = zeros(size(W{i}));
        mom_b{i} = zeros(size(b{i}));
    end

    decay = decayRate;
    startEta = eta;

    startCost = ComputeCost(X, Y, W, b, lambda);
    disp(['startcost: ', num2str(startCost)]);
    for i=1:n_epochs
        for j=1:N/n_batch
            j_start = (j-1)*n_batch + 1;
            j_end = j*n_batch;
            Xbatch = X(:, j_start:j_end);
            Ybatch = Y(:, j_start:j_end);

            [s1, H, P, mean, variance, sNorm] = EvaluateClassifier(Xbatch,
W, b);
            [grad_W, grad_b] = ComputeGradientsBatchNorm(Xbatch, H, s1,
Ybatch, P, W, lambda, mean, variance, sNorm);

            for k=1:layers
                if k ~=layers
                    if i == 1 && j == 1
                        meanAv{k} = mean{k};
                        varianceAv{k} = variance{k};
                    end
                    meanAv{k} = alpha * meanAv{k} + (1-alpha)*mean{k};
                    varianceAv{k} = alpha * varianceAv{k} + (1-
alpha)*variance{k};
                end

                mom_W{k} = mom_W{k}*rho + eta*grad_W{k};
                W{k} = W{k} - mom_W{k};
```

```
                mom_b{k} = mom_b{k}*rho + eta*grad_b{k};
                b{k} = b{k} - mom_b{k};
            end

        end

        eta = eta * decay;
        costTrain(i) = ComputeCost(X, Y, W, b, lambda);

        if costTrain(i)>3*startCost
            Wstar = W;
            bstar = b;
            disp(['Cost was to big: ', num2str(costTrain(i)), ' while start
cost was: ', num2str(startCost)])
            return
        end
        disp(['epoch: ', num2str(i), '/', num2str(n_epochs), '     Cost: ',
num2str(costTrain(i))]);
        costVal(i) = ComputeCost(Xval, Yval, W, b, lambda, meanAv,
varianceAv);

        hold on
        plot(1:i, costTrain(1:i), 1:i, costVal(1:i));
        title(['Cost. lambda: ', num2str(lambda), ' rho: ', num2str(rho),
', eta: ', num2str(startEta), ' decay: ', num2str(decay)]);

        drawnow

    end
    Wstar = W;
    bstar = b;

    plot(1:n_epochs, costTrain, 1:n_epochs, costVal);
    title(['Cost. lambda: ', num2str(lambda), ' rho: ', num2str(rho), ',
eta: ', num2str(startEta), ' decay: ', num2str(decay)]);
    xlabel('Epochs')
    ylabel('Cost')
    legend('training data', 'validation data')
    hold off
    acc = ComputeAccuracy(Xval, yval, W, b)

end
```