DD2424 - Assignement 3

Pierre Rudin

June 7, 2017

1 Introduction

The aim of this assignment is to implement the methods used in previous assignments so that they can be used with any number of layers, and there after train a 3 layer network using a mini-batch gradient descent with momentum and batch normalization, so that the network can classify images from the CIFAR-10 data set. This will be done by implementing the calculations and algorithms given by the instructions for this assignment in Matlab.

1.1 Data set

CIFAR-10 is a data set of 60000 labeled 32 x 32 pixel coloured images separated in 5 training batches and 1 test batch, each batch holding 10000 images. In this assignment 2 training batches and the test batch i used.

1.2 Algorithm

This algorithm make use of some pretty simple linear algebraic equations. Since the equations are given by the instructions this section will contain my implementation step by step.

1. The data sets are loaded and the network initialized. The data set loads to $\mathbf{X}$ (size d x N), $\mathbf{Y}$ (K x N) and $\mathbf{y}$ (1 x N). The network is initialized by creating $\mathbf{W_1}$ ($nodes_{hiddenLayer1}$ x d), $\mathbf{W_{2,...,n\text{-}1}}$ ($nodes_{thisLayer}$ x $nodes_{previousLayer}$) $\mathbf{W_n}$ (K, $nodes_{hiddenLayern-1}$), $\mathbf{b_1}$ ($nodes_{hiddenLayer1}$, 1), $\mathbf{b_{2,...,n\text{-}1}}$($nodes_{thisLayer}$, 1) and $\mathbf{b_n}$ (K x 1) which will contain randomly generated numbers. This is also where we create the mini-batches (**XBatches** and **YBatches**).

2. Take the j:th batches from **XBatches** and **YBatches** and make label predictions on each image in **XBatches**.

3. Compute gradients with batch normalization added on the predictions and batches from previous step.

4. Add momentum

5. Update the network.
6. Repeat from step 2 until end of epochs or other condition is satisfied.

2 Results

2.1 Analyticall gradient checking

The analytically computed gradients where compared to numerically computed gradients, to compute the numerical gradients the code given with the assignment was used.

2.1.1 2-Layer Network

	Sum	Mean	Min	Max
b1	2.821483e-01	5.642965e-03	2.116609e-04	2.344918e-02
b2	4.500311e-06	4.500311e-07	4.481301e-07	4.540629e-07
W1	1.003890e+02	6.535743e-04	2.087295e-09	5.148941e-03
W2	1.062019e-04	2.124039e-07	1.658791e-07	2.622531e-07

Table 1: Absolute difference between numerically and analytically computed gradients on a 2-layer network

With batch normalization the difference between the analytically and numerically calculated gradients are bigger then in previous assignments. But since the mean difference still are in the range of 1e-3 or smaller I would still consider the analytically calculated gradients to be quiet accurate.

2.1.2 3-Layer Network

	Sum	Mean	Min	Max
b1	1.555098e-01	3.110196e-03	8.404835e-05	1.305131e-02
b2	4.115022e-01	1.371674e-02	3.729078e-04	4.355665e-02
b3	4.499778e-06	4.499778e-07	4.492763e-07	4.504875e-07
W1	1.126712e+02	7.335363e-04	8.897409e-09	4.812008e-03
W2	1.043999e+01	6.959991e-03	5.697405e-06	5.041149e-02
W3	3.832258e-05	1.277419e-07	7.448231e-08	1.879620e-07

Table 2: Absolute difference between numerically and analytically computed gradients on a 3-layer network

Adding a layer increases the difference between the methods of calculating the gradients. This however is expected, and it is concluded that the numbers for 3 layers looks good.

2.1.3 4-Layer Network

	Sum	Mean	Min	Max
b1	2.708227e-01	2.708227e-03	3.645406e-05	9.005303e-03
b2	3.209452e-01	6.418905e-03	1.289368e-04	2.986614e-02
b3	3.332370e-01	1.110790e-02	4.138029e-06	3.213377e-02
b4	4.499423e-06	4.499423e-07	4.491167e-07	4.509904e-07
W1	2.191894e+02	7.135072e-04	3.616336e-09	6.332208e-03
W2	2.316350e+01	4.632700e-03	9.506812e-08	3.893993e-02
W3	7.959463e+00	5.306309e-03	8.669500e-06	2.761229e-02
W4	3.708043e-05	1.236014e-07	7.681426e-08	2.101081e-07

Table 3: Absolute difference between numerically and analytically computed gradients on a 4-layer network

Adding yet another layer increases the difference between the methods of calculating the gradients even further. This, again, however is expected, and it is concluded that the numbers for 4 layers looks good.

2.2 Batch Normalization

To test the effects of Batch Normalization, 3 test runs where made with Batch Normalization, and 3 with out Batch Normalization. Each run is then presented with a graph in the section bellow.

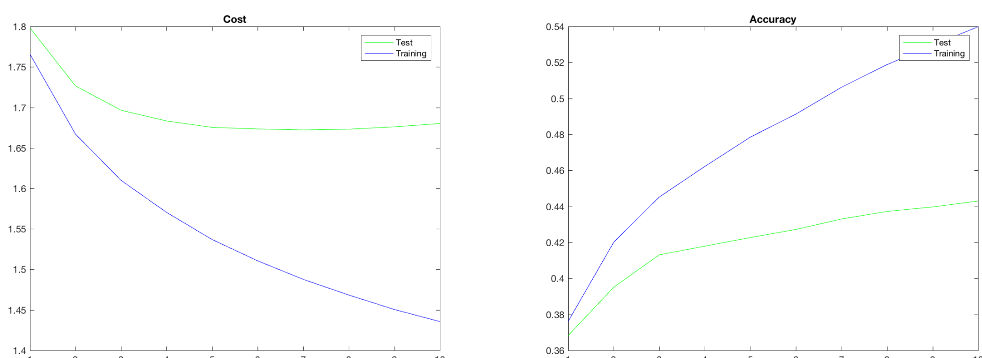


Figure 1: Graphs displaying cost and accuracy development for each epoch, with Batch Normalization and $\eta = .025$.

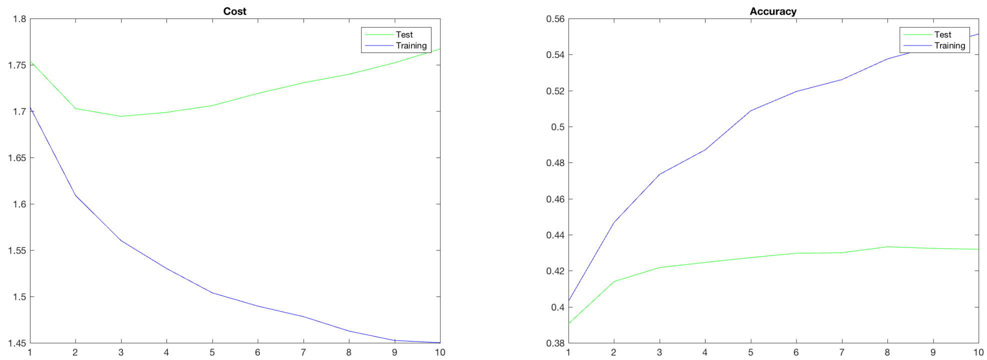


Figure 2: Graphs displaying cost and accuracy development for each epoch, with Batch Normalization and $\eta = .05$.

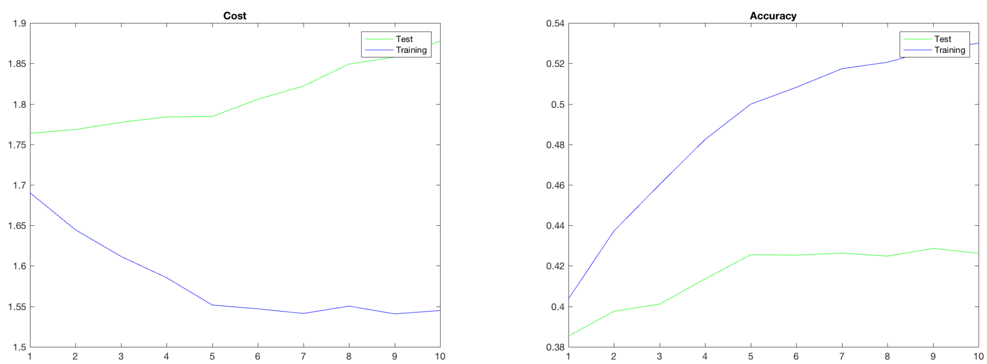


Figure 3: Graphs displaying cost and accuracy development for each epoch, with Batch Normalization and $\eta = .1$.

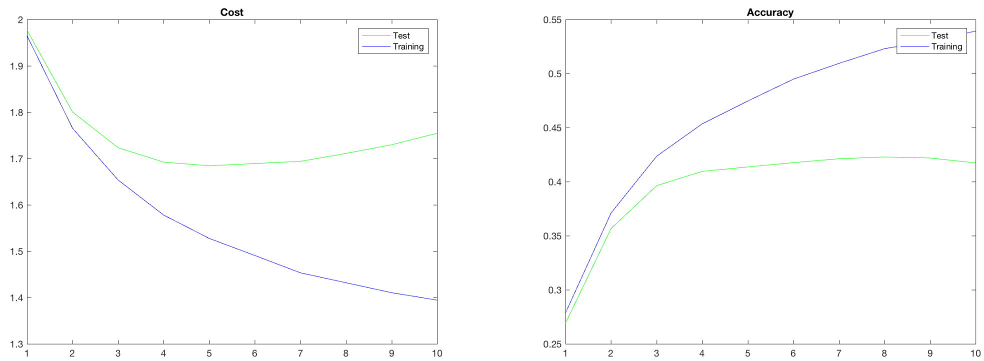


Figure 4: Graphs displaying cost and accuracy development for each epoch, without Batch Normalization and $\eta = .025$.

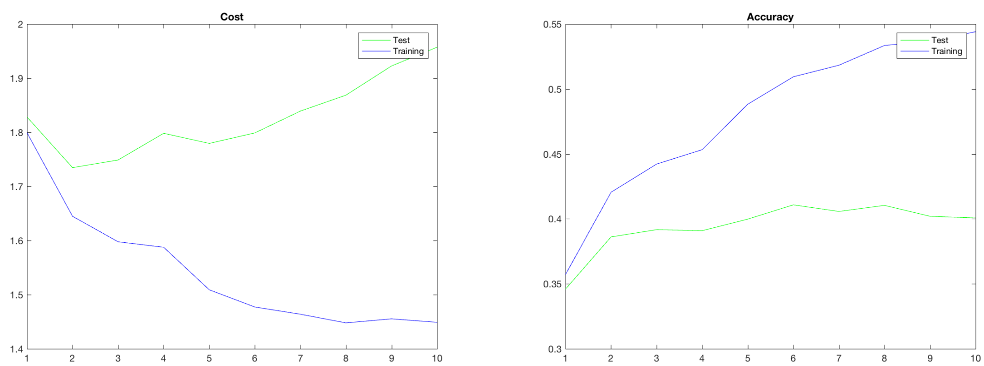
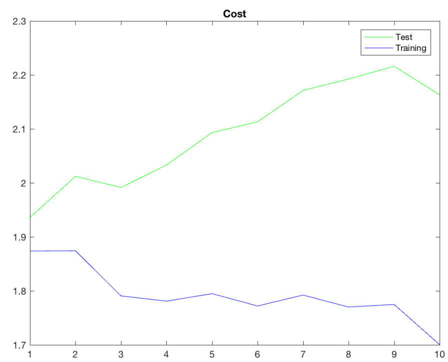


Figure 5: Graphs displaying cost and accuracy development for each epoch, without Batch Normalization and $\eta = .05$.



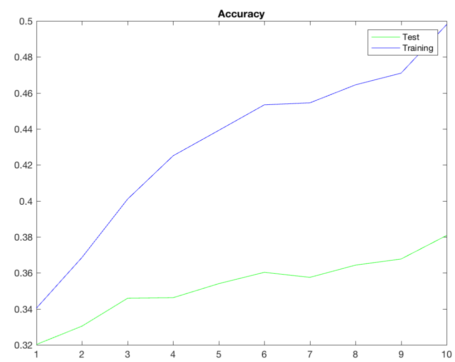


Figure 6: Graphs displaying cost and accuracy development for each epoch, without Batch Normalization and $\eta = .1$.

As we can see in the graphs, the gap between the cost on the training data and the test data is smaller when Batch Normalization is used. It's also noted that the overall accuracy on the test data is higher when Batch Normalization is used.

2.3 Coarse search

The initial search for our parameters starts with these values;

$\eta = [.01, .025, .05, .1, .25, .5, .75]$

$\lambda = [0, .001, .0025, .005, .01, .025, .05, .1]$

Which combined adds up to 56 different combination. Each combination is used to train the network for 5 epochs.

2.3.1 #1

Parameters:

$\eta = .1$

$\lambda = 0$

Accuracy test: .4181 Accuracy train: .4695

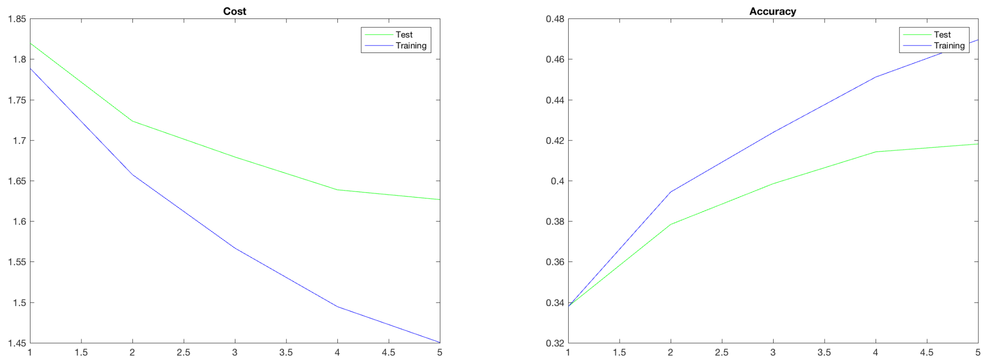


Figure 7: Graphs displaying cost and accuracy development for each epoch, when $\eta = .1$ and $\lambda = 0$.

2.3.2 #2

Parameters:

$\eta = .1$

$\lambda = .0005$

Accuracy test: .414 Accuracy train: .4665

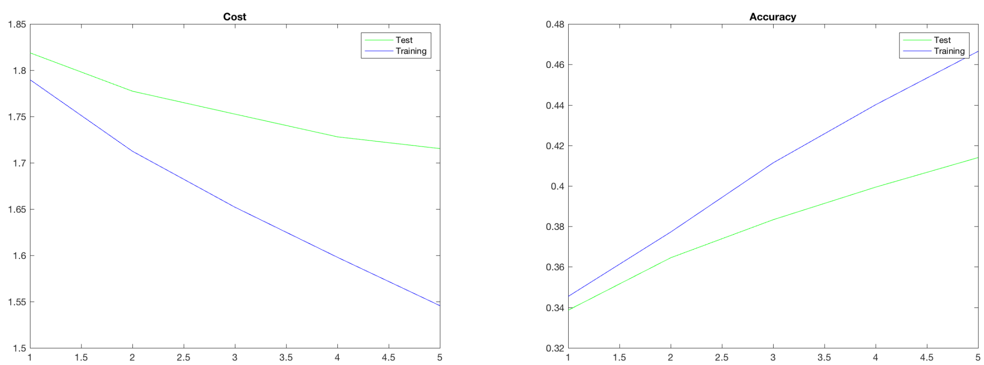


Figure 8: Graphs displaying cost and accuracy development for each epoch, when $\eta = .1$ and $\lambda = .0005$.

2.3.3 #3

Parameters:

$\eta = .05$

$\lambda = 0$

Accuracy test: .3852 Accuracy train: .4102

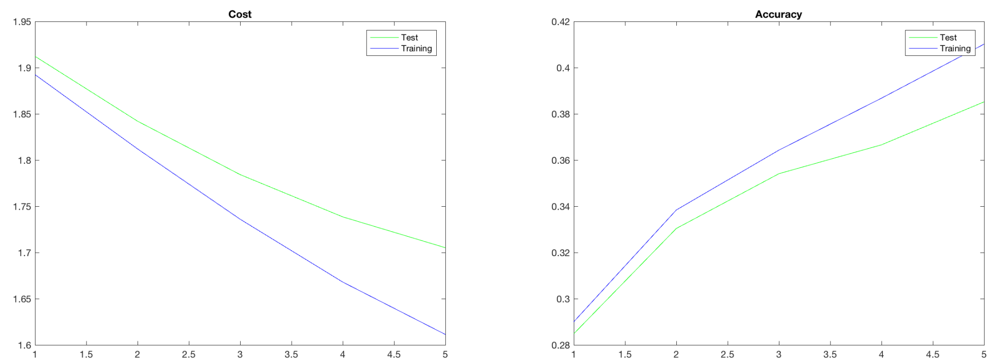


Figure 9: Graphs displaying cost and accuracy development for each epoch, when $\eta = .05$ and $\lambda = 0$

2.4 Fine search

With the result from the Coarse search in mind, the following parameters where chosen for the Fine search;

η = (.15-.03) .* rand(1,8) + .03

λ = .001 .* rand(1,8)

Which combined adds up to 64 different combination. Each combination is used to train the network for 10 epochs.

2.4.1 #1

Parameters:

$\eta = .045828$

$\lambda = .000662$

Accuracy test: .4266, Accuracy train: .4744

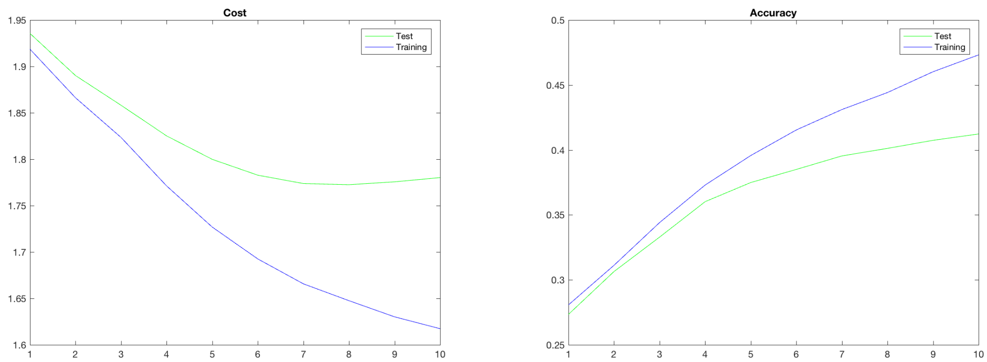


Figure 10: Graphs displaying cost and accuracy development for each epoch, when $\eta = .045828$ and $\lambda = .000662$.

2.4.2 #2

Parameters:

$\eta = .045828$

$\lambda = .000378$

Accuracy test: .42, Accuracy train: .5008

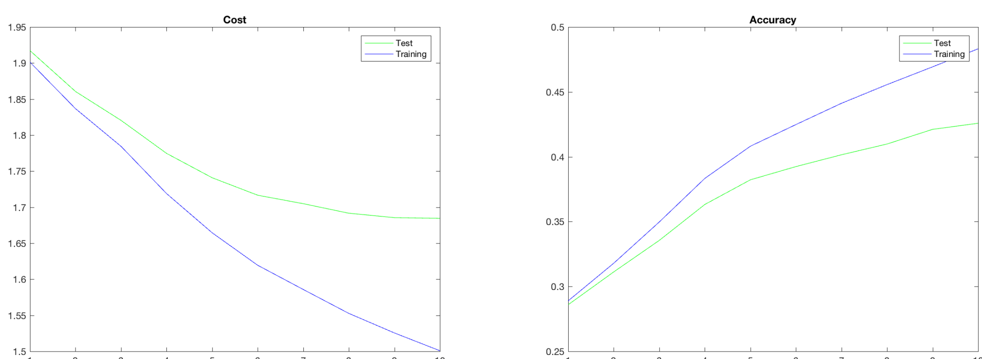


Figure 11: Graphs displaying cost and accuracy development for each epoch, when $\eta = .045828$ and $\lambda = .000378$.

2.4.3 #3

Parameters:

$\eta = .045828$

$\lambda = .000261$

Accuracy test: .4092, Accuracy train: .5033

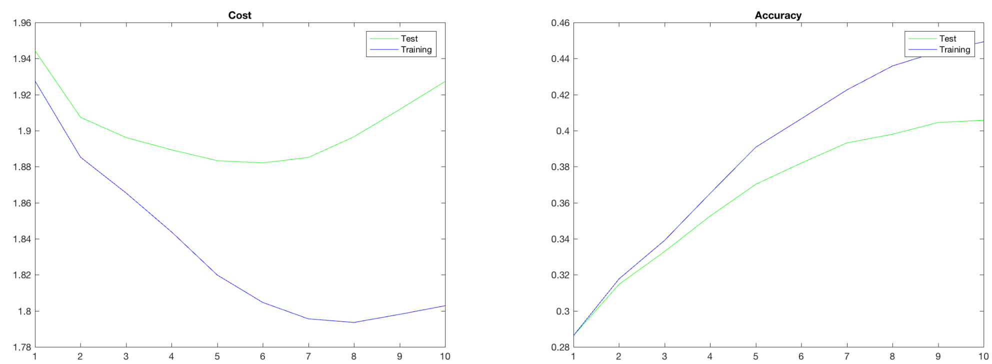


Figure 12: Graphs displaying cost and accuracy development for each epoch, when $\eta = .045828$ and $\lambda = .000261$.

2.5 Best parameters

When the best parameters are found, a further run is done where the network is trained for 30 epochs on the full training set except for the last 1000 images that are used for validation. Parameters:

$\eta = .045828$

$\lambda = .000662$

Accuracy test: .432, Accuracy train: .619778

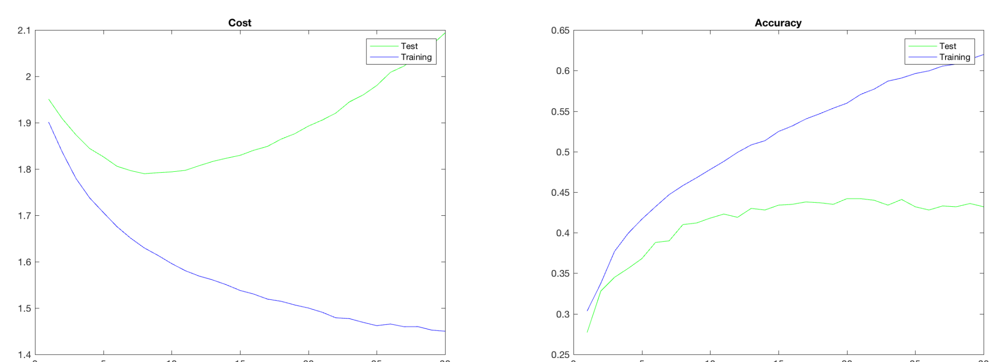


Figure 13: Graphs displaying cost and accuracy development for each epoch, when $\eta = .045828$ and $\lambda = .000662$.

3 Code

3.1 Main

```
clear all;
clc;
close all;

addpath Datasets/cifar-10-batches-mat;

% Parameters
n_batch = 50;
n_epochs = 10;
h = 1e-5; %
nodes_in_hidden_layers = [50, 30]; % Number of nodes in hidden
    layers

% Hyper parameters
eta = 0.01; % learning rate
lambda = 0.001; % regularization
decay_rate = .998; % decay in learning rate
rho = .8; % Momentum

% Data setup
[X,Y,y,mean_X] = LoadBatch('data_batch_1.mat');
[XValid, YValid, yValid] = LoadData('data_batch_2.mat', mean_X);
[XTest,YTest,yTest] = LoadData('test_batch.mat', mean_X);
[XBatches, YBatches] = GetMiniBatches(X, Y, n_batch);
[W, b] = InitModel(X,nodes_in_hidden_layers);

trainingLoop(XBatches,YBatches,W,b,n_epochs,eta,lambda,rho,
    decay_rate,nodes_in_hidden_layers,X,Y,y,XTest,YTest,yTest,h);
```

3.2 LoadBatch()

```
function [X, Y, y, mean_X] = LoadBatch(filename)
    A = load(filename);
    X = double(A.data') / 255;
    y = A.labels';
    y = y + uint8(ones(1,length(y))); % Add one to simplify
        indexing

    mean_X = mean(X, 2);
```

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

    % Create image label matrix
    Y = zeros(10, length(X));
    for i = 1:length(Y)
        Y(y(i),i) = 1;
    end;
```

3.3 LoadData()

```
function [X, Y, y] = LoadData(filename, mean_X)
    A = load(filename);
    X = double(A.data') / 255;
    y = A.labels';
    y = y + uint8(ones(1,length(y))); % Add one to simplify
        indexing

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

    % Create image label matrix
    Y = zeros(10, length(X));
    for i = 1:length(Y)
        Y(y(i),i) = 1;
    end;
```

3.4 InitModel()

```
function [W, b] = InitModel(X, nodes_in_hidden_layers)
    W = cell(length(nodes_in_hidden_layers)+1, 1);
    b = cell(length(nodes_in_hidden_layers)+1, 1);
    [d, ~] = size(X);

    W{1} = normrnd(0,.001,nodes_in_hidden_layers(1),d);
    b{1} = zeros(nodes_in_hidden_layers(1),1);

    for i = 2:length(nodes_in_hidden_layers)
        W{i} = normrnd(0,.001,nodes_in_hidden_layers(i),
            nodes_in_hidden_layers(i-1));
        b{i} = zeros(nodes_in_hidden_layers(i),1);
    end;

    W{length(nodes_in_hidden_layers)+1} = normrnd(0,.001,10,
        nodes_in_hidden_layers(length(nodes_in_hidden_layers)));
    b{length(nodes_in_hidden_layers)+1} = zeros(10,1);
```

3.5 GetMiniBatches()

```
function [XBatches, YBatches] = GetMiniBatches(Xtrain, Ytrain,
    n_batch)
    [d,N] = size(Xtrain);
    [K,~] = size(Ytrain);

    XBatches = zeros(N/n_batch,d,n_batch);
    YBatches = zeros(N/n_batch,K,n_batch);

    for j=1:N/n_batch
        j_start = (j-1)*n_batch + 1;
        j_end = j*n_batch;
        Xbatch = Xtrain(:, j_start:j_end);
        Ybatch = Ytrain(:, j_start:j_end);

        XBatches(j,:,:) = Xbatch;
        YBatches(j,:,:) = Ybatch;
    end

    % Permute to simplify picking out image representations from
        the
    % matrices.
    XBatches = permute(XBatches,[3 2 1]);
    YBatches = permute(YBatches,[3 2 1]);
```

3.6 trainingLoop()

```

function [W,b,costs_train,costs_test,accs_train,accs_test] =
    trainingLoop(XBatches,YBatches,W,b,n_epochs,eta,lambda,rho,
    decay_rate,nodes_in_hidden_layers,X,Y,y,XTest,YTest,yTest,h)
    epsilon = 1e-5;

    accs_train = double.empty(n_epochs,0);
    accs_test = double.empty(n_epochs,0);
    costs_train = double.empty(n_epochs,0);
    costs_test = double.empty(n_epochs,0);

    n = length(nodes_in_hidden_layers);

    %Momentum
    moment_W = cell(n+1,1);
```

```
    moment_b = cell(n+1,1);
    mu_exp = cell(n+1,1);
    v_exp = cell(n+1,1);
    for i = 1:n+1
        moment_W{i} = zeros(size(W{i}));
        moment_b{i} = zeros(size(b{i}));
    end


    [~,~,l] = size(XBatches);

    for i = 1:n_epochs
        for j = 1:l
            % Get j:th batch
            XBatch = XBatches(:,:,j)';
            YBatch = YBatches(:,:,j)';

            [P, mu_exp, v_exp] = EvaluateClassifier(XBatch, W, b,
                epsilon, 'train', mu_exp, v_exp);
            [grad_W, grad_b, mu_exp, v_exp] = ComputeGradients(
               XBatch, YBatch, P, W, b, lambda,
               nodes_in_hidden_layers,epsilon, 'train', mu_exp,
               v_exp);


            % Calculate momentum
            for m = 1:n+1
                moment_W{m} = eta * grad_W{m} + rho * moment_W{m
                    };
                moment_b{m} = eta * grad_b{m} + rho * moment_b{m
                    };
            end

            % Update W's and b's
            for m = 1:n+1
                W{m} = W{m} - moment_W{m};
                b{m} = b{m} - moment_b{m};
            end

        end;
        eta = decay_rate * eta;

```

```
        [P, mu_exp, v_exp] = EvaluateClassifier(X, W, b, epsilon, 'train', mu_exp, v_exp);
        accs_train(i) = ComputeAccuracy(P, y);
        costs_train(i) = ComputeCost(X, Y, W, b, lambda, epsilon, 'train', mu_exp, v_exp);

        [PTest, ~, ~] = EvaluateClassifier(XTest, W, b, epsilon, 'test', mu_exp, v_exp);
        accs_test(i) = ComputeAccuracy(PTest, yTest);
        costs_test(i) = ComputeCost(XTest, YTest, W, b, lambda, epsilon, 'test', mu_exp, v_exp);

       %Displays the cost and accuracy of each epoch
        fprintf('Cost train: %f\n', costs_train(i));
        fprintf('Accuracy train: %f\n', accs_train(i));
        fprintf('Cost test: %f\n', costs_test(i));
        fprintf('Accuracy test: %f\n\n', accs_test(i));

        fprintf('Epoch %f is done.\n', i);
    end;

    % % Plots evolution of the cost
    figure(2);
    x = 1:1:n_epochs;
    plot(x,costs_test,'g',x,costs_train,'b');
    title('Cost')
    legend('Test','Training')

    % Plots evolution of the accuracy
    figure(3);
    x = 1:1:n_epochs;
    plot(x,accs_test,'g',x,accs_train,'b');
    title('Accuracy')
    legend('Test','Training')

    close all;
```

3.7 EvaluateClassifier()

```
function [P, mu_exp, v_exp] = EvaluateClassifier(X, W, b, epsilon, mode, mu_exp, v_exp)
    hiddenLayer = X;
    for i = 1 : length(W) - 1
```

```
        [hiddenLayer, ~, ~, ~, mu_exp{i}, v_exp{i}] = MakeHiddenLayer(hiddenLayer, W{i}, b{i}, epsilon, mode, mu_exp{i}, v_exp{i});
    end;
    s = W{length(W)}*hiddenLayer+b{length(W)};
    P = exp(s) ./ sum(exp(s));
```

3.8 MakeHiddenLayer()

```
function [hiddenLayer, s0, mu, v, mu_exp, v_exp] = MakeHiddenLayer(X, W, b, epsilon, mode, mu_exp, v_exp)
    s = W * X + b;
    s0 = s;
    [s, mu, v] = BatchNormalize(s, epsilon, mode, mu_exp, v_exp);
    hiddenLayer = max(0,s);

    if strcmp(mode,'train')
        alpha = .99;
        mu_exp = alpha * mu + (1 - alpha) * mu;
        v_exp = alpha * v + (1 - alpha) * v;
    end
```

3.9 ComputeGradients()

```
function [grad_W, grad_b, mu_exp, v_exp] = ComputeGradients(X, Y, P, W, b, lambda, nodes_in_hidden_layers,epsilon, mode, mu_exp, v_exp)
    [hY, ~] = size(Y);
    [hX, lX] = size(X);

    n = length(nodes_in_hidden_layers);
    hidden_layers = cell(n,1);
    mu = cell(n-1,1);
    v = cell(n-1,1);
    S = cell(n-1,1);
    hidden_layers{1} = X;
    for i = 2:n+1
        [hidden_layers{i}, S{i-1}, mu{i-1}, v{i-1}, mu_exp{i-1}, v_exp{i-1}] = MakeHiddenLayer(hidden_layers{i-1}, W{i-1}, b{i-1}, epsilon, mode, mu_exp, v_exp);
    end;

    grad_W = cell(1, n + 1);
    grad_b = cell(1, n + 1);
```

```
    grad_W{1} = zeros(nodes_in_hidden_layers(1),hX);
    grad_b{1} = zeros(nodes_in_hidden_layers(1),1);

    for i = 2:n
        grad_W{i} = zeros(nodes_in_hidden_layers(i), nodes_in_hidden_layers(i-1));
        grad_b{i} = zeros(nodes_in_hidden_layers(i),1);
    end;

    grad_W{length(nodes_in_hidden_layers) + 1} = zeros(hY, nodes_in_hidden_layers(n));
    grad_b{length(nodes_in_hidden_layers) + 1} = zeros(hY, 1);

    gs = zeros(lX,nodes_in_hidden_layers(n));

    % Calculate gradients for last layer.
    for i = 1:lX
        y = Y(:,i);
        p = P(:,i);

        [hp,~] = size(p);
        diag_p = eye(hp) .* p;
        g = -((y' / (y' * p)) * (diag_p - p * p'));

        grad_b{n+1} = grad_b{n+1} + g';
        grad_W{n+1} = grad_W{n+1} + g' * hidden_layers{n+1}(:,i)';

        g = g * W{n+1};
        s = S{n}(:,i);
        [hs, ~] = size(s);
        diag_s = eye(hs) .* (s>0);
        g = g * diag_s;

        gs(i,:) = g;
    end;

    grad_b{n+1} = grad_b{n+1} ./ lX;
    grad_W{n+1} = grad_W{n+1} ./ lX + 2 * lambda .* W{n+1};


```

```
    for i = n:-1:1
        x = hidden_layers{i};
        gs = BatchNormBackPass(gs,S{i},mu{i},v{i},epsilon);


        grad_b{i} = sum(gs,1)' ./ lX;
        grad_W{i} = gs' * x' ./ lX + 2 * lambda .* W{i};

        gs = gs * W{i};
        if i > 1
            for j = 1:lX
                s = S{i-1}(:,j);
                [hs, ~] = size(s);
                diag_s = eye(hs) .* (s>0);
                gs(j,:) = gs(j,:) * diag_s;
            end;
        end;
    end;
```

3.10 ComputeCost()

```
function J = ComputeCost(X, Y, W, b, lambda)
    P = EvaluateClassifier(X, W, b);
    c = 0;
    [~,x] = size(X);
    for i = 1:x
        y = Y(:,i);
        p = P(:,i);
        c = c + -log(y'*p);
    end;
    J = c/x + lambda * sum(sum(W{1}.^2)) + lambda * sum(sum(W
        {2}.^2));
```

3.11 ComputeAccuracy()

```
function acc = ComputeAccuracy(P, y)
    [~,prediction] = max(P);
    acc = sum(prediction==y) / length(P);
```

3.12 BatchNormalize()

```
function [s, mu, v] = BatchNormalize(s,epsilon,mode,mu,v)
    if strcmp(mode,'train')
        mu = mean(s, 2);
```

```
        v = var(s,0,2);
    end

    s2 = s - mu;
    v2 = eye(length(v)) .* (v + epsilon);
    v2 = (v2 ^ (-1/2));
    s =  v2 * s2;
```

3.13 BatchNormBackPass()

```
function g = BatchNormBackPass(g,s,mu,v,epsilon)

    [n, m] = size(g);
    for i = 1:n
        diags = eye(m) .* (s(:,i) - mu)';
        diagv = eye(m) .* (v + epsilon);

        gradv = - sum(g(i,:) * (diagv ^ (-3 / 2)) * diags, 1) /
            2;
        gradmu = - sum(g(i,:) * (diagv ^ (-1 / 2)), 1);

        g(i,:) = g(i,:) * (diagv ^ (-1 / 2)) + 2 * (gradv * diags
            + gradmu) / n;
    end;
```