KTH Royal Institute of Technology

DD2424
Deep Learning in Data Science

Year 2017 - 2018

Assignment 2

Picó Oristrell Sandra 940531 – 2308 sandrapo@kth.se

Professors :
Sullivan Josephine

August 5, 2018

Contents

The main goal of this second assignment is to train and test a two-layer network with multiple outputs to classify images from the CIFAR-10 dataset. As in the first assignment, the network will be trained using mini-batch gradient descent with a cost function that computes the cross-entropy loss and with an L_2 regularization term on the weight matrix. The main structure of the code will be similar to assignment 1, but adding more parameters as now we are implementing one extra layer.

In order to explain and show the results obtained in a proper way I will divide the report into different subsections.

At the same time, in the appendix section, I will append all the implemented functions to be able to check the code ,if needed.

1 Background

The first section of this assignment will be focused on explaining the background needed for the assignment implementation.

1.1 Background 1: Mathematical background.

Following the same structure than in the first assignment, the network that we are going to implement will be a classifier where the output will correspond to a vector of probabilities. Then, the main structure of the network will be as follows:

Figure 1: Main structure of the implemented network.

Then, the predicted class will correspond to the output label with higher probability, i.e the *arg max* of the probability's vector.

In order to compute the output vector we will implement the following two-layer neural network:

$$s_1 = W_1 x + b_1 \tag{1}$$
$$h = max(0, s_1) \tag{2}$$
$$s = W_2 h + b_2 \tag{3}$$
$$\mathbf{p} = softmax(s) = \frac{exp(s)}{1^T exp(s)} \tag{4}$$

This means that during the training phase we will learn the parameters W_1, W_2, b_1 and b_2.

Then, defining the model as: $\Theta = [W_1, W_2, b_1, b_2]$, we will solve the following optimization problem to be able to set all the mentioned trainable parameters:

$$\mathbf{\Theta}^* = argmin_{\Theta} J(D, \lambda, \Theta) \tag{5}$$

Where *J* is a cost function that minimize the cross-entropy plus a regularization term on *W*.

$$J(D, \lambda, \Theta) = \frac{1}{\mid D \mid} \sum_{x,y \in D} \log(y^T p) + \lambda \sum_{l=1}^{2} \sum_{i,j} W_{l i,j}^2 \tag{6}$$

The problem will be solve via mini-batch gradient descent with momentum.

For the mini-batch implementation we will begin with sensible random initialization for W and b, and then estimate the parameters for k = 1,2:

$$W_k^{(t+1)} = W_k^{(t)} - \eta \frac{\partial J(^{(t+1)}, \lambda, \Theta)}{\partial W_k} \tag{7}$$

$$b_k^{(t+1)} = b_k^{(t)} - \eta \frac{\partial J(^{(t+1)}, \lambda, \Theta)}{\partial b_k} \tag{8}$$

1.2 Background 2: Speeding up training, add momentum.

The vanilla version of the mini-batch gradient descent is really slow when using a sensible learning rate and with the size of data that we are going to use for this implementation. For this reason, we will need to speed up the training.

One of the possible ways to do it is adding a momentum term in the update step.

To achieve it, you first need to initialize a momentum vector (matrix) v_0 for each parameter of the network. The vector will be initialized to zero and will have the same dimension as the correspondence parameter. Once initialized, per each time step t:

$$v_t = p * v_{t-1} + \eta * \frac{\partial J}{\partial \theta} \tag{9}$$

$$\theta_t = \theta_{t-1} - v_t \tag{10}$$

where η is the learning rate, p $\in$ [0, 1] and θ is a generic placeholder to represent one of the parameters of the model.

In this assignment we will not implement an adaptive learning rate but we will decay the learning rate by some factor after each epoch.

1.3 Data structure

As in the previous assignment, the data that we are going to use is from the CIFAR-10 dataset. The dataset has a size of 10000 x 3072 following the next structure:

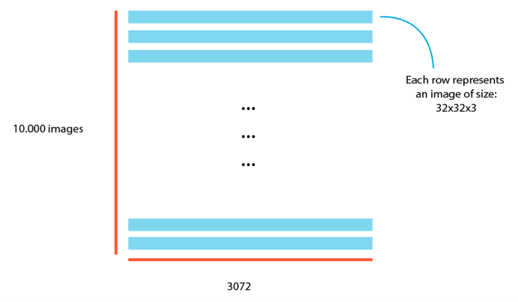

Figure 2: Data structure of CIFAR-10 dataset

2 Exercises

2.1 Exercise 1: Read in the data and initialize the parameters of the network.

To begin with this assignment we will split the data as follows:

- **Training data:** data_batch_1.mat
- **Validation data:** data_batch_2.mat
- **Test data:** test_batch.mat

To achieve it, we are going to use the function LoadBatch, already implemented for Assignment 1. Nevertheless, we will also need to pre-process the input data to have zero mean. We need to compute the mean for training data and then subtract this mean to the validation and test sets.

$$mean\_X = mean(X, 2); \tag{11}$$

$$X = X - repmat(mean\_X, [1, size(X, 2)]); \tag{12}$$

Next step will be initialize and set up all the parameters of the network. In this 2 layer neural network we will have m=50 nodes in the hidden layer. As the two weight matrices and bias vectors will have different dimensions, we will use cell arrays to store them.

The weight matrices will be initialized with a Gaussian distribution with 0 mean and 0.001 of standard deviation. On the other hand, bias vectors will be initialized to zero.

Then, the first assignment will be just the initialization of the network. The implementation can be found in the appendix section.

2.2 Exercise 2: Compute the gradients for the network parameters

After initialize and define all the network parameters, we will start writing the functions to compute the gradients. To do so, we will re-use the functions already implemented in Assignment 1.

Specifically, we are re-writing the function: ComputeGradients and transforming it to be able to compute a two-layer network using cell-arrays. EvaluateClassifier and ComputeCost are also modified.

Once the two mentioned functions are already implemented, it is important to check if the gradients are calculated properly. To be able to compare it, we will use the same methodology as in the first assignment; compare the gradients achieved with the numerical ones.

The achieved values are as follows:

	Values
g_$\mathbf{W}_1$	1.9894e-05
g_$\mathbf{b}_1$	7.1271e-04
g_$\mathbf{W}_2$	5.0976e-08
g_$\mathbf{b}_2$	6.2052e-10

As the values achieved are small enough, we will consider that the gradients are properly implemented.

The last experiment that we are going to implement to be sure that gradient computations and mini-batch gradient descent algorithm are okay is to train the network and check if it is possible to over-fit to training data and get a small loss after enough number of epochs.

Specifically, we are going to use the following configurations:

- Small amount of training data: 100 examples.
- Regularization term turned off: lambda = 0.
- Enough number of epochs: 300.
- Reasonable learning rate: 0.01

As can be seen in the following plot, when we train the two layer network using the previous parameters,we are able to over-fit into the training data:

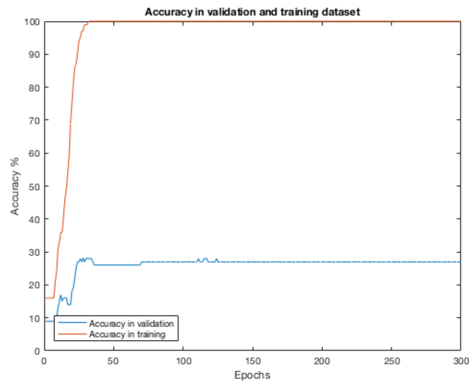

Figure 3: Training and testing accuracy with 100 examples.

We can conclude that the training is working because we are able to over-fit with only 29 epochs.

2.3 Exercise 3: Add momentum to your update step

During this third exercise, we have implemented the momentum vectors into the MiniBatch function to speed up the learning.

We will check its behaviour using different values of rho = [0.5,0.9,0.99] and fixing the learning rate into 0.001. The number of epochs will be set up to 10.

At the same time, to fully check the importance of it and the speed up achieved using momentum, I will do the following experiments using all the training data of the data_batch_1.mat dataset.

Here we can see the comparison between using momentum or without using it.

- **With momentum:**

  - **rho = 0.5**

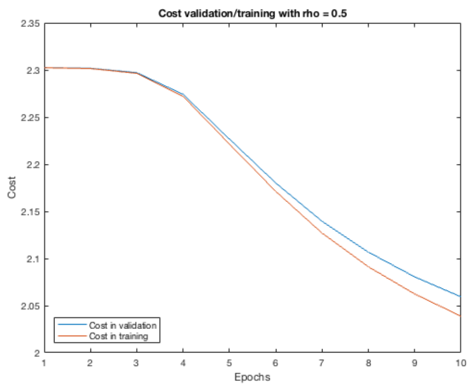

Figure 4: Cost with 10 epochs, rho = 0.5

  - **rho = 0.9**

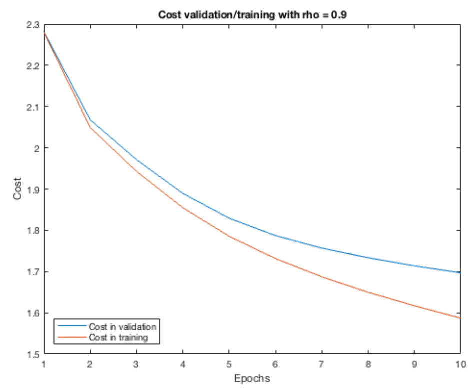

Figure 5: Cost with 10 epochs, rho = 0.9

- rho = 0.99

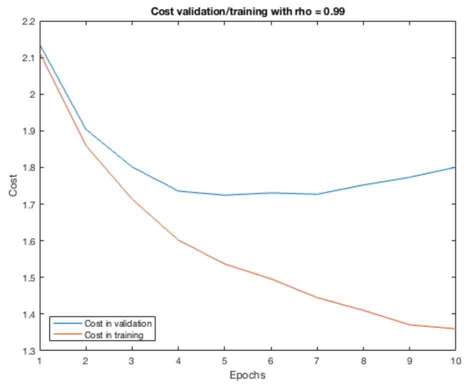

Figure 6: Cost with 10 epochs, rho = 0.99

- **Without momentum:**

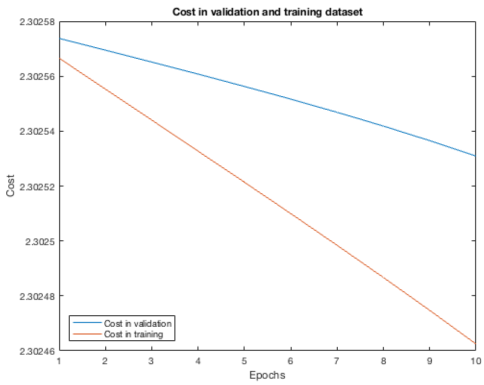

Figure 7: Cost with 10 epochs with no momentum.

As can be concluded from the previous images, using momentum helps to converge faster than not using it. Also, with momentum and rho = 0.90 we achieve the best performance.

2.4 Exercise 4: Training your network

After doing the necessary experiments to verify that the momentum vectors and the gradients were well implemented, we will proceed to train the network correctly.

The first part of this assignment will be focused on finding a reasonable range of values for the learning rate parameter.

- **Find reasonable range of values for learning rate.**

  During this exercise, we are going to use all the samples from the training data; i.e not only using 100 examples of the data.

  As specified in the assignment description, we will use *rho* = 0.9, a small value for the regularization term and also only 10 epochs.

  To obtain a reasonable value of the learning rate, we will test all the next values:

  eta_values = [0.1,0.01,0.001,0.0001,1e-05,1e-06,1e-7];

  The obtained results can be seen in the next figure:

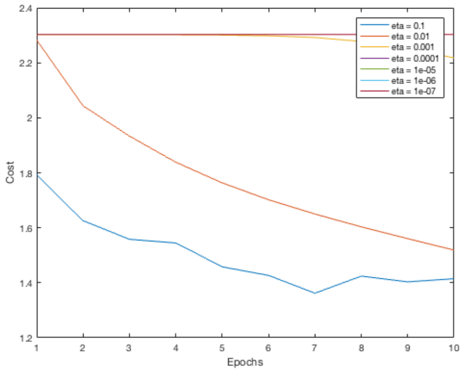

Figure 8: Training loss for different learning rate values.

  As can be seen in the figure previously attached, the value of eta greatly affects the loss obtained while training in the network. In the mentioned plot, we can conclude that when the value of the learning rate is too small; as in the cases of 1e-5,1e-6 and 1e-7, we practically do not learn; the loss is exactly the same during the first 10 epochs of the network.

  Therefore, for the next experiment, we will consider that the first range of values of learning rate must be from 0.001 to 0.1.

- **Coarse random search.**

  In this step we are going to find the best combinations for lambda and eta values, fixing rho = 0.9.

  The main intention will be performing training and then checking the learned network's best performance on the validation set; via the accuracy on it.

  The first search will be focused over the feasible learning-rates that we identified in the previous experiment with a search over a very broad range of values for lambda.

  Specifically, the range of values that we used for this first search are the next ones:

Network parameters	
rho	0.9
epochs	10
eta range values	0.001 to 0.1
lambda range values	1e-7 to 0.1
number of pairs	70

  Using the mentioned configurations, the 3 best results obtained using this first search are:

Eta	**Lambda**	**Validation accuracy**
0.011778	0.000323	43.54%
0.013877	0.000165	43.28%
0.018709	0.007597	43.20%

- **Coarse-to-fine random search to set lambda and eta.**

  During this search, we used a small range of values for lambda, to achieve better and accurate results. Specifically, we have been based on the previous best results. Then, during the coarse-to-fine random search, the parameters and the results obtained are as follows:

Network parameters	
rho	0.9
epochs	10
eta range values	0.01 to 0.07
lambda range values	1e-6 to 0.005
number of pairs	70

  The results obtained in the coarse-to-fine random search are the next ones:

Eta	**Lambda**	**Validation accuracy**
0.014001	0.003347	44.00%
0.019811	0.003421	44.00%
0.018525	0.000001	43.82%

  We can state that during this second search, we are improving the achieved results.

- **Best hyper-parameters**

  Finally, using the best hyper-parameters combination found previously, we trained the model for longer and we checked the performance obtained on the test set.

  During the training, we also break out early once the training cost is ever > 3 * original training cost.

Also, we are using only three of the training sets as training data instead of using all of them since exceeds Matlab maximum array dimensions.

Using the best hyper-parameters selection:

rho	lambda	eta
0.9	0.003347	0.014001

As a result of training the network during 30 epochs and with the previous specified hyper-parameters selection, we obtain 49.46% of accuracy in testing and 55.64% of accuracy in training data.

On the other hand,the validation and training loss achieved per each epoch, can be state in the figure:

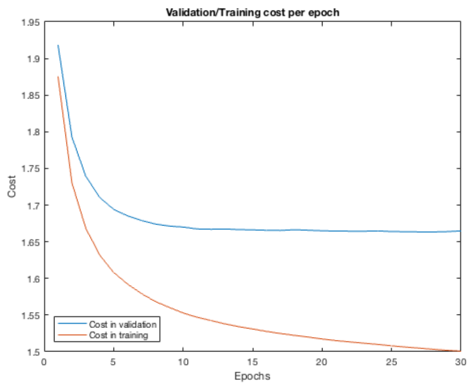

Figure 9: Final training and validation loss for 30 epochs.

3 Conclusions

During this assignment I learned about the implementation of a two-layer neural network, but above all the importance of correctly selecting the network hyper-parameters to be able to train and get better performance.

It should also be noted that without momentum, the experiments would have been much slower.

Therefore, with all the results obtained during the different experiments, we can conclude that the implementation of the 2-Layer neural network is properly implemented and verified.

Appendices

A Assignment 2 implementation:

```
%Author: Sandra Pic  Oristrell
%Data: 13 of August 2018.
%Assignment 2: 2 layer neural network.

%% Exercise 1: Read in the data and initialize the parameters of the network.
%Load the data using the function implemented in assignment 1.

[X_training, Y_training, y_training] = LoadBatch('./data_batch_1.mat');
[X_validation, Y_validation, y_validation] = LoadBatch('./data_batch_2.mat');
[X_test, Y_test, y_test] = LoadBatch('./test_batch.mat');

%Apply more pre-processing to the raw input data.
%Transform it to have zero mean.
mean_X = mean(X_training, 2);
X_training = X_training - repmat(mean_X, [1, size(X_training, 2)]);
X_validation   = X_validation   - repmat(mean_X, [1, size(X_validation,   2)]);
X_test   = X_test   - repmat(mean_X, [1, size(X_test,   2)]);

%Variables
d = size(X_training,1);
N = size(X_training,2);
K = size(Y_training,1);

%Data structure for the parameters of the network and initialize the
%values.

%50 nodes in the hidden layer
m = 50;
lambda = 0;
%Initialize the network:
[W,b] = Initialize_Network(m,d,K);

%% Exercise 2: Compute the gradients for the network parameters.

%Re-write or update gradients functions from assignment 1.
%% 2.1 Check that the gradients are implemented properly.

batch_size = 100;

[P,h] = EvaluateClassifier(X_training(:,1:batch_size),W,b);
[grad_W, grad_b] = ...
    ComputeGradients(X_training(:,1:batch_size),Y_training(:,1:batch_size),P,h,W,b,lambda);
[ngrad_W, ngrad_b] = ...
    ComputeGradsNumSlow(X_training(:,1:batch_size),Y_training(:,1:batch_size),W,b,lambda,1e-5);

%Check the comparison between the numerical and the computed gradients:
eps = 1e-10;
%Layer 1:
gradient_b1_comparison = sum(abs(ngrad_b{1} - grad_b{1})/max(eps, sum(abs(ngrad_b{1}) ...
    + abs(grad_b{1}))));
gradient_W1_comparison = sum(sum(abs(ngrad_W{1} - grad_W{1})/max(eps, ...
    sum(sum(abs(ngrad_W{1}) + abs(grad_W{1}))))));
%Layer 2:
gradient_b2_comparison = sum(abs(ngrad_b{2} - grad_b{2})/max(eps, sum(abs(ngrad_b{2}) ...
    + abs(grad_b{2}))));
gradient_W2_comparison = sum(sum(abs(ngrad_W{2} - grad_W{2})/max(eps, ...
    sum(sum(abs(ngrad_W{2}) + abs(grad_W{2}))))));
```

```
%Check that the gradients have small number.
fprintf("Results for the calculated gradients:");
fprintf("Layer 1:");
fprintf("W1: %f",gradient_W1_comparison);
fprintf("b1: %f",gradient_b1_comparison);
fprintf("Layer 2:");
fprintf("W2: %f",gradient_W2_comparison);
fprintf("b2: %f",gradient_b2_comparison);

%% 2.2 Try if you can overfit in training data with 200 epochs and reasonable learning ...
    rate.

GDparams.eta = 0.01;
GDparams.n_batch = 1;
GDparams.n_epochs = 300;
lambda = 0;

%Save all the information per each epoch.
cost_training_list = zeros(1, GDparams.n_epochs);
cost_validation_list = zeros(1, GDparams.n_epochs);
accuracy_training_list = zeros(1, GDparams.n_epochs);
accuracy_validation_list = zeros(1, GDparams.n_epochs);
epochs_list = zeros(1, GDparams.n_epochs);

%Try with only 100 examples.
examples = 100;
X_train = X_training(:,1:examples);
Y_train = Y_training(:,1:examples);
y_train = y_training(:,1:examples);

X_val = X_validation(:,1:examples);
Y_val = Y_validation(:,1:examples);
y_val = y_validation(:,1:examples);

for i = 1:GDparams.n_epochs
    [W,b] = MiniBatchGD(X_train, Y_train,y_train,GDparams,W,b,lambda);
    fprintf("Epoch: %d\n",i);
    epochs_list(i) = i;
    cost_training_list(i) = ComputeCost(X_train, Y_train, W, b, lambda);
    accuracy_training_list(i) = ComputeAccuracy(X_train,y_train,W,b);
    accuracy_validation_list(i) = ComputeAccuracy(X_val,y_val,W,b);
    cost_validation_list(i) = ComputeCost(X_val, Y_val, W, b,lambda);
end

%Plot cost in training and validation per epoch.
title_text = "Cost in validation and training dataset";
PlotCost(epochs_list, cost_validation_list, cost_training_list,title_text);

%Plot accuracy in training and validation per epoch.
title_text = "Accuracy in validation and training dataset";
PlotAccuracy(epochs_list,accuracy_validation_list, accuracy_training_list,title_text);

%Final accuracy in validation dataset and training dataset:
accuracy_final_training = ComputeAccuracy(X_train,y_train,W,b);
fprintf("Final training accuracy: %f %", accuracy_final_training);

accuracy_final_validation = ComputeAccuracy(X_val,y_val,W,b);
fprintf("Final validation accuracy: %f %",accuracy_final_validation);

%%  Exercise 3: Add momentum to your update step.
%To help speed up training times you should add momentum terms into your mini-batch ...
    update steps

GDparams.eta = 0.001;
```

```
GDparams.n_batch = 100;
GDparams.n_epochs = 10;
lambda = 0;

eta_decay_rate = 0.95;

%Compare rho values = [0.5,0.9,0.99];
rho_values = [0.50,0.9,0.99];

for rho = rho_values
    [W,b] = Initialize_Network(m,d,K);
    GDparams.eta = 0.01;
    cost_training_list = zeros(1, GDparams.n_epochs);
    cost_validation_list = zeros(1, GDparams.n_epochs);
    accuracy_training_list = zeros(1, GDparams.n_epochs);
    accuracy_validation_list = zeros(1, GDparams.n_epochs);
    epochs_list = zeros(1, GDparams.n_epochs);
    for i = 1:GDparams.n_epochs
        [W,b] = MiniBatchGD_withMomentum(X_training, ...
            Y_training,y_training,GDparams,W,b,lambda,rho);
        fprintf("Epoch: %d\n",i);
        epochs_list(i) = i;
        cost_training_list(i) = ComputeCost(X_training, Y_training, W, b, lambda);
        accuracy_training_list(i) = ComputeAccuracy(X_training,y_training,W,b);
        accuracy_validation_list(i) = ComputeAccuracy(X_validation,y_validation,W,b);
        cost_validation_list(i) = ComputeCost(X_validation, Y_validation, W, b,lambda);
        GDparams.eta  = GDparams.eta * eta_decay_rate;
    end
    %Plot cost in training and validation per epoch.
    title_text = "Cost validation/training with rho = " + num2str(rho);
    PlotCost(epochs_list, cost_validation_list, cost_training_list,title_text);
    %Plot accuracy in training and validation per epoch.
    title_text = "Accuracy validation/training with rho = " + num2str(rho);
    PlotAccuracy(epochs_list,accuracy_validation_list, accuracy_training_list,title_text);
    %Final accuracy in validation dataset and training dataset:
    accuracy_final_training = ComputeAccuracy(X_training,y_training,W,b);
    fprintf("Final training accuracy: %f %", accuracy_final_training);
    accuracy_final_validation = ComputeAccuracy(X_validation,y_validation,W,b);
    fprintf("Final validation accuracy: %f %",accuracy_final_validation);
end

%% Exercise 4: Training your network
% All the experiments will be using all the examples.

%%  4.1 Find reasonable values for the learning rate:


%Regularization term to small value:
lambda = 0.000001;
rho = 0.9;
eta_values = [0.1,0.01,0.001,0.0001,0.00001,0.000001,0.0000001];
GDparams.n_batch = 100;
GDparams.n_epochs = 10;

final_cost_training_list = {};
for eta = eta_values
    GDparams.eta = eta;
    cost_training_list = zeros(1, GDparams.n_epochs);
    [W,b] = Initialize_Network(m,d,K);
    fprintf("Eta value: %f\n",eta);
    for i = 1:GDparams.n_epochs
        fprintf("Epoch: %d\n",i);
        [W,b] = MiniBatchGD_withMomentum(X_training, ...
            Y_training,y_training,GDparams,W,b,lambda,rho);
        cost_training_list(i) = ComputeCost(X_training, Y_training,W,b,lambda);
```

```
    end
    final_cost_training_list{end+1} = cost_training_list;
end
epochs_list = zeros(1,GDparams.n_epochs);
for i = 1: GDparams.n_epochs
    epochs_list(i) = i;
end
%Plot the cost per each case of eta.
PlotCost_Eta_values(epochs_list, eta_values, final_cost_training_list);

%% 4.2 Coarse-search random.

GDparams.n_batch = 100;
GDparams.n_epochs = 10;
decay_rate= 0.95;
rho=0.9;
n_pairs= 70;
%Learning rate range:
eta_max = 0.1;
eta_min = 0.001;
%Lambda range:
lambda_max = 0.1;
lambda_min = 1e-7;

validation_accuracy_list = zeros(1,n_pairs);
eta_values_list = zeros(1,n_pairs);
lambda_values_list = zeros(1,n_pairs);

for j = 1:n_pairs
    fprintf("\n Pair number: %d \n ", j);
    eta_exp = log10(eta_min) + (log10(eta_max) - log10(eta_min))*rand(1, 1);
    eta = 10^eta_exp;
    lambda_exp = log10(lambda_min) + (log10(lambda_max) - log10(lambda_min))*rand(1, 1);
    lambda = 10^lambda_exp;

    GDparams.eta = eta;
    [W,b] = Initialize_Network(m,d,K);

    for i=1: GDparams.n_epochs
        fprintf("Epoch: %d\n", i);
        [W, b] = MiniBatchGD_withMomentum(X_training, Y_training,y_training, GDparams, ...
            W, b, lambda, rho);
        GDparams.eta = GDparams.eta*decay_rate;
    end
    validation_accuracy_list(j) = ComputeAccuracy(X_validation,y_validation,W,b);
    eta_values_list(j) = GDparams.eta;
    lambda_values_list(j) = lambda;
end

%Sort the array:
[validation_accuracy_list,validation_indexs] = sort(validation_accuracy_list,'descend');
eta_values_list = eta_values_list(validation_indexs);
lambda_values_list = lambda_values_list(validation_indexs);

%Write file:
filename = 'exercise_4.2_coarse_search.txt';
fid = fopen(filename,'wt');
fprintf(fid,"\nAccuracy values:\n");
fprintf(fid,'%f\t',validation_accuracy_list);
fprintf(fid,"\nEta values:\n");
fprintf(fid,'%f\t',eta_values_list);
fprintf(fid,"\nLambda values:\n");
fprintf(fid,'%f\t',lambda_values_list);
fprintf(fid,"\n");
fclose(fid);
```

```
%% 4.2 Coarse-fine search random.
%Better values for lambda search.

GDparams.n_batch = 100;
GDparams.n_epochs = 10;
decay_rate= 0.95;
rho=0.9;
n_pairs= 70;
%Learning rate range:
eta_max = 0.07;
eta_min = 0.01;
%Lambda range:
lambda_max = 0.005;
lambda_min = 1e-6;

validation_accuracy_list = zeros(1,n_pairs);
eta_values_list = zeros(1,n_pairs);
lambda_values_list = zeros(1,n_pairs);

for j = 1:n_pairs
    fprintf("\n Pair number: %d \n ", j);
    eta_exp = log10(eta_min) + (log10(eta_max) - log10(eta_min))*rand(1, 1);
    eta = 10^eta_exp;
    lambda_exp = log10(lambda_min) + (log10(lambda_max) - log10(lambda_min))*rand(1, 1);
    lambda = 10^lambda_exp;

    GDparams.eta = eta;
    [W,b] = Initialize_Network(m,d,K);

    for i=1: GDparams.n_epochs
        fprintf("Epoch: %d\n", i);
        [W, b] = MiniBatchGD_withMomentum(X_training, Y_training,y_training, GDparams, ...
            W, b, lambda, rho);
        GDparams.eta = GDparams.eta*decay_rate;
    end
    validation_accuracy_list(j) = ComputeAccuracy(X_validation,y_validation,W,b);
    eta_values_list(j) = GDparams.eta;
    lambda_values_list(j) = lambda;
end

%Sort the array:
[validation_accuracy_list,validation_indexs] = sort(validation_accuracy_list,'descend');
eta_values_list = eta_values_list(validation_indexs);
lambda_values_list = lambda_values_list(validation_indexs);

%Write file:
filename = 'exercise_4.2_coarse_fine_search.txt';
fid = fopen(filename,'wt');
fprintf(fid,"\nAccuracy values:\n");
fprintf(fid,'%f\t',validation_accuracy_list);
fprintf(fid,"\nEta values:\n");
fprintf(fid,'%f\t',eta_values_list);
fprintf(fid,"\nLambda values:\n");
fprintf(fid,'%f\t',lambda_values_list);
fprintf(fid,"\n");
fclose(fid);

%% 4.3 Best hyper-parameters for training the network.
%Best combination

m= 50;

%Train with all the training data:
[X_training1, Y_training1, y_training1] = LoadBatch('./data_batch_1.mat');
```

```
[X_training2, Y_training2, y_training2] = LoadBatch('./data_batch_2.mat');
[X_training3, Y_training3, y_training3] = LoadBatch('./data_batch_3.mat');
[X_training4, Y_training4, y_training4] = LoadBatch('./data_batch_4.mat');
[X_training5, Y_training5, y_training5] = LoadBatch('./data_batch_5.mat');

%Test dataset
[X_test, Y_test, y_test] = LoadBatch('./test_batch.mat');

%except 1000 samples for validation: using the validation dataset
%(data_batch_2.mat)
X_validation = X_training2(:, 1:1000);
Y_validation = Y_training2(:, 1:1000);
y_validation = y_training2(:, 1:1000);
X_training2 = X_training2(:, 1001:10000);
Y_training2 = Y_training2(:, 1001:10000);
y_training2 = y_training2(:, 1001:10000);

%All training data together:
X_training = [X_training1, X_training2, X_training3];
Y_training = [Y_training1, Y_training2, Y_training3];
y_training = [y_training1, y_training2, y_training3];

%Pre-processing:
mean_X = mean(X_training, 2);
X_training = X_training - repmat(mean_X, [1, size(X_training, 2)]);
X_validation   = X_validation   - repmat(mean_X, [1, size(X_validation,   2)]);
X_test  = X_test  - repmat(mean_X, [1, size(X_test,  2)]);

%Variables
d = size(X_training,1);
N = size(X_training,2);
K = size(Y_training,1);

GDparams.n_batch = 100;
GDparams.n_epochs = 30;
decay_rate= 0.95;
rho= 0.9;
%Best combination:
GDparams.eta = 0.014001;
lambda = 0.003347;

%Plot the training and validation cost after each epoch of training
cost_training_list = zeros(1, GDparams.n_epochs);
cost_validation_list = zeros(1, GDparams.n_epochs);
[W,b] = Initialize_Network(m,d,K);
%Original training cost
original_training_cost = ComputeCost(X_training, Y_training,W,b,lambda);
for i=1:GDparams.n_epochs
    [W, b] = MiniBatchGD_withMomentum(X_training, Y_training,y_training, GDparams, W, ...
        b, lambda, rho);
    cost_training_list(i) = ComputeCost(X_training, Y_training,W,b,lambda);
    cost_validation_list(i) = ComputeCost(X_validation, Y_validation,W,b,lambda);
    GDparams.eta = GDparams.eta*decay_rate;
    if cost_training_list(i) > 3*original_training_cost
        fprintf("Cost_training(i) > 3*original_training_cost");
        i = GDparams.n_epochs;
    end
end

epochs_list = zeros(1, GDparams.n_epochs);
for i=1:GDparams.n_epochs
   epochs_list(i) = i;
end

%Plot the loss per epochs in validation and in training:
```

```
title_text = "Validation/Training cost per epoch";
PlotCost(epochs_list, cost_validation_list, cost_training_list,title_text)

%Final accuracy in testing and training:
final_accuracy_test = ComputeAccuracy(X_test,y_test,W,b);
fprintf("Final accuracy in test: %f % \n", final_accuracy_test);
final_accuracy_training = ComputeAccuracy(X_training,y_training,W,b);
fprintf("Final accuracy in training: %f % \n", final_accuracy_training);

%% Functions implementation


function PlotCost_Eta_values(epochs_list, eta_values, final_cost_training_list)
    figure;
    for i = 1:length(eta_values)
        text_label = "eta = " + num2str(eta_values(i));
        plot(epochs_list,final_cost_training_list{i},'DisplayName',text_label);
        hold on;
    end
    hold off;
    xlabel('Epochs');
    ylabel('Cost');
    legend;
end

%Plot accuracy for validation/test - training dataset per epoch.
function PlotAccuracy(epochs, accuracy_validation, accuracy_training,title_text)
    figure;
    plot(epochs,accuracy_validation,epochs,accuracy_training);
    title(title_text);
    xlabel('Epochs');
    ylabel('Accuracy %');
    legend('Accuracy in validation', 'Accuracy in training','Location','southwest');
end

%Plot the loss/cost for validation/test -training dataset per epoch.
function PlotCost(epochs, cost_validation, cost_training,title_text)
    figure;
    plot(epochs,cost_validation,epochs,cost_training);
    title(title_text);
    xlabel('Epochs');
    ylabel('Cost');
    legend('Cost in validation', 'Cost in training','Location','southwest');
end

%Function to initialize the values of the Weight matrices.
function [W,b] = Initialize_Network(m,d,K)
    W1 = randn(m,d)*0.001;
    W2 = randn(K,m)*0.001;
    b1 = zeros(m, 1);
    b2 = zeros(K, 1);
    W = {W1,W2};
    b = {b1, b2};
end

function [X,Y,y] = LoadBatch(filename)
    A = load(filename);
    X = double(A.data');
    X= X/255;
    y = double(A.labels') + 1;
    vec = ind2vec(y);
    Y = full(vec);      %One-hot representation
end

%Function to compute the cost of the 2-layer network.
```

```
function J = ComputeCost(X,Y,W,b,lambda)
    [P,h] = EvaluateClassifier(X,W,b);
    crossentropy_term = sum(diag(-log(double(Y)'*P)));
    reg_term = sum(W{1}(:).^2)+sum(W{2}(:).^2);
    J = (1/(size(X,2))*crossentropy_term)+(lambda*reg_term);
end

function [P,h] = EvaluateClassifier(X,W,b)
    s1 = W{1}*X + b{1};
    h = max(0, s1);
    s = W{2} * h + b{2};
    P = exp(s)./sum(exp(s));
end

%Function to compute the accuracy in a 2-layer neural network:
function acc = ComputeAccuracy(X,y,W,b)
    [P,h] = EvaluateClassifier(X, W, b);
    [¬,pred_idx] = max(P);
    acc=((sum(pred_idx==y)/size(X,2)*100));
end

function [grad_W, grad_b] = ComputeGradients(X, Y, P, h, W,b,lambda)
    k_x = size(X,2);
    W1 = cell2mat(W(1));
    b1 = cell2mat(b(1));
    W2 = cell2mat(W(2));
    b2 = cell2mat(b(2));
    grad_W1 = zeros(size(W1));
    grad_W2 = zeros(size(W2));
    grad_b1 = zeros(size(b1));
    grad_b2 = zeros(size(b2));

    for i= 1:k_x
        P_i = P(:, i);
        h_i = h(:, i);
        Y_i = Y(:, i);
        X_i = X(:, i);
        g = -(Y_i-P_i)';
        grad_b2 = grad_b2 + g';
        grad_W2 = grad_W2 + g'*h_i';

        h_i(find(h_i > 0)) = 1;
        g = g*W2*diag(h_i);

        grad_b1 = grad_b1 + g';
        grad_W1 = grad_W1 + g'*X_i';
    end
    grad_b1 = grad_b1/k_x;
    grad_W1 = grad_W1/k_x + 2*lambda*W1;
    grad_b2 = grad_b2/k_x;
    grad_W2 = grad_W2/k_x + 2*lambda*W2;

    grad_b = {grad_b1, grad_b2};
    grad_W = {grad_W1, grad_W2};
end

function [grad_W, grad_b] = ComputeGradsNumSlow(X, Y, W, b, lambda, h)
    grad_W = cell(numel(W), 1);
    grad_b = cell(numel(b), 1);
    for j=1:length(b)
        grad_b{j} = zeros(size(b{j}));
        for i=1:length(b{j})
            b_try = b;
            b_try{j}(i) = b_try{j}(i) - h;
            c1 = ComputeCost(X, Y, W, b_try, lambda);
```

```
                b_try = b;
                b_try{j}(i) = b_try{j}(i) + h;
                c2 = ComputeCost(X, Y, W, b_try, lambda);
                grad_b{j}(i) = (c2-c1) / (2*h);
            end
        end
        for j=1:length(W)
            grad_W{j} = zeros(size(W{j}));
            for i=1:numel(W{j})
                W_try = W;
                W_try{j}(i) = W_try{j}(i) - h;
                c1 = ComputeCost(X, Y, W_try, b, lambda);
                W_try = W;
                W_try{j}(i) = W_try{j}(i) + h;
                c2 = ComputeCost(X, Y, W_try, b, lambda);
                grad_W{j}(i) = (c2-c1) / (2*h);
            end
        end
    end


    %Mini batch function implementation without momentum and learning rate
    %decay.
    function [Wstar, bstar] = MiniBatchGD(X,Y,y,GDparams,W,b,lambda)
        N = size(X,2);
        eta = GDparams.eta;
        n_batch = GDparams.n_batch;

        for j=1:N/n_batch
            j_start = (j-1)*n_batch + 1;
            j_end = j*n_batch;
            indx = j_start:j_end;
            Xbatch = X(:,indx);
            Ybatch = Y(:,indx);
            [P,h] = EvaluateClassifier(Xbatch,W,b);
            [grad_W, grad_b] = ComputeGradients(Xbatch,Ybatch,P,h, W,b,lambda);
            W{1} = W{1} - eta*grad_W{1};
            W{2} = W{2} - eta*grad_W{2};
            b{1} = b{1} - eta*grad_b{1};
            b{2} = b{2} - eta*grad_b{2};
        end
        bstar = b;
        Wstar = W;
    end

    function [v_W, v_b] = InitializeMomentum(W,b)
        v_b = {zeros(size(b{1})), zeros(size(b{2}))};
        v_W = {zeros(size(W{1})), zeros(size(W{2}))};
    end

    function [Wstar, bstar] = MiniBatchGD_withMomentum(X, Y,y,GDparams,W,b,lambda,rho)
        N = size(X,2);
        eta = GDparams.eta;
        n_batch = GDparams.n_batch;
        [v_W,v_b] = InitializeMomentum(W,b);

        for j=1:N/n_batch
            j_start = (j-1)*n_batch + 1;
            j_end = j*n_batch;
            indx = j_start:j_end;
            Xbatch = X(:,indx);
            Ybatch = Y(:,indx);
            [P,h] = EvaluateClassifier(Xbatch,W,b);
            [grad_W, grad_b] = ComputeGradients(Xbatch,Ybatch,P,h, W,b,lambda);
            %Update with momentum:
```

```
        v_W{1} = rho*v_W{1} + eta*grad_W{1};
        v_b{1} = rho*v_b{1} + eta*grad_b{1};
        v_W{2} = rho*v_W{2} + eta*grad_W{2};
        v_b{2} = rho*v_b{2} + eta*grad_b{2};
        W{1} = W{1} - v_W{1};
        b{1}  = b{1} - v_b{1};
        W{2} = W{2} - v_W{2};
        b{2}  = b{2} -  v_b{2};
    end
    bstar = b;
    Wstar = W;
end
```