KTH Royal Institute of Technology

DD2424
Deep Learning in Data Science

Year 2017 - 2018

Assignment 1

Picó Oristrell Sandra 940531 – 2308 sandrapo@kth.se

Professors :
Sullivan Josephine

April 11, 2018

Contents

The main intention of this assignment is to train and test a one layer network to classify images from the CIFAR-10 dataset. The algorithm used is a 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.

In order to explain and show the results obtained in a proper way I will divide this report into different subsections. The first section will be focused on the background of the assignment where I will explain the main functions of the classifier and the data that we are analyzing. To continue, I will specified the functions implemented. As I am doing the assignment in Matlab, the functions have the same structure as in the assignment description.All of them are attached in the appendix section. Finally I will report all the results obtained in the exercise 1.

1 Background

As mentioned above, in this section I will explain a little bit about the background needed for this assignment.

1.1 Mathematical background:

The network that we are going to implement will be a classification problem where the output will be a vector of probabilities.

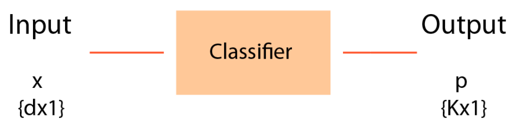


Figure 1: Main structure of the network used

Then, the predicted class will correspond to the label with higher probability, i.e the *arg max* of the probability's vector. Nevertheless, in order to compute the output vector we will use the following formulas:

$$s = Wx + b \tag{1}$$

$$\mathbf{p} = \frac{exp(s)}{1^T exp(s)} \tag{2}$$

This means that during the training phase we will learn the parameters *W* and *b*. To set this parameters we will solve the following optimization problem:

$$W^*, \mathbf{b}^* = argmin_{W,b} J(D, \lambda, W, b) \tag{3}$$

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

$$J(D, \lambda, W, b) = \frac{1}{\mid D \mid} \sum_{x,y \in D} \log(y^T p) + \lambda \sum_{i,j} W_{ij}^2 \tag{4}$$

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

1.2 Data structure:

As mentioned before, 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:

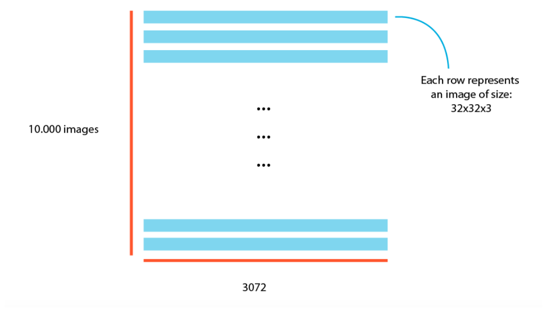


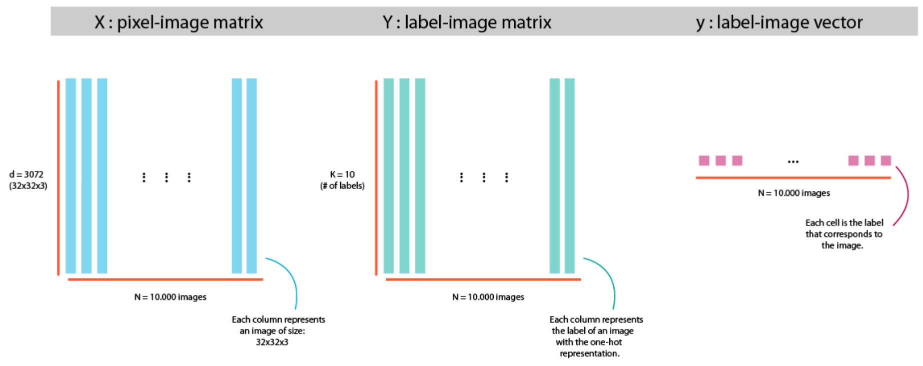
Figure 2: Data structure of CIFAR-10 dataset

2 Implementation of the functions

To be able to train the multi-linear classifier we will need to write some sub-functions. The purpose of this section is to explain them. Nevertheless, you can find the code of each attached in the appendix section.

- **function [X,Y,y] = LoadBatch(filename)**

  This function reads a filename and returns the image and the label data in the following format:



X : pixel-image matrix
Y : label-image matrix
y : label-image vector
d = 3072 (32x32x3)
N = 10.000 images
Each column represents an image of size: 32x32x3
K = 10 (# of labels)
N = 10.000 images
Each column represents the label of an image with the one-hot representation.
N = 10.000 images
Each cell is the label that corresponds to the image.

Figure 3: X,Y and y structure

- **function P = EvaluateClassifier(X,W,b)**

  This function generates the P matrix. As explained above, the P matrix will be a matrix where each column will show the probability for each label for a specific image.

- **function J = ComputeCost(X,Y,W,b,lambda)**

  This function computes the cost based on the cross-entropy loss and the regularization term on W. This function will be called when we implement the gradient descent algorithm.

- **function acc = ComputeAccuracy(P,y)**

  To be able to compute the accuracy of our classifier we need to count how many outputs are correct , comparing with the truth labels and compute the percentage.

- **function [grad_W,grad_b] = ComputeGradients(X,Y,P,W,lambda)**

In this function we are going to compute the analytic gradients. The calculus are based on the lecture's slides. This gradients are first checked through the numerical gradients computed in the ComputeGradsNumSlow.

- **function [Wstar,bstar] = MiniBatchGD(X,Y,GDparams,W,b,lambda)**

  Finally, this function will implement the vanilla version of the mini-batch gradient descent algorithm. This algorithm will learn the network parameters: W and b. The main structure of the code is specified in the description of the assignment.

  GDparams is an object that contains the parameter values needed for the MiniBatchGD function: n_batch,eta and n_epochs.

3 Exercise 1:

3.1 Check the gradients

The first top exercise is to be sure that the analytical gradients are properly calculated. In order to do so, I decide to compute the relative error specified in the description. This error is computed as follows:

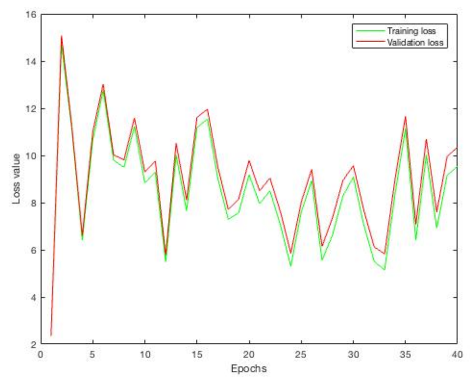
$$\frac{\mid g_a - g_n \mid}{max(eps, \mid g_a \mid + \mid g_n \mid)} \tag{5}$$

where g_n is the numerically computed gradient and g_a is the analytically computed gradient. If the relative error obtained is $<$ 1e-6 then we are sure that the implementation is correct. The implementation can be found in the appendix section (Assignment1_1).

3.2 Loss, accuracy and weights visualization

Once I was sure that the gradients were correctly implemented I plot the accuracy, the loss and the weights visualization per each configuration specified in the assignment description. The implementation can be found in the appendix section (Assignment1_2). The obtained results are the following ones:

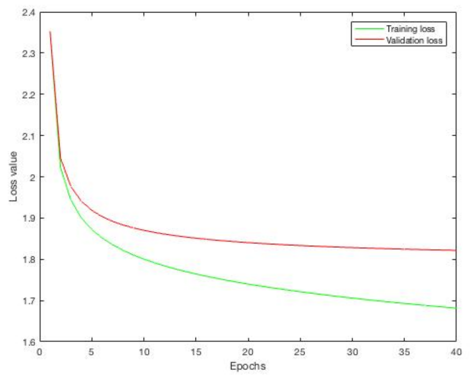
- **No-Regularization:**
  - **lambda=0, n epochs=40, n batch=100, eta=.1**
    - *Loss:*



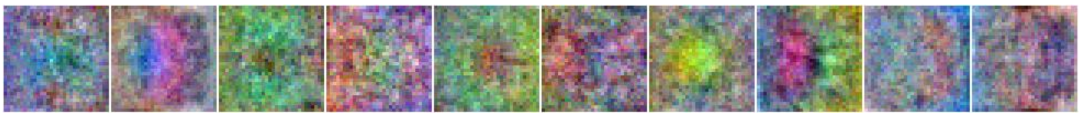
- - *Weights:*

    - *Accuracy:* Training Accuracy:31.59% Test Accuracy:26.2%

  - **lambda=0, n epochs=40, n batch=100, eta=.01**
    - *Loss:*

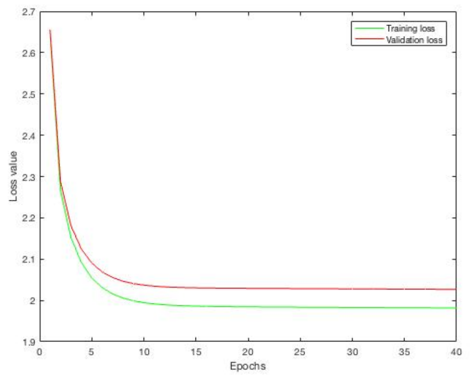


    - *Weights:*



    - *Accuracy:* Training Accuracy:41.56% Test Accuracy:36.87%

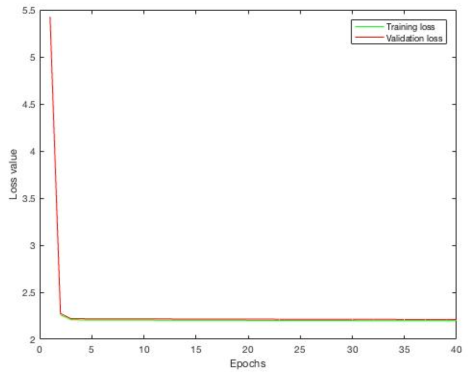
- **Regularization:**
  - **lambda=.1, n epochs=40, n batch=100, eta=.01**
    - *Loss:*



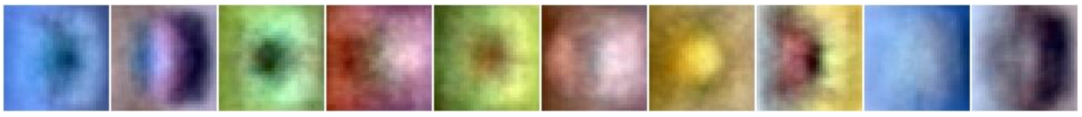
    - *Weights:*

    - *Accuracy:*Training Accuracy:34.18% Test Accuracy:33.37%

  - **lambda=1, n epochs=40, n batch=100, eta=.01**
    - *Loss:*



    - *Weights:*



    - *Accuracy:*Training Accuracy:22.24% Test Accuracy:21.92%

3.3 Conclusions

Regarding the results obtained in the previous section we can state two main conclusions regarding the learning rate and the regularization term. A high learning rate implies that the loss cost function specified will converge faster. However, that also means that it will be unstable. This behavior can be found in the first loss graph. Better performance is achieved with a low learning rate.

On the other hand, regularization term implies reducing the over-fitting effect. In the two last cases we have seen that the space between validation loss and training loss is less than in the two cases where we did not have regularization. As we don't have over-fitting,we have better generalization.

Appendices

A Functions implementation:

A.1 LoadBatch function

```
function[X,Y,y] = LoadBatch(filename)
    %Open the dataset
    addpath Datasets/;
    dataset = load(filename);
    %X contains the image pixel data. Size 3072x10000. Double type between
    %0 and 1.
    %Reshape the pixel data into a 3072x10000 array
    I = reshape(dataset.data.',32,32,3,10000);
    I = permute(I,[2,1,3,4]);
    X = reshape(I,[3072,10000]);
    %Normalize the vector
    X = double(X);
    X = X./255.0;

    %y is a vector of 10000 containing the label for each image.
    %better to encode it between 1-10 instead of 0-9.
    y = dataset.labels;
    for i = 1:10000
        y(i,1) = y(i,1) + 1;
    end

    %Y is a 10x10000 array with the one_hot representation.
    %Extract the labels dataset
    Y = dataset.labels;
    %One-hot representation
    num_labels = 10;     %Number of labels
    Y_one_hot = zeros(size(Y,1),num_labels);
    Y_one_hot = reshape(Y_one_hot,[10,10000]); %Reshape the one_hot representation.
    for i = 1:10000
        row = Y_one_hot(:,i);
        label = Y(i,:);
        row(label+1) = 1;
        Y_one_hot(:,i) = row;
    end
    Y = Y_one_hot;

return
```

A.2 EvaluateClassifier function

```
function P = EvaluateClassifier(X,W,b)
    %Be sure that it is double.
    W = double(W);
    X = double(X);
    b = double(b);
    %Softmax to generate a vector with the probability of each label.
    s = W*X + b;
    K = size(b);
    K = K(1);
    %Compute the P matrix.
    %Each column of P represents the probability of each label /image.
    P = exp(s)./((ones(1,K,'double'))*exp(s));
return
```

A.3 ComputeCost function

```
function J = ComputeCost(X,Y,W,b,lambda)
    %Size values
    K = 10;
    %In this case, N will represent the D subset.
    [d,N] = size(X);

    %Compute P
    W = double(W);
    X = double(X);
    b = double(b);
    s = W*X + b;
    K = size(b);
    K = K(1);
    P = exp(s)./((ones(1,K,'double'))*exp(s));

    %Cross-Entropy loss
    crossEntropy_term = 0;
    for i = 1:N
        y = Y(:,i);
        p = P(:,i);
        py = y.'*p;
        crossEntropy_term = crossEntropy_term -log(py);
    end

    %Regularization term.
    reg_term = 0;
    for row = 1:K
        for column = 1:d
            reg_term = reg_term + W(row,column)^2;
        end
    end
    reg_term = lambda*reg_term;

    %Calculate J.
    J = 1./abs(N)*crossEntropy_term + reg_term;

return
```

A.4 ComputeAccuracy function

```
function acc = ComputeAccuracy(P,y)
    %size P = KxN
    [K,N] = size(P);
    %size y = Nx1
    %To store the prediction using the P matrix.
    pred = zeros(N,1,'double');
    %Compute the result using the P matrix.
    for i = 1: N
        [argvalue, argmax] = max(P(:,i));
        pred(i,1) = argmax;
    end
    correct = 0;
    for i = 1: N
```

```
        if pred(i,1) == y(i,1)
            correct = correct + 1;
        end
    end
    acc = (correct/N)*100;
return
```

A.5 ComputeGradients function

```
function [grad_W,grad_b] = ComputeGradients(X,Y,P,W,lambda)
    % grad_W: gradient matrix of the cost J respect to W.
    % grad_b: gradient vector of the cost J respect to b.

    %Initialize the gradients with the respective sizes.
    grad_W = zeros(size(W));
    grad_b = zeros(size(W, 1), 1);

    %Compute gradients from the slides (lectures).
    for i = 1 : size(X, 2)
        Xi = X(:, i);
        Pi = P(:, i);
        Yi = Y(:, i);
        g = -Yi'/(Yi'*Pi)*(diag(Pi) - Pi*Pi');
        grad_b = grad_b + g';
        grad_W = grad_W + g'*Xi';
    end

    %Normalize the gradients.
    grad_b = grad_b/size(X, 2);
    grad_W = grad_W/size(X, 2) + (2*lambda)*W;
end
```

A.6 MiniBatchGD function

```
function [Wstar,bstar] = MiniBatchGD(X,Y,GDparams,W,b,lambda)
    %Size of X is : dxN
    N = size(X, 2);

    %GDparams is an object containing the parameter values n_batch, eta and
    %n_epochs.
    eta = GDparams.eta;
    n_batch = GDparams.n_batch;
    GDparams.n_epochs;

    %Structure of the assignment description.
    for j=1:N/n_batch
        j_start = (j-1)*n_batch + 1;
        j_end = j*n_batch;
        inds = j_start:j_end;
        Xbatch = X(:, inds);
        Ybatch = Y(:, inds);
        %EvaluateClassifier to be able to compute the gradients at each
        %mini-batch computation.
        P = EvaluateClassifier(Xbatch, W, b);
        [grad_W, grad_b] = ComputeGradients(Xbatch, Ybatch, P, W, lambda);
        %Update the values based with the learning-rate.
        W = W - eta*grad_W;
        b = b - eta*grad_b;
    end
    %Parameters that we would like to return
    Wstar = W;
    bstar = b;
end
```

B Exercise 1 implementation:

B.1 Assignment 1.1:

```
function Assignment1_1
    %%Exercise 1.1
    %Extract data from the datafiles.
    addpath Datasets/;
    training_name = 'data_batch_1.mat';
    test_name = 'test_batch.mat';
    validation_name = 'data_batch_2.mat';
    %X,Y and y data for each file.
    [X_train,Y_train,y_train] = LoadBatch(training_name);
    [X_test,Y_test,y_test] = LoadBatch(test_name);
    [X_validation,Y_validation,y_validation] = LoadBatch(validation_name);

    %Initialize the parameters of the model W and b. [size(W) = 10x3072
    %size(b) = 10x1]
    %Initialize each entry to have Gaussian random values with 0 mean and
    %standard deviation 0.01.
    c = 0;
    a = 0.01;
    K = size(Y_train,1);
    d = size(X_train,1);
    b = a.*randn(K,1)+c;
    W = a.*randn(K,d)+c;

    %Parameters to check:
    lambda = 0;
    batch = 100;
    %Small positive number
    eps = 1e-10;

    %Compute the numerically gradients.
    [grad_b_n, grad_W_n] = ComputeGradsNumSlow(X_train(:, 1 : batch), Y_train(:, 1 : ...
        batch), W, b, lambda, 1e-6);
    %Generate P to be able to compute the gradients with our function
    %ComputeGradients.
    P = EvaluateClassifier(X_train(:, 1 : batch),  W, b);
    %Analytical gradients.
    [grad_W, grad_b] = ComputeGradients(X_train(:, 1 : batch), Y_train(:, 1 : batch), ...
        P,  W, lambda);
    %Compare them to be sure that the gradients are computed properly.
    %Compare them through the computation of the relative error.
    %|ga-gn| / (max(eps,|ga|+|gn|)) where eps is a small positive number.
    %Check this is small.
    gradient_W = max(max(abs(grad_W_n - grad_W)./max(eps, abs(grad_W_n) + abs(grad_W))));
    gradient_b = max(abs(grad_b_n - grad_b)./max(eps, abs(grad_b_n) + abs(grad_b)));

    %Check if gradients are computed correctly.
    %Small number choosed: 1e-6 (specified in the description of the
    %assignment)

    %Gradient W:
```

```
    if gradient_W < 1e-6
        fprintf("Correct gradient W!");
    else
        fprintf("Incorrect gradient W!");
    end
    %Gradient b:
    if gradient_b < 1e-6
        fprintf("Correct gradient b!");
    else
        fprintf("Incorrect gradient b!");
    end
end
```

B.2 Assignment 1.2:

```
function Assignment1_2
    %Now we know for sure that the gradients calculations are ok.
    %Exercise 1.2 : Training a multi-linear classifier.
    addpath Datasets/;
    training_name = 'data_batch_1.mat';
    test_name = 'test_batch.mat';
    validation_name = 'data_batch_2.mat';
    %X,Y and y data for each file.
    [X_train,Y_train,y_train] = LoadBatch(training_name);
    [X_test,Y_test,y_test] = LoadBatch(test_name);
    [X_validation,Y_validation,y_validation] = LoadBatch(validation_name);

    %Initialize the parameters of the model W and b. [size(W) = 10x3072
    %size(b) = 10x1]
    %Initialize each entry to have Gaussian random values with 0 mean and
    %standard deviation 0.01.
    c = 0;
    a = 0.01;
    K = size(Y_train,1);
    d = size(X_train,1);
    b = a.*randn(K,1)+c;
    W = a.*randn(K,d)+c;

    %n_batch, eta,n_epochs,lambda aprameters
    lambda = 0;
    GDparams.eta = 0.1;
    GDparams.n_batch =100;
    GDparams.n_epochs =40;

    %Calculate the training and validation loss.
    Loss_validation = zeros(1,GDparams.n_epochs);
    Loss_train = zeros(1,GDparams.n_epochs);
    for j=1: GDparams.n_epochs
        Loss_validation(j) = ComputeCost(X_validation, Y_validation, W, b, lambda);
        Loss_train(j) = ComputeCost(X_train, Y_train, W, b, lambda);
        [Wstar, bstar] = MiniBatchGD(X_train, Y_train, GDparams, W, b, lambda);
        W=Wstar;
        b=bstar;
    end
    %Plot the training and the validation loss.
    %Assignment description : around 2.0...
    figure()
    plot(1 : GDparams.n_epochs, Loss_train,'g')
    hold on
    plot(1 : GDparams.n_epochs, Loss_validation,'r')
    hold off
    xlabel('Epochs');
    ylabel('Loss value');
    legend('Training loss', 'Validation loss');

    %After training, just compute the accuracy of your learnt classifer on
    %the test data. (Assignemnt description = 36.39%)

    P = EvaluateClassifier(X_train,W,b);
    accuracy_train = ComputeAccuracy(P,y_train);
    disp(['Training Accuracy:' num2str(accuracy_train) '%'])
    P = EvaluateClassifier(X_test,W,b);
    accuracy_test = ComputeAccuracy(P, y_test);
    disp(['Test Accuracy:' num2str(accuracy_test) '%'])

    %Visualization of the weight matrix as an image.
    K = 10;
    for i = 1 : K
        im = reshape(W(i, :), 32, 32, 3);
        s_im{i} = (im - min(im(:))) / (max(im(:)) - min(im(:)));
        s_im{i} = permute(s_im{i}, [2, 1, 3]);
    end
    figure()
    montage(s_im, 'size', [1, K])

    %Do the same with the following configurations:
    %lambda = 0, n_epochs = 40, n_batch = 100, eta = 0.1
    %lambda = 0, n_epochs = 40, n_batch = 100, eta =0.01
    %lambda = 0.1, n_epochs =40, n_batch = 100, eta = 0.01
    %lambda = 1, n_epochs = 40, n_batch = 100, eta = 0.01
end
```