Assignment 2

Diego González Morín

diegogm@kth.se

1. Objective

The main objective of this assignment is to train and test a two layer network with multiple outputs for image classification. The training will be done using the Mini-Batch Gradient descent algorithm computing a cost function minimization. This cost function computes the cross-entropy loss of the classifier, applied to the labelled training data, and a L2 regularization term on the weight matrix. The dataset we will use for training, validation and test is the CIFAR-10 dataset.

2. Method

The assignment is divided in three parts: a first part in which the different functions that were written for the first lab will be adapted for this 2 layer network, a second part that, that will experiment with different values of eta and lambda to find the optimal combination to use in the training, and a last part in which the network will be trained and tested.

2.1. Function implementation and verification

The first step then was to adapt all the functions implemented in the first lab to work on a two layer network. This functions are:

- `function [X, Y, y] = LoadBatch(filename)` – takes a input filename of a dataset file, loads the file, extract the pixels data in a matrix with dimensions d(dimension of each image)xN(number of images), the one-hot representation of the images with dimension K(number of labels)xN(number of images) and the vector y that contains the labels for every image N.
- `function P = EvaluateClassifier(X, W, b)` – Computes the probability of each label for each image, using the weight matrix, the vector b and the input data. The size of P is K(number of labels)xn(number of pictures in the input data).
- `function J = ComputeCost(X, Y, W, b, lambda)` – Computes the cost/loss of the network depending on the weight matrix (W) the vector b, the regularization term (lambda), the one-hot representation(Y) and the input data. This function is based on the equations available in the lecture notes. The output is a scalar.
- `function acc = ComputeAccuracy(X, y, W, b)` – Computes the proportion of images of the test data that were correctly classified by the net, using the input data(X), the labels for image in the set(y), the weight matrix(W) and the vector b.
- `function [grad_W, grad_b] = ComputeGradients(X, Y, P, W, lambda)` – Computes the gradients for the Weight matrix W and the vector b. The implementation of this method is based on the operations available in the lecture notes. We have that grad_W has size K(labels)xd(dimension of each image) and grad_b has dimension K(labels).

- `function [Wstar, bstar, val_loss, train_loss] = MiniBatchGD(X, Y, X_val, Y_val, GDparams, W, b, lambda, display)` – Trains the input Weights and b vector using the Mini Batch Gradient descent algorithm. To do so, it computes the probability matrix, then the gradients, and the W and b are then updated accordingly to this gradients and the learning rate eta. It did not use the validation data to avoid over fitting, however the addition of this parameter would improve the performance of the network.

2.2. ComputeGradients function debugging.

This is the key function, as the Gradient descent is based in this function and its debugging can be hard. To successfully debug this function, the recommended option was to compare the performance of the function (analytical gradient calculation) with the results obtained with the numerical gradient computation. If the difference was small enough (< 1e-8) we could consider that the function worked well. We can observe in Figure 1, the representation of the difference between the analytical and numerical computations. Is easy to see how this error remains low enough for all the mini batches used.

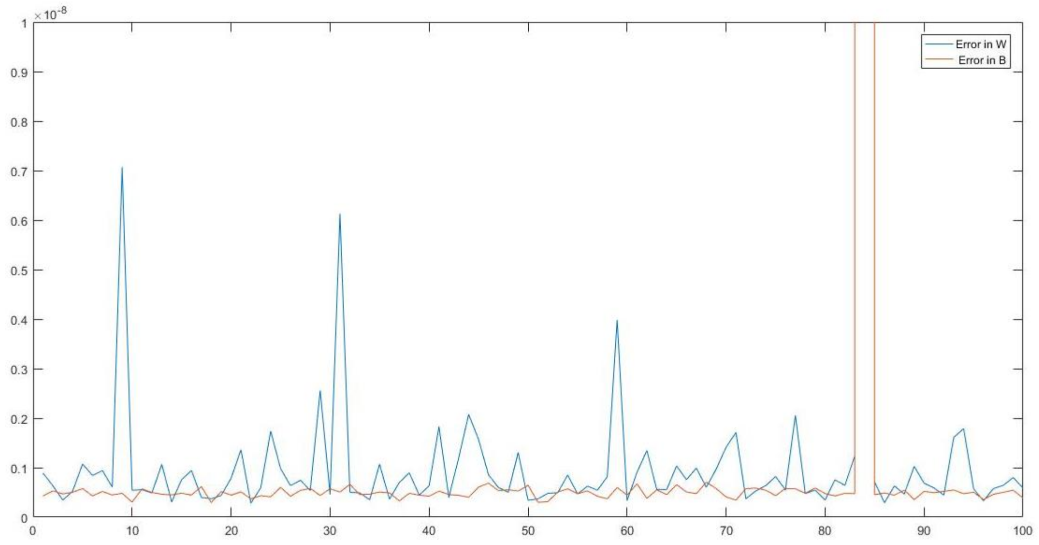

Figure 1: Error between the analytic and the numerical computations of the gradients of the Weight matrix W and bias vector b.

We can observe that the error remains smaller than 1E-8 except in one term. We can assume that this is an isolated error and the GradientsComputations is working well.

However, and as it was recommended in the lab notes, we double checked its performance by training the network on a small amount of training data and check if we can overfit to the training data and get a very low loss after around 200 epochs. In Figure 2 we can observe how the network overfits the training data, as the training loss is very low while the validation loss increases considerably.

The network overfits completely to the training data as the training loss is almost zero and the validation loss increases considerably. It can be assumed that, after satisfying both requirements (numerical compare and network overfitting), the gradients are well implemented.

2.3. Adding the momentum term.

One more improvement that has to be added in this network are the momentum terms in the mini-batch update in order to speed up the training times. After this momentum term was added, another sanity check was performed: the network was trained to fit the training data as it was done before, setting the rho values to {.5, .95, .99}. The implementation was correct as we can see in figure 3 how the network overfits to the training data, and now the learning is faster. Note that a decay for the eta term was added too.

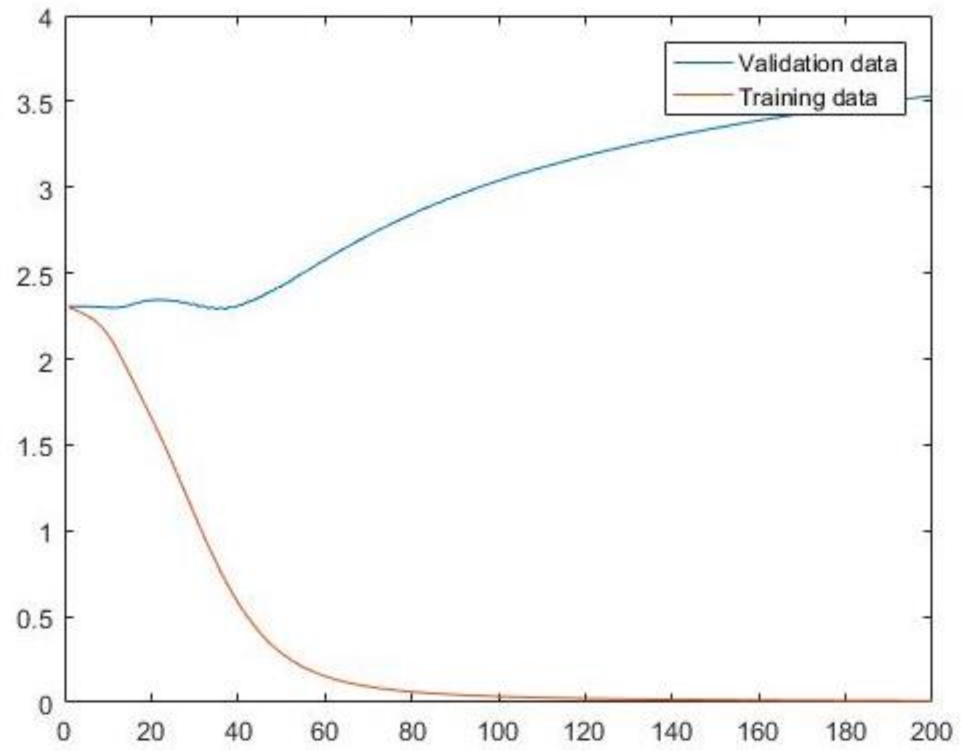

Figure 2: Validation and training losses after training a small subset of the first batch (100 images) for 200 epochs

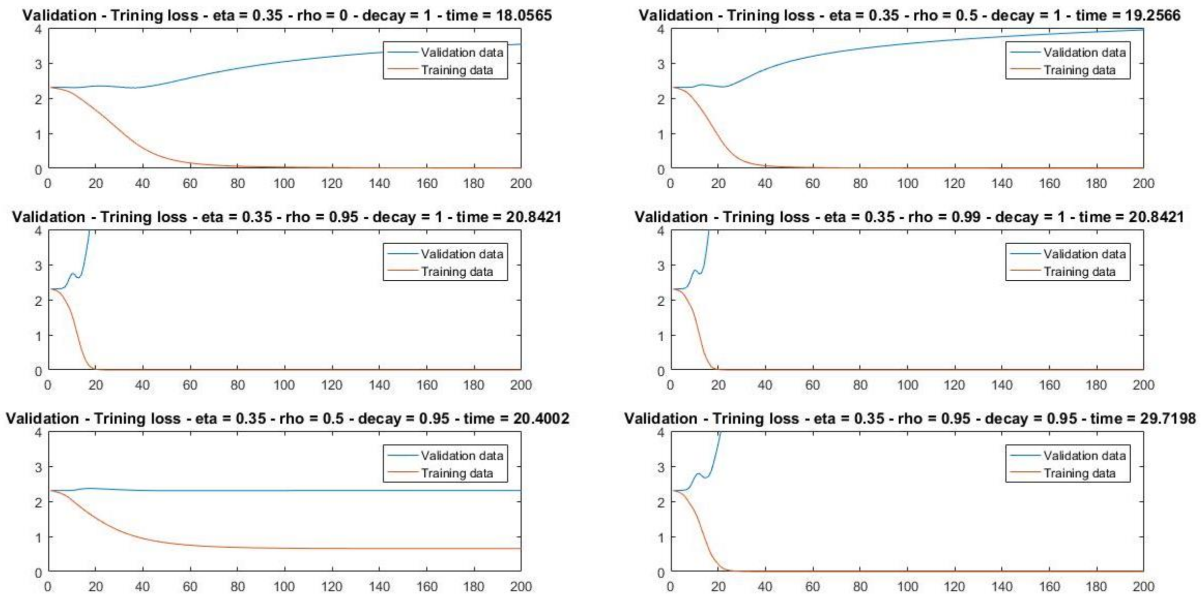

Figure 3: Validation and training losses for different combinations of rho and eta decay.

We can observe in Figure 3, that after the addition of the momentum term, the learning is quicker as it overfits sooner. We can also observe that the higher the rho, the quicker it overfits. Is also important to observe how the learning speed decreases if we add the eta decay.

3. Parameter tuning

Once the functions were correctly working, and the GradientComputation function's performance, that compute the gradients of the Weight matrix and Bias vector, was checked, the next step was to tune the main parameters of the network. During the parameter tuning, the training data was the entire data batch (data_batch_1.mat).

3.1. Find a reasonable range of values for the learning rate

The first step in the process of parameters tuning is to find a reasonable range of values for the learning rate. To do so, we are going to set the regularization term to a very small value (.000001). After the network is initialized randomly using Xavier's initialization, the next step is to perform a quick search by hand to find the rough bounds for reasonably values of the learning rate. After several experiments, the training loss is shown in Figure 4. This training losses were calculated with the data in the first batch (data_batch_1.mat) and only 2 epochs, as this number of epochs was enough to estimate a coarse range for the etas values.

In figure 4 we can observe that for eta = 0.01, the evolution of the loss is considerably slower compared to the rest of loss. Also, from eta >= 0.11 we are getting values of the training loss that stop decreasing after 2 epochs. This is way, the chosen range of feasible values of eta were 0.02 - 0.1.

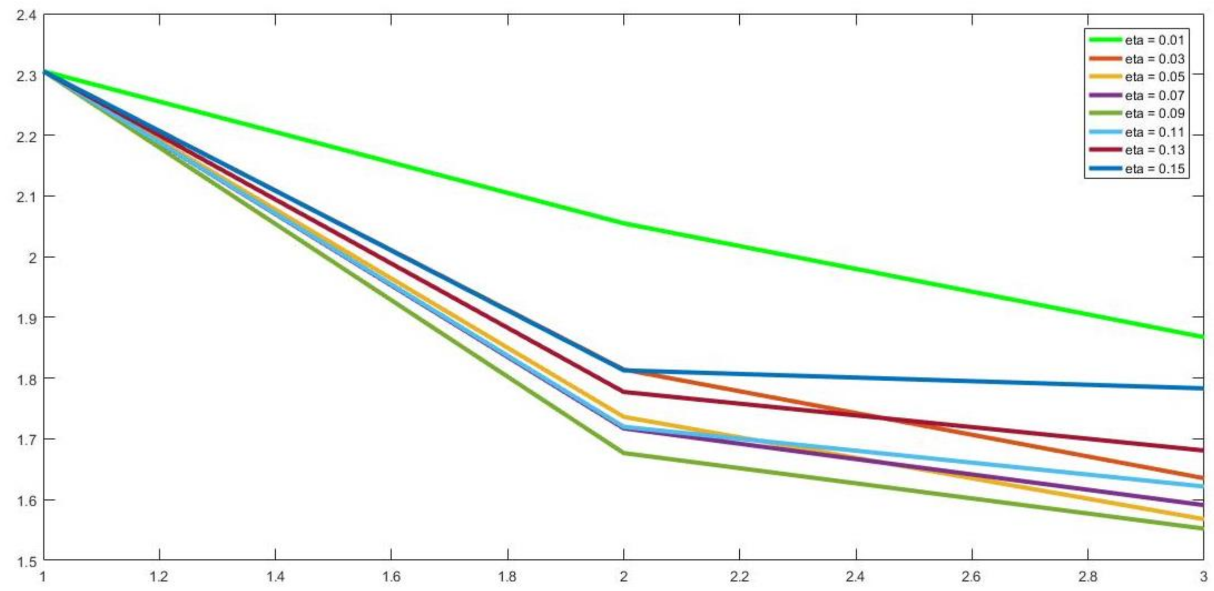

Figure 4: Training loss after training a network for 2 epochs for different values of eta and with lambda fixed to 0.000001.

3.2. Coarse to fine random search to set lambda and eta

After the range of reasonable values of eta is estimated, the next step is to fine tune both lambda and eta parameters. To do this, a random search will be performed. After the network is randomly initialized using Xavier initialization, it is trained several times for different parameters of eta and lambda. The etas will be chosen randomly within the estimated range of feasible-rates of eta in tandem with a search over a very broad range of values of lambda. To make sure that all the range of values was included when randomly pick the etas, the estimated range of feasible-rates of eta was made wider. Therefore, the eta used in this first random search was eta = 0.005 – 0.2 instead of eta = 0.02 – 0.1 calculated. As we are doing a random tandem search, each sample of eta and lambda will be randomly picked using the following expression:

```
e = e_min + (e_max – e_min)*rand(1,1);
```

To estimate the tandem of optimal parameters, 60 random picked pairs were used, to train the randomly initialized network (with the same seed always), during only one epoch. Only one epoch was used as this is still a coarse search, and this was a quick way to make the ranges of eta and lambda narrower. All the scores both of the validation loss and the accuracy of the network on the validation data were saved. The pair of parameters the gave the better results (higher accuracies or lower losses) were stored. Then, the higher and lower boundaries of eta and lambda that gave the first 15 better results were computed and stored. This range of optimal parameters was eta = 5.38E-3 – 0.1442 , lambda = 1.2927E-4 - 0.0087 . This boundary values of the parameters were used to repeat the process with a narrower range, to try to get closer to the optimal parameters. As previously, we are using a slightly wider range, to make sure that all the possible optimal parameters are chosen during the random search. We used: eta = 5.38E-4 – 14.42 , lambda = 1.2927E-5 - 0.0087 This second random search was done with 60 pairs of parameters during 3 epochs. Only two epoch more were used as the main improvements in the accuracy when training the network with only one batch of data occur during the first 3 epochs. In this case, the range calculated was eta = 0.0319 – 0.0858 , lambda = 3.8382E-4 - 0.0061, being the optimal pairs of parameters the ones shown in table 1. To compute this final range, only the best 8 combinations were chosen.

Eta_range	Lambda_range	Epochs	Number of pairs trained/considered	Eta estimated	Lambda Estimated
0.02 – 0.1	0.0005 – 0.1	1	60/15	0.00538-0.141	0.000129- 0.0087
0.00053 – 0.141	0.000013 – 0.0087	3	60/8	0.0319-0.0858	0.000383-0.0061

Table 1: Ranges for Eta and Lambda

3.3. Find the 3 best combination of parameters

The objective in this last section of the parameters tuning is to find the 3 best combination the 8 optimal tandems. To do so, 8 randomly initialized networks (using Xavier's initialization) will be trained using each of the 8 pairs of parameters calculated in the step before during 10 epochs. The value of the parameter pairs, the number of epochs, and final accuracy on the test data is shown in the table 2. Rho and Eta_decay has fixed values of 0.9 and 0.95 respectively.

Eta	Lambda	Epochs	Accuracy on test data
0.0319	0.0043	10	44.57%
0.0447	0.0037	10	44.35%
0.0501	0.0024	10	44.11%
0.0386	0.0057	10	44.11%
0.390	0.0061	10	43.84%
0.0613	0.0015	10	42.32%
0.0858	0.0004	10	41.38%
0.0683	0.0010	10	40.82%

Table 2: Accuracy on the test data after training for 10 epochs for different combinations of eta and lambda.

We have that the best combinations of eta-lambdas are:

- Eta = 0.0319 – Lambda = 0.0043
- Eta = 0.0447 – Lambda = 0.0037
- Eta = 0.0501 – Lambda = 0.0024

In figures 5 we can see the training and validation loss evolution during the training of a networks during 10 epochs with the best pair of parameters.

4. Training

After the best tandem of parameters was calculated, a new randomly initialized network was trained using this parameters and all the training data available. The dataset was then organized as follows:

- Trainning set : data_batch_1.mat, data_batch_3.mat, data_batch_4.mat, first 9000 images of data_batch_2.mat.
- Validation set: last 1000 images of data_batch_2.mat.
- Test set: test_batch.mat

The best performances were obtained with the pair of paramters eta = 0.0319 and lambda = 0.0043, with rho, as before, set to 0.9 and the eta decay to 0.95. The network was trained for 30 epochs, obtaining a final accuracy on the test set of 50.12%. In figure 6 we can see the training and validation loss during training.

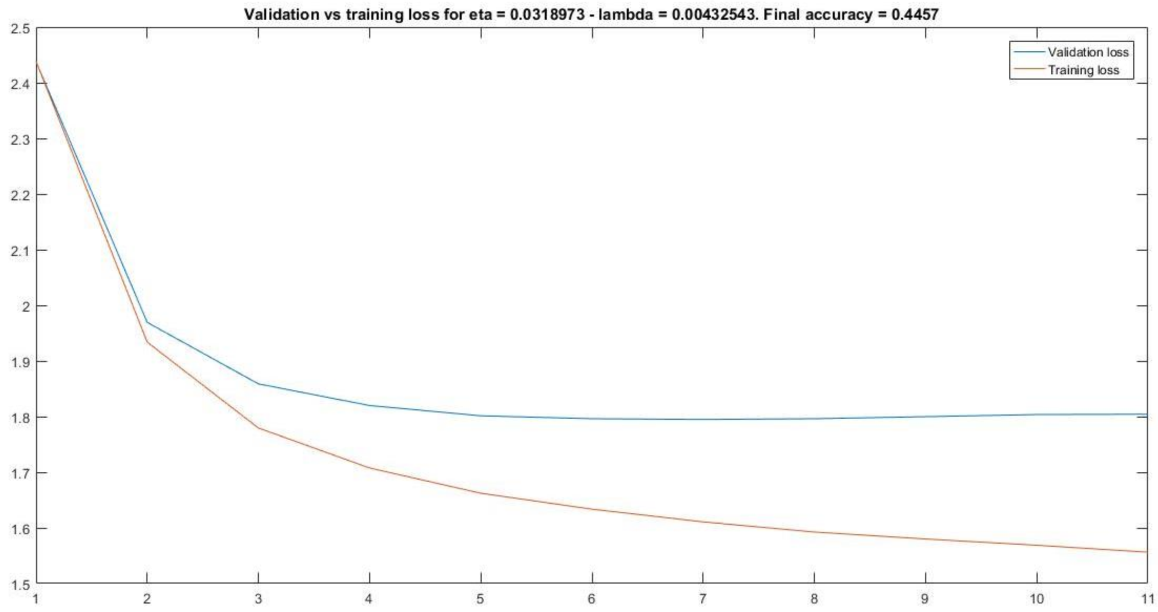

Figure 5: Validation and training loss during the training over 10 epochs, for eta = 0.0318 and lambda 0.00432 using only one batch for training.

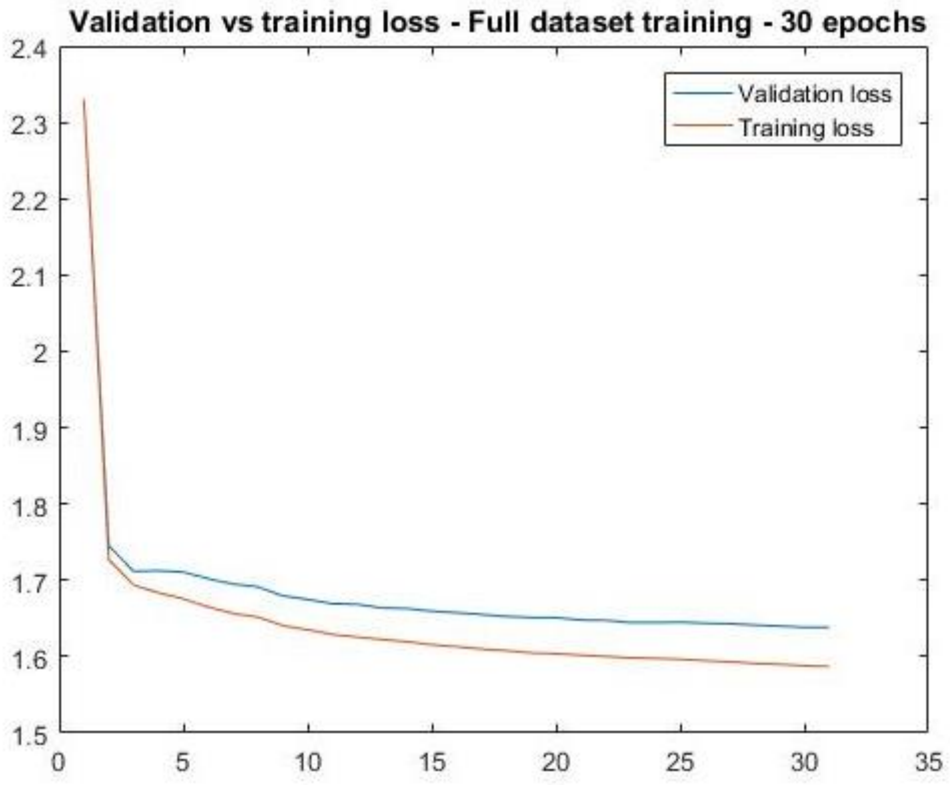

Figure 6: Validation and training loss during the training over 10 epochs, for eta = 0.0318 and lambda 0.00432 using all the available batches for training.