Deep Learning in Data Science, DD2424

Short report on lab assignment 1

Mattia Evangelisti

April 5, 2023

Contents

1 Introduction

This assignment is about building, training, and testing a one-layer network with multiple outputs to classify images from the `CIFAR-10` dataset.
I used `data batch 1` for training, `data batch 2` for validation, and `test batch` for testing. The neural network has one input layer (as many nodes as image pixels) and one output layer (as many nodes as possible labels). Cross-entropy loss function and L2 regularization were used as well.

2 Preprocessing

The first step was to load the data and transform them into a matrix of doubles of size dxn where `n` is the number of images (10000) and `d` the dimensionality of each image (32x32x3). Labels were also loaded and represented in one-hot encoding as well.

After that, the raw input data (training and validation) was preprocessed to obtain zero mean.

Then I implemented other support functions:

- `initializeWeights(input_size, output_size)` to randomly initialize the weight matrix sampling from a Gaussian with zero mean and standard deviation equal to 0.01.
- `evaluateClassifier(X, W, b)` that calculates the probability of the labels for the current network function.
- `computeCost(X, Y, W, b, lambda)` that compute the cost function for a set of images, i.e., it implements:

$$J(D, \lambda, W, b) = \frac{1}{|D|} \sum_{(x,y)\in D} l_{cross}(x, y, W, b) + \lambda \sum_{ij} W_{ij}^2 \tag{1}$$

- `computeAccuracy(X, y, W, b)` which computes the accuracy of the network's predictions.

3 Training and Evaluation

I implemented the function to analytically compute the gradients of the cost function w.r.t W and b, i.e. the function implements the functions:

$$\frac{\partial J(\beta^{(t+1)}, \lambda, W, b)}{\partial W} = \frac{1}{|\beta^{t+1}|} \sum_{(x,y)\in\beta^{(t+1)}} \frac{\partial l_{cross}(x, y, W, b)}{\partial W} + 2\lambda W \tag{2}$$

$$\frac{\partial J(\beta^{(t+1)}, \lambda, W, b)}{\partial b} = \frac{1}{|\beta^{t+1}|} \sum_{(x,y)\in\beta^{(t+1)}} \frac{\partial l_{cross}(x, y, W, b)}{\partial b} \tag{3}$$

Then I checked that the results obtained by the function correspond to the correct gradients. To do this, I compared the obtained results (analytical method) with the gradients computed with

the finite difference model (numerical method). The weights were initialized in the same way for both methods and the results were checked using both absolute and relative errors in two different scenarios:

1. first 20 dimensions of the first image of the training dataset without regularization:
    - For weights: 100.0% of absolute errors below 1e-6 and the maximum absolute error is 1.5863768881851925e-08
    - For bias: 100.0% of absolute errors below 1e-6 and the maximum absolute error is 4.648442673838016e-08
    - For weights: 100.0% of relative errors below 1e-6 and the maximum relative error is 1.330840327540334e-07
    - For bias: 100.0% of relative errors below 1e-6 and the maximum relative error is 2.259799157508011e-07
2. first 5 images of the training dataset with regularization:
    - For weights: 100.0% of absolute errors below 1e-6 and the maximum absolute error is 4.817284007807565e-08
    - For bias: 100.0% of absolute errors below 1e-6 and the maximum absolute error is 5.950048174302447e-08
    - For weights: 93.80533854166667% of relative errors below 1e-6 and the maximum relative error is 0.015300343829114159
    - For bias: 100.0% of relative errors below 1e-6 and the maximum relative error is 4.968814168759987e-07

After I noticed the obtained results were in line with the numeric ones, I wrote the code to perform the mini-batch gradient descent algorithm to learn the network's parameters. As suggested in the text I randomly shuffled the training data before each epoch.
On the next page are the learning curves (cross-entropy loss and accuracy) for the required scenarios.

We can notice how a large learning rate (first case) leads to instability and the loss function does not seem to converge. Adding the regularization we reduce the gap between training and validation curves and increase the steadiness of the curves, but we have worse performances. This is because L2 regularization is not particularly useful when we have a simple model and a large dataset.

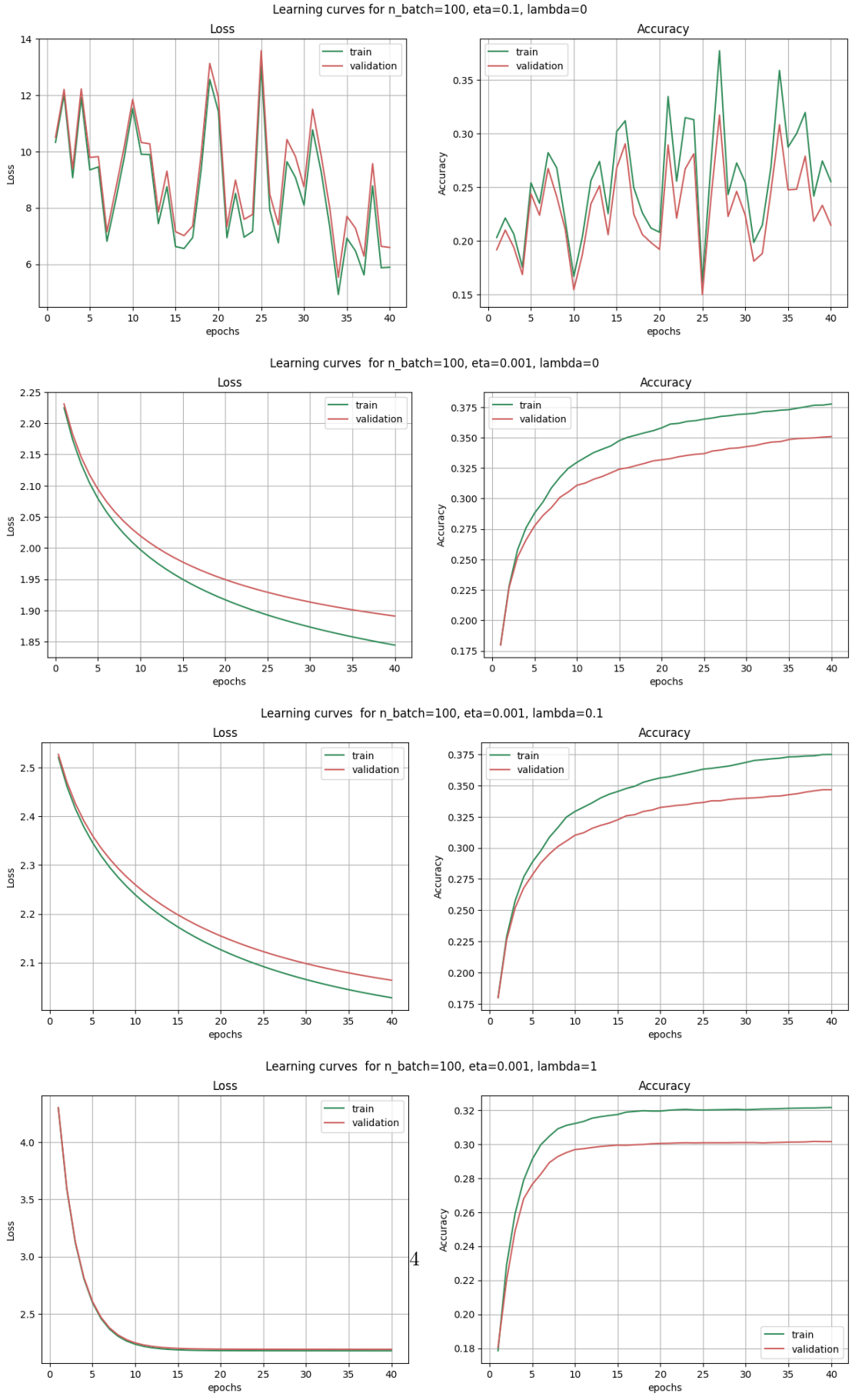

Figure 1: Learning curves for the 4 scenarios

Below we can what the learned weight matrices look like and we can notice how the regularization forces the weights to be closer to the center f the distribution (see Figure 4) and hence we obtain better images. When we have regularization we are even able to distinguish some images (for example automobile image)

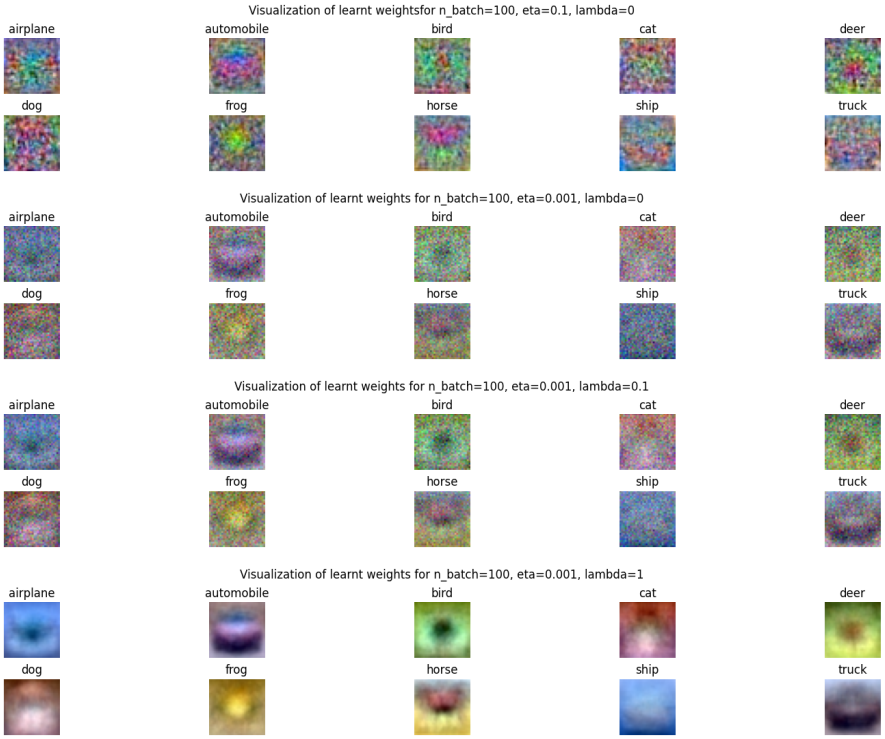

Figure 2: Learnt weight matrix for the 4 scenarios

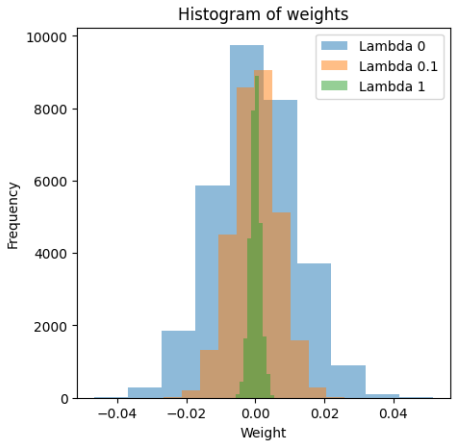

Figure 3: Caption

Finally, when measuring the accuracy on the test set I obtained the following results:

Parameters	Accuracy
`lambda=0, n_epochs=40, n_batch=100, eta=0.1`	21.23%
`lambda=0, n_epochs=40, n_batch=100, eta=0.001`	36.52%
`lambda=0.1, n_epochs=40, n_batch=100, eta=0.001`	35.59%
`lambda=1, n_epochs=40, n_batch=100, eta=0.001`	31.84%

Table 1: Test accuracies of the different parameters configurations

4 Bonus Points

4.1 Improve performance of the network

In this section, I decided to implement three possible tricks to increase the performances and they will be tested against the parameters configuration that scored the best result in the previous part:
`lambda=0, n_epochs=40, n_batch=100, eta=0.001`

4.1.1 a) Use all available data for training

In the first point, I tried and succeeded to improve the network's accuracy by using all available data for training, decreasing the size of the validation set to 1000 samples. The performance is improved since more data has been used for training.

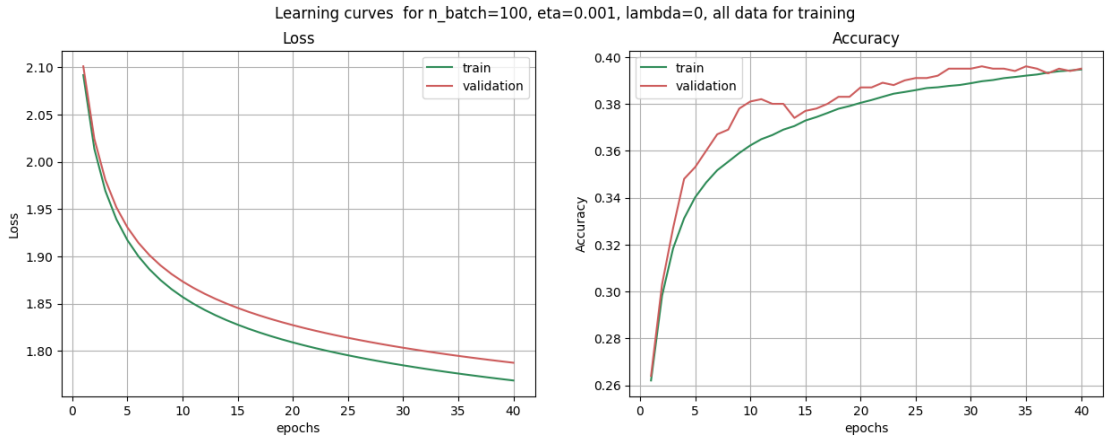

Figure 4: Learning curves with all data used for training

4.1.2 c) Grid search

In the second point, I performed a grid search in order to find the best combination of the following parameters: `eta, n_batch, lambda`. On the next page are reported the results of the grid search. As before, we can notice how the best results are for lambda = 0, i.e., without weights regularization. The obtained results are not the best possible, since for sake of time the grid search was run on a restricted set of combinations. Better results could be obtained by using a wider range of values for each parameter.

	lambda	eta	n_batch	loss_train	loss_val
3	0.00	0.00100	20.0	1.969848	2.070894
8	0.00	0.01000	200.0	1.977117	2.079010
4	0.00	0.00100	100.0	2.018722	2.091686
7	0.00	0.01000	100.0	1.984793	2.100608
12	0.01	0.00100	20.0	2.010443	2.106006
17	0.01	0.01000	200.0	2.017852	2.113668
5	0.00	0.00100	200.0	2.056273	2.118618
13	0.01	0.00100	100.0	2.054765	2.126772
16	0.01	0.01000	100.0	2.025494	2.132118
14	0.01	0.00100	200.0	2.090138	2.152029
6	0.00	0.01000	20.0	2.040420	2.164963
21	0.10	0.00100	20.0	2.093716	2.168369
26	0.10	0.01000	200.0	2.101416	2.174992
15	0.01	0.01000	20.0	2.085351	2.191003
25	0.10	0.01000	100.0	2.117525	2.194618
22	0.10	0.00100	100.0	2.153544	2.219389
0	0.00	0.00001	20.0	2.225953	2.246756
24	0.10	0.01000	20.0	2.179487	2.248327
9	0.01	0.00001	20.0	2.256673	2.277444
1	0.00	0.00001	100.0	2.282790	2.284759
2	0.00	0.00001	200.0	2.293876	2.292194
23	0.10	0.00100	200.0	2.240572	2.298996
10	0.01	0.00001	100.0	2.313481	2.315450
11	0.01	0.00001	200.0	2.324586	2.322904
18	0.10	0.00001	20.0	2.511931	2.532431
19	0.10	0.00001	100.0	2.585312	2.587284
20	0.10	0.00001	200.0	2.598765	2.597092

Figure 5: Grid search results

4.1.3 d) Decaying the learning rate

In this last point, I applied the technique called step decay, which consists in decaying the learning rate by a factor of 10 after every *nth* epoch or when the validation seems to plateau. For this test, I decided to start with a high learning rate (eta = 0.1) in order to better visualize the effect of the learning rate on the network's performances:

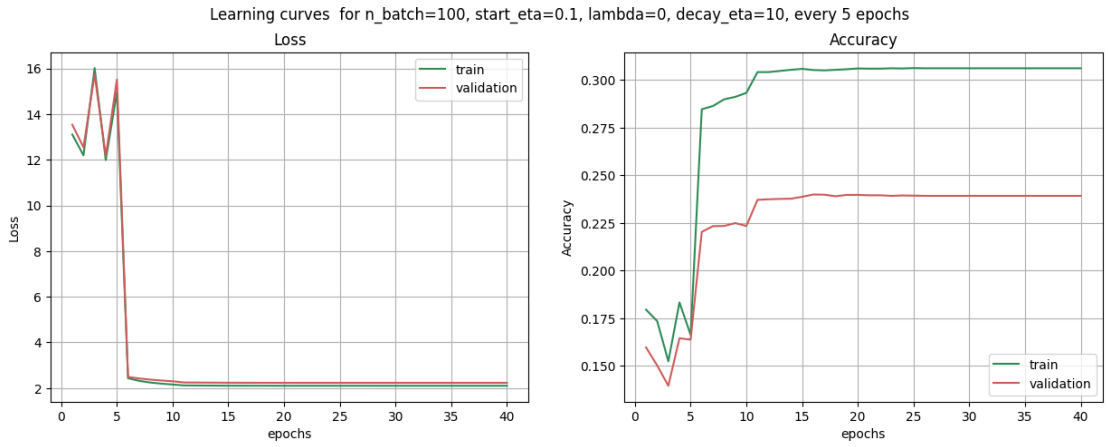

Figure 6: Caption

In the image above we can notice how the decay does not boost our network performances with respect to using a constant 0.001 learning rate. This could be more useful with more training epochs and a smaller decrease factor.

Overall the best performance was obtained using all data for training:
accuracy = 38.75%

4.2 Train network - multiple binary cross-entropy losses

In this last section, we are asked to implement a K binary cross-entropy loss function:

$$l_{multiplebce}(x, y) = -\frac{1}{K}\sum_{k=1}^{K}[(1 - y_k)log(1 - p_k) + y_k log(p_k)] \tag{4}$$

and its derivative results in:

$$\begin{aligned} \frac{\partial l_{cross}}{\partial p_i} = -\frac{y_i}{p_i} + \frac{1 - y_i}{1 - p_i} &= \frac{p_i - y_i}{p_i(1 - p_i)} \\ \frac{\partial p_i}{\partial s_i} &= p_i * (1 - p_i) \\ \frac{\partial l_{cross}}{\partial s_i} &= \frac{\partial l_{cross}}{\partial p_i} * \frac{\partial p_i}{\partial s_i} \\ &= p_i - y_i \end{aligned} \tag{5}$$

hence we can conclude that the gradient computation result is identical to the categorical cross-entropy and there is no need to rewrite the code.

Below is the test accuracy for the network when using binary cross-entropy loss.

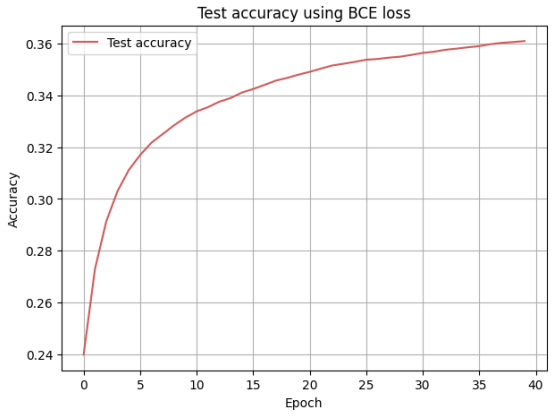

Figure 7: Test accuracy using binary cross-entropy loss

Comparing this result with the ones obtained in Table 1, we can notice how the overall accuracy is the same when solving this specific task. The precise final accuracy for categorical loss is 36.52%, while for the binary cross-entropy, I obtained a 36.27%, so we can conclude that the general performance is the same.

Finally, I want to say that during the training, the binary cross-entropy function showed a smaller gap between training and validation performance, hence I would affirm that the cross-entropy model is less prone to over-fit.

For the last point, I report below the histograms of the probability for the ground truth class for the examples correctly and those incorrectly classified.

Figure 8: Classification histograms for softmax cross-entropy (left) and binary cross-entropy (right)