Deep Learning in Data Science, DD2424

Short report on lab assignment 2

Mattia Evangelisti

April 19, 2023

Contents

1 Introduction

This assignment is about building, training, and testing a two-layers 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 Read the Data & Initialize the Parameters of the Network

In the first part of this assignment, I followed the same steps as in the first one:

- I loaded the data and transformed 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.
- I calculated the mean and the standard deviation and normalized the training, validation, and test data to obtain zero mean data.
- I wrote a function to initialize the weights. Different from the first assignment, we now have a network with one hidden layer and consequently two bias vectors and two weight matrices. I initialized the biases to zero and the weights are randomly drawn from a Gaussian distribution with zero mean and standard deviation `1/sqrt(q)` for layer 1 and `1/sqrt(m)` for layer 2, where `m` is the number of nodes in the hidden layer.

3 Compute the gradients for the network parameters

In this section, I adapted the functions from Assignment 1 in order to make them work with a two layers network.
More specifically I adapted the `evaluateClassifier` function in order to make it work with two layers, i.e., this function is now computing both the final `P` value and the intermediary activation values.
Using the same logic I also updated the function `computeGradients`, which now calculates the biases and weights gradients for each layer of the network.
The other support function, such as the ones to calculate loss and cost functions have been slightly modified as well to work in a two-layers network.
Then I checked the correct functioning of my analytical calculation of the gradients. To do so I compared the obtained results with the ones obtained by calculating the gradients with the finite difference methods.

After running both functions with a fixed seed and using only the first 20 samples of the training data, I obtained the following results:

```
For W1: 100.0% of absolute errors below 1e-6 and maximum absolute error is 2.501802741083914e-07
For b1: 100.0% of absolute errors below 1e-6 and maximum absolute error is 1.6988281989482878e-07
For W2: 100.0% of absolute errors below 1e-6 and maximum absolute error is 2.4640263258587924e-09
For b2: 100.0% of absolute errors below 1e-6 and maximum absolute error is 4.642751090661035e-07


For W1: 96.6% of relative errors below 1e-6 and maximum relative error is 7.989086681842825e-06
For b1: 90.0% of relative errors below 1e-6 and maximum relative error is 6.584976578136217e-06
For W2: 100.0% of relative errors below 1e-6 and maximum relative error is 1.6573063279013553e-07
For b2: 10.0% of relative errors below 1e-6 and maximum relative error is 2.261867019633793e-06
```

Figure 1: Errors between numerical and analytical gradients calculation

After having checked the correctness of the obtained results, I trained my network with a vanilla mini-batch gradient descent on 100 data samples without regularization.

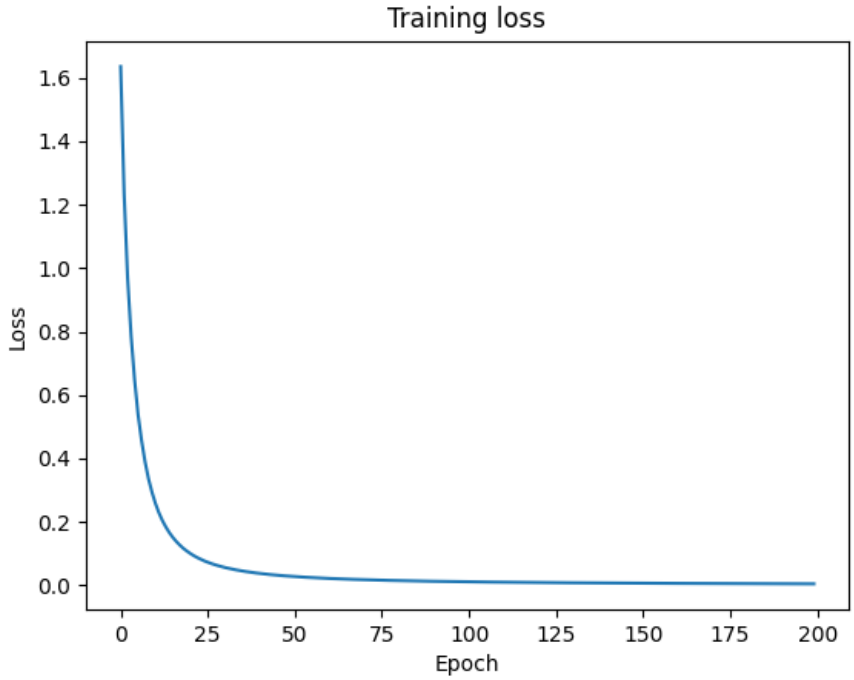

Figure 2: Training loss using eta = 0.01 and 200 epochs training

We can notice how the training loss rapidly converges to a very small value, indicating that my gradient computations and mini-batch gradient descent algorithm are correct.

4 Train your network with cyclical learning rates

In the third section of this assignment, I implemented the mini-batch gradient descent with a cyclical learning rate.
More specifically the learning rate at each update is defined as follows:

1. at iteration t of training if $2ln_s \leq t \leq (2l+1)n_s$ for some $l \in \{0, 1, 2, ...\}$ set:

$$n_t = n_{min} + \frac{t + 2ln_s}{n_s}(n_{max} - n_{min}) \tag{1}$$

2. while if $(2l+1)n_s \leq t \leq 2(l+1)n_s$ for some $l \in \{0, 1, 2, ...\}$ set:

$$n_t = n_{max} + \frac{t - (2l+1)n_s}{n_s}(n_{max} - n_{min}) \tag{2}$$

The main idea of cyclical learning rate is that during training the learning rate is periodically changed in a systematic fashion from a small value to a large one and then from this large value back to the small value.
One complete cycle will take $2n_s$ update steps where n_s is known as the step size. Normally training is run for a set number of complete cycles and is stopped when the learning rate is at its smallest.
If we plot the learning rate over time we obtained the following:

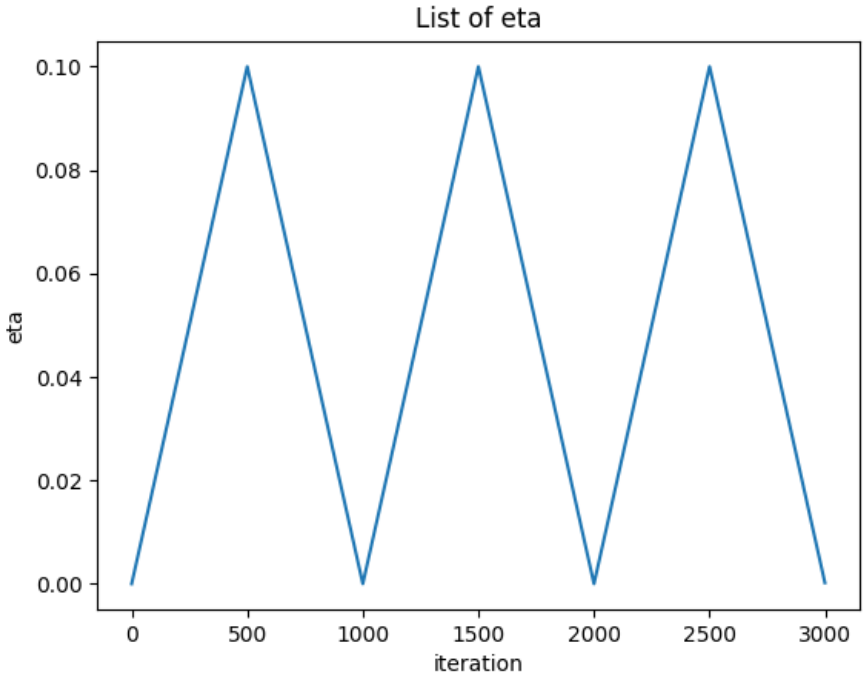

Figure 3: Learning rate plot for three cycles of training

After I implemented the algorithm, I trained the network for one cycle, setting `eta min` = 1e-5, `eta max` = 1e-1, and n_s=500 and the batch size to 100.
In the image below the plots for the cost, loss, and accuracy using $\lambda = 0.01$ are reported.

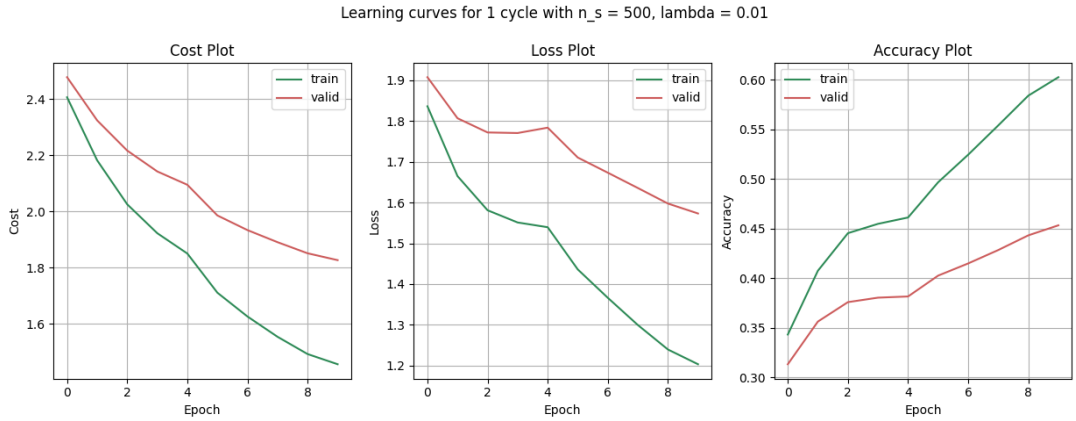

Figure 4: Cost, Loss and Accuracy plots for one training cycle

The obtained results are perfectly in line with what was expected and hence I moved to the next point of the work. At the end of the training, I obtained a test accuracy of `45.15%`

5 Train your network for real

I trained my network again using the same hyperparameters as before, but with a `stepsize=800` and for three cycles. We can clearly notice how the curves vary as the n_t varies. After the three cycles of training, I obtained a test accuracy of `47.01%`

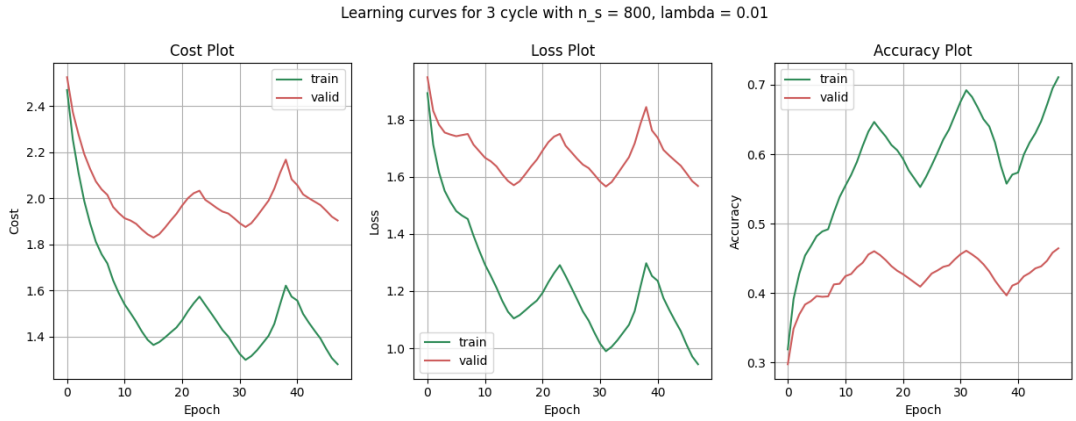

Figure 5: Cost, Loss and Accuracy plots for one training cycle

5.1 Coarse-to-fine random search to set `lambda`

In this part, I developed and performed a random search to find the best value for the `lambda` parameter. In order to do that I iterated over a set of randomly generated values for lambda and for each lambda I trained my network over different random initializations.
I performed my search over 10 lambda values and 10 different random initialization for each value. At each iteration, I trained my network over two cycles and used all available data for training except for 5000 images used as a validation set. For every lambda value, I measured the mean accuracy over the different random initialization. I tested the following lambda values [`0.09422310548966127, 0.022427976461626444, 0.013834969686475122, 0.0028763506984820003, 0.0005695302306079795, 0.0003578763205021341, 0.0002694186051499005, 6.815505724693517e-05, 2.8086493375109866e-05, 1.836194825120896e-05`] and obtained the following results:

Lambda	Accuracy Train	Accuracy Val
`0.09422310548966127`	39.23%	38.33%
`0.022427976461626444`	49.98%	47.32%
`0.013834969686475122`	52.57%	49.01%
`0.0028763506984820003`	57.65%	51.20%
`0.0005695302306079795`	58.79%	51.25%
`0.0003578763205021341`	58.91%	51.16%
`0.0002694186051499005`	58.92%	51.26%
`6.815505724693517e-05`	59.00%	51.11%
`2.8086493375109866e-05`	58.96%	51.15%
`1.836194825120896e-05`	59.02%	51.11%

Table 1: Validation accuracies of the different lambda values

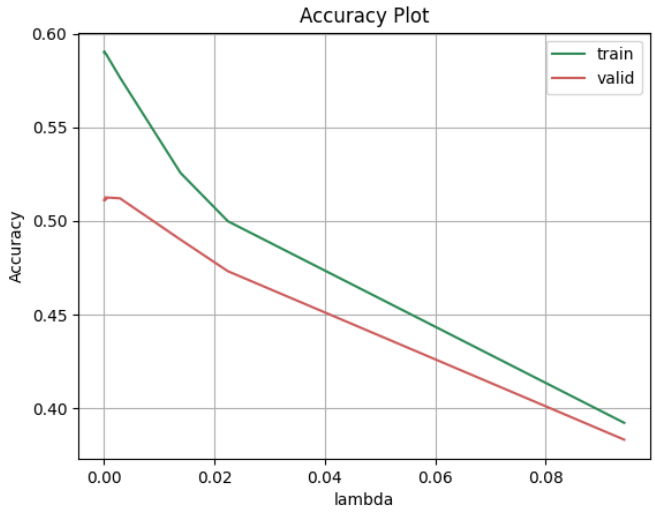

Figure 6: Accuracy plot for different lambda values

From the table and the plot above we can notice how the best results are achieved for `lambda = 0.0002694186051499005`.
After having achieved this result I performed a fine random search focused on the good parameters found before. I used the following lambda values: [`0.0001, 0.0002, 0.0003, 0.0004, 0.0005, 0.0006, 0.0007, 0.0008, 0.0009, 0.001`] and I obtained the following results:

Lambda	Accuracy Train	Accuracy Val
`0.0001`	62.83%	50.88%
`0.0002`	62.66%	51.02%
`0.0003`	62.65%	51.27%
`0.0004`	62.52%	50.96%
`0.0005`	62.34%	51.28%
`0.0006`	62.38%	51.33%
`0.0007`	62.19%	51.23%
`0.0008`	62.11%	51.42%
`0.0009`	62.03%	51.33%
`0.001`	61.79%	51.54%

Table 2: Validation accuracies of the different lambda values

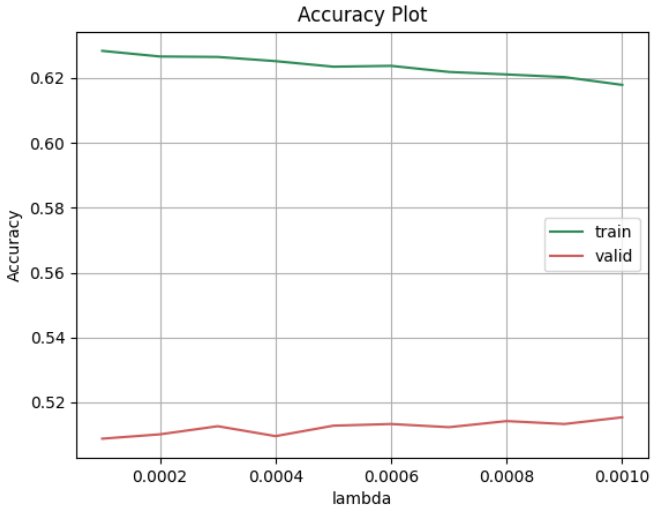

Figure 7: Accuracy plot for different lambda values

After the fine search, I found out `lambda=0.001` to be the best result., obtaining a `51.54%` accuracy on the validation set.

Finally, for my best found `lambda` setting, I trained the network on all the training data except for 1000 examples in a validation set for 3 cycles.

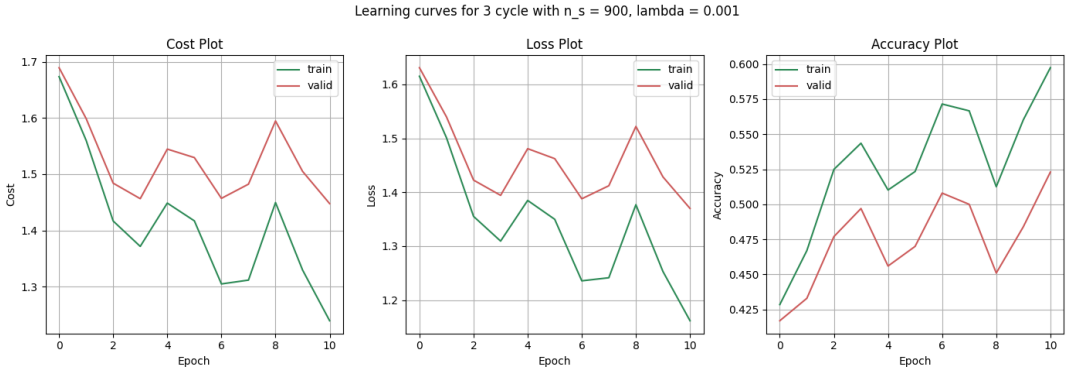

Figure 8: Cost, Loss and Accuracy plots for three cycles

The network in this configuration achieved a `51.40%` accuracy on the test set.

6 Bonus Points

6.1 1. Optimize the performance of the network

6.1.1 a) More hidden nodes

In this point, I tried to significantly increase the number of nodes in the hidden layer to see how that affects the performance of the network.

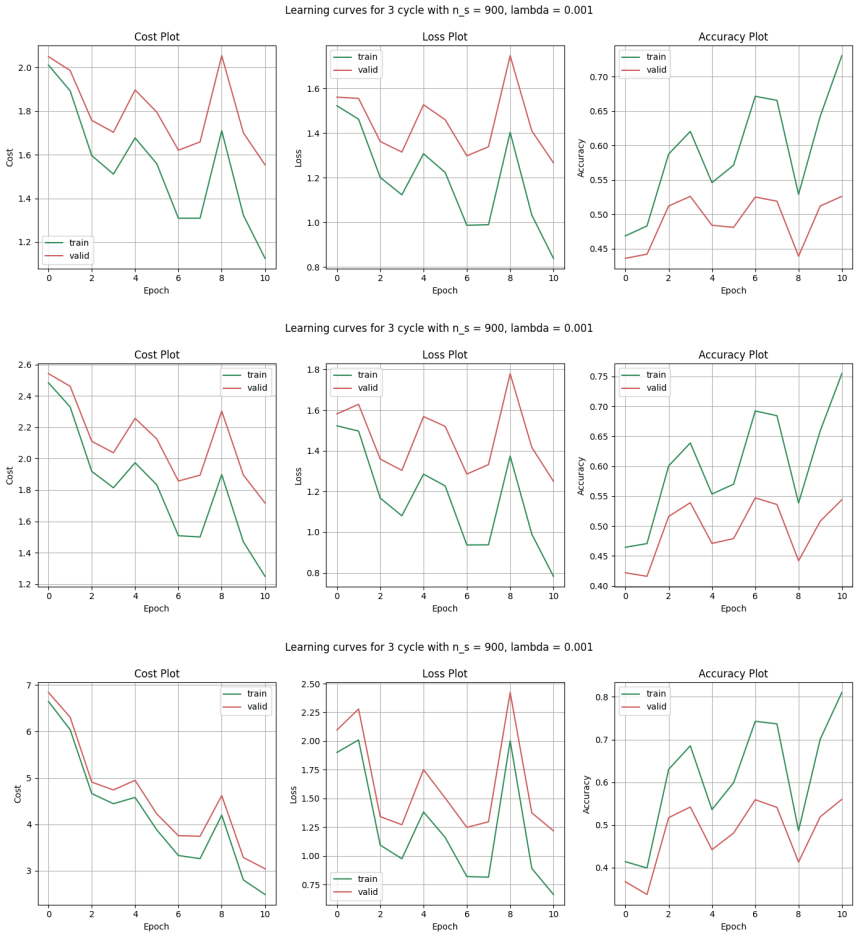

Figure 9: Learning curves for the 500 (top), 1000 (center), and 5000 (bottom) hidden nodes

We can notice how increasing the number of hidden nodes lead to an increase in performance, achieving respectively `55.90%`, `56.50%`, and `57.34%` test accuracy.

6.1.2 b) Dropout

After I increased the number of hidden nodes, I applied dropout with a probability of 30% of shutting down a node to the network with a high number of nodes in the hidden layer.

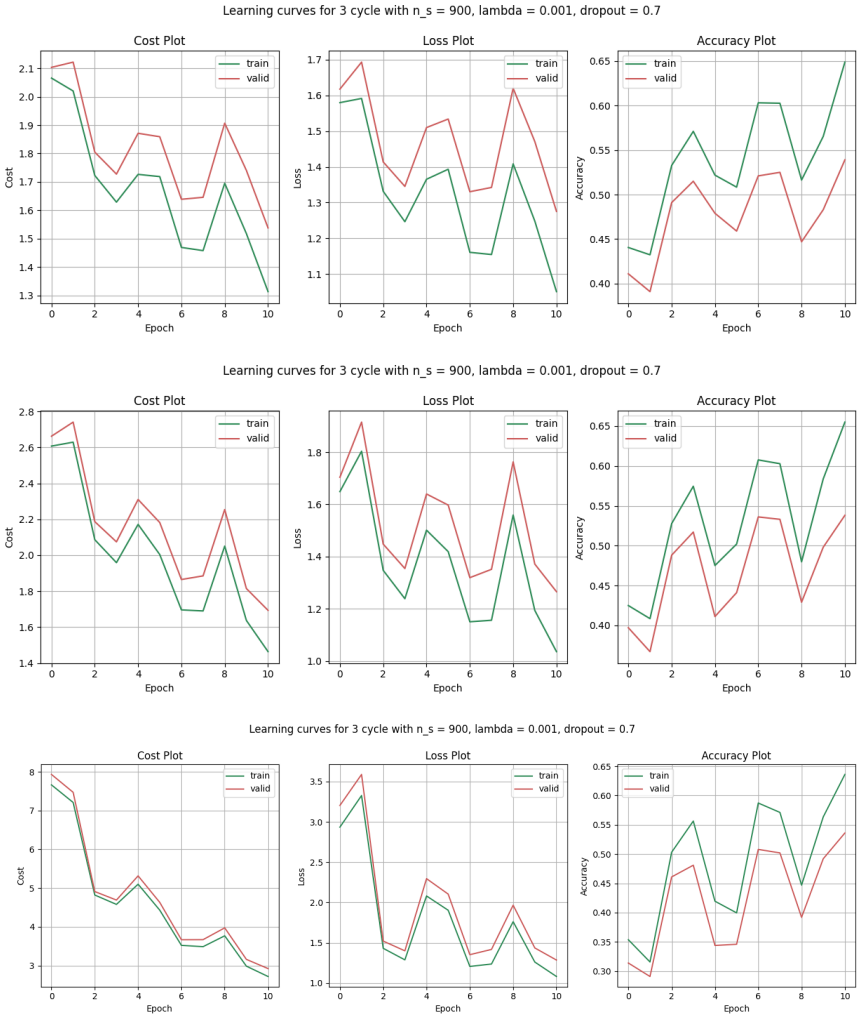

Figure 10: Learning curves for the 500 (top), 1000 (center), and 5000 (bottom) hidden nodes with dropout

I achieved respectively `55.63%`, `55.47%`, and `54.06%` test accuracy. We can notice how the test performance slightly decreased, but the gap between training and validation curves is decreased, hence the network is less prone to overfit.

6.1.3 c) Data Augmentation

The third improvement I implemented is data augmentation. More specifically I randomly mirrored and/or translated the images during training with a 30% probability.

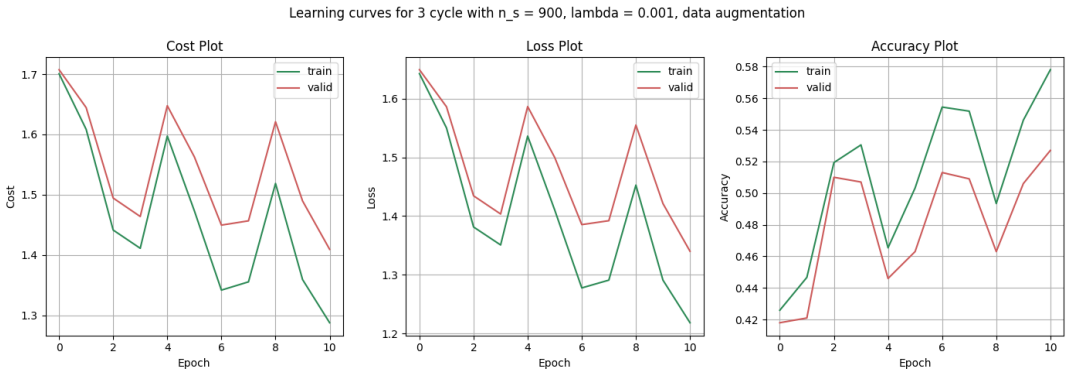

Figure 11: Cost, Loss and Accuracy plots for three cycles with 50 hidden nodes

The described configuration achieved a `52.37%` test accuracy.
Besides the slight increment in performance, we can also notice a reduction in the gap between training and validation curves.

6.2 2. Semi-extensive testing

In this last section, I combined the knowledge I acquired from the previous work and did some testing in order to find the best-performing 2-layer network. More in detail I found out that:

- The amount of regularization influences the performance of the network. With this rather simple model, I found `lambda = 0.001` to achieve the best results.
- Networks with more hidden nodes achieve better results.
- Dropout negatively impacts networks' performance but reduces the training and validation results gap.
- Data augmentation positively impacts the results, even if the accuracy increase was not too significant in my case.
- Using more data for training boosts the results.

After the testing process, I found out that in my case training for more than three cycles does not improve the performance. That said I trained my network using all available data, but a small validation subset. I run the training for three cycles with the following parameters:

`lambda`	`0.001`
`n_hidden`	`5000`
`dropout`	`None`
`augmentation`	`40%`

Table 3: Best network parameters

Summing up I run my network with a wide (5000 nodes) hidden layer, with a 0% probability of deactivating a node and with a 30% probability of mirroring/translating and I also added a data shuffle at the beginning of every batch.
With the cited parameters I achieved the following results:

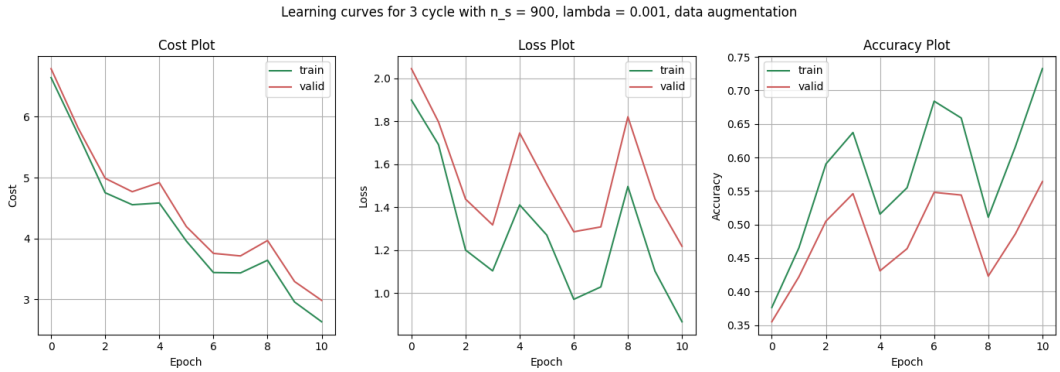

Figure 12: Cost, Loss and Accuracy plots for three cycles with the best parameters setting

The best network achieved a final test accuracy of `58.07%`, which is the best-obtained result. Summing up using a wide hidden layer will lead to better performance, but the network will be more computationally heavy, and hence more time will be required for training.
We can, however, notice that thanks to the data augmentation the network generalizes better with respect to the results in Section 6.1.1 and hence scores a better test accuracy.