Deep Learning in Data Science, DD2424

Short report on lab assignment 3

Mattia Evangelisti

May 08, 2023

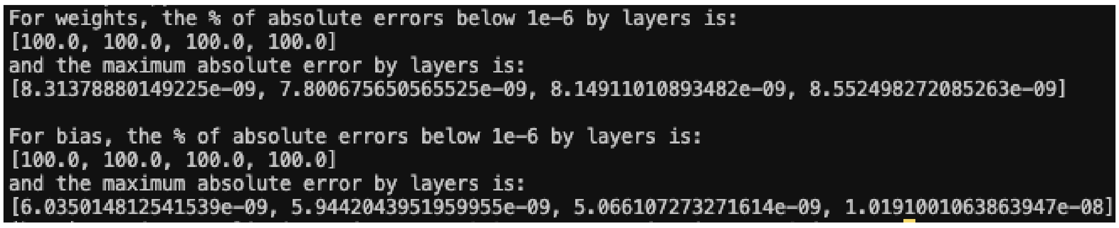
Contents

1 Introduction

This assignment is about building, training, and testing a multi-layers network with multiple outputs to classify images from the `CIFAR-10` dataset.
I used all the data, but 5000 samples for training, 5000 samples 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 Upgrade Assignment 2 code to train & test k-layer networks

In this first part of the assignment, I modify and adapted my code from the previous assignment in order to work with a k-layer network.
Most of the work in this part was to adapt the existing functions to cycle through the k layers instead of having a fixed number as in the previous assignment. More in detail I changed the functions to initialize the parameters of my networks, the functions for the forward pass (`evaluateClassifier`), the function for the mini-batch training (`fit`), and for the calculation of the gradients.
To check the correctness of my new gradients calculations, I checked the analytical results against the ones obtained through numerical computations for a 2-layer, 3-layer, and 4-layer network. Below I report the results obtained from testing with a 4-layer network:



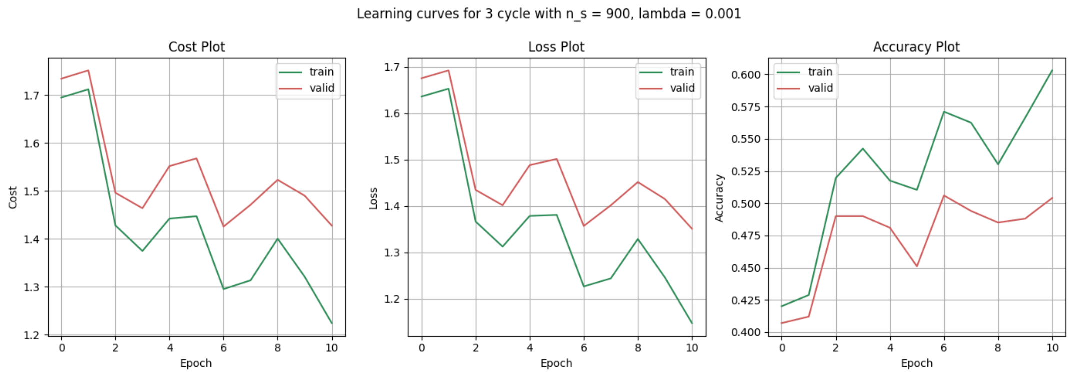
```
For weights, the % of absolute errors below 1e-6 by layers is:
[100.0, 100.0, 100.0, 100.0]
and the maximum absolute error by layers is:
[8.31378880149225e-09, 7.800675650565525e-09, 8.14911010893482e-09, 8.552498272085263e-09]

For bias, the % of absolute errors below 1e-6 by layers is:
[100.0, 100.0, 100.0, 100.0]
and the maximum absolute error by layers is:
[6.035014812541539e-09, 5.9442043951959955e-09, 5.066107273271614e-09, 1.0191001063863947e-08]
```

Figure 1: Gradients check

3 Can I train multi-layer networks?

After verifying the correctness of my gradient calculations, I checked if I could replicate the results obtained with a 2-layer network. To do so I set the same parameters I used in the previous assignment, i.e., one hidden layer with 50 nodes, $\lambda = 0.001$, $n_s = 900$, and training for three cycles. I obtained a `51.98%` test accuracy, which is almost the same obtained in the previous assignment which was equal to `51.50%`
I report the curves obtained from the previous and current architecture on the following page.



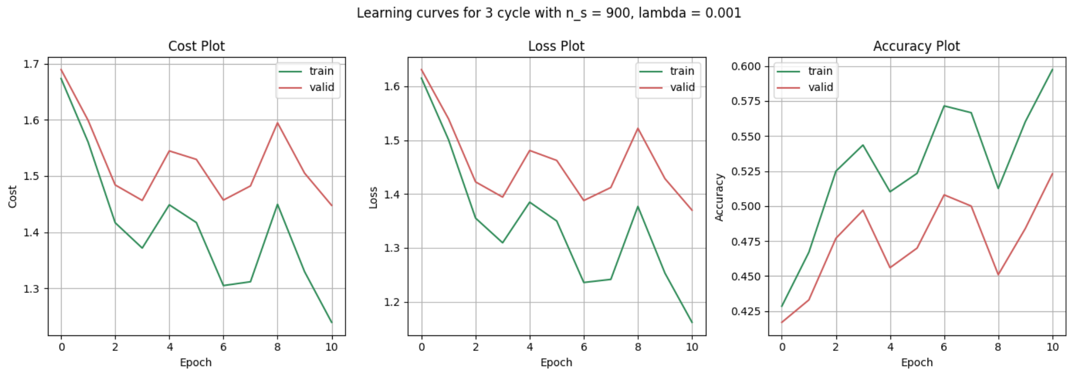


Figure 2: Current (top) and previous (bottom) architecture

After I verified I was able to reproduce the results obtained in the previous assignments, I run two other experiments with the following parameters:

- 3-layer network with [50,50] nodes in the hidden layers and `n_batch=100`, `eta_min=1e-5`, `eta_max=1e-1`, `lambda=0.005`, `n_s=5*45000/n_batch`, training for two cycles and using Xavier initialization.
- 9-layer network with [50,30,20,20,10,10,10,10] nodes in the hidden layers and `n_batch=100`, `eta_min=1e-5`, `eta_max=1e-1`, `lambda=0.005`, `n_s=5*45000/n_batch`, training for two cycles and using Xavier initialization.

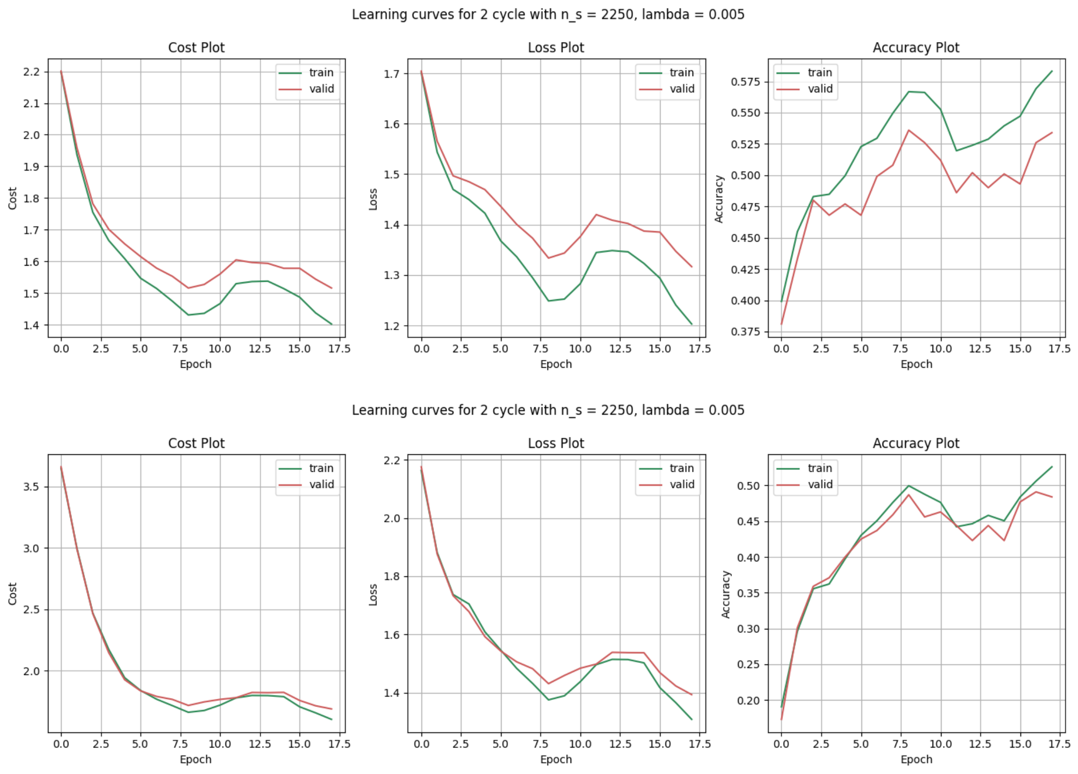


Figure 3: 3-layers (top) and 9-layers (bottom) networks

We can notice how the performance of the network drops when we increase the number of nodes, I indeed obtained a `52.04%` test accuracy with the 3-layer network and `48.16%` with the 9-layer network.

4 Implement batch normalization

At this point in the assignment, I implemented Batch Normalization, which can overcome the problem of poor performance in a multi-layer network.

Batch normalization involves the following steps:

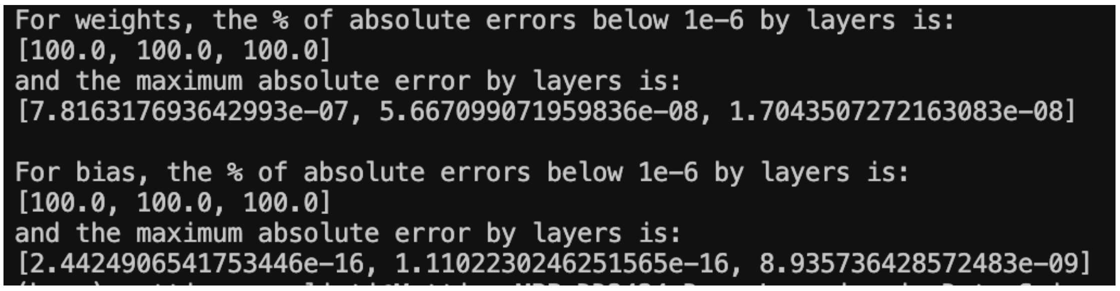
- Compute the mean and variance of the input features for each mini-batch during training. Mean: $\mu_\beta = \frac{1}{m}\sum x_i$ where x_i are the input features and m is the mini-batch size. Variance: $\sigma_\beta^2 = \frac{1}{m}\sum(x_i - \mu_\beta)^2$
- Normalize the input features using the computed mean and variance. Normalized input: $\frac{\hat{x}_i=(x_i-\mu_\beta)}{\sqrt{(\sigma_\beta^2+\epsilon)}}$, where ϵ is a small constant added for numerical stability

- Apply a linear transformation to the normalized inputs using learned parameters gamma and beta. This step allows the model to learn an appropriate scale and shift for the normalized inputs. Output: $y_i = \gamma \hat{x}_i + \beta$

During training, BN computes the mean and variance of each feature within a mini-batch. In the inference phase, when new data points are passed through the network, we use a running average of the mean and variance computed during training. This is known as the population mean and population variance, which are updated using exponential moving averages.

Population mean: $\mu = (1 - momentum) * \mu + momentum * \mu_\beta$
Population variance: $\sigma^2 = (1 - momentum) * \sigma^2 + momentum * \sigma_\beta^2$

I tested the modified code by checking the analytical gradients against the ones calculated in a numerical way for a 2-layer, and 3-layer. Below I report the results obtained from testing with a 3-layer network:



```
For weights, the % of absolute errors below 1e-6 by layers is:
[100.0, 100.0, 100.0]
and the maximum absolute error by layers is:
[7.816317693642993e-07, 5.667099071959836e-08, 1.7043507272163083e-08]

For bias, the % of absolute errors below 1e-6 by layers is:
[100.0, 100.0, 100.0]
and the maximum absolute error by layers is:
[2.4424906541753446e-16, 1.1102230246251565e-16, 8.935736428572483e-09]
```

Figure 4: Gradients check with batch normalization

Then I trained again the 3-layer and 9-layer networks using batch normalization and in the following page, I report the training curves for the two architectures.
With batch normalization, I obtained a `52.49%` test accuracy with the 3-layer network and `51.47%` with the 9-layer network.
We can notice how the batch normalization influenced most the 9-layer network, which indeed had a higher increase in performance.

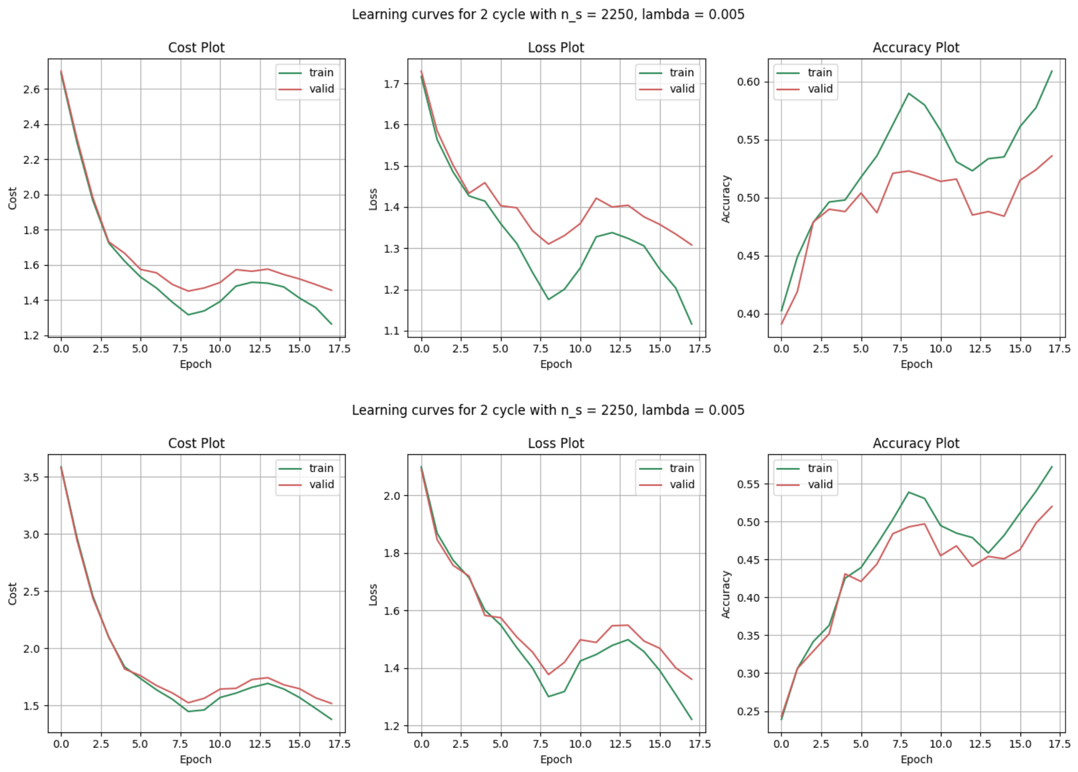


Figure 5: 3-layers (top) and 9-layers (bottom) networks with batch normalization

4.1 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 [`0.08544670917128065, 0.06797930226110538, 0.045904492843848094, 0.04159985166856119, 0.009926151798168227, 0.006084147303403797, 0.0008653470078030258, 0.0001227703773447407, 5.020302237476644e-05, 1.23619054774414e-05`] and I obtained the following results:

```
'train_acc':  [0.4883061224489796, 0.5017959183673469, 0.5171428571428571,
0.5225306122448979, 0.5886326530612245, 0.6148163265306122, 0.6831632653061225,
0.7237959183673469, 0.7560816326530613, 0.7794489795918368]
'valid_acc':  [0.475, 0.476, 0.49, 0.49, 0.522, 0.526, 0.527, 0.522, 0.516, 0.497]
```

Best lambda: 0.0008653470078030258
Best train accuracy: 0.6831632653061225
Best valid accuracy: 0.527

Then I run a fine search investigating better values in the range on the best lambda found before. I tested the following lambda values [`0.0005, 0.0006, 0.0007, 0.0008, 0.0009, 0.001, 0.002, 0.003, 0.004, 0.005`] and I obtained the following results:

```
'train_acc': [0.6337142857142857, 0.683469387755102, 0.7043265306122449,
0.7136122448979592, 0.7179183673469388, 0.7169387755102041, 0.6837755102040817,
0.654734693877551, 0.6426530612244898, 0.6342040816326531]
'valid_acc': [0.55, 0.53, 0.526, 0.517, 0.528, 0.528, 0.527, 0.542, 0.538, 0.537]
```

Best lambda: 0.0005
Best train accuracy: 0.6337142857142857
Best valid accuracy: 0.55

After that, I trained my 3-layer network with the best value of lambda, i.e., with `lambda = 0.0005`:

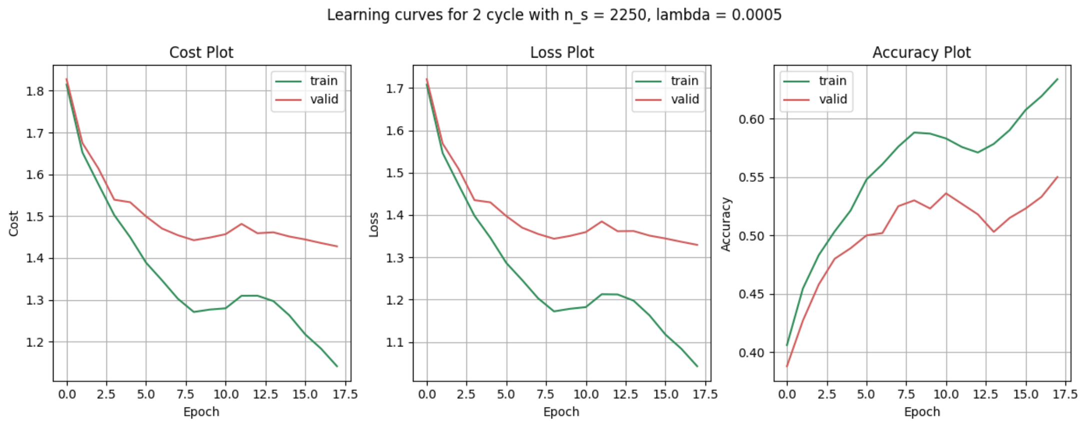


Figure 6: Learning curves for best lambda settings

The network with this lambda configuration achieved a 53.80% accuracy on the test set.

4.2 Sensitivity to Initialization

In the last part of the assignment, for each training regime instead of using He initialization, I initialized each weight parameter to be normally distributed with sigmas equal to the same value `sig` at each layer. For three runs set `sig=1e-1, 1e-3 and 1e-4` respectively and train the network with and without BN and see the effect on the final test accuracy.

sig=1e-1

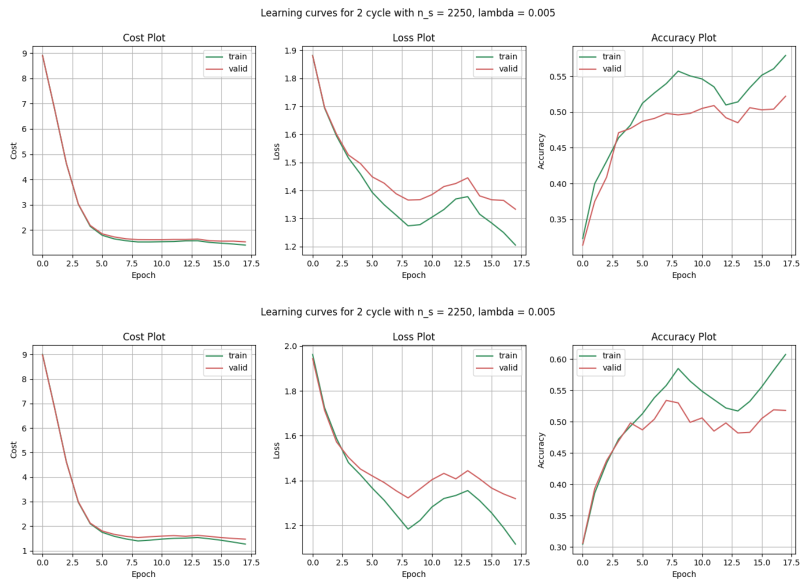


Figure 7: `sig=1e-1` without (top) and with (bottom) BN

sig=1e-3

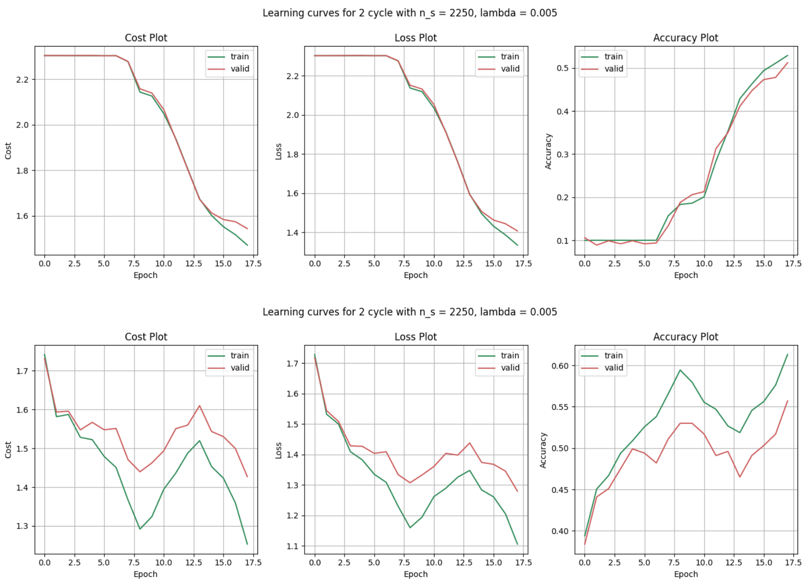


Figure 8: `sig=1e-3` without (top) and with (bottom) BN

`sig=1e-4`

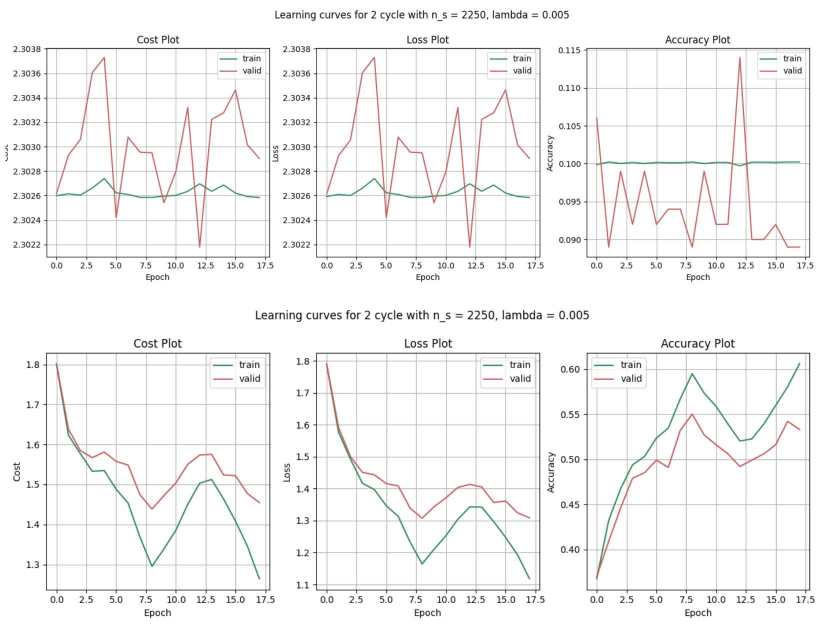


Figure 9: `sig=1e-3` without (top) and with (bottom) BN

We can notice how initializing all weights to a fixed parameter does decrease the performance of the network, especially for smaller values of the `sig` parameter. However, it is visible how batch normalization seems to patch this problem.

When training with BN, the performance is generally less sensitive to the choice of initialization. This is because BN helps mitigate the issues of vanishing and exploding gradients. By normalizing the activations, BN prevents the distribution of activations from changing too much during training. This leads to more stable gradients and faster convergence.

5 Bonus Points

5.1 1a) Batch normalization after non-linear activation functions

In this section, I modified my code in order to apply batch normalization to the scores after the non-linear activation function has been applied.

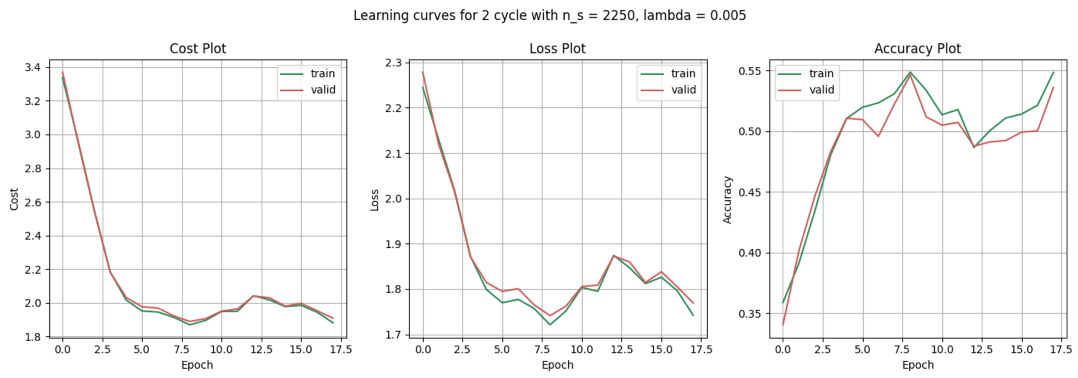


Figure 10: Learning curves for 3-layer network with batch normalization after ReLu

I achieved `53.04%` accuracy on the test set, which is a slight increase with respect to the base model.

5.2 1b) PreciseBN

In this section, I applied a technique known as *PreciseBN*, which consists of calculating means and standard deviation with a fixed model over a large chunk of training data instead of using the running average approach.

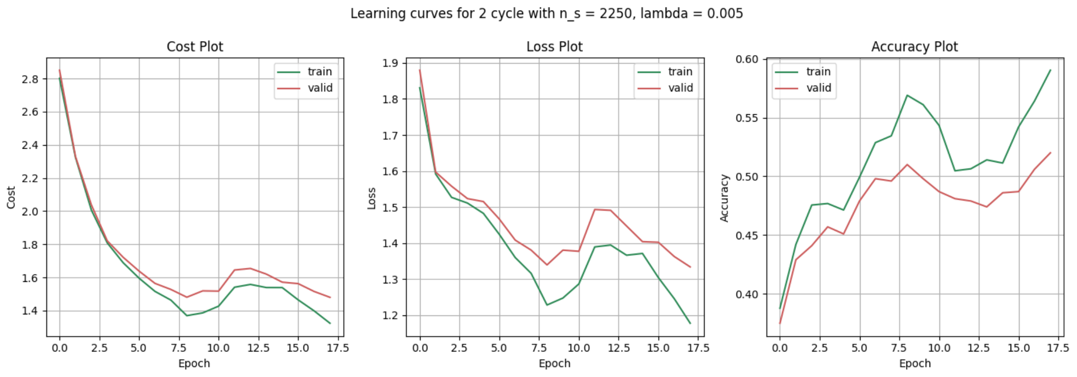


Figure 11: Learning curves for 3-layer network using PreciseBN

I achieved `52.12%` accuracy on the test set, which is a very similar result to the one achieved using the running average method.

5.3 2a) Adam optimizer and learning rate decay

In this section, I tried to use Adam optimizer and decay learning rate with a 7-layer network. Since I implemented decay learning rate, I trained my network without cyclical learning rate, with 50 units in each hidden layer and for 20 epochs.

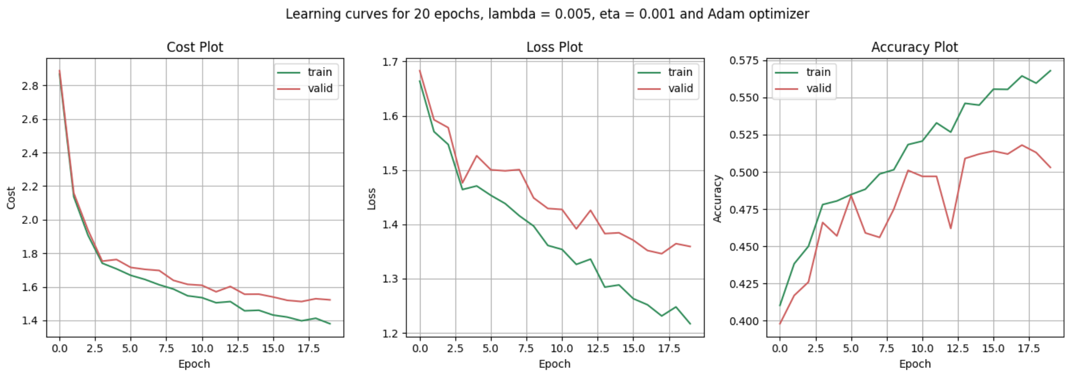


Figure 12: Learning curves for 7-layer network with adam optimizer

I achieved `52.08%` accuracy on the test set, which in my case is not a great improvement with respect to what is shown in the next section.

5.4 2b) Increasing the number of hidden nodes at each layer

In this part of the assignment, I tried to train a 7-layer network with more units in the hidden layers.

I tried to use 100, 250, and 500 units in each layer:

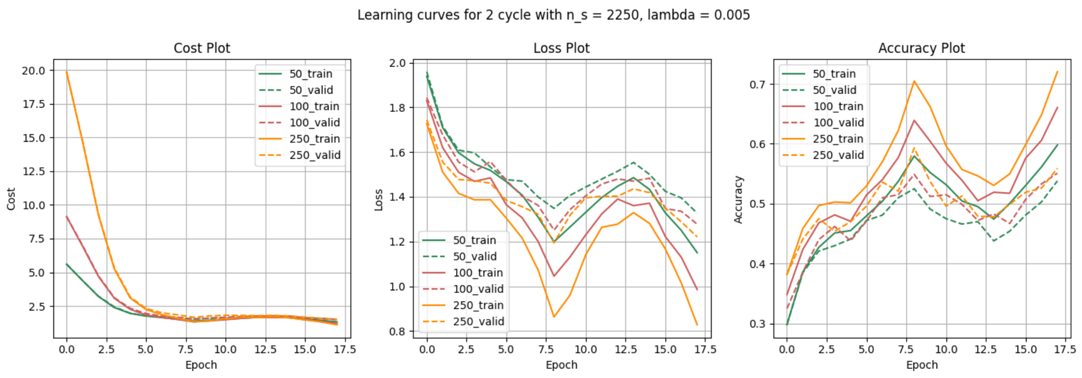


Figure 13: Learning curves for 7-layer network with more hidden nodes

I achieved `51.88%`, `55.24%`, and `57.63%` accuracy on the test set.

We can clearly notice how incrementing the number of nodes in the hidden layers boosts the performance of the network. I am confident that with an even higher number of nodes and the right amount of normalization, I could have achieved better results, but for time's sake, I limited my search to a relatively small number of nodes.