DD2424 Deep Learning in Data Science

Assignment 1

Ali Shibli

April 2021

1 Introduction

In this assignment, we build a 2 layer neural network from scratch for classification task. The dataset being used is CIFAR-10 dataset, that consists of 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The hidden layer will contain 50 units.

2 Functions Implemented

2.1 LoadBatch

This function is simply to load the dataset from disk. There are 5 batches of files and 1 testing batch when we download from source. For this assignment, the first batch is used for training, second for validation, and third for testing (if required).

2.2 Normalize

This function basically pre-processes the dataset. It centeres the points (vectors) around 0 and divides by their standard deviation.

2.3 initialize_weights

This function initializes the weight matrix W and the biases vector b. We use a gaussian random initialization with mean=0 and std=0.01.

2.4 ReLu

This is the activation function used in the hidden layer units. It implements the following function:

$$f(x) = max(0, x) \tag{1}$$

2.5 Softmax

This is the activation function used in the output layer. It implements the following function:

$$f(x) = \frac{exp(s)}{1^T exp(s)} \tag{2}$$

2.6 EvaluateClassifier

This function computes the predictions for a particular input data (batch) in terms of a probability distriubtion for each class label. In other words, given an input matrix (or vector) X, the weights matrices W_1 and W_2 for the hidden and output layers respectively, and the biases b_1 and b_1 for the hidden the output layers, the function returns a vector containing the probability of each label from the target labels. Then we choose the label with highest probability as our class guess. Mathematically, it is computed as follows:

$$f(x) = softmax(W_2.relu(W_1X + b_1) + b_2) \tag{3}$$

2.7 ComputeCost

This function computes the cost function (or loss) of our network. It derives from our predictions in the EvaluateClassifier function, and compares the results with the ground truth results. This cost is a combination of two costs. For the first part we are using cross entropy loss function. For the second term, it is the regularization loss with the corresponding λ parameters to help in generalization and regularization of the network.

2.8 ComputeAccuracy

This function computes the accuracy of the predictions. That is, the True Positives plus the True Negatives out of the total predicted examples.

2.9 ComputeGradients

This function is the main part (or heart) of the algorithm. This is where the parameters of the network W and b get updated at each iteration. In particular, we follow these steps to compute the gradients of these parameters:

1. Compute $G_{batch} = -(Y_{batch} - P_{batch})$
2. Compute gradient w.r.t. W_2 as $\frac{dl}{dW_2} = \frac{1}{n} G_{batch} H_{batch}^T$
3. Compute gradient w.r.t. b_2 as $\frac{dl}{db_2} = \frac{1}{n} G_{batch} 1_n$
4. Propagate the gradient backwards through the second layer as $G_{batch} = W_2^T G_{batch}$, then $G_{batch} = G_{batch}.Ind(H_{batch} > 0)$
5. Compute gradient w.r.t. W_1 as $\frac{dl}{dW_1} = \frac{1}{n} G_{batch} X_{batch}^T$

6. Compute gradient w.r.t. b_1 as $\frac{dl}{db_1} = \frac{1}{n} G_{batch} 1_n$

This calculation so far is analytical calculation. Another way is by numerical calculations, using functions pre-implemented in Matlab, we convert them to Python code. To verify our gradient computations, we furthermore experiment with the following two given functions:

- ComputeGradsNum: faster but less accurate results
- ComputeGradsNumSlow: slower but more accurate results

We use two metrics in the comparison between our computed gradients and the gradients obtained from the predifined functions. The first one is simply the average of the number of errors in the results, where the errors are simply the points which have a difference $> \epsilon = 10^{-6}$. The second metric is the relative error between our computed gradients and the obtained gradients. We take the also the points whose difference is larger than the threshold $\epsilon = 10^{-6}$.

As a first run, we start with no regularization term, that is, $\lambda = 0.0$. In addition, instead of taking the entire dataset, we take the first 20 dimensions of the first training example. The results were as follow:

- Using error as number of points where difference is less than epsilon = 1e-6
  - Between ComputeGradients and ComputeGradNum:
    - There are 0 errors for weights 1 (error rate = 0.0%)
    - There are 0 errors for biases 1 (error rate = 0.0%)
    - There are 0 errors for weights 2 (error rate = 0.0%)
    - There are 0 errors for biases 2 (error rate = 0.0%)
  - Between ComputeGradients and ComputeGradNumSlow:
    - There are 520 errors for weights 1 (error rate = 52.0%)
    - There are 0 errors for biases 1 (error rate = 0.0%)
    - There are 260 errors for weights 2 (error rate = 52.0%)
    - There are 0 errors for biases 2 (error rate = 0.0%)
- Using relative error as number of points where difference is less than epsilon = 1e-6
  - Between ComputeGradients and ComputeGradNum:
    - There are 42 errors for weights 1 (error rate = 4.2%)
    - There are 7 errors for biases 1 (error rate = 14.000000000000002%)
    - There are 0 errors for weights 2 (error rate = 0.0%)
    - There are 9 errors for biases 2 (error rate = 9.0%)
  - Between ComputeGradients and ComputeGradNumSlow:

- ∗ There are 520 errors for weights (error rate = 0.52%)
- ∗ There are 0 errors for biases (error rate = 0.0%)
- ∗ There are 260 errors for weights (error rate = 0.52%)
- ∗ There are 0 errors for biases (error rate = 0.0%)

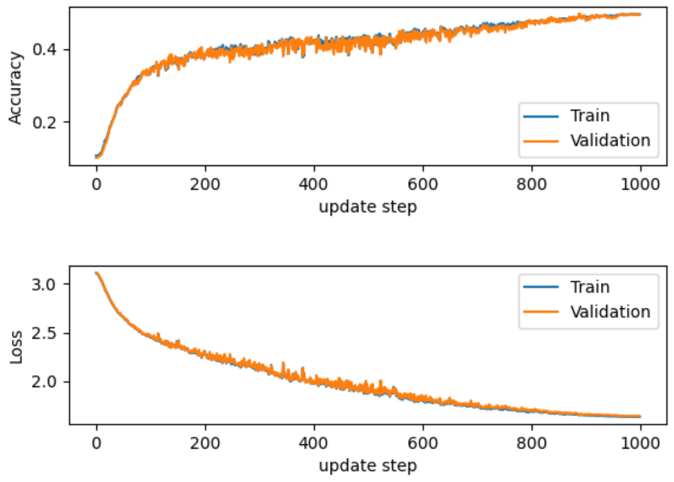
2.10 MiniBatchGD

This function is the high level computation of the algorithm. We run over the different batch sizes of the data, computing the gradients at each iteration, and updating the weights and biases until the number of epochs or cycles is reached. Note that we are implementing cyclical learning rate update.

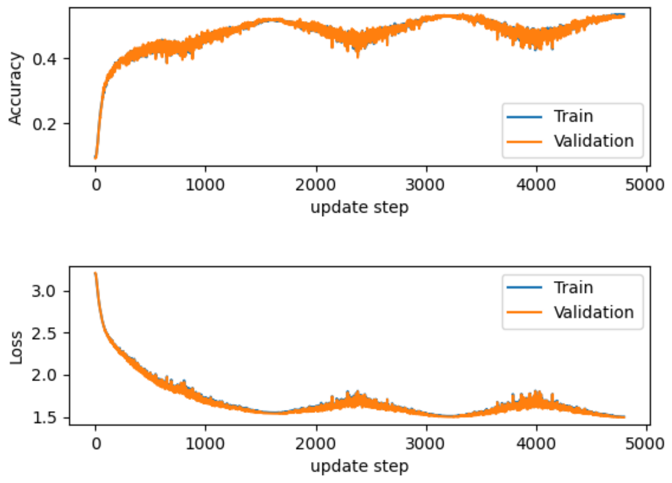
3 Results

3.1 Cyclical Learning Rates

We can visualize the cost and learning curves in the following graphs, when we are training on the first batch of images, validating on the second, and testing on the testing batch of images.



(a) The parameters for this model are lambda=0.01, eta_min=1e-5,eta_max=1e-1, step_size=500, batch_size=100, and 1 cycle (this corresponds to figure 3 in the assignment description). The results at the end of the training were as Iter 1000/1000 : loss= 1.6359175132456165 acc= 0.49536 val_loss= 1.6444249063528553 val_acc= 0.4954. The testing accuracy on the test set was 0.4784.



(b) The parameters for this model are lambda=0.01, eta_min=1e-5,eta_max=1e-1, step_size=800, batch_size=100, and 3 cycles (this corresponds to figure 4 in the assignment description). The results at the end of the training were as Iter 4800/4800 : loss= 1.5017215509556965 acc= 0.5368 val_loss= 1.4946561872950146 val_acc= 0.53088. The testing accuracy on the test set was 0.511.

Figure 1: Case 1 results

We note how the cyclical learning with multiple cycles propagates between 3 cycles in the loss and accuracy curves. In addition, we note that as we increase the step size and number of cycles, we get higher accuracy on the test set.

3.2 Coarse Random Search to set Lambda

In this part, we are optimizing our learning to find the best possible lambda fit. For this purpose, we devise an interval for lambda values and search for lambda on log scale, by setting a min and max value for lambda, and taking 8 different values in between. This is illustrated in:

- l = l_min + (l_max - l_min) * rand(8)
- lambdas_coarse = 10^l
- lambdas_coarse.sort()

The minimum and maximum learning rates are as before, 1e-5 and 1e-1 respectively. The batch size is 500 and the number of cycles is 2.

We report the top 3 performing networks per lambda values on the testing set accuracy. The top three results are listed as:

1. lambda = 7.141445874680704e-05 (second value in the list). The test accuracy is 0.4907.
2. lambda = 3.239347447896492e-05 (first value in the list). The test accuracy is 0.4894.
3. lambda = 0.0014859207773860062 (fourth value in the list). The test accuracy is 0.4893.

3.3 Fine Random Search to set Lambda

In this part as well, we are optimizing our learning to find the best possible lambda fit. For this purpose, we devise an interval uniformly from 0.001 to 0.008 as: lambdas_fine = np.arange(0, 0.008, 0.001)

The minimum and maximum learning rates are as before, 1e-5 and 1e-1 respectively. The batch size is 500 and the number of cycles is 2.

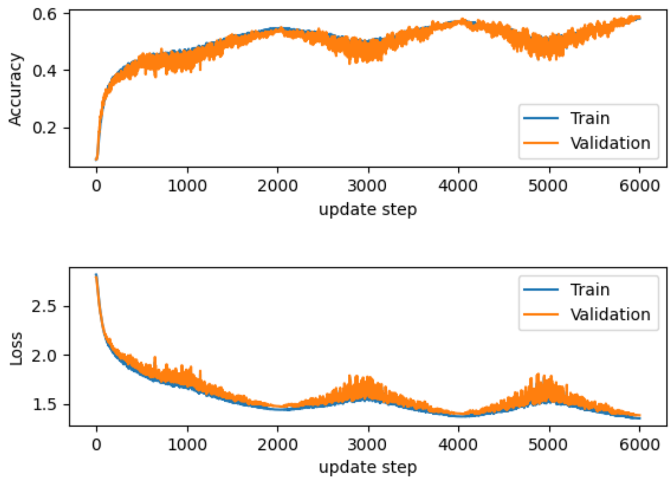
We report the top 3 performing networks per lambda values on the testing set accuracy. The top three results are listed as:

1. lambda = 0.004. The test accuracy is 0.4911.
2. lambda = 0.001. The test accuracy is 0.4899.
3. lambda = 0.003. The test accuracy is 0.4897.

3.4 Final Training with best lambda setting

The best lambda that we can guess from the previous 2 subsections is lambda=0.004, which achieved an accuracy of 49.11% on the testing set.

For this part, we use same paramater settings as before, except that we use a smaller batch size of 100 (before we were testing many cases that would be computationally expensive and take time).



(a) Accuracy and loss graphs for train and validation sets

Final accuracy on testing set: 0.5237.