DD2424 Deep Learning in Data Science

Assignment 3

Ali Shibli

May 2021

1 Introduction

In this assignment, we build a k 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. We will be varying the number of hihdden layers to test the effect of complexity of the network on the classification task. In addition, we implement batch normalization to normalize the batches at each layer as proposed by [1] to increase the performance of the network.

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 weights of the network. We have two cases, one with batch normalization and without normalization. With normalization we initialize the weights matrix W, the biases vector b, and the gamma and beta arrays that are used in normalization. Finally the weights are distributed normally with either Xavier initialization or He initialization.

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 distribution for each class label. In other words, given an input matrix (or vector) X, the weights matrices W for the hidden and output layers respectively, and the biases b for the hidden the output layers, as well as gamma and beta in case of batch normalization, 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.

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. for i=k-1,k-2,...,2:
    a) Compute gradient w.r.t. W_i as $\frac{dl}{dW_i} = \frac{1}{n} G_{batch} X_{batch}^{(l-1)^T}$

   b) Compute gradient w.r.t. b_i as $\frac{dl}{db_i} = \frac{1}{n} G_{batch} 1_n$
   c) Propagate the gradient backwards through the layers as $G_{batch} = W_i^T G_{batch}$
   d) Then $G_{batch} = G_{batch}.Ind(X_{batch}^{i-1} > 0)$

3. Finally $\frac{dl}{dW_1} = \frac{1}{n} G_{batch} X_{batch}^T$
4. and $\frac{dl}{db_1} = \frac{1}{n} G_{batch} 1_n$

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 given functions:

- 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 $\epsilon = 10^{-6}$.

We show, as required, the results for k layer networks with batch normalization for k=2,3,4. The number of nodes in each hidden layer is set to 50 nodes. We set 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 10 training instances. The results were as follow:

- For k=2:
  - Using error as number of points where difference is less than epsilon = 1e-6
    * There are [0, 0] errors for weights (error rate = [0.0, 0.0] (in %))
    * There are [0, 0] errors for biases (error rate = [0.0, 0.0] (in %))
    * There are [0] errors for gammas (error rate = [0.0] (in %))
    * There are [0] errors for betas (error rate = [0.0] (in %))
  - Using relative error as number of points where difference is less than epsilon = 1e-6
    * There are [191, 38] errors for weights (error rate = [19.1, 7.6] (in %))
    * There are [2, 2] errors for biases (error rate = [4.0, 20.0] (in %))
    * There are [5] errors for gammas (error rate = [10.0] (in %))
    * There are [3] errors for betas (error rate = [6.0] (in %))
- For k=3:
  - Using error as number of points where difference is less than epsilon = 1e-6

        - There are [0, 0, 0] errors for weights (error rate = [0.0, 0.0, 0.0] (in %))
        - There are [0, 0, 0] errors for biases (error rate = [0.0, 0.0, 0.0] (in %))
        - There are [0, 0] errors for gammas (error rate = [0.0, 0.0] (in %))
        - There are [0, 0] errors for betas (error rate = [0.0, 0.0] (in %))
    - Using relative error as number of points where difference is less than epsilon = 1e-6
        - There are [236, 586, 32] errors for weights (error rate = [23.599999999999998, 23.44, 6.4] (in %))
        - There are [2, 0, 5] errors for biases (error rate = [4.0, 0.0, 50.0] (in %))
        - There are [5, 2] errors for gammas (error rate = [10.0, 4.0] (in %))
        - There are [7, 2] errors for betas (error rate = [14.000000000000002, 4.0] (in %))

- For k=4:
    - Using error as number of points where difference is less than epsilon = 1e-6
        - There are [0, 0, 0] errors for weights (error rate = [0.0, 0.0, 0.0] (in %))
        - There are [0, 0, 0] errors for biases (error rate = [0.0, 0.0, 0.0] (in %))
        - There are [0, 0] errors for gammas (error rate = [0.0, 0.0] (in %))
        - There are [0, 0] errors for betas (error rate = [0.0, 0.0] (in %))
    - Using relative error as number of points where difference is less than epsilon = 1e-6
        - There are [236, 586, 32] errors for weights (error rate = [23.599999999999998, 23.44, 6.4] (in %))
        - There are [2, 0, 5] errors for biases (error rate = [4.0, 0.0, 50.0] (in %))
        - There are [5, 2] errors for gammas (error rate = [10.0, 4.0] (in %))
        - There are [7, 2] errors for betas (error rate = [14.000000000000002, 4.0] (in %))

2.10 BatchNormBackPass

This function implements the backpropagation through the network to get the new batches G_{batch} after normalization.

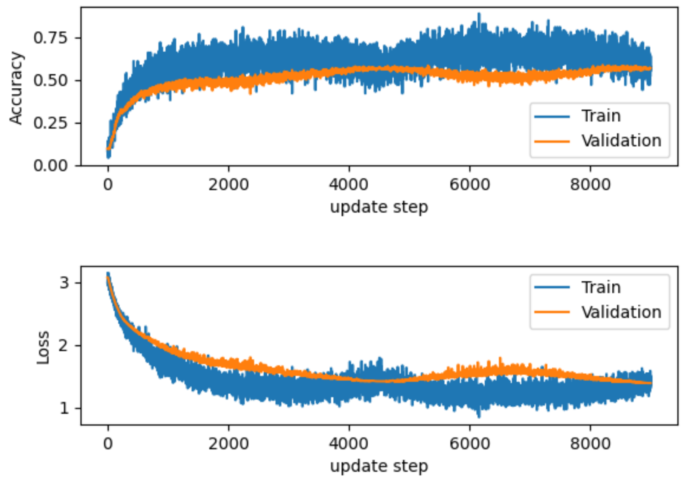
2.11 MiniBatchGD

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

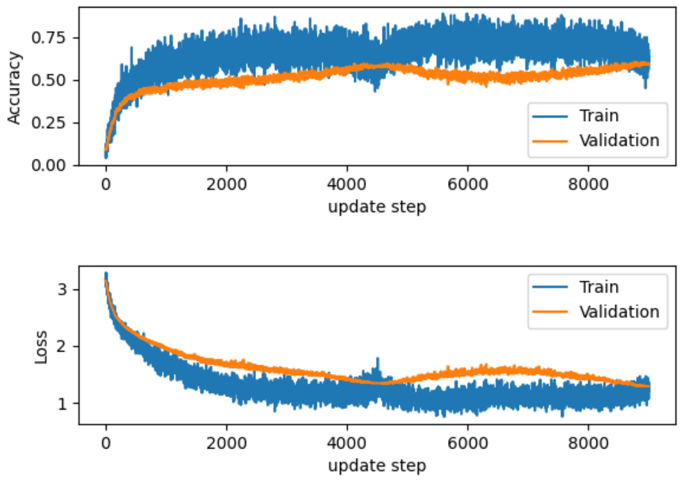
3.1 3 layer network

This network has 2 hidden layers + 1 input layer + 1 output layer, where the hidden layers have 50 units each with the relu activation function.
For the network without batch normalization, 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=0.005, eta_min=1e-5,eta_max=1e-1, batch_size=100, step_size=$\frac{5*45000}{batch\_size}$, and 2 cycles (this corresponds to figure 3 in the assignment description). The results at the end of the training were as Iter 9000/9000 : loss= 1.4934917851319578 acc= 0.57 val_loss= 1.3882495091159022 val_acc= 0.568. The testing accuracy on the test set was 0.5328.

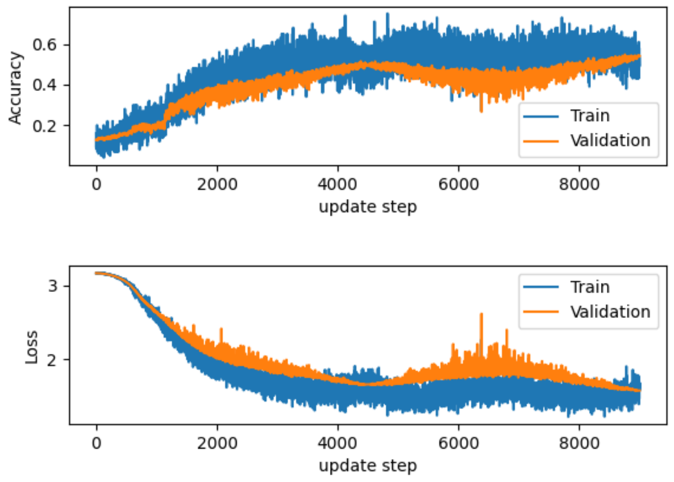
For the network with batch normalization, we can visualize the cost and learning curves in the following graphs, when we are training on the same parameters as before:



(b) The parameters for this model are lambda=0.005, eta_min=1e-5,eta_max=1e-1, batch_size=100, step_size=$\frac{5*45000}{batch\_size}$, and 2 cycles (this corresponds to figure 3 in the assignment description). The results at the end of the training were as Iter 9000/9000: loss= 1.3252312501797041 acc= 0.61 val_loss= 1.291725814923054 val_acc= 0.5922. The testing accuracy on the test set was 0.5312.

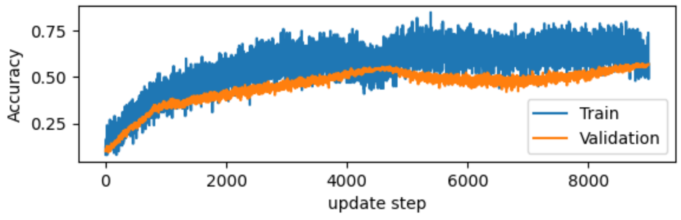
3.2 9 layer network

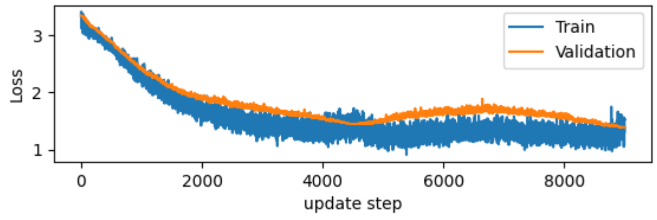
This network has 9 hidden layers + 1 input layer + 1 output layer, where the hidden layers are distributed as = [50, 30, 20, 20, 10, 10, 10, 10] units per layer, each with the relu activation function.
For the network without batch normalization, we can visualize the cost and learning curves in the following graphs, when we are training on the first batch of images, validating on the second, and testing on the testing batch of images.



(c) The parameters for this model are lambda=0.005, eta_min=1e-5,eta_max=1e-1, batch_size=100, step_size=$\frac{5*45000}{batch\_size}$, and 2 cycles (this corresponds to figure 3 in the assignment description). The results at the end of the training were as Iter 9000/9000: loss=1.672425892845959 acc=0.53 val_loss=1.5772876216641571 val_acc=0.5398 Test Accuracy: 0.4764

For the network with batch normalization, we can visualize the cost and learning curves in the following graphs, when we are training on the same parameters as before:





(d) The parameters for this model are lambda=0.005, eta_min=1e-5,eta_max=1e-1, batch_size=100, step_size=$\frac{5*45000}{batch\_size}$, and 2 cycles (this corresponds to figure 3 in the assignment description). The results at the end of the training were as Iter 9000/9000: loss=1.520889739884565 acc=0.57 val_loss=1.3537592956539148 val_acc=0.5655. Test Accuracy: 0.5182

3.3 Coarse Random Search to set Lambda with Batch Norm for 3 layer network

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 performing networks per lambda values on the testing set accuracy. This corresponds to:

1. lambda = 0.0005365922242846594 (l=5). The test accuracy is 0.5128.

3.4 Fine Random Search to set Lambda with Batch Norm for 3 layer network

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, 0.0001, 0.0008, 0.00025)

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.

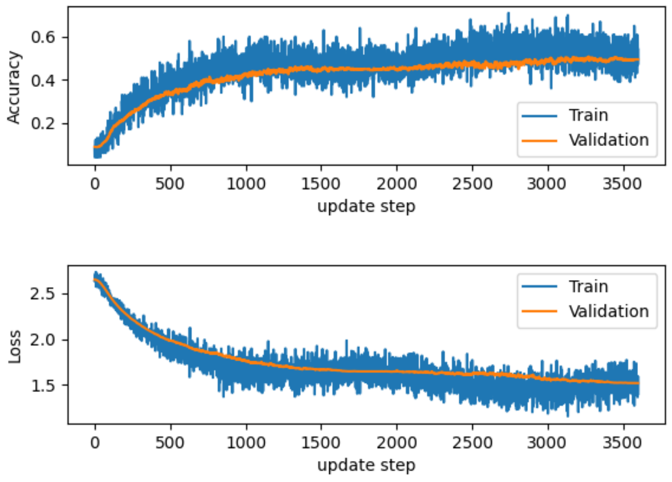
We report the top performing network per lambda values on the testing set accuracy. This corresponds to:

1. lambda = 0.0001. The test accuracy is 0.521.

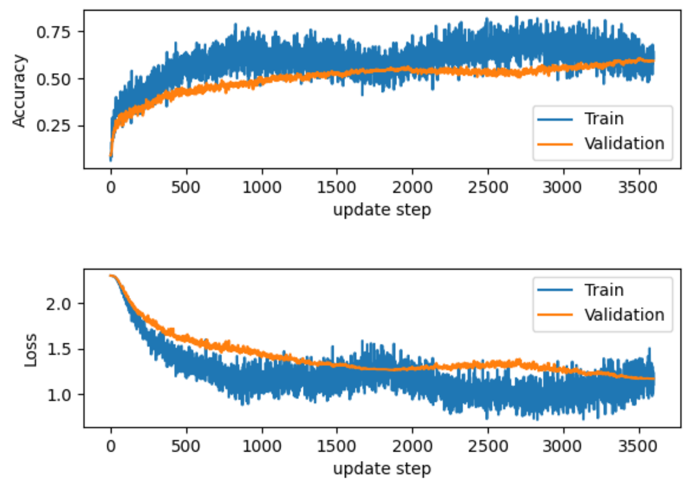
3.5 Sensitivity to Initialization

For this part, we use same paramater settings as before, except the number of samples is $\frac{2*45000}{batch\_size}$. In addition, we use the best lambda value we get, which is 0.0001. Finally, we check the sensitivity of initialization to normal distribution instead of He initialization to the weights matrices, with varying sigma sigs=[1e-1,1e-3,1e-4]. We apply this setting on two cases. First with batch normalization, and then without batch normalization. First, with batch normalization, we can can visualize the plots as follows with the corresponding test accuracies:

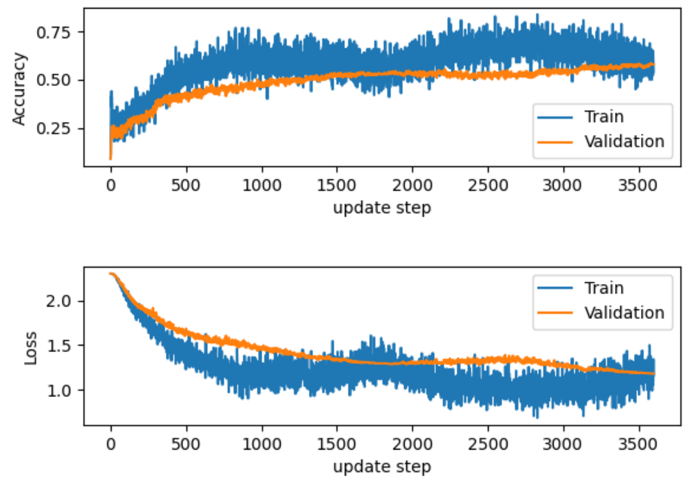
sigma=1e-1. The results at the end of the training were as Iter 3600/3600: loss=1.4002606453433202 acc=0.54 val_loss=1.51854141913791 val_acc=0.49494. Test



Accuracy: 0.4812]



(f) sigma=1e-3.The results at the end of the training were as IIter 3600/3600: loss=1.1063458752104511 acc=0.64 val_loss=1.1696663352758028 val_acc=0.593. Test Accuracy: 0.525

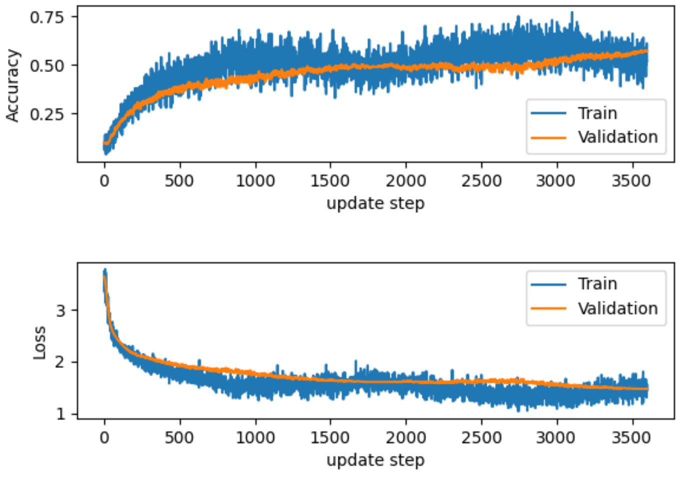


(g) sigma=1e-3.The results at the end of the training were as Iter 3600/3600: loss=1.1629339040561382 acc=0.54 val_loss=1.1818119521327897 val_acc=0.579. Test Accuracy: 0.5204

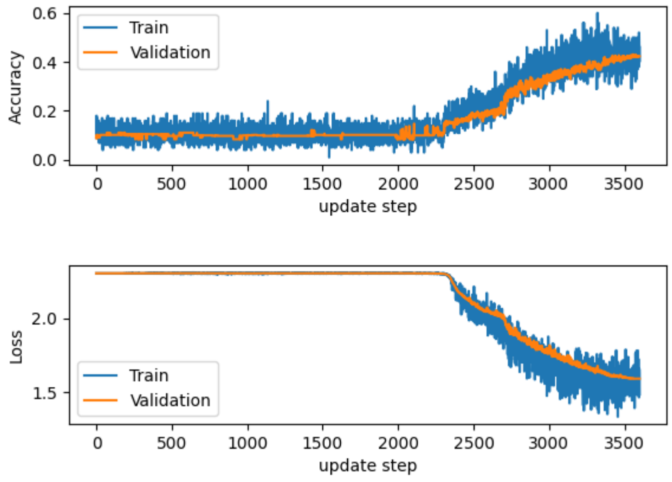
From the plots we notice how the test accuracy rises slightly as the sigma decreases. This emphasizes the importance of a relatively smaller variance when initializing the

weights matrix of the network.
Next we plot the graphs without batch normalization as follows:

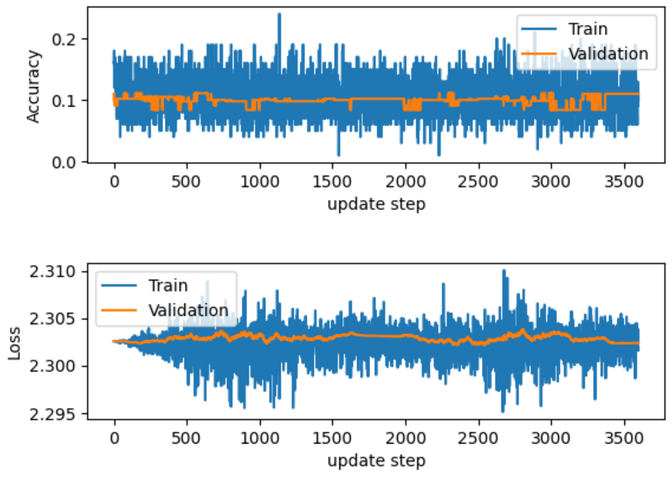
sigma=1e-1. The results at the end of the training were as Iter 3600/3600: loss=1.4477025446155163 acc=0.52 val_loss=1.4716155156451605 val_acc=0.579. Test



Accuracy: 0.4726]



(i) sigma=1e-3.The results at the end of the training were as Iter 3600/3600: loss=1.4871929433728168 acc=0.46 val_loss=1.5933355583555475 val_acc=0.423. Test Accuracy: 0.4119



(j) sigma=1e-3.The results at the end of the training were as Iter 3600/3600: loss=2.3018439434488758 acc=0.1 val_loss=2.302396848473235 val_acc=0.11114. Test Accuracy: 0.1

We notice that in contrast to batch normalization, the accuracies are generally slightly lower than these with batch normalization. In addition, a lower sigma leads to much lower accuracy on the test set.

References

[1] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.