Course: Deep Learning **(14-11-2021)**

Submission Assignment 1

Coordinator: Jakub Tomczak *Name:* Venkat Mohit Sornapudi, *Netid:* 2721697

1 Question answers

Question 1 *Work out the local derivatives of both, in scalar terms. Show the derivation. Assume that the target class is given as an integer value.*

We have: $l = -log(y_c)$, $y_i = \frac{e^{o_i}}{\sum_j e^{o_j}}$
So, by evaluating partial differentiation using chain rule we get the following in one step:

$$\frac{\partial y_i}{\partial o_j} = \begin{cases} -\frac{e^{o_j+o_i}}{\left(\sum_k e^{o_k}\right)^2} & \text{if } i \neq j \\ \frac{\sum_{k\neq i} e^{o_j+o_i}}{\left(\sum_k e^{o_k}\right)^2} & \text{if } i = j \end{cases}$$

$$\frac{\partial l}{\partial y_i} = \begin{cases} 0 & \text{if } i \neq c \\ -\frac{1}{y_c} & \text{if } i = c \end{cases}$$

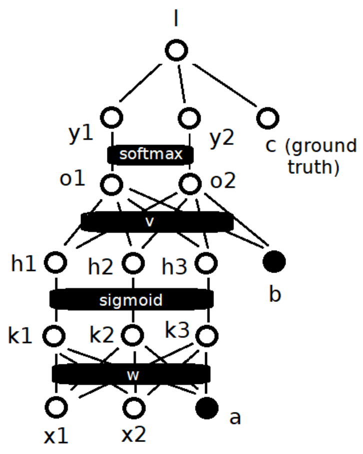
Question 2 *Work out the derivative $\frac{\partial l}{\partial o_i}$. Why is this not strictly necessary for a neural network, if we already have the two derivatives we worked out above?*

$$\frac{\partial l}{\partial o_j} = \sum_i \frac{\partial l}{\partial y_i}\frac{\partial y_i}{\partial o_j} = -\frac{1}{y_c}\frac{\partial y_c}{\partial o_j} = \begin{cases} \frac{e^{o_i}}{\sum_k e^{o_k}} = y_j & \text{if } j \neq c \\ -\frac{\sum_{k\neq c} e^{o_k}}{\sum_k e^{o_k}} = y_c - 1 & \text{if } j = c \end{cases}$$

If we have $\frac{\partial l}{\partial y_i}$ and $\frac{\partial y_i}{\partial o_j}$ values we can get the same result by implementing the step 1 or step 2 (of the calculation done above) in the code. In other words, we can do appropriate matrix multiplication of the matrices corresponding to $\frac{\partial l}{\partial y_i}$ and $\frac{\partial y_i}{\partial o_j}$. So, we can model the neural network without even calculating the above explicitly. But, implementing the end result of this calculation makes the code more efficient.

Question 3 *Implement the network drawn in the image below, including the weights. Perform one forward pass, up to the loss on the target value, and one backward pass. Show the relevant code in your report. Report the derivatives on all weights (including biases). Do not use anything more than plain python and the math package.*

The given neural network can be represented as follows:



The given neural network is implemented using the following initialisation:

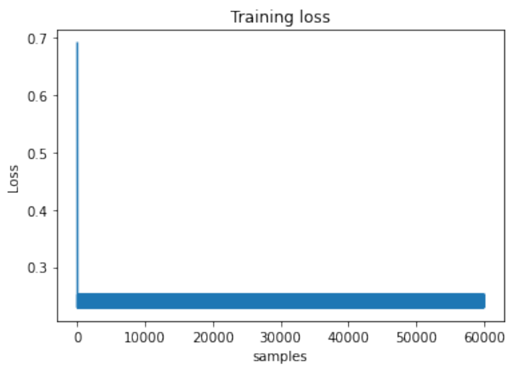
x = [1,-1], a = [0,0,0], w = [[1., 1., 1.], [-1., -1., -1.]], k = [0,0,0],
h = [[0,0,0], b = [0,0], v = [[1,1],[-1,-1],[-1,-1]], o = [0,0], y = [0,0],
grad_a = [0,0,0], grad_w = [[0,0,0],[0,0,0]], grad_k = [0,0,0], grad_h = [0,0,0],
grad_b = [0,0,0], grad_v = [[0,0],[0,0],[0,0]]*3, grad_o = [0,0], grad_y = [0,0]

The resultant gradients are:

grad_w = [[-0.0, -0.0, -0.0], [-0.0, -0.0, -0.0]]
grad_a = [0.0, 0.0, 0.0]
grad_v = [[-0.44039853898894116, 0.44039853898894116],
[-0.44039853898894116, 0.44039853898894116],
[-0.44039853898894116, 0.44039853898894116]]
grad_b = [-0.5, 0.5]

Question 4 *Implement a training loop for your network and show that the training loss drops as training progresses.*

Training loop has been implemented. The loss quickly drops and fluctuates at near zero values thus giving a thick horizontal line in Training loss curve as shown below:



Question 5 *Implement a neural network for the MNIST data. Use two linear layers as before, with a hidden layer size of 300, a sigmoid activation, and a softmax activation over the output layer, which has size 10.*

A vectorized version of fully connected neural network (MLP) with one hidden layer of size 300 has been implemented using SGD i.e. by taking one gradient step for each sample in the training data.

Question 6 *Work out the vectorized version of a batched forward and backward.*

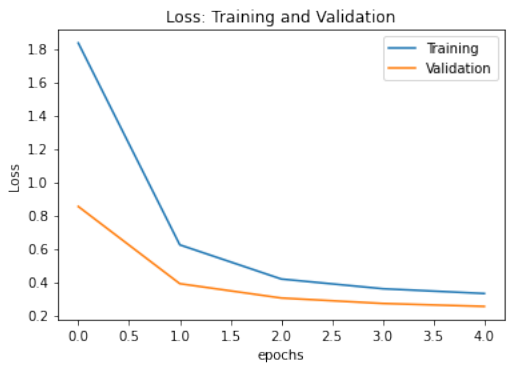
The vectorised version of MLP is implemented. (Found at the end of the submitted ipython notebook.)

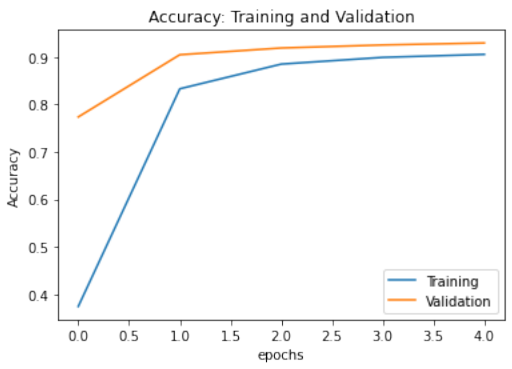
Question 7 *Train the network on MNIST and plot the loss of each batch or instance against the timestep.*

This part (Analysis) has been done by using SGD over each instance (not a mini-batch descent).

1. Compare the training loss per epoch to the validation loss per epoch. What does the difference tell you?

By training over 5 epochs the resultant loss and accuracy curves as below:



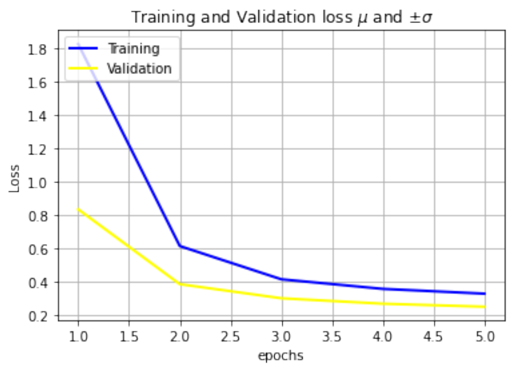


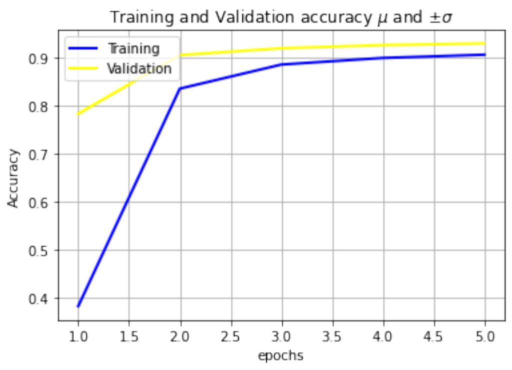
Surprisingly, here, validation loss is lesser than the training loss (vice-versa for accuracy). This might be due to overfitting over validation set samples. It can happen only when validation set is very similar to training set. If that is the case, as training loss is calculated after a forward pass on training samples where as validation loss is calculated after a backward pass on training samples and then a forward pass on validation samples, the validation loss can be lower. Only this argument seems to explain this behaviour.

2. Run the SGD method multiple times (at least 3) and plot an average and a standard deviation of the objective value in each iteration . What does this tell you?

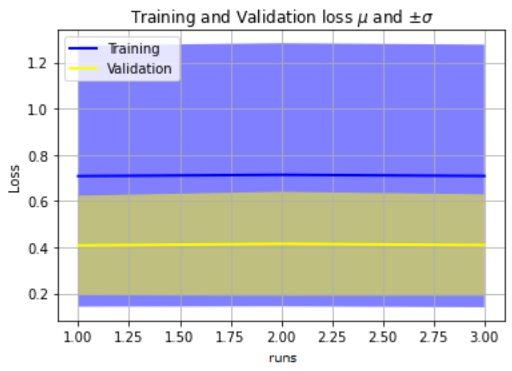
By training and validating for 3 times (as weights are initialised randomly, each run has different weights) the loss and accuracy show the following behaviour:

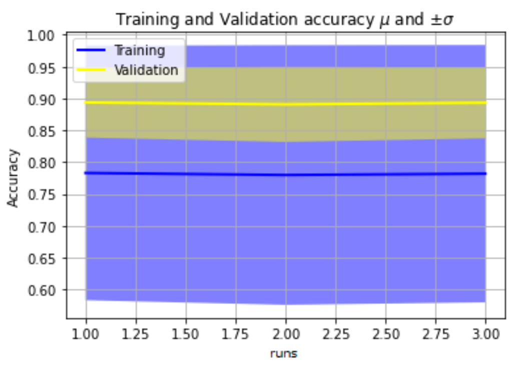
(As it isn't clear what iterations means here, I have generated following 4 graphs)





Analysis over runs:
The loss and accuracy in training and validation show same behaviour as in question 7.2 experiment. (Explanation is already given in 7.2 answer)



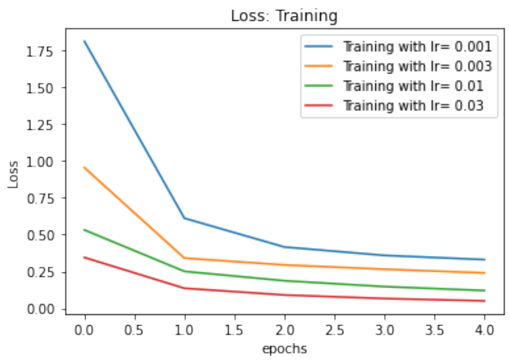


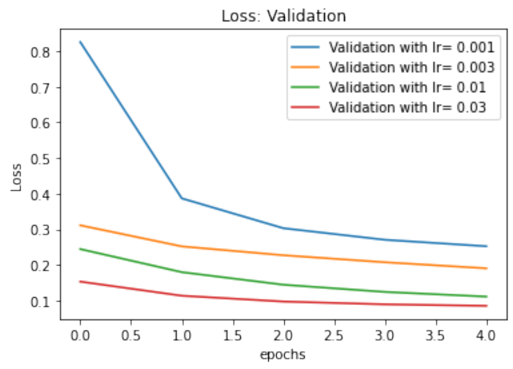
Analysis over epochs:
In the above graphs, the area filled (for 1 sigma) by standard deviation of validation (shown in yellow) loss

and accuracy is overlapped (in green) by training counterparts. So, the standard deviation of training loss and accuracy are higher.
Here,the loss and accuracy in training and validation show similar behaviour as in question 7.2 experiment i.e. validation loss mean is less than training loss mean (vice-versa for accuracy).

3. Run the SGD with different learning rates (e.g., 0.001, 0.003, 0.01, 0.03). Analyze how the learning rate value influences the final performance.

Upon running SGD using learning rates as 0.001, 0.003, 0.01, 0.03, the loss curves are as follows:





While using SGD, if learning rate is too high, there is a chance that the descent may skip the global minimum. If learning rate is too low, there is a chance that the descent may not reach the global minimum. It can happen when the descent can't escape a local minimum (which is not the global minimum) or it is so slow that it couldn't reach the global minimum in the given number of epochs.

Using the above argument, we can interpret that descents with the learning rates 0.001, 0.003, 0.01 are not reaching the global minimum as fast as the descent with learning rates 0.03. So, it can be concluded that for the given number of epochs 0.03 is the best learning rates. Note that the curves for 0.001, 0.003, 0.01 as learning rate are promising. The best learning rate for a given number of epochs (hyperparameter) doesn't guarantee that it would be the best one for a different number of epochs. But, here, as the curves are monotonic and well spaced we can assume that 0.03 would be the best (compared other values that are taken in the experiments). For getting the best learning rate and number of epochs combination a grid search can be done over the two hyperparameters.

4. Based on these experiments, choose a final set of hyperparameters, load the full training data with the canonical test set, train your model with the chosen hyperparameters and report the accuracy you get.

From the above experiments, it is observed that by taking learning rate as 0.03 gives the least loss. So, the hyperparameter i.e. learning rate is set to 0.03 to train and test on the MNIST dataset. The resultant training loss, test loss, training accuracy, and test accuracy are 0.0425528, 0.07912055, 0.9869833, 0.9754 respectively.