UNIVERSITY OF PENNSYLVANIA

ESE 546: PRINCIPLES OF DEEP LEARNING

FALL 2020

[11/03] HOMEWORK 4

DUE: 11/18 WED 1.30P ET

Changelog

Instructions

Read the following instructions carefully before beginning to work on the homework.

- You will submit solutions typeset in LaTeX on Gradescope (strongly encouraged). You can use hw_template.tex on Canvas in the "Homeworks" folder to do so. If your handwriting is *unambiguously legible*, you can submit PDF scans/tablet-created PDFs.
- Please start a new problem on a fresh page and mark all the pages corresponding to each problem. Failure to do so may result in your work not graded completely.
- Clearly indicate the name and Penn email ID of all your collaborators on your submitted solutions.
- For each problem in the homework, you should mention the total amount of time you spent on it.
- You can be informal while typesetting the solutions, e.g., if you want to draw a picture feel free to draw it on paper clearly, click a picture and include it in your solution. Do not spend undue time on typesetting solutions.
- You will see an entry of the form "HW 4 PDF" where you will upload the PDF of your solutions. You will also see entries like "HW 4 Problem 1 Code" and "HW 4 Problem 2 Code" where you will upload your solution for the respective problems. **For each programming problem, you should create a fresh Google Colab notebook**. This notebook should contain all the code to reproduce the results of the problem and you will upload the .ipynb file obtained from Colab. Name your notebook to be "pennkey_hw3_problem4.ipynb", e.g., I will name my code for Problem 4 as "pratikac_hw3_problem4.ipynb".
- **Remember that you should include all the relevant plots in the PDF, without doing so you will not get full credit. Your PDF solutions should be completely self-contained, we should not have have to look at/run the Colab notebook to check your solutions.**

Credit The points for the problems add up to 120. You only need to solve for 100 points to get full credit, i.e., your final score will be $\min(\text{your total points}, 100)$.

Problem 1 (60 points). (Can be done on your laptop) In this problem, we will implement logistic regression for classifying two classes (zero and one) from MNIST. You may not use any routines from PyTorch other than the ones that help download the data. Use Numpy to code up the optimizers.

(a) **(0 points)** First, prepare the dataset. Select all the samples belonging to the first two classes from MNIST's training dataset, this will be your training dataset. Similarly, create a validation dataset for the two classes by picking samples corresponding to the first two classes from the validation set of the MNIST dataset. You can subsample input images from 28×28 to 14×14 if you need.

(b) **(15 points)** Logistic regression solves for

$$\underset{w \in \mathbb{R}^d, w_0 \in \mathbb{R}}{\operatorname{argmin}} \; \frac{1}{n} \sum_{i=1}^{n} \log \left(1 + e^{-y_i \; (w^\top x_i + w_0)} \right) + \frac{\lambda}{2} \left(\|w\|_2^2 + w_0^2 \right) \tag{1}$$

where $x \in \mathbb{R}^{196}$. Set $y_i = 1$ for MNIST images labeled zero and $y_i = -1$ for MNIST images labeled one. Initialize the weights randomly for both the following parts but make sure that they are the same for both gradient descent and gradient descent with Nesterov's acceleration in part (c). You can try a few different values of λ and pick the one that gives the best validation error for the following parts.

Optimize the objective in (1) using gradient descent (note, not stochastic gradient descent) and plot the training *loss* as a function of the number of parameter updates on a semi-log scale (log scale on the Y-axis). This plot should be a straight line. As we saw in the class, the slope of this line should be about $-\kappa^{-1}$ for gradient descent. Compute the slope of the line in your plot and mention it clearly.

(c) **(10 points)** Write down the Hessian of the loss function in (1). Without assuming any special conditions about the dataset $\{(x_i, y_i)\}_{i=1,\ldots,n}$, is this problem strongly convex? What is the best strong convexity parameter for the loss function in (1)?

(d) **(15 points)** Optimize the objective in (1) again, this time using gradient descent with Nesterov's acceleration. If we knew the condition number κ, what momentum coefficient would we use for this problem? It is difficult, although possible, to evaluate κ, so let's use

$$\kappa = \frac{L}{m}$$

where we treat L as a hyper-parameter, i.e., we *choose* a value for L and set m to be the solution of part (b). Again, choose the value of L by trying out a few values and looking at the slope of the training loss for each setting. (Hint: the momentum parameter is typically between 0.75-0.95).

The slope of the semi-log plot of training loss versus the number of parameter updates in this case will be about $-\kappa^{-1/2}$ for Nesterov's updates. Note that we do not know the correct κ in this problem, so your slope may not match the value you chose for κ above. You should however see that the slope of the plot for Nesterov's acceleration is better than the slope of the plot you obtained in part (b).

(e) **(20 points)** We will now optimize the same problem with stochastic gradient descent (SGD). Use a batch-size $b = 128$ and optimize (1) using SGD with and without Nesterov's acceleration. Plot the training loss against the number of parameter updates on a semi-log scale (log scale on the Y-axis) for (i) gradient descent with Nesterov's updates (can be the same plot from your previous solutions), (ii) SGD without Nesterov's acceleration and, (iii) SGD with Nesterov's acceleration. Is the convergence for (iii) faster than that of (ii)? Comment on the differences for the convergence curves of (i) and (ii).

Problem 2 (60 points). (You need Colab/AWS for this problem) In this problem, we will see a heuristic that helps find out a good schedule for the learning rate. You will use the All-CNN network from Homework 2 in
https://gist.github.com/pratikac/68d6d94e4739786798e90691fb1a581b and train it on the CIFAR-10

dataset. Here is one learning rate schedule that has been noticed to work in practice

$$\eta(t) = \begin{cases} 10^{-4} + \frac{t}{T_0}\,\eta_{\max} & \text{if } t \leq T_0 \\ \eta_{\max} \cos\left(\frac{\pi}{2}\frac{t-T_0}{T-T_0}\right) + 10^{-6} & \text{if } T_0 \leq t \leq T. \end{cases} \tag{2}$$

Here t is the number of weight updates of the network. This is called the cosine learning rate schedule with a warmup. The first phase until $t \leq T_0$ is called the warmup phase and the second phase where the learning rate decreases along a cosine is called the annealing phase. The constants 10^{-4} and 10^{-6} at the beginning and the end are your choices; they do not typically matter much in practice.

There are three parameters in (2): the length of warmup T_0, the maximum learning rate $\eta_{\max}$ and the total number of weight updates T. Let us eliminate one parameter and set

$$T_0 = \frac{T}{5}.$$

For a mini-batch size of 128 in CIFAR-10, there are 391 weight updates in every training epoch. We will train for 50 epochs for this problem which gives

$$T = 50 \times 391 = 19,550.$$

The only parameter left to choose is $\eta_{\max}$.

(a) **(5 points)** Why does the cross-entropy loss for CIFAR-10 start at around 2.3 for the All-CNN network with the default PyTorch initialization?

(b) **(15 points)** We saw in the lecture that for gradient descent to converge, in particular to make monotonic progress, the learning rate is limited by the smoothness coefficient of the loss function. It is difficult to estimate the Lipschitz constant of the gradients of a typical deep network and therefore to pick a good learning rate: pick it too small and you train too slowly, pick it to be too large and the loss does not go down quickly enough. We will use exactly this property to pick $\eta_{\max}$. Here is the algorithm, often called the "learning-rate finder".

(1) Fix all hyper-parameters, e.g., dropout probability, weight-decay and momentum to some reasonable values. Say, pick weight-decay to be 10^{-3}, momentum coefficient to be 0.9 and dropout to 0.5.

(2) Start from $\eta(0) = 10^{-5}$. For the mini-batch of the training set at time t, set the learning rate to

$$\eta(t+1) = 1.1\,\eta(t);$$

you can also pick some other constant greater than 1 (typically, something around 1.1 works well) to increase the learning rate. Record the average training loss of each mini-batch separately and the learning rate $\eta(t)$ that was used for it for about 100 iterations. Plot the training loss (Y-axis) as a function of the learning rate (X-axis); use a log-scale for the X-axis. You should see $\ell(w^t)$ start off at high value, decrease rapidly with the increasing learning rate and then start increasing again after a certain value. The local minimum of this plot is the maximum learning rate one can use for this particular network, with this dataset and hyper-parameters.

Run the above algorithm and draw the plot in part (ii). Identify the global minimum of this plot (doing it manually is enough) and call the corresponding learning rate η^*. You do not need to be very precise in identifying the global minimum, so long as you eyeball something close to the global minimum you can proceed to the next step.

(c) **(20 points)** We cannot use the learning rate η^* directly to choose a value of η_{max} in (2). This is because the network had already trained for a few time-steps before we tried out η^* in part (a.ii). We should be more conservative in picking the maximum learning rate and will set

$$\eta_{\text{max}} = \frac{\eta^*}{10}.$$

Train the network with this value of η_{max} for 100 epochs as we had calculated above. Take care to ensure that you change the learning rate according to the schedule in (2) *before each weight update*. Note that you *cannot* simply create a new PyTorch optimizer because it will reset the momentum buffer. You will have to do something like

```
for g in optimizer.param_groups:
    g['lr'] = lr
```

before feeding each mini-batch. Plot the learning rate, training and validation loss and error as a function of the number of weight updates. You should compute the validation loss and error after every epoch.

(d) **(20 points) (This part will require some time to train)** There are heuristics in the optimization process that help pick hyper-parameters in practice. We picked the momentum parameter to be $\rho = 0.9$ in the previous two parts. We will now show experimentally that if

$$\frac{\eta_{\text{max}}}{1-\rho}$$

is kept unchanged, the validation error more or less remains the same. You will train the network for 50 epochs and measure the validation error at the end of the 50 epochs for three settings:

(i) η_{max} and $\rho = 0.9$
(ii) $\eta_{\text{max}} \leftarrow 5\eta_{\text{max}}$ and $\rho = 0.5$
(iii) $\eta_{\text{max}} \leftarrow \eta_{\text{max}}$ and $\rho = 0.5$

You will notice that the validation error is about the same for the first two but increases for the third setting.

Note: There are more such heuristics, e.g.,

$$\frac{\eta\,\lambda}{\mathscr{b}(1-\rho)};$$

is called the effective learning rate in the hyper-parameter optimization literature and often determines the validation error if everything else remaining the same. Here λ is the coefficient of weight-decay, ρ is the momentum parameter as above and $\mathscr{b}$ is the batch-size.

Such heuristics are incredibly useful in practice, for instance if one were to search for the best value for these 4 hyper-parameters using a grid-search, even with only 3 candidate values for each of them, one would need to train the network 81 times. You can perform a bisection search with the knowledge of this invariant to significantly reduce this cost. Note that these heuristics work well for image-classification problems, they do not work so well for recurrent networks.