UNIVERSITY OF PENNSYLVANIA

ESE 546: PRINCIPLES OF DEEP LEARNING

FALL 2020

[09/29] HOMEWORK 2

DUE: 10/12 MON 1.30P ET

Changelog

- **10/06 4pm: Problem 4 has been removed from this homework because we have not covered the material yet; we will include Problem 4 in the next homework.**
- **09/29 2p: Line 146. Fixed subquestion number typo.**
- **10/02 7p: Line 48: Fixed loss function notation**

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 2 PDF" where you will upload the PDF of your solutions. You will also see entries like "HW 2 Problem 5 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_hw2_problem5.ipynb", e.g., I will name my code for Problem 5 as "pratikac_hw2_problem5.ipynb". **Remember that you should also include the relevant plots in the PDF, without doing you will not get full credit for the problem.**
  - To be safe, make sure that the option "Omit code cell output when saving this notebook" is unchecked in the "Notebook Settings" tab in Colab. This way, the instructors may be able to give you partial credit for incorrect output/plots.

- **This is very important**. Note that the instructors will download your notebook and execute it on Colab themselves, so your notebook should be such that it can be executed independently without any errors to create all output/plots required in the problem.

Notice that even if Problem 5 contains multiple sub-parts, you should submit only one Colab notebook. In these cases, you will demarcate clearly within your notebook what is the output cell of each sub-problem. Jupyter notebooks can handle markdown very well so use it to write your solutions and narrative liberally. You should make sections/sub-sections in the Jupyter notebook for us to find your answers/plots easily. As an example, see Rahul Ramesh's Colab notebook at https://github.com/rahul13ramesh/cis520-dataset/blob/master/Clean_CIS520_Project.ipynb for his CIS 520 project. Notice how the code cells are preceded by text cells with titles appropriately written in bold. You can have plots for your debugging in the notebook but please comment them when you submit it to Gradescope.

Credit The points for the problems add up to 105 (Problem 4 was removed on 10/06). 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 (15 points). The torchvision library at https://pytorch.org/docs/stable/torchvision/models.html implements a number of popular architectures that you can use quickly in your code. In this problem, you will take a deeper look at residual networks at https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py. Understand how this architecture is coded up.

(a) (2 points) Note down all the peculiarities that you notice in the code. E.g., which one is better: using a batch-normalization layer before ReLU or after ReLU; what does this code do?

(b) (2 points) What do the calls "model.train()" and "model.eval()" do and where do you use them in a typical training and validation code? Why did we not have them in HW 1 when we wrote our own library for training deep networks?

(c) (3 points) Draw a rough picture of the resnet-18 architecture and note down the number of parameters in each layer; you will find it easier to to write a function that computes the number of parameters in each layer of the network.

(d) (3 points) Weight decay should not be applied to the biases of the different layers in the network, argue why this is the case.

(e) (5 points) Write the code to iterate over all the network layers in resnet-18 and separate out the parameters in three groups: (i) batch-norm affine transform parameters, (ii) biases of convolutional and fully-connected layers, and (iii) all the rest. There is no need to submit the code separately as a Jupyter notebook in this case, since these are a few lines of code just copy them out into your PDF solutions.

Problem 2 (20 points). Non-convex optimization problems are harder than convex optimization problems. There are however a few special non-convex problems that are easy. We will look at one of them here, namely unconstrained matrix factorization. Given a matrix $X \in \mathbb{R}^{m \times n}$ we would like to decompose it into two matrices of rank at most r

$$X = AB$$

where $A \in \mathbb{R}^{m \times r}$ and $B \in \mathbb{R}^{r \times n}$. Think of arranging all your data as columns of X. Columns of the matrix A are like elements of a dictionary, they correspond to different patterns in the data and are called atoms. The matrix B chooses which patterns to collect together in order to create a particular datum, i.e., column of X. Solving for factors A, B is usually done with constraints, e.g., B is typically forced to be a sparse matrix which enables regenerating data X using as few atoms as possible. We will solve a simpler problem:

$$A^*, B^* = \underset{A \in \mathbb{R}^{m \times r},\ B \in \mathbb{R}^{r \times n}}{\operatorname{argmin}} \|X - AB\|_F^2.$$

where $\|\cdot\|_F$ denotes the Frobenius norm.

(a) (4 points) Why is the above problem not convex?

(b) (8 points) The global optimum for this loss function can be obtained easily in spite of it being non-convex; find it. You may find it useful to write down the SVD of X.

(c) (8 points) Is the solution to the above optimization problem unique? Given one solution A^*, B^* name one way using which you can obtain another solution.

Problem 3 (15 points). Consider a loss function $\ell : \mathbb{R}^p \to \mathbb{R}$ that is invariant with respect to the scaling of the norm of its parameters $w \in \mathbb{R}^p$, i.e.,

$$\ell(\lambda w) = \ell(w)$$

for any scalar $\lambda > 0$.

(a) (5 points) Show that for such a function, the gradient scales down by λ, i.e.,

$$\frac{\mathrm{d}\ell(\lambda w)}{\mathrm{d}(\lambda w)} = \frac{1}{\lambda}\,\frac{\mathrm{d}\ell(w)}{\mathrm{d}w}.$$

We saw that is the case for the parameters of fully-connected layers (and thereby convolutional ones as well) with batch-normalization. You will find it useful to write down the definition of the derivative.

Due to this scale invariance, weight decay cannot affect the parameters of these layers at all. However, in practice, networks with batch-normalization have a much better validation error with weight-decay than without. This seems contradictory: if weight decay does not change the loss function in the presence of BN then how can it obtain a better validation error? We will try to understand this next.

(b) (10 points) Consider the gradient descent update

$$w^{t+1} = w^t - \eta\,\nabla\,\ell(w^t) \tag{1}$$

and show that if we define the direction of the weights as

$$v^t = \frac{w^t}{\|w^t\|_2}$$

the update in (1) can be written as

$$v^{t+1} = v^t - \frac{\eta}{\|w^t\|_2^2}\,\left(I - v^t v^{t\top}\right)\nabla\,\ell(v^t) + \mathcal{O}(\eta^2). \tag{2}$$

The coefficient

$$\eta' := \frac{\eta}{\|w^t\|_2^2}.$$

is the effective learning rate. The multiplier $I - v^t v^{t\top}$ is a matrix that transforms the gradient $\nabla\,\ell(v^t)$ linearly, we can think of this matrix as a rotation and scaling of the gradient.

Observe now that the weights may grow unbounded because batch-normalization does not differentiate between weights of different magnitudes. If this happens, the effective learning rate η' decreases drastically which essentially freezes the *direction* of the weights v^t. Weight decay counteracts this effect, it encourages the magnitude of the weights to go down which keeps the effective learning rate large and allows the direction of the weights to change.

You will do well to remember that in the presence of batch-normalization, only the direction of the weights in a deep network matters, the magnitude of the weights does not matter.

Problem 4 (30 points, THIS PROBLEM IS NOT A PART OF HOMEWORK 2. YOU DO

NOT HAVE TO SUBMIT IT. IT WILL BE INCLUDED IN THE NEXT HOMEWORK.). This problem will show that co-coercivity of the gradient and its Lipschitz continuity are equivalent. Assume that the function $f(x)$ is convex.

(a) (10 points) For a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$, shat that co-coercivity of ∇f implies Lipschitz continuity of ∇f. I.e., show that for all x, y

$$\langle \nabla f(x) - \nabla f(y), x - y \rangle \geq \frac{1}{L} \|\nabla f(x) - \nabla f(y)\|^2 \Rightarrow \|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|.$$

Hint: use the Cauchy-Schwartz inequality.

(b) (10 points) Show the converse, i.e., Lipschitz continuity of ∇f implies co-coercivity.

(c) (10 points) For a twice-differentiable function $f(x)$ with L-Lipschitz gradients and strong-convexity parameter m, show that

$$m \leq \|\nabla^2 f(x)\|_2 \leq L$$

for all x. Hint: use the mean-value theorem on a line that joins two points x, y.

Problem 5 (40 points). Use Google Colab for this problem. Neural networks are high-dimensional classifiers. While we may be able to train and regularize SVMs to have a large margin between two classes, it is often difficult to do the same for neural networks. A consequence of this is that, roughly speaking, all samples in typical training datasets lie close to the decision boundaries after training. This makes it quite easy to make minor perturbations to the input image—perturbations that are imperceptible to the human eye—and send the sample across the decision boundary so as to cause the network to mis-predict. We will synthesize such adversarial perturbations in this problem. You can read more about it at https://arxiv.org/abs/1412.6572; however be wary of the heuristic generalizations in this paper that are incorrect.

We will find the best adversarial perturbation to a given image x and its target y, this amounts to solving the optimization problem

$$\max_{\|x' - x\|_\infty \leq \epsilon} \ell(x', y) \tag{3}$$

where x' is the *variable of optimization* and it is the adversarially perturbed image corresponding to x, the quantity $\ell(x', y)$ is the loss computed on the image x' for the label y. We have chosen to make the parameters of the classifier w implicit for sake of clarity. This optimization problem searches for all images within an ϵ-ball of the original image x. We will use the ℓ_∞-norm

$$\|x\|_\infty = \max_k |x_k|$$

in this problem. Let us perform the Taylor expansion of the objective (3)

$$\ell(x', y) = \ell(x, y) + \epsilon\, d^\top \nabla \ell(x, y) + \mathcal{O}(\epsilon^2);$$

here $d = (x' - x)/\epsilon$. We can now write an approximate problem for finding adversarial perturbations as

$$\max_{\|d\|_\infty \leq 1} d^\top \nabla \ell(x, y).$$

Notice that the constraint implies that any element of the vector d can be perturbed by at most 1; there is no limit on the number of elements perturbed. The value of d that maximizes this objective is therefore the "signed gradient"

$$d_k = \frac{\nabla \ell(x, y)_k}{|\nabla \ell(x, y)_k|};$$

if $\nabla \ell(x, y)_k < 0$, the corresponding $d_k < 0$ and vice versa. The maximal objective is $\|\nabla \ell(x, y)\|_1$. The perturbation d is what we want to compute.

(a) (20 points) We will first train a convolutional neural network for this problem with all the bells and whistles. You can use the model code provided at https://gist.github.com/pratikac/68d6d94e4739786798e90691fb1a581b. You can use the example training code provided in https://colab.research.google.com/drive/1pCfUX2l-jikaOMUi_lgqN6Si6BKWPjfJ?usp=sharing. This is a small model with about 1.6M parameters. Train this model on the CIFAR-10 dataset for 100 epochs, you should try to get a validation error below 12%. You can use data augmentation such as mirror flips and brightness and contrast changes to improve your validation accuracy. Plot the training and validation losses and errors as a function of the number of epochs. Some hints for choosing hyper-parameters:

- Learning rate of 0.1 for the first 40 epochs, then 0.01 for the next 40 epochs and then 0.001 for the final 20 epochs.
- Weight decay of 10^{-3}.
- No need to perform data augmentation, although you can do so if you wish.
- Use SGD with Nesterov's momentum of 0.9 to train the network.

Make sure you save the parameters of the network because we will need them for the next part.

(b) (10 points) We will next compute the backprop gradient of the loss with respect to the input. We know that code of the form

```
...
yh = net.forward(x)
loss = loss.forward(yh, y)

loss.backward()
```

computes the gradient of the loss with respect to the weights, i.e., it computes $\overline{w}$ in our notation. You can get the gradient $\overline{x}$ very easily by adding the following line after loss.backward().

```
dx = x.grad.data.clone()
```

Plot this gradient dx for a few input images which the network classifies correctly and also for a few images which the network misclassifies. Comment on the similarities or the differences.

Note that each pixel of the RGB image x lies in $[0, 255]$, we will pick $\epsilon = 8$. Pick a particular mini-batch $\{x_1, \ldots, x_b\}$ with $b = 100$. For every image in this mini-batch perform the "5-step signed gradient attack", i.e., perturb that image 5 times using the signed gradient, at each step you feed in the perturbed image from the previous step and perturb it a bit more. Your pseudo-code will look as follows.

```
xs, ys = mini-batch of inputs and targets
for x,y in zip(xs, ys):
    for k in range(5):
        # forward propagate x through the network
        # backprop the loss
        dx = ...
        x += eps*sign(dx)
        # record loss on the perturbed image
        ell = loss(x, y)
```

Plot the loss on the perturbed images as a function of the number of steps in the attack averaged across your mini-batch.

(c) (10 points) Compute the accuracy of the network on 1-step perturbed images, i.e., for every image in the validation set, perturb the image using a 1-step attack and check the prediction of the network. How does this accuracy on adversarially perturbed images compare with the accuracy on the clean validation set?

(d) (0 points) The human brain also has a lot of neurons and is likely a high-dimensional classifier. Are humans susceptible to adversarial perturbations? Can you give examples of images that fool the human visual system? Are these "small", i.e., is $\|\epsilon/x\|_\infty$ small for these examples?

Problem 6 (15 points). Training of recurrent neural networks (RNNs) is often difficult because of the vanishing or exploding gradient problem. We will study this in a simple setting without any nonlinearities.

(a) (5 points) If the input to an RNN at the t^{th} timestep is $x^t \in \mathbb{R}^d$ and the hidden vector is $z^t \in \mathbb{R}^p$, the hidden vector z^{t+1} is given by

$$z^{t+1} = \sigma\left(w_x x^t + w_z z^t\right)$$

where $w_x \in \mathbb{R}^{p\times d}$ and $w_z \in \mathbb{R}^{p\times p}$ are weights and σ is a nonlinearity. If $\sigma(z) = z$, i.e., there is no nonlinearity, and $w_x = 0$ then the update boils down to

$$z^{t+1} = w_z z^t.$$

Write down the back-propagation gradient for w_z if the loss function is only a function of the hidden vector at time T , i.e., the loss function is $\ell(z^T)$. Compute the conditions on the weight matrix w_z under which the gradient explodes and vanishes.

(b) (2 points) Argue how the nonlinearity $\sigma(\cdot)$ affects the exploding/vanishing gradients. Which nonlinearities are well-suited for training RNNs?

(c) (3 points) Updates to the weights are computed using SGD as

$$w_z^{\text{new}} = w_z^{\text{old}} - \eta\,\frac{\mathrm{d}\ell}{\mathrm{d}w_z}.$$

We would like to protect the weights w_z^{new} from blowing up even if the gradient $\overline{w_z}$ explodes. Can you think of a way to do this? Similarly, can you modify these updates to handle the vanishing gradient problem?

(d) (5 points) Explain how an LSTM solves the problem of vanishing gradients.