UNIVERSITY OF PENNSYLVANIA

ESE 546: PRINCIPLES OF DEEP LEARNING

HOMEWORK 2

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.
- Clearly indicate the name and Penn email ID of all your collaborators on your submitted solutions.
- 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.
- For each problem in the homework, you should mention the total amount of time you spent on it. This helps us keep track of which problems most students are finding difficult.
- 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 1 PDF" where you will upload the PDF of your solutions. You will also see entries like "HW 1 Problem 1 Code" and "HW 1 Problem 3 Code" where you will upload your solution for the respective problems.
- **For each programming problem/sub-problem, you should create a fresh .py file**. This file should contain **all** the code to reproduce the results of the problem/sub-problem, e.g., it should save the plot that is required (correctly with all the axes, title and legend) as a PDF in the same directory. You will upload the .py file as your solution for "HW 1 Problem 3 Code" or "HW 1 Problem 3 Code". Name your file as pennkey_hw1_problem3.py, e.g., I will name my code as pratikac_hw1_problem3.py. Note, we will not accept .ipynb files (i.e., Jupyter notebooks), you should only upload .py files. If you are using Google Colab to do your homework (and I suggest that you don't...), you can export the notebook to a .py file.
- **In addition to submitting the code, you should append the entire Python code for the particular problem to the solution in the PDF. If you are using Latex, you can do something like the screenshot below. The instructors will execute the code to check it. Your code should run without any errors and should create all output/plots required in the problem.**

```
\includepackage{pythonhighlight}

\begin{python}

a = np.array(10)

...

\end{python}
```

Credit The points for the problems add up to 102. You only need to solve for 100 points to get full credit, i.e., your final score will be min(your total points, 100).

Problem 1 (20 points). The torchvision library at https://pytorch.org/vision/stable/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) Do you notice peculiarities that you notice in the code, e.g., which one is better, using a batch-normalization layer before ReLU or after ReLU?

(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) Write code to iterate over all layers of the network and count the number of parameters in each layer of the network. Execute this code for the Resnet-18 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 in this case, since these are a few lines of code just copy them out into your PDF solutions.

(f) (5 points) Augmentations are increasingly becoming much more important for deep learning that what we initially believed. And therefore there are a number of sophisticated ways to perform random augmentation. Go through the paper https://arxiv.org/abs/1909.13719 (you don't have to read the details in order to do this question) and its implementation in torchvision at https://pytorch.org/vision/main/generated/torchvision.transforms.RandAugment.html. The code of this implementation is at https://pytorch.org/vision/main/_modules/torchvision/transforms/autoaugment.html#RandAugment.

Take any one image of your choice (you can use the same image of the astronaut as that of the examples) and show the result of augmenting this image using the following 10 augmentations: (a) ShearX, (b) ShearY, (c) TranslateX, (d) TranslateY, (e) Rotate, (d) Brightness, (e) Color, (f) Contrast, (g) Sharpness, (h) Posterize, (i) Solarize and (j) Equalize. You will use existing functions from torchvision to perform these augmentations (see the code of RandAugment linked here to understand how to call each of them). Each of these augmentations has some parameters that you will have to choose in order to run these corresponding augmentation functions. Any reasonable values are okay, feel free to experiment. The goal of this problem is to understand how these augmentations work. You don't have to submit code in this case, pictures in the PDF correctly annotated with the parameters of the augmentation that was used to create them are fine.

Problem 2 (12 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) (2 points) Show that the above problem is not convex. You can give a counter-example.

(b) (6 points) The global optimum for this problem can be computed easily in spite of it being non-convex, find it. You may find it useful to write down the SVD of X.

(c) (4 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 (40 points). Use Google Colab to train the neural network, but after debugging everything on your laptop first. 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

$$\max_{\|x'-x\|_\infty \leq \epsilon} \ell(x', y) \tag{1}$$

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 series expansion of the objective in Eq. (1)

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

here $d = (x' - x)/\epsilon$. We can now write an optimization 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"

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

where $|\cdot|$ denotes the element-wise absolute value. 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 next.

(a) (20 points) We will first train a convolutional neural network for this problem with all the bells and whistles. You can use the code in allcnn.py provided at
https://gist.github.com/pratikac/68d6d94e4739786798e90691fb1a581b.
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 10%. You should use a GPU on Colab for training; else your code will be very slow. Roughly speaking, running for 100–200 epochs will take 2–4 hours, so be patient. 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}.
- You will need to perform data augmentation, at least mirror flips, and cropping and padding.
- You will also need to use dropout and batch-normalization.
- 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 $\mathrm{d}x$ 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.

```python
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 4 (30 points, You can try to do this problem on your laptop when you are debugging but Colab with GPU will train much faster). We will implement a model to predict the next character in a given sentence. We will use a few different sources of data to build this model (think carefully about how you will create a train and test set)

(1) The complete works of Shakespeare, which you can get at https://www.gutenberg.org/ebooks/100.txt.utf-8 which is hosted at https://www.gutenberg.org/ebooks/100.
(2) Leo Tolstoy's War and Peace as our training dataset. You can get the text file of the entire book from https://www.gutenberg.org/ebooks/2600.txt.utf-8 which is hosted at https://www.gutenberg.org/ebooks/2600.
(3) Any English book of your choice which is sufficiently long, e.g., its text file is at least 1 MB. You can find text files for many books at Project Gutenberg https://www.gutenberg.org.

Many different ways of creating the train and test set are possible (they will give similar values for the eventual loss). We will not do cross-validation in this problem, so just create one train set and one test set (which is itself used as a validation set).

(a) (5 points) First observe that characters in the text can be letters, numbers, punctuation, and newlines. We will represent each character using its one-hot encoding; you should do

```python
m = length(set(all_chars))
```

to get the correct number of unique characters. This is known as the size of the vocabulary in NLP. Do not worry about cleaning the dataset; neural networks are surprisingly good at handling unclean data. You will write a function that takes a part of the text input, say a sequence of 32 characters, and converts it into this embedding.

(b) (20 points) We now have written our problem in the standard form for an RNN where we are given a long sequence of data $x_1, x_2 \ldots$. Each element here is the embedding of a character or a punctuation mark. You will train an RNN with one hidden layer (feel free to use a deeper network if you'd like) which predicts the one-hot vector of the next *character* as output, given the past sequence. This corresponds to the operations

$$\begin{aligned}
h_{t+1} &= \tanh\left(w_h h_t + w_x x_{t+1} + b_h\right) \\
\mathbb{R}^{75} \ni \hat{y}_t &= \text{softmax}(w_y h_t + b_y) \\
\ell(\hat{y}_t, y_t) &= \ell(\hat{y}_t, x_{t+1}) = -\log(\hat{y}_t)_{x_{t+1}} \\
\ell &= \frac{1}{T}\sum_{t=1}^{T} \ell(\hat{y}_t, y_t).
\end{aligned}$$

Notice that we are going to predict softmax output logits $\hat{y}_t \in \mathbb{R}^{75}$ over the entire vocabulary. The loss is simply the cross-entropy loss of predicting the correct next character.

Code the RNN using PyTorch and train it. Plot the training loss and error of a mini-batch as a function of the number of weight updates (you will see a noisy plot, that's okay). The validation loss/error of an RNN is calculated the same way as the training loss/error, it is the cross-entropy loss on data that was not a part of the training set. Report the validation loss and error over a future sequence of 32 characters (averaged over the size of the mini-batch), every 1000 weight updates.

(c) (5 points) You should also report a few examples of the RNN for generating new sentences. To do so initialize the hidden state to zero and roll the RNN forward by setting x_{t+1} in the first equation above to be the prediction of previous step $\hat{y}_t$. You will notice that although the RNN obviously does not do a good job of generating correct words, it gets syntactic things like punctuation more or less correct.

Some tips: You should use the inbuilt nn.RNN module to build the recurrent network. You should also use torch.optim.Adam as the optimizer instead of SGD (we will study this soon) with a learning rate of 10^{-3}. Set the length of the sequence in the mini-batch to be $T = 32$. You may have to clip the gradients before calling optim.step() using the function torch.nn.utils.clip_grad_norm_. For more tips read the code at https://github.com/pytorch/examples/tree/master/time_sequence_prediction and https://github.com/pytorch/examples/blob/master/word_language_model/main.py carefully.