HW0

Submission instructions: Submit a PDF with the results of your experiments (summary, any relevant plots, etc). At the top of the PDF, inlcude links to your code as Github gists.

(For jupyter notebooks: drag-and-drop them onto gist, and make sure the file extension is `.ipynb` so it renders properly. Include the **output cells** for jupyter notebooks, so we know the results of running your code).

Getting Started

This first "warm up" problem will get you setup running deep learning experiments on a GPU.

Compute: Paperspace

We will need a GPU to train models quickly. If you have an existing GPU setup (eg via your lab), you can use that. Otherwise, we reccomend signing up for Paperspace (free).

- Create a free Paperspace account.
- Create a new Notebook instance, with the following settings: Runtime=pytorch, Machine type="Free P5000"
- DO NOT "add card" to your account – no payment is neccesary.

Paperspace gives you a virtual machine running JupyterLab, with a GPU attached. Instances auto-shutdown after 12 hours, or if you close the browser.

You can use this instances both for interactive development (Jupyter notebooks), or for launching long-running training jobs (via the terminal). We reccomend you use git for all your code, instead of relying on Paperspace for storage.

Framework: PyTorch

Deep learning frameworks like PyTorch make it easy to define models, compute gradients, and perform computations on the GPU. You can use any framework you're familiar with, though we will only "officially support" PyTorch.

To learn PyTorch, we reccomend working through the following tutorials: You can run these on your Paperspace VM:
1. https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
2. https://pytorch.org/tutorials/beginner/nn_tutorial.html

Note that there is no "magic" in these deep learning libraries, they just make certain operations (like differentiation) very convenient. It is possible to build and train neural-networks entirely from scratch, in < 200 lines of python. See micrograd by Andrej Karpathy.

Train CIFAR-10

We included a minimal notebook which trains a 5-layer CNN on CIFAR-10:
https://github.com/boazbk/mltheoryseminar/blob/main/code/hw0/simple_train.ipynb

Make sure you can run this on your GPU (/Paperspace):
- Clone the class repo: Open Terminal in JupyterLab, and `git clone https://github.com/boazbk/mltheoryseminar.git`
- Run the above notebook. It should reach ~85% test accuracy in ~4 minutes.
- Play around with parameters in this notebook to get a better feeling of training models (try changing the model definition, the batch sizes, the learning rate schedule, the number of train samples, the optimizer, etc). Record any interesting behaviors you notice.
- *Optional*: The network we define has Batch Normalization layers. Remove these layers and try to train the network. It should still work, but may be more sensitive to choices like initialization, layer widths, and learning rates. If you are interested, we recommend this survey on optimization considerations in neural networks. It is an open problem to understand neccesary and sufficient conditions that ensure neural-networks can be properly optimized with SGD (see the suggestions in the survey – can you replicate their predictions with experiments?)

Extra Tips

- Practical tips on training NNs, from Andrej Karpathy.
- How to Train Your ResNet by David Page, walks through the design and debugging process of training a state-of-the-art network. Note the careful experimental design at each stage.
- For longer-running training jobs, you may want to create a stand-alone training script instead of running jobs in notebooks.
- We *highly reccomend* using wandb (free) to track and log your experiments. For example, we like to use wandb to track train/test errors of models as they train, and automatically plot them in the web UI.
- You may like Pytorch Lightning which removes some pytorch boilerplate.

Problem 1: Robustness

Let's do a basic robustness check on the CNN you trained in the Warm Up:
What happens if you flip all of the images in the test set vertically?
Does an upside-down cat still look like a cat to your CNN?
- Using the same CNN you trained previously, report how much the test accuracy

drops when test images are flipped vertically.
- Report the accuracy drops per-class (are upside down dogs easier than cars?)
- Suppose you know that your CNN may be vision-impared in this way. Can you modify the training procedure to help robustness in this case? Try to train your CNN to minimize the maximum error across both original and flipped images. Report the original and flipped test accuracies of your best model, and describe your training procedure. (Hint: try data-augmentations. Pytorch Transforms may help).

Problem 2: Random Labels

In this problem, we will study the generalization gap of neural networks in the under and over-parameterized regime. This problem is based on the paper Zhang et. al..

Dataset: For this problem, we will consider label noised version of the CIFAR-10 dataset. That is, randomize the labels of p fraction of the dataset:

$$\bar{y}_i = \begin{cases} y_i & \text{with probability 1-p} \\ Uniform(0, 1...9) & \text{with probability p} \end{cases}$$

We will consider three noise levels $p \in \{0, 0.5, 1.0\}$.

Models + Training: We will use the same convolutional neural network architecture we trained in Problem 1, with two different widths (input paramter `c` in the function `make_cnn`)

1. Under-parameterized (c = 4): Cannot 'interpolate' the training dataset (i.e. reach zero training error)
2. Over-parameterized (c = 64): Can interpolate the training dataset

For both the above models and the three noise levels, train the networks long enough that they reach low training error. You may need to experiment with different learning rates or more epochs.

- Plot the final training error and test error of the models with p. Do you notice any differences in the two models?
- Discuss potential reasons behind these differences.

We strongly recommend using `wandb` for this problem. Using `wandb`, you can log the errors during training and compare them across hyperparameters. Here is an easy-to-follow guide for the same https://docs.wandb.ai/quickstart.

Problem 3: Linear Regression

In this problem we will compare various ways to solve a linear system, including via SGD.

Consider a verison of MNIST, where we want to classify images of digits $x \in \mathbb{R}^{784}$ as either even or odd, labeled as $y \in \{+1, -1\}$. Let's try to solve this with linear regression.

Specifically, we will unwrap the 28x28 pixel images into a vector in $\mathbb{R}^{784}$, and try to learn a linear model $f_\beta(x) = \langle \beta, x \rangle$ to predict whether the digit is even or odd. Take $n = 500$ samples from MNIST, denoted $\{(x_i, y_i)\}$ for pairs of images and their labels. Then, we will try to find a linear model f_β that minimizes the Mean-Squared-Error of predictions on the train set of n samples:

$$\text{argmin}_\beta \frac{1}{n} \sum_{i=1}^{n} (f_\beta(x_i) - y_i)^2$$

We can re-write this: collect the samples into a data matrix $X \in \mathbb{R}^{n \times 784}$. Let $Y \in \{+1, -1\}^n$ be the matrix of labels for these samples. Then, we wish to find a minimizer

$$\text{argmin}_\beta ||X\beta - Y||^2$$

where $\beta \in \mathbb{R}^{784}$. For convenience, we have provided code that sets up X, Y at: https://github.com/boazbk/mltheoryseminar/blob/main/code/hw0/mnist.ipynb

Now:

0. Compute the numeric (exact) rank of the matrix X.
1. This problem is "underspecified" (or "overparameterized"), meaning there are multiple possible solutions β with 0 loss. Find one solution using numpy's linear algebra package. (Hints: you can use either np.linalg.pinv or np.linalg.solve). Describe how the method works "under the hood", in as much detail as you wish. Compute and report the *test* mean-squared error of your classifier. That is, what is:

   $$\mathbb{E}_{x,y\sim D}[(f_\beta(x) - y)^2]$$

   where the expectation is over the MNIST (test) distribution.
2. Analytically compute the gradient of the objective

   $$L(\beta) := ||X\beta - Y||^2$$

   and (computationally) try to minimize L using GD or SGD. Does it work? Describe your experimental setting and results.
3. Investigate and describe why you think SGD did or did not work. (Hint: compute the spectrum of the matrix X, and think about its condition number)

4. *Optional*: What happens if we parameterize a linear model in a nonlinear way? For example, suppose we parameterize the weight vector as $\beta := W_1\beta_1$ where $W_1 \in \mathbb{R}^{784\times 784}$ and $\beta_1 \in \mathbb{R}^{784}$. The objective function $\tilde{L}(\beta_1, W_1) := L(W_1\beta_1)$ is now non-convex in its parameters, although it describes the same family of linear models. Try to apply SGD to optimize $\tilde{L}$. Does it reach a different solution from parts (1) and (2)?

5. *Optional*: Experiment with a "kernalized" version of the above problem, where you learn a model $F_\beta(x) = \langle \beta, \phi(x) \rangle$ for some high-dimentional embedding ϕ. Can you come up with an embedding that achieves smaller test loss than using raw pixels (as in part 1)? (Try using a random features embedding. Then try using an exact Gausian kernel, instead of a random-features approximation to the Gaussian kernel.).