COMS 4721 Spring 2022 Homework 4

Due at 11:59 PM on April 29, 2022

Grade for a *late submission* is scaled by factor of $(0.75)^{(\#\text{ days late, rounded up to an integer})}$

About this assignment. This homework assignment is a mix of some problems that involve programming and data analysis using Python, as well as some problems that do not involve programming but may require a bit of mathematical reasoning, simple derivations, and/or "calculations" (usually done by hand).

Data files needed for the programming/data analysis parts are available on Courseworks (under "Files").

Working in pairs is explicitly permitted on this assignment (subject to the conditions detailed in the syllabus).

Submission. For the programming and data analysis portions, please create and use a Jupyter Notebook to solve these problems. Include all of the code you write for this assignment in the notebook, and add appropriate comments so that the code is easy to understand. Make sure your answers to the various questions are clearly stated and labeled in separate Markdown cells (even if the answer also appears as an output of a code cell). You will need to submit the Jupyter Notebook on Courseworks as part of your homework submission.

You will also need to create a PDF document containing your solutions to *all problems* (including those involving programming and data analysis). **This PDF document should also contain the code and results from the Jupyeter Notebook.** Please read the homework policy (in the syllabus) for information about how to write-up your solutions, and please see the instructions for properly converting a Jupyter Notebook to PDF. You will need to submit this document on Gradescope.

- For redundancy purposes, please make sure your name and UNI appear prominently on the first page of the PDF. If you are working in pairs, both students' names and UNIs should appear in this manner.
- On Gradescope, for each problem, you need to **select all of the pages that contain your solutions** for that problem. If part of your solution is on an unselected page, we will consider it missing. See the instructions for submitting a PDF on Gradescope.
- On Gradescope, if you are working in pairs, then only one student in the pair should make the submission. That student must make sure to **add both students in the pair to the submission**. This is something that has to be done at the time of submission within the Gradescope submission system. See the instructions for adding group members on Gradescope. Do not forget to do this, or else one student will not receive credit for the homework submission!
- On Courseworks, submit only the Jupyter Notebook (a `.ipynb` file). Do not include any of the data files—we already have those `:-)`
- On Courseworks, if you are working in pairs, then only one student in the pair should make the submission of the Jupyter Notebook. (We are only using this to collect your Jupyter Notebooks; the grading will happen on Gradescope.) If both students submit, we'll only look at one of the submissions (and we'll pick one arbitrarily and ignore the other).
- Please submit both your write-up and Jupyter Notebook by the deadline. Do not wait until the last minute to make your submissions!

Please consult the Gradescope Help Center if you need further help with submitting on Gradescope, and the Canvas Guides if you need further help with submitting on Courseworks.

1 Pseudoinverse

The *(Moore-Penrose) pseudoinverse* of a matrix $A \in \mathbb{R}^{n \times d}$, typically denoted by $A^\dagger$ (or sometimes A^+), can be defined via the singular value decomposition of A as follows. Let r denote the rank of A, and let the singular value decomposition of A be written as

$$A = USV^\intercal$$

where $U = [\vec{u}_1 | \cdots | \vec{u}_r] \in \mathbb{R}^{n \times r}$ is the matrix of (orthonormal) left singular vectors of A, $V = [\vec{v}_1 | \cdots | \vec{v}_r] \in \mathbb{R}^{d \times r}$ is the matrix of (orthonormal) right singular vectors of A, and $S = \text{diag}(s_1, \ldots, s_r) \in \mathbb{R}^{r \times r}$ is the diagonal matrix of (positive) singular values of A.

The pseudoinverse $A^\dagger \in \mathbb{R}^{d \times n}$ of the matrix A is

$$A^\dagger = VS^{-1}U^\intercal.$$

Observe that $AA^\dagger A = A$ and $A^\dagger AA^\dagger = A^\dagger$.

Now consider a matrix $A \in \mathbb{R}^{n \times d}$ (with SVD $A = USV^\intercal$ as above) and vector $\vec{b} \in \mathbb{R}^n$. We are interested in determining the minimum Euclidean norm solution to the normal equations:

$$A^\intercal A\vec{w} = A^\intercal \vec{b}.$$

Problem 1.1 (1 point). Suppose $\vec{w}$ satisfies the normal equations. Briefly explain why $\vec{w}$ must also satisfy

$$V^\intercal \vec{w} = S^{-1}U^\intercal \vec{b}. \tag{1}$$

It's also easy to check that if $\vec{w}$ satisfies Equation (1), then it must also satisfies the normal equations.

Recall (from lecture) that the minimum Euclidean norm solution must be in the row space of A.

Problem 1.2 (1 point). Explain why $A^\dagger \vec{b}$ is in the row space of A and also satisfies Equation (1).

Problem 1.3 (1 point). Can there exist another vector $\vec{q}$ (different from $A^\dagger \vec{b}$) in the row space of A that also satisfies Equation (1). Briefly explain why or why not. And explain why this implies that $A^\dagger \vec{b}$ is the minimum Euclidean norm solution to the normal equations.

2 Low-rank approximation

Suppose you want to find a rank-1 matrix $\vec{b}\vec{c}^\intercal$ that minimizes

$$J(\vec{b}, \vec{c}) := \frac{1}{nd}\|A - \vec{b}\vec{c}^\intercal\|_{\mathrm{F}}^2$$

for a given matrix $A \in \mathbb{R}^{n\times d}$. (The scaling by $1/(nd)$ in J allows the interpretation as an average squared error.) Recall that $\|M\|_{\mathrm{F}}$ denotes the Frobenius norm of the matrix $M \in \mathbb{R}^{n\times d}$:

$$\|M\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{d}\sum_{j=1}^{d} M_{i,j}^2}.$$

By the Schmidt/Eckart-Young Theorem, we know this objective is minimized by $s_1\vec{u}_1\vec{v}^\intercal$, where s_1 is the top singular value of A, $\vec{u}_1$ is the top left singular vector of A, and $\vec{v}_1$ is the top right singular vector of A. (We can set $\vec{b} = s_1\vec{u}_1$ and $\vec{c} = \vec{v}_1$, for instance, but this is not the only possibility!)

Let's see how this plays out on the MNIST data, available in the file `ocr.mat`. It can be loaded as follows:

```
from scipy.io import loadmat
A = loadmat('ocr.mat')['testdata']
```

(Don't worry about the fact that this is the "test data".)

You can view an image (say, the first one) with the following commands:

```
import matplotlib.pyplot as plt
from matplotlib import cm
plt.imshow(A[0].reshape((28,28)), cmap=cm.gray_r)
plt.show()
```

> **Problem 2.1 (1 point).** Determine the minimum value of the objective J for the 10000×784 matrix loaded as `A` above. Compute this two ways. First way: evaluate the objective function J at $\vec{b} = s_1\vec{u}_1$ and $\vec{c} = \vec{v}_1$. Second way: use the singular values of A together with the formula from the Schmidt/Eckart-Young Theorem. Do you get the same answers? You may use any software package you like to compute the SVD, but report which software package you use in your write-up.

You'll use this data set (as the matrix A) for the remaining problems in this section.

2.1 Gradient descent

We can try to use gradient descent to minimize the objective function J directly. Note that the objective function has a *saddle point* at $(\vec{b}, \vec{c}) = (\vec{0}, \vec{0})$. This means the gradient of J at this point is zero, and yet it is (usually) not a extreme point of J (local minimum or local maximum). Gradient descent starting at such a point will not get very far. It is common in such cases to initialize gradient descent at randomly chosen values. However, to aid reproducibility, in the next problem you'll initialize gradient descent at very specific values.

> **Problem 2.2 (2 point).** Implement gradient descent for the objective described above as applied to the MNIST data. Run 250 iterations of gradient descent starting from $\vec{b}^{(0)} = (1, 1, \ldots, 1)$ and $\vec{c}^{(0)} = (1, 1, \ldots, 1)$, with step size $\eta = 1$. Make a plot of the objective value achieved as a function of the number of iterations, and report the final objective value.

You may use the automatic differentiation facilities in PyTorch if you like, but it is not required. However, you should not use any library functions that directly implement gradient descent (or other optimization procedures). Note that if you do use PyTorch, it is recommended that you do not compute the objective function using a straightline program that *explicitly* computes the $n \times d$ matrix $\vec{b}\vec{c}^\intercal$. *Hint*: You can rewrite

the objective J in a way that avoids the expression $\vec{b}\vec{c}^\top$ by using the fact that $\|M\|_F^2 = \mathrm{tr}(M^\top M)$, and also the cyclic property of the trace. (This may be helpful even if you are not using PyTorch.)

Problem 2.3 (1 point). Experiment with the step size η in your code from the previous problem to see if you can get gradient descent to converge more quickly, but still starting from $\vec{b}^{(0)} = (1, 1, \ldots, 1)$ and $\vec{c}^{(0)} = (1, 1, \ldots, 1)$. Report what step sizes you tried, and for each step size, how many iterations were needed to reach the same objective value.

2.2 Alternating minimization

Now let's consider using a different optimization method: alternating minimization.

Problem 2.4 (1 point). Write the formula for the gradient of J with respect to $\vec{b}$, i.e., $\frac{\partial J}{\partial \vec{b}}$. Determine the value of $\vec{b}_*$ for which $\frac{\partial J}{\partial \vec{b}}(\vec{b}_*, \vec{c}) = \vec{0}$. The answer may be given in terms of $\vec{c}$.

Problem 2.5 (1 point). Write the formula for the gradient of J with respect to $\vec{c}$, i.e., $\frac{\partial J}{\partial \vec{c}}$. Determine the value of $\vec{c}_*$ for which $\frac{\partial J}{\partial \vec{c}}(\vec{b}, \vec{c}_*) = \vec{0}$. The answer may be given in terms of $\vec{b}$.

The alternating minimziation updates are as follows. In iteration t:

1. Set $\vec{b}^{(t)}$ to the value $\vec{b}_*$ such that $\frac{\partial J}{\partial \vec{b}}(\vec{b}_*, \vec{c}^{(t-1)}) = \vec{0}$.
2. Set $\vec{c}^{(t)}$ to the value $\vec{c}_*$ such that $\frac{\partial J}{\partial \vec{c}}(\vec{b}^{(t)}, \vec{c}_*) = \vec{0}$.

This algorithm may seem familiar!

Problem 2.6 (2 point). Directly implement the alternating minimization algorithm as described above. Run it for 5 iterations, starting with $\vec{b}^{(0)} = (1, 1, \ldots, 1)$ and $\vec{c}^{(0)} = (1, 1, \ldots, 1)$. Make a plot of the objective value achieved as a function of the number of iterations, and report the final objective value.

2.3 Clustering/quantization

Let's return to the general rank-k approximation problem, so the goal is to find $B \in \mathbb{R}^{n \times k}$ and $C \in \mathbb{R}^{k \times d}$ that minimize the objective function

$$J(B, C) = \|A - BC\|_F^2.$$

However, as we saw in lecture, sometimes it is interesting to consider further restrictions on the factors B and C so that they have some interesting semantics. Here is a common restriction: allow $C \in \mathbb{R}^{k \times d}$ to be anything, but require each row of $B \in \mathbb{R}^{n \times k}$ to have a single non-zero entry whose value is one. For example, if $n = 9$ and $k = 5$, then B might look like

$$B = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

When B is subject to these restrictions, we have the following interpretation of BC as an approximation to A: the i-th row of A is approximated by one of the rows of C (as specified by the location of the non-zero entry in the i-th row of B). This is called *clustering* or *quantization*.

SVD won't necessarily be useful for finding a solution that respects these constraints on B. But we can devise an optimization algorithm based on the alternating minimization strategy.

Problem 2.7 (1 point). Suppose $B \in \mathbb{R}^{n \times k}$ is fixed, and it obeys the constraints described above (i.e., every row has a single non-zero entry whose value is one). Write a formula/mathematical expression that explicitly describes the value of $C \in \mathbb{R}^{k \times d}$ that minimizes the objective $J(B, C)$. Explain why your answer is correct.

Problem 2.8 (1 point). Now suppose $C \in \mathbb{R}^{k \times d}$ is fixed (and recall it is unconstrained). Write a formula/mathematical expression that explicitly describes the value of $B \in \mathbb{R}^{n \times k}$ that minimizes the objective $J(B, C)$, subject to the restrictions on B described above. Explain why your answer is correct.

Problem 2.9 (2 point). Implement the alternating minimization algorithm as suggested by the previous two problems, with $k = 20$. (Holding C fixed, find B to minimize the objective; then holding B fixed, find C to minimize the objective; repeat.) You will need to initialize C somehow. Do this by setting the j-th row of C is equal to the j-th row of A (for all $j = 1, \ldots, k$). Here, we're assuming $k \leq n$, of course. (In practice, one would use a more clever, possibly randomized, method of initialization.) We don't need to initialize B, since it will be determined in the first step of the algorithm. Run the algorithm until it converges (i.e., there are no more changes to B or C).

- How many iterations are required? (If t is the first iteration such that at the end of iteration t, you find that B and C are the same as what they were at the end of iteration $t - 1$, then report t.)
- What is the final objective value?
- Make a plot of the objective value achieved as a function of the number of iterations.
- It turns out every row of C will be a convex combination (i.e., weighted average) of some rows of A. So we can actually treat each row as an 28×28 pixel image. Display all $k = 20$ such images side-by-side. Does it seem like all 10 digits are represented?

Problem 2.10 (2 point). Repeat Problem 2.9 with $k = 10$.

3 Neural networks

In this problem, you will train neural networks on synthetic and OCR data using PyTorch. Template code (`hw4-nn.py`) is provided on Courseworks. (Feel free to turn this into a Jupyter notebook as appropriate.) A PyTorch tutorial is available here: https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

Some PyTorch functions/classes/methods that may be useful:

- `torch.Tensor`
- `torch.Tensor.backward`
- `torch.Tensor.view`
- `torch.nn.BCEWithLogitsLoss`
- `torch.nn.Linear`
- `torch.nn.Module`
- `torch.nn.Module.parameters`
- `torch.nn.Module.zero_grad`
- `torch.relu`

3.1 One-layer neural network on XOR data

First, start with a one-layer neural network $f_\theta \colon \mathbb{R}^2 \to \mathbb{R}$ given by

$$f_\theta(x) := W_1 x + b_1, \quad \text{for all } x \in \mathbb{R}^2,$$

where $\theta = (W_1, b_1) \in \mathbb{R}^{1\times 2} \times \mathbb{R}^1$ are the parameters of the function. (We are using an identity activation function for the output node.) This is just an affine function! We shall use it as a binary classifier via

$$x \mapsto \mathbf{1}\{f_\theta(x) > 0\}.$$

Implement such a neural network in PyTorch, as well as a gradient descent procedure to fit the parameters to the training data $(x_1, y_1), \ldots, (x_4, y_4) \in \mathbb{R}^2 \times \{0, 1\}$,

$$\begin{aligned}(x_1, y_1) &= ((-1, -1), 0),\\ (x_2, y_2) &= ((+1, -1), 1),\\ (x_3, y_3) &= ((-1, +1), 1),\\ (x_4, y_4) &= ((+1, +1), 0).\end{aligned}$$

This is the "XOR" pattern that we gave as an example of a non-linearly separable data set. This data set is provided by the `XOR_data()` function in the template code. Run 25 iterations of gradient descent with step size $\eta = 1.0$ on the average "logistic loss" objective $J(\theta)$, defined as follows:

$$J(\theta) = \frac{1}{4}\sum_{i=1}^{4}\Big(y_i \ln(1 + \exp(-f_\theta(x_i))) + (1 - y_i)\ln(1 + \exp(f_\theta(x_i)))\Big).$$

In PyTorch, the "logistic loss" is implemented as `torch.nn.BCEWithLogitsLoss`. Use the default PyTorch initialization scheme to set the initial parameters for gradient descent, but do so *immediately after* setting the random number seed using `torch.manual_seed(0)`. (This will help reproducibility!) Record the objective values and error rates prior to training, and also after every iteration of gradient descent.

Problem 3.1 (2 points). Report the following for the one-layer neural network described above:

- The objective value at initialization
- The objective value after 25 iterations of gradient descent
- The training error rate at initialization
- The training error rate after 25 iterations of gradient descent

Also plot the objective value as a function of the iteration number, and separately plot the training error rate as a function of the iteration number.

3.2 Two-layer neural network on XOR data

Next, consider the two-layer neural network function $f_\theta \colon \mathbb{R}^2 \to \mathbb{R}$ given by

$$f_\theta(x) := W_2\sigma(W_1 x + b_1) + b_2, \quad \text{for all } x \in \mathbb{R}^2,$$

where $\theta = (W_1, b_1, W_2, b_2) \in \mathbb{R}^{2\times 2} \times \mathbb{R}^2 \times \mathbb{R}^{1\times 2} \times \mathbb{R}^1$ are the parameters of the function, and $\sigma \colon \mathbb{R} \to \mathbb{R}$ is the ReLU function $\sigma(z) := \max\{0, z\}$, which is applied coordinate-wise to vector arguments. (Again, we are using an identity activation function for the output node.) Implement this two-layer neural network in PyTorch and a corresponding training procedure for the same XOR data.

Problem 3.2 (2 points). Repeat Problem 3.1 but using the two-layer neural network.

3.3 Three-layer neural network on handwritten digits

Next, consider a three-layer neural network $f_\theta \colon \mathbb{R}^{64} \to \mathbb{R}$ with the following architecture:

- Layer 1: fully-connected (linear) layer with 64 inputs and 64 outputs, with a bias term. Use the ReLU activation function.
- Layer 2: fully-connected (linear) layer with 64 inputs and 32 outputs, with a bias term. Use the ReLU activation function.
- Layer 3: fully-connected (linear) layer with 32 inputs and 1 output, with a bias term. Use identity (i.e., no) activation function.

In other words,

$$f_\theta(x) := W_3\sigma(W_2\sigma(W_1 x + b_1) + b_2) + b_3, \quad \text{for all } x \in \mathbb{R}^{64},$$

where $\theta = (W_1, b_1, W_2, b_2, W_3, b_3) \in \mathbb{R}^{64\times 64} \times \mathbb{R}^{64} \times \mathbb{R}^{32\times 64} \times \mathbb{R}^{32} \times \mathbb{R}^{1\times 32} \times \mathbb{R}^1$ are the parameters of the function, and $\sigma \colon \mathbb{R} \to \mathbb{R}$ is the ReLU function $\sigma(z) := \max\{0, z\}$, which is applied coordinate-wise to vector arguments.

Implement this network in PyTorch and a corresponding training procedure, with the following changes.

- Instead of the XOR data set, use the "digits" data set that is prepared using the `digits_data()` function in the template code. This data set is obtained using scikit-learn: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html. The first 180 examples are reserved as test examples; the remaining 1617 examples are used as training examples. The binary label to predict is 1 for odd digits, and 0 for even digits.
- You will have to "reshape" the inputs, as each image is given as an 8×8 array, but your network takes 64-dimensional vectors as inputs. (The template code shows how to do this.)
- In your gradient descent implementation, use a step size of $\eta = 0.1$ (instead of 1.0), and use 500 iterations (instead of 25).

Problem 3.3 (2 points). Repeat Problem 3.1 for the three-layer neural network on the "digits" data set. In addition, report the test error rate (using the test examples) at initialization and after the 500 iterations of gradient descent, and also make a plot of the test error rate as a function of the iteration number.