Homework 7 and 8
APPM 5630 Spring 2021
Advanced Convex Optimization

Due date: Friday, Mar 19 at midnight (via Gradescope) **Instructor**: Prof. Becker
Theme: Convex optimization **Solution date**: 3/5/22

Instructions Collaboration with your fellow students is allowed and in fact recommended, although direct copying is not allowed. The internet is allowed for basic tasks. Please write down the names of the students that you worked with. An arbitrary subset of these questions will be graded.

Homework submission instructions at github.com/stephenbeckr/convex-optimization-class/tree/master/Homeworks. You'll turn in a PDF (either scanned handwritten work, or typed, or a combination of both) to **gradescope**, using the link from the Canvas assignment.

Reading Read chapter 5 in [BV2004].

Homework 7: Duality and TV denoising

Problem 1: [Optional but recommended] Problem 5.16 in [BV2004] on exact penalties (and read problem 5.14)

Problem 2: 2D total-variation (TV) is a seminorm that acts on an image (i.e., matrix) $X \in \mathbb{R}^{n_1 \times n_2}$ as

$$\mathrm{TV}(X) = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} \sqrt{(X_{i+1,j} - X_{i,j})^2 + (X_{i,j+1} - X_{i,j})^2}$$

where we define $X_{i,j} = 0$ if either $i > n_1$ or $j > n_2$ (i.e., zero boundary conditions). This is used as penalty that discourages changes in value in the matrix, and hence can be used to regularize optimization problems involving images that have large regions of constant intensity (see Homework 8's phantom image for an example). The above definition is known as "isotropic TV" because it is invariant under orthogonal rotations of the image. There are other variants (e.g., other boundary conditions, differentiable but non-isotropic variants, higher-order TV to reduce staircasing effects, etc.). See An introduction to continuous optimization for imaging by Chambolle and Pock (*Acta Numerica*, 2016) for a recent survey-level article on TV and related topics in variational imaging.

One way to think of the TV operator is

$$\mathrm{TV}(X) = \sum_{i=1}^{n_1 n_2} \left\| \begin{bmatrix} y_i^h \\ y_i^v \end{bmatrix} \right\|_2 \quad \text{or} \tag{1}$$

$$= \sum_{i=1}^{n_1 n_2} \|y_i^h + \sqrt{-1} y_i^v\|_1 \quad \text{(this is the complex } \ell_1 \text{ norm)} \tag{2}$$

where

$$y^h = L_h(X)$$
$$y^v = L_v(X)$$

and L_h and L_v are the horizontal and vertical standard forward finite-difference operators. In other words, the map $X \mapsto (L_h(X), L_v(X)) = (y^h, y^v)$ is a *discrete gradient* operator and its adjoint will be, up to a negative, a form of a *discrete divergence*. (Note: I am using the notation $\sqrt{-1}$ instead of i since we've used i as an index variable.) If we vectorize X in column-major order, and let the $n \times n$ matrix D_n be defined

$$D_n = \begin{bmatrix} -1 & 1 & 0 & \dots & 0 \\ 0 & -1 & 1 & \dots & 0 \\ 0 & 0 & -1 & \dots & 0 \\ \vdots & & & \ddots & 1 \\ 0 & \dots & & & -1 \end{bmatrix}$$

and I_n be the $n \times n$ identity matrix, then we can define

$$L_h = D_{n_2} \otimes I_{n_1}, \quad L_v = I_{n_1} \otimes D_{n_1}$$

where $\otimes$ is the Kronecker product (this is `kron` in Matlab and `numpy.kron` in Python)

$$A \otimes B \overset{\text{def}}{=} \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}.$$

If you haven't had a numerical PDE class, then this Kronecker product notation might be unmotivated. The reason is that for general matrices of the right size, using column-major vectorization, then $(B^T \otimes A)\text{vec}(X) = \text{vec}(AXB)$ hence $L_v \cdot \text{vec}(X) = \text{vec}(D_{n_1} X I_{n_2}^T) = \text{vec}(D_{n_1} X)$, meaning we are taking discrete differences of the rows of X. Similarly $L_h \cdot \text{vec}(X) = \text{vec}(I_{n_1} X D_{n_2}^T)$ so we see that we are essentially applying D_{n_2} to X^T, i.e., taking discrete differences of the columns of X.

If you use the Eq (1) interpretation, then define the discrete gradient operator L as either $L : X \mapsto (L_h(X), L_v(X))$ with $\varphi : \mathbb{R}^2 \to \mathbb{R}$ as $\varphi(y_1, y_2) = \sqrt{y_1^2 + y_2^2}$, and define $g(y) = \sum_{i=1}^{n_1 n_2} \varphi(y_{i,1}, y_{i,2})$ with domain $\mathbb{R}^{n_1 n_2 \times 2}$.

Or if you chose to use the Eq (2) interpretation, define the discrete gradient operator L as $L : X \mapsto L_h(X) + \sqrt{-1} L_v(h)$ and $\varphi : \mathbb{C} \to \mathbb{R}$ as $\varphi(z) = |z| \overset{\text{def}}{=} \sqrt{\Re(z)^2 + \Im(z)^2}$ where $\Re$ means taking the real-part and $\Im$ means taking the imaginary part. Define $g(y) = \sum_{i=1}^{n_1 n_2} \varphi(y_i)$ with domain $\mathbb{C}^{n_1 n_2}$.

Using either interpretation, we can now write $\text{TV}(X) = g(L(X))$. Then the TV denoising problem is, given a noisy image Y, find a nearby proposed image X that has small TV semi-norm:

$$\begin{aligned} \min_{X \in \mathbb{R}^{n_1 \times n_2}} \quad & \frac{1}{2} \|X - Y\|_F^2 \\ \text{subject to} \quad & g(L(X)) \le \tau. \end{aligned} \tag{P}$$

This form of the problem is not amenable to efficient first-order methods because it is not easy to project onto these constraints. However, an appropriate dual problem will be efficient. First rewrite the problem as

$$\begin{aligned} \min_{X, z} \quad & \frac{1}{2} \|X - Y\|_F^2 \\ \text{subject to} \quad & g(z) \le \tau, \quad L(X) = z. \end{aligned} \tag{P'}$$

$\boxed{\text{Find the dual of this problem (P').}}$ (Hint: only dualize the equality constraint; that is, keep the inequality constraint implicit in the Lagrangian and do not introduce a dual variable for it).

Homework 8: FISTA and TV denoising

See the end of the homework for more information on FISTA. We build a bit on the logistic regression from last homework. You may use the files on the class github website, or your classmates' files, to setup the spam classification problems from the previous homework.

Problem 1: Re-run the logistic regression on the spam data from Homework 6, but use any of Nesterov's accelerated methods instead of plain gradient descent (please code this yourself, do not use a software package). You may use a fixed stepsize (a backtracking line search is optional). Also run the logistic regression using an existing implementation of a derivative-free method, such as the Nelder-Mead simplex (direct search) method, which is implemented for you in Matlab via `fminsearch`, and in Python via `scipy.optimize.fmin`. [Note: the Nelder-Mead simplex algorithm is *unrelated* to "the" simplex method for linear programming]. Do the same pre-processing as Homework 6.

Deliverables: A printed write-up, include a plot of the objective value in the logistic regression problem using (1) last week's gradient descent method, (2) the new Nesterov accelerated method, and (3) the derivative-free code. You may want to plot the y-axis in log scale (e.g., `semilogy` in Matlab, or `matplotlib.pyplot.semilogy` in Python). You can include relevant parts of code if you want, but it's not necessary. To be clear, you should implement the Nesterov method yourself, but the derivative-free method can use a package.

Problem 2: Re-run the logistic regression on the spam data, this time adding a ℓ_1 penalty, and using a proximal gradient method. Specifically, solve

$$\min_w \ \ell(w; y, X) + \lambda \|w\|_1$$

for $\lambda = 5$. Your code should allow for a generic proximity function that maps

$$(y, t) \mapsto \operatorname{prox}_{tg}(y) = \operatorname*{argmin}_x \ tg(x) + \frac{1}{2}\|x - y\|_2^2.$$

In our case, $g(x) = \lambda\|x\|_1$ and

$$\operatorname{prox}_{tg}(y) = \operatorname{sign}(y) \cdot \lfloor |y| - t\lambda \rfloor_+$$

where each operation is done component-wise, and $\lfloor a \rfloor_+ = \max(a, 0)$. Don't forget the t factor! You can use a standard proximal gradient method, or an accelerated method like FISTA. You may use a fixed stepsize of $1/L$. Do the same pre-processing as Homework 6.

Deliverables: A printed write-up, include a plot of the optimal weights w obtained both with and without the ℓ_1 regularization, and also the testing/training classification accuracy with and without the regularization.

Problem 3: TV denoising in CVX/CVXPY. Implement TV denoising of the MRI phantom image using CVX or CVXPY. Using these solvers, we cannot scale to large problems (if you wanted to, follow the ideas of Homework 7 and solve with a proximal method), so we will work with a 150×150 image. Specifically:

In Matlab, you can generate the MRI phantom image with
`Y = phantom('Modified Shepp-Logan',150)`. For python users, I have saved several file formats of this image and put them on Canvas (e.g., a pickle pkl file, a png file, an ascii file with whitespace separator, and a .mat file which can be read using the `scipy.io.loadmat` library — use whichever format is easiest for you). The image Y should have values in $[0, 1]$.

Add a simple version of salt-and-pepper noise (a more realistic noise model would be shot noise). Randomly pick 10% of the pixels, and add a uniform random number in $[0, 1]$ to these pixels. See Fig. 1. Call this noisy version `Y_noisy`. You'll use this for the value of Y in Eq. (P).

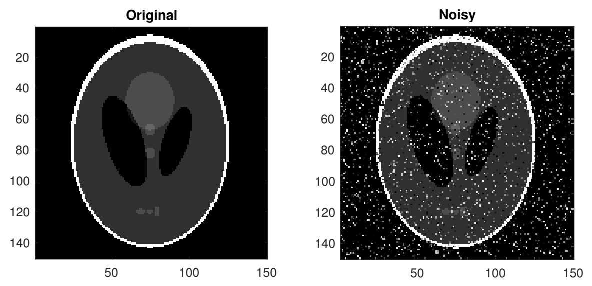

Figure 1: MRI phantom test image, with and without noise. An experiment with this test image was the beginning of the theory of compressed sensing; see Robust Uncertainty Principles: Exact Signal Reconstruction from Highly Incomplete Frequency Information by Candès, Romberg, Tao 2004.

Make the discrete gradient linear operator described in Homework 7 and represent it with a sparse matrix. You may want to use `spdiags` (for D_n) and `kron` (in python, `scipy.sparse.spdiags` and `scipy.sparse.kron`).

Using the discrete gradient, make a CVX/CVXPY compatible function to represent the TV semi-norm. In cvx, you may use either the complex version and `norm( ,1)` which supports the complex ℓ_1 norm, or a combination of `sum` and `norms( , 2, 2)`. In python, use `cvx.sum_entries`, `cvx.norm2`, and `cvx.hstack`.

Finally, solve the TV denoising problem (P) (from Homework 7) using τ as 1/4 the value of the TV semi-norm of `Y_noisy`. Include constraints that $0 \leq X_{i,j} \leq 1$.

Deliverables: There need not be a lot of code as we are using CVX/CVXYPY, so please print out the code. Include figures of your noisy image as well as the denoised image given from solving (P).

References for Nesterov's method: Note that we informally refer to Nesterov's accelerated method and FISTA interchangeably. FISTA (and also another paper from Nesterov around the same time) was important because it allowed a proximity operator term. The basic method looks like this (for $\min_x f(x) + g(x)$) and $t = 1/L$ where ∇f is L-Lipschitz continuous:

$$x_{k+1} = \text{prox}_{tg}\left(y_k - t\nabla f(y_k)\right)$$
$$y_{k+1} = x_{k+1} + \frac{k}{k+3}\left(x_{k+1} - x_k\right).$$

For Nesterov's method, there are several variants of **Nesterov's method** to choose from. See Gradient-Based Algorithms with Applications to Signal Recovery Problems by Amir Beck and Marc Teboulle (2010) page 23 for "FISTA" code (both fixed stepsize and backtracking), or Lecture 9. Accelerated proximal gradient methods in Lieven Vandenberghe's EE236C course. For more variants of Nesterov, see the TFOCS paper by Becker, Candés and Grant (2011). For some discussion of the intuition, see Sébastien Bubeck's "Revisiting Nesterov's Acceleration" blog post, which discusses Bubeck's own work, as well as citing some recent papers about differential equation interpretations (Su, Boyd, Candès 2015). This last interpretation has been very popular; about 5 submissions to ICML 2017 discuss it; see also A Variational Perspective on Accelerated Methods in Optimization (2016).

Some of the motivation for Nesterov's method comes from Polyak's 1964 "Heavy Ball Method." For the case of quadratics, this can be analyzed simply and is an exercise in Bertsekas' *Nonlinear programming* book. See also Ben Recht's analysis of the heavy ball method.

Another interesting connection is that recent methods, like Hogwild!, which are parallel computing methods that do not lock variables and update asychronously (hence they are fast, but computations may be inaccurate) can be see as variants of Nesterov's method. See Asynchrony begets momentum, with an application to deep learning 2016. They conclude that if you have asynchrony, you need to reduce the amount of momentum.

Some downsides to accelerated methods:

1. They are faster in the presence of strong convexity, but need to know the strong convexity constant, unlike gradient descent (which exploits it automatically). There are several ways around this (assuming that you don't know the strong convexity constant, which is useful, since sometimes we only have local strong convexity or restricted strong convexity). A popular approach is to restart the weight counter back to 0; see Adaptive Restart for Accelerated Gradient Schemes by O'Donoghue, Candès '12.

2. Though $f(x_k)$ converges, it is not shown that (x_k) itself converges (unless one assumes strong convexity). There is a variant of FISTA where the iterates do provably converge; see On the convergence of the iterates of "FISTA".

3. They may be more sensitive to computational errors in some cases, especially deterministic errors. Some recent work proposes modifications that make the methods more stable; see Stability of over-relaxations for the Forward-Backward algorithm, application to "FISTA" (2015).