Homework #3, EECS 598-006, W20. Due **Thu. Jan. 30**, by 4:00PM

1. [19] Gradient descent for smooth cost functions

EECS 551 students implemented the gradient descent (GD) method for minimizing a least-squares (LS) cost function. In this problem you will implement gradient descent for more smooth cost functions, *i.e.*, cost functions having a gradient that is Lipschitz smooth, and then apply it to both a least-squares problem (as a warm-up test) and then, in the next problem, to a 1D signal denoising problem. Later we will apply it (and other faster algorithms) to 2D image denoising after we learn more about how to handle large-scale problems.

(a) [10] Write a JULIA function `gd` that implements the GD iteration $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \frac{1}{L} \nabla \Psi(\boldsymbol{x}_k)$, given a function g that computes the cost function gradient $g(\boldsymbol{x}) = \nabla \Psi(\boldsymbol{x})$ and the Lipschitz constant L and an initial guess $\boldsymbol{x}_0$.
Your function should take an optional named argument `fun` for evaluating `fun(x,iter)` each iteration, and return a tuple of the final $\boldsymbol{x}$ and an array of type `Array{Any,1}` of length `niter+1` with the values $\mathrm{fun}(\boldsymbol{x}_0), \ldots, \mathrm{fun}(x_{\mathrm{niter}})$. If `fun` is not provided, then the second output argument is an array of `undef` values.
Often a user may want to save multiple values each iteration, such as $\Psi(\boldsymbol{x}_k)$ and $\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|$. So we have the routine return a 1D `Array` of length `niter+1` where each entry in the array is whatever the `fun` returns each iteration, which might differ across iterations. (In MATLAB you would use a “cell” for this, whereas in JULIA, `Array{Any}` is the general array container type. To allow this generality, create the `out` variable as follows:

```
out = Array{Any}(undef, niter+1)
```

Then each iteration you can use something like this to save:

```
out[iter+1] = fun(x, iter)
```

To elaborate on the impetus for this generality, suppose the user wants to compute the cost function every iteration, and save the current iteration $\boldsymbol{x}_k$ every 10th iteration. Then the user might use something like this:

```
fun = (x,iter) -> (mod(iter,10) == 0 ? x : [], cost(x))
```

assuming `cost(x)` is a function that evaluates $\Psi(\boldsymbol{x})$.
Another possibility is that the user might want to display $\boldsymbol{x}$ every 20th iteration, to see how the algorithm is proceeding. This can be done with

```
fun = (x,iter) -> (mod(iter,20) == 0 ? display(x) : [])
```

There are many possibilities of how a user might want to probe or archive what is happening to $\boldsymbol{x}_k$ during an iterative algorithm, so we will be including this type of general function as an optional argument in all the iterative algorithms implemented in this course.
Your file should be named `gd.jl` and should contain the following function:

```
"""
    (x,out) = gd(g::Function, L::Real, x0 ; niter::Int=?, fun::Function=?)

Perform gradient descent to "solve" a minimization problem
having a L-Lipschitz smooth gradient

In
* `g` function that computes gradient `g(x)` of cost function
* `L` Lipschitz constant of cost function gradient
* `x0` initial guess

Option
* `niter` # number of iterations; default 100
* `fun` user-defined function to be evaluated with two arguments `(x,iter)`
It is evaluated at `(x0,0)` and then after each iteration

Output
* `x` final iterate
* `out` `[fun(x0,0), fun(x1,1), ..., fun(x_niter,niter)]`
(all `undef` by default).  An array of length `niter+1`
"""
function gd(g::Function, L::Real,
    x0::Union{Number,AbstractVector{<:Number}} ;
    niter::Int=100,
    fun::Function = (x,iter) -> undef)
```

Submit your solution to mailto:eecs556@autograder.eecs.umich.edu.
Hint. Use the next part as a way to debug *before* you submit to the autograder. You should be able to predict what the plots in the next part look like and use them to self-test your code. Then you should be able to get your code to pass on the first try after using the next part to debug it. In the real world there are no autograders and you must devise your own tests to sanity check your code! The next part is a good example of such a check.

(b) [3] Apply your JULIA function to solve the LS problem $\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} \frac{1}{2} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2$ iteratively for the following test data.

```
using LinearAlgebra: norm, opnorm
using Random: seed!
using Plots
#include("gd.jl") # include your function

seed!(0); M = 100; N = 50; A = randn(M,N); y = randn(M); xh = A \ y
cost = # ?
fun = # ?
grad = # ?
L = # ?

niter = 200
xrun, out = gd( #? )
# thecost = [out[k][2] for k=1:niter+1] # this might be useful
```

Make plots of both the log NRMSD $\log_{10}(\|\boldsymbol{x}_k - \hat{\boldsymbol{x}}\| \,/\, \|\hat{\boldsymbol{x}}\|)$ and the log cost function $\log_{10}(\Psi(\boldsymbol{x}_k) - \Psi(\hat{\boldsymbol{x}}))$ versus iteration k for $k = 0, \ldots, 200$.
Initialize with $\boldsymbol{x}_0 = \boldsymbol{0}$.
You should call your `gd` function exactly once, using an appropriate `fun`, to get all of the quantities needed for making these plots.
Submit a screenshot of your code to gradescope to confirm efficiency.

(c) [3] Submit a screenshot of your NRMSD plot.

(d) [3] Submit a screenshot of your cost plot.

2. [22] **Gradient descent for 1D edge-preserving signal denoising**

(a) [3] Continuing the previous problem, consider the 1D edge-preserving denoising application where we want to recover $\boldsymbol{x}$ from the model $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{\varepsilon}$ using the optimization problem

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x} \in \mathbb{R}^N} \Psi(\boldsymbol{x}), \quad \Psi(\boldsymbol{x}) = \frac{1}{2} \|\boldsymbol{y} - \boldsymbol{x}\|_2^2 + \beta R(\boldsymbol{x}), \quad R(\boldsymbol{x}) = \sum_{n=2}^{N} \psi(x_n - x_{n-1}, \delta), \tag{1}$$

where ψ denotes the **Fair potential**.

Determine a Lipschitz constant for the gradient of Ψ that you can compute easily without calling `opnorm` (or `svd` or `svdvals` etc.) functions, because those do not scale to large problems.

(b) [10] Write a JULIA function that uses your `gd` code for GD to minimize this cost function. Your function must return $\hat{\boldsymbol{x}}$ and the cost function evaluated at each iteration.

Hint. The function `spdiagm` is useful.

Your file should be named `dn1gd.jl` and should contain the following function:

```
"""
    (x,cost) = dn1gd(y ; x0=?, niter=?, reg=?, del=?)

Perform 1D edge-preserving signal denoising using GD,
to "solve" the minimization problem
`argmin_x 1/2 |y - x|^2 + reg * sum_{n=2}^N pot(x_n - x_n-1,del)`
where `pot()` is the Fair potential with parameter `del`
(Uses `gd` function from previous problem.)

In
* `y` 1D noisy signal

Option
* `x0` initial guess; default `y`
* `niter::Int` # number of iterations; default 100
* `reg::Real` regularization parameter; default 1
* `del::Real` potential function parameter; default 2

Out
* `x` final iterate
* `cost` `[niter+1]` cost function each iteration
"""
function dn1gd(y::AbstractVector ;
    x0::AbstractVector = y,
    reg::Real = 1,
    del::Real = 2,
    niter::Int=100)
```

Submit your solution to `mailto:eecs556@autograder.eecs.umich.edu`.

(c) [3] Apply your 1D denoising method `dn1gd` with $\delta = 0.1$ and $\beta = 8$ to the 1D noisy signal generated by the following code, using 200 iterations.

```
using Random: seed!
seed!(0); N = 100; x = cumsum(mod.(1:N, 30) .== 0); y = x + 0.2 * randn(N)
```

Make a plot of $\log_{10}(\Psi(\boldsymbol{x}_k))$ versus iteration k to confirm that your method is working and that we have enough iterations.

(d) [3] Make a single plot that shows $\boldsymbol{x}$, $\boldsymbol{y}$, and $\hat{\boldsymbol{x}}$. In the legend of this plot, show the initial NRMSE of the noisy signal $\|\boldsymbol{y} - \boldsymbol{x}\| \,/\, \|\boldsymbol{x}\|$ and the final NRMSE of the denoised signal $\|\hat{\boldsymbol{x}} - \boldsymbol{x}\| \,/\, \|\boldsymbol{x}\|$.

Hint. The NRMSE should be reduced by roughly 2×, and $\hat{\boldsymbol{x}}$ should look much more like $\boldsymbol{x}$ than the noisy signal $\boldsymbol{y}$ does.

(e) [3] Suppose we used $\psi(z) = |z|$ instead of the Fair potential here. Is your GD algorithm applicable? Why or why not?

(f) [0] What property of the true signal $\boldsymbol{x}$ in this problem makes it well suited to this denoising method?

3. [26] **1D total-variation (TV) signal denoising**

(a) [10] Continue the previous problem, where the 1D denoising cost function has the form (1), but now with $\psi(z) = |z|$. This choice corresponds to 1D **total variation** (TV) regularization. The LASSO method `lasso_cls` from a previous HW is inapplicable because it was for $\|\boldsymbol{x}\|_1$ not $\|\boldsymbol{T}\boldsymbol{x}\|_1$. However, the Ch. 1 notes show a way to rewrite (1) so that one can indeed apply a slightly modified version of `lasso_cls` that uses a special nonnegative diagonal weighting matrix $\boldsymbol{W}$:

$$\|\boldsymbol{W}\boldsymbol{z}\|_1 = \|\mathsf{diag}\{\boldsymbol{w}\}\,\boldsymbol{z}\|_1 = \sum_{n=1}^{N} w_n\,|z_n|\,.$$

Make a modified version of `lasso_cls` that expects a *vector* input `w` instead of a scalar regularization parameter β. Your file should be named `lasso_w_cls.jl` and should contain the following function:

```
"""
  xh,out = lasso_w_cls(A, y, w ; niter, x0, step, fun)

Iterative algorithm for the LASSO problem with a (nonnegatively) weighted l1 norm:
`argmin_x 1/2 |A x - y|^2 + |diag(w) x|_1`

Uses the constrained least-squares approach and gradient projection method
where we write `x = u - v = max(x,0) - max(-x,0)` and `|diag(w)x|_1 = w'u + w'v`
Only suitable when `A` and `y` are real so that `x` is also real!

In
* `A` `M x N` real matrix
* `y::AbstractVector{<:Real}` vector of length `M`
* `w::AbstractVector{<:Real}` regularization parameter vector (diagonal weights)

Option
* `x0::AbstractVector{<:Real}` initial guess (default A'y)
* `niter::Int` number of iterations (default 10)
* `step::Real` step size (default 0 means use `1/Lipschitz`)
* `fun::Function` user-defined function evaluated at `x0,0` and at each iteration
default: `(x,iter) -> undef`

Out
* `xh` "solution" of minimization problem after `niter` iterations
* `out` `[fun(x0,0), fun(x1,1), ..., fun(x_niter,niter)]`
"""
function lasso_w_cls(A, y::AbstractVector{<:Real}, w::AbstractVector{<:Real} ;
    niter::Int=10,
    x0::AbstractVector{<:Real} = A'y,
    step::Real=0,
    fun::Function = (x,iter) -> undef)
```

Submit your solution to mailto:eecs556@autograder.eecs.umich.edu.
Hint. The general cost function this function is minimizing has a unique minimizer with a simple analytical solution in the special case when $\boldsymbol{A}$ is unitary. You can use that special case to help debug your code.

(b) [10] Now use your `lasso_w_cls` as the core of an algorithm for solving the 1D denoising problem (1) with TV regularization, following the ideas in the Ch. 1 notes.
Your file should be named `dn1tv.jl` and should contain the following function:

```
"""
    (x,cost) = dn1tv(y ; x0=?, niter=?, reg=?)

Perform 1D TV signal denoising using trick with LASSO
to "solve" the minimization problem
`argmin_x 1/2 |y - x|^2 + reg * sum_{n=2}^N reg * |x_n - x_n-1|`
```

```
(Uses `lasso_w_cls` function from previous problem.)

In
* `y` 1D noisy signal

Option
* `x0` initial guess; default `y`
* `niter` # number of iterations; default 100
* `reg` regularization parameter; default 1

Out
* `x` final estimate
* `cost` `[niter+1]` cost function each iteration
"""
function dn1tv(y::AbstractVector ;
    x0::AbstractVector = y,
    reg::Real = 1,
    niter::Int = 100)
```

Submit your solution to `mailto:eecs556@autograder.eecs.umich.edu`.

(c) [3] Apply your `dn1tv` to the same 1D data as in the previous problem. Use $\beta = 2$.
Show a plot of the cost function $\log_{10}(\Psi(\boldsymbol{x}_k))$ versus iteration for $k = 0, \ldots, 9000$.

(d) [3] Make a single plot that shows $\boldsymbol{x}$, $\boldsymbol{y}$, and $\hat{\boldsymbol{x}}$. In the legend of this plot, show the appropriate NRMSE values.
Hint. If it is working, the NRMSE should be a bit lower than with the Fair potential, but the convergence is quite slow.

Non-graded problem(s)

4. [0] **Analysis vs synthesis regularization**

As discussed in the course notes, a typical analysis regularizer has the form

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} \frac{1}{2} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \beta \|\boldsymbol{T}\boldsymbol{x}\|_1 ,$$

where $\boldsymbol{A} \in \mathbb{F}^{M \times N}$, whereas a typical synthesis regularizer has the form

$$\hat{\boldsymbol{x}} = \boldsymbol{D}\hat{\boldsymbol{z}}, \quad \hat{\boldsymbol{z}} = \arg\min_{\boldsymbol{z}} \frac{1}{2} \|\boldsymbol{A}\boldsymbol{D}\boldsymbol{z} - \boldsymbol{y}\|_2^2 + \beta \|\boldsymbol{z}\|_1 .$$

When $\boldsymbol{D}\boldsymbol{T} = \boldsymbol{I}$ and $\boldsymbol{T}\boldsymbol{D} = \boldsymbol{I}$, *i.e.*, when $\boldsymbol{D}$ and $\boldsymbol{T}$ are both invertible and $\boldsymbol{D} = \boldsymbol{T}^{-1}$, then it is easy to verify that these two approaches yield the same solution (or set of solutions if the minimizer is not unique). This problem explores whether we can relax the assumption that *both* $\boldsymbol{D}\boldsymbol{T} = \boldsymbol{I}$ and $\boldsymbol{T}\boldsymbol{D} = \boldsymbol{I}$ and still get equivalent solution(s).

Specifically, we focus on the case where $\boldsymbol{D} = \boldsymbol{B}$ and $\boldsymbol{T} = \boldsymbol{B}'$ where $\boldsymbol{B}$ is a $N \times K$ matrix with $N \leq K$ (wide) called a **Parseval tight frame**. (See F18 EECS 551 notes for details.)

Consider the two following generalizations of the above forms. The (constrained) synthesis form is

$$\hat{\boldsymbol{x}}_1 = \boldsymbol{B}\hat{\boldsymbol{z}}_1, \quad \hat{\boldsymbol{z}}_1 = \arg\min_{\boldsymbol{z} \in \mathcal{Z}} h(\boldsymbol{z}), \quad h(\boldsymbol{z}) \triangleq f(\boldsymbol{B}\boldsymbol{z}) + g(\boldsymbol{z})$$

where $f(\boldsymbol{x})$ is a data-fit term, $g(\boldsymbol{z})$ is some type of sparsity regularizer, and the set $\mathcal{Z}$ is

$$\mathcal{Z} \triangleq \left\{ \boldsymbol{z} \in \mathbb{F}^K \ : \ \boldsymbol{B}'\boldsymbol{B}\boldsymbol{z} = \boldsymbol{z} \right\} .$$

Using the same f and g as above, the analysis form is:

$$\hat{\boldsymbol{x}}_2 = \arg\min_{\boldsymbol{x} \in \mathbb{F}^N} \Psi(\boldsymbol{x}), \quad \Psi(\boldsymbol{x}) \triangleq f(\boldsymbol{x}) + g(\boldsymbol{B}'\boldsymbol{x}).$$

(a) As a warm-up, express $\hat{\boldsymbol{x}}_1$ and $\hat{\boldsymbol{x}}_2$ in terms of $\boldsymbol{A}$, $\boldsymbol{B}$, β and $\boldsymbol{y}$, in the quadratic case where $f(\boldsymbol{x}) = \frac{1}{2} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2$ and $g(\boldsymbol{z}) = \beta \frac{1}{2} \|\boldsymbol{z}\|_2^2$. For this subproblem, ignore the constraint by letting $\mathcal{Z} = \mathbb{F}^K$. Hint. For a Parseval tight frame, $\boldsymbol{B}\boldsymbol{B}' = \boldsymbol{I}_N$, whereas $\boldsymbol{B}'\boldsymbol{B} \neq \boldsymbol{I}_K$. This is the sense in which we are generalizing the case where $\boldsymbol{D}\boldsymbol{T} = \boldsymbol{T}\boldsymbol{D} = \boldsymbol{I}_N$.

(b) Apply the **push-through identity** (see EECS 551 notes) twice, or use some other approach, to show that in the quadratic case of (a) we have $\hat{\boldsymbol{x}}_1 = \hat{\boldsymbol{x}}_2$, despite the fact that we used the unconstrained formulation where $\mathcal{Z} = \mathbb{F}^K$ in (a).

(c) In (a) does $\boldsymbol{B}'\boldsymbol{B}\hat{\boldsymbol{z}}_1 = \hat{\boldsymbol{z}}_1$? Prove or disprove with a counter-example.

(d) Now consider the general case where f and g are arbitrary convex functions.
In this case, $h(\boldsymbol{z})$ and $\Psi(\boldsymbol{x})$ might have multiple global minimizers, so we will not try to show that $\hat{\boldsymbol{x}}_1 = \hat{\boldsymbol{x}}_2$.
Instead, show that $\hat{\boldsymbol{x}}_1$ and $\hat{\boldsymbol{x}}_2$ are *equivalent* solutions in the sense that $\Psi(\hat{\boldsymbol{x}}_1) = \Psi(\hat{\boldsymbol{x}}_2)$ and $h(\hat{\boldsymbol{z}}_1) = h(\boldsymbol{B}'\hat{\boldsymbol{x}}_2)$.
Hint. First find a simple relationship between $h(\cdot)$ and $\Psi(\cdot)$.
Hint. Writing $\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x} \in \mathcal{X}} \Psi(\boldsymbol{x})$ does not necessarily imply that $\hat{\boldsymbol{x}}$ is unique, but it does mean that $\Psi(\hat{\boldsymbol{x}}) \leq \Psi(\boldsymbol{x}), \ \forall \boldsymbol{x} \in \mathcal{X}$.

(e) Does your conclusion in part (d) imply equivalence of $\hat{\boldsymbol{x}}_1$ and $\hat{\boldsymbol{x}}_2$ for the following *constrained* formulations for $p \geq 1$?
Prove or disprove with a small counter-example. (Assume $\boldsymbol{B}$ is a Parseval tight frame.)

$$\hat{\boldsymbol{x}}_1 = \boldsymbol{B}\hat{\boldsymbol{z}}_1, \quad \hat{\boldsymbol{z}}_1 = \arg\min_{\boldsymbol{z}} \|\boldsymbol{z}\|_p \text{ sub. to } \|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{B}\boldsymbol{z}\|_2 < \varepsilon \text{ and } \boldsymbol{B}'\boldsymbol{B}\boldsymbol{z} = \boldsymbol{z}$$
$$\hat{\boldsymbol{x}}_2 = \arg\min_{\boldsymbol{x}} \|\boldsymbol{B}'\boldsymbol{x}\|_p \text{ sub. to } \|\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}\|_2 < \varepsilon.$$

This equivalence was claimed (without proof) by “Prof. VC” at the IEEE SSP workshop (held in Ann Arbor) in 2012.

(f) [0] Optional challenge. Revisit (d) in the case where $\boldsymbol{B}$ is not a tight frame, and/or when $\mathcal{Z} = \mathbb{F}^K$.