Advanced Machine Learning GR5242
Fall 2024

Homework 2

Due: Friday October 11, 7 pm, NY time.

Note: All questions carry equal weight.

Collaboration: You may collaborate on this assignment with peers from any course section. However, you must write up your own solutions individually for submission.

Homework submission: Assignments should be submitted through Gradescope, which is accessible through Courseworks. Please submit your homework by publishing a notebook that cleanly displays your code, results and plots to pdf or html. For the non-coding questions, you can typeset using LaTeX or markdown, or you can include neatly scanned answers. **Please make sure to put everything together into a single pdf file for submission.** Late submission is not accepted.

Problem 1: Loss Function Example*

Note: this is an open-ended question. You get full point for attempting it.

Give an example of a machine learning model class along with a loss function (defining a risk to minimize) other than the following two examples: linear regression (with square loss) and neural network (with logistic loss).

Problem 2: Differentiation

In what follows we will use the convention that the gradient is a column vector, which is the transpose of the Jacobian in dimension 1. In fact that any vector $x \in \mathbb{R}^d$ is a column vector.

(a) Let's consider the function $g(x) = Ax$, where $x \in \mathbb{R}^d$ and $A \in \mathbb{R}^{k \times d}$. In the special case $k = 1$, show that the gradient of g is equal to $A^\top$.

(b) Now, consider the case $k > 1$, where we might write $g(x) = Ax \doteq [g_1(x), g_2(x), ..., g_k(x)]^\top$. Recall that the *Jacobian* is a generalization of the gradient to multivariate functions g. That is, the Jacobian of g is the matrix of partial derivatives whose $(i,j)^{th}$ entry is $\frac{\partial g_i(x)}{\partial x_j}$. How does the Jacobian matrix relate to the gradients of the components g_i of g? Argue from there that the Jacobian matrix of g above is given as $J_g(x) = A$.

(c) Now consider the function $g(x) = x^\top Ax$, where $x \in \mathbb{R}^d$ and $A \in \mathbb{R}^{d \times d}$. Show that the gradient of g is given as $\nabla g(x) = Ax + A^\top x$ (it then follows that when A is symmetric, $\nabla g(x) = 2Ax$).
Hint : You can write $x^\top Ax = \sum_{i,j} A_{i,j} \cdot x_i x_j$.

Problem 3: Compare Gradient Descent method and Newton Methods

Consider the following function over $x \in \mathbb{R}^2$ (depicted in Figure 1):

$$F(x) = x^\top \Sigma x + \log(1 + \exp(-\mathbf{1}^\top x)), \text{ where } \mathbf{1} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \text{ and } \Sigma = \begin{bmatrix} 5 & 0 \\ 0 & \frac{1}{2} \end{bmatrix}.$$

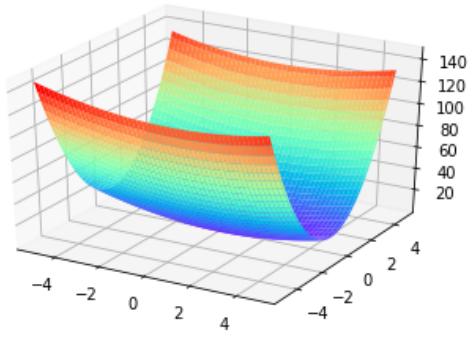

Figure 1: Problem 3, a plot of the given function F.

The problem asks you to implement Gradient Descent and Newton methods in Python to minimize the above function F, and in each case, to plot $F(x_t)$ as a function of iterations $t \in [1, 2, \ldots, 30]$ (all on the same plot for comparison).

Implementation details: *For either methods, and any step size, start in iteration $t = 1$ at the same point $x_1 = (0, 0)$, for fairer comparison.*

Here we can calculate $2/\beta = 0.195$, i.e., in terms of the *smoothness* constant β of F (via its Hessian). According to theory, we should set the step size η for Gradient Descent to $\eta < 2/\beta$. In the case of Newton, which uses a better approximation, we can use a larger step size.

Therefore, on the same figure, plot $F(x_t)$ against $t \in [1, 2, \ldots, 30]$ for the following 4 configurations, and label each curve appropriately as given below (NM1 through GD0.2):

(NM1) Newton Method with constant step size $\eta = 1$

(GD0.1) Gradient Descent Method with constant step size $\eta = 0.1$

(GD0.19) Gradient Descent Method with constant step size $\eta = 0.19$

(GD0.2) Gradient Descent Method with constant step size $\eta = 0.2$

The code for the function $F(x)$, its Gradient and Hessian Matrix are given below.

```
Sigma = np.array([[5,0],[0,0.5]])
II = np.array([[1,1]])
# x is a 2 by 1 array starting from np.array([[0.0],[0.0]])
def func0(x):
    return np.dot(np.dot(x.transpose(), Sigma), x)+math.log(1+math.exp(-np.
                                                    dot(II,x)))
def First_derivative(x):
    x1 = x[0][0]
    x2 = x[1][0]
    ex = math.exp(-(x1+x2))
    return np.array([[10*x1-ex/(1+ex)],[x2-ex/(1+ex)]])
def Second_derivative(x):
    x1 = x[0][0]
    x2 = x[1][0]
    ex = math.exp(-(x1+x2))
```

```
    ex = ex/(1+ex)**2
    return np.array([[10+ex,ex],[ex,1+ex]])
```

Problem 4: Taylor's Remainder Theorem

Consider a convex function $F : \mathbb{R}^d \to \mathbb{R}$. Suppose its Hessian is positive definite, i.e. $\nabla^2 F \succeq 0$, and is continuous for every x. Assume that F achieves its minimum at x^*. Show that

$$F(x) - F(x^*) \leq (x - x^*)^\top \nabla F(x)$$

Hint : Use Taylor's Remainder Theorem (which holds for instance when $\nabla^2 F$ is continuous).

Problem 5: Regular SGD vs Adagrad optimization on a Regression problem

Consider a random vector $X \sim \mathcal{N}(0, \Sigma)$ with covariance matrix $\Sigma = \text{diag } (\sigma_{*1}^2, \sigma_{*2}^2, \ldots, \sigma_{*d}^2)$. Here we let $d = 10$ and $\sigma_{*i} = 2^{-i} \cdot 10$. Let the response Y be generated as

$$Y = w_*^\top X + \mathcal{N}(0, \sigma^2),$$

for an optimal vector $w_* = [1, 1, 1, ..., 1]^\top$ (supposed to be unknown), and $\sigma = 1$. Consider the following Ridge objective, defined over i.i.d. data $\{(X_i, Y_i)\}_{i=1}^n$ from the above distribution:

$$F(w) = \frac{1}{n} \sum_{i=1}^{n} (Y_i - w^\top X_i)^2 + \lambda \|w\|^2,$$

for a setting of $\lambda = 0.1$. We want to minimize F over choices of w to estimate w_*.

You are asked to implement the following two optimization approaches, namely SGD (Stochastic Gradient Descent) and Adagrad, given in the two pseudocodes below. The only differences between the two approaches are in the setting of the step sizes η_t (in the case of Adagrad, η_t is a diagonal matrix of step sizes).

For both procedures, we need the following definition of a *stochastic gradient*. First write

$$F(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w), \text{ that is, we let } f_i(w) = (Y_i - w^\top X_i)^2 + \lambda \|w\|^2.$$

Thus, the term $f_i(w)$ is just in terms of the random data sample (X_i, Y_i).

We define the *stochastic gradient* (evaluated at w) at time t by $\nabla f_t(w)$ (that is at index $i = t$). At step t in both of the procedures below, when the current w is w_t, we use the common notation

$$\tilde{\nabla} F(w_t) \doteq \nabla f_t(w_t),$$

Algorithm 1: Regular SGD

The steps below use a parameter β to be specified;
At time t=1: set $w_1 = 0$;
while $t \leq n$ **do**
  Compute $\tilde{\nabla} F(w_t)$ on t^{th} datapoint (X_t, Y_t);
  Set $\eta_t = \sqrt{\frac{1}{\beta \cdot \sigma_t^2}}$, where $\sigma_t^2 = \sum_{s=1}^{t} \|\tilde{\nabla} F(w_s)\|^2$ (sum over past $\tilde{\nabla}$'s);
  Update $w_{t+1} = w_t - \eta_t \tilde{\nabla} F(w_t)$;
  t = t+1;
end

Algorithm 2: Adagrad

The steps below use a parameter β to be specified;
At time $t = 1$: set $w_1 = 0$;
while $t \leq n$ **do**
  Compute $\tilde{\nabla} F(w_t)$ on t^{th} datapoint (X_t, Y_t);
  $\forall i \in [d]$, set $\eta_{t,i} = \sqrt{\frac{1}{\beta \cdot \sigma_{t,i}^2}}$, where $\sigma_{t,i}^2 = \sum_{s=1}^{t} (\tilde{\nabla}_i F(w_s))^2$ (sum over i^{th} coordinate of past $\tilde{\nabla}$'s);
  Update $w_{t+1} = w_t - \eta_t \tilde{\nabla} F(w_t)$, where now $\eta_t = \text{diag}\ (\eta_{t,1}, \ldots, \eta_{t,d})$;
  t = t+1;
end

(a) What is $\tilde{\nabla} F(w_t)$ in terms of (X_t, Y_t) and w_t?

(b) Implement the above two procedures in Python, using $\beta = 2(\sigma_{*1}^2 + \lambda)$ (smoothness measure).

(c) **Experiment:** Run each of the two procedures on $n = 1000$ datapoints, generated from the above distribution, using the sampler provided below. Repeat 10 times, every time starting at $w_1 = \mathbf{0}$ as specified. Plot $\|w_t - w_*\|$ against $t = 1, 2, \ldots, 1000$, averaged over the 10 repetitions. Show error bars (i.e., std over repetitions at each t).

Report the plots in (c) for the two procedures on the same figure for comparison.

Data sampler code:

```
## Code for generator/sampler
import numpy as np
import random
import time
from numpy import linalg as LA
import statistics
#initialization
sigma = 1
d = 10
c_square = 100
cov = np.diag([(0.25**i)*c_square for i in range(1,d+1)])
mean = [0]*d
#coeficient given
w = np.array([1]*d)

# Sampler function
def sampler(n):
    #data X generator
    np.random.seed(int(time.time()*100000)%100000)
    X = np.random.multivariate_normal(mean, cov, n)
    #data Y generator
    Y = np.matmul(X, w)+np.random.normal(0, sigma**2, n)
    return (X,Y)
```