Labs
Machine Learning Summer School
July 2017

EPFL
School of Computer and Communication Sciences
Machine Learning and Optimization Laboratory
Martin Jaggi
mlo.epfl.ch

Problem Set (Optimization Lab)

Setup, data and sample code. Obtain the contents of the github repository

github.com/epfml/opt-shortcourse

In the `template` folder of each part, we have provided datasets and sample code useful for this exercise. Start from this code, and fill in the blank parts. The code can be run from the notebooks `run.ipynb` for each exercise.

1 Setting Up a Cost Function

In this exercise, we will focus on simple linear regression which takes the following form,

$$y_n \approx f(x_{n1}) = w_0 + w_1 x_{n1}. \tag{1}$$

We will use height as the input variable x_{n1} and weight as the output variable y_n. The coefficients w_0 and w_1 are also called *model parameters*. We will use a mean-square-error (MSE) function defined as follows,

$$\mathcal{L}(w_0, w_1) = \frac{1}{2N} \sum_{n=1}^{N} \big(y_n - f(x_{n1})\big)^2 = \frac{1}{2N} \sum_{n=1}^{N} \big(y_n - w_0 - w_1 x_{n1}\big)^2. \tag{2}$$

Our goal is to find $w_0^\star$ and $w_1^\star$ that minimize this *cost*.

Let us start by the array data type in *NumPy*. We store all the (y_n, x_{n1}) pairs in a vector and a matrix as shown below.

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \qquad \widetilde{\boldsymbol{X}} = \begin{bmatrix} 1 & x_{11} \\ 1 & x_{21} \\ \vdots & \vdots \\ 1 & x_{N1} \end{bmatrix} \tag{3}$$

a) Complete the implementation of the notebook function `compute_loss(y, tx, w)`. You can start by setting $\boldsymbol{w} = [1, 2]^\top$, and test your function.

2 Gradient Descent

Consider least squares linear regression as defined above. We derive the following expressions for the gradient (the vector of partial derivatives)

$$\frac{\partial \mathcal{L}(w_0, w_1)}{\partial w_0} = \frac{1}{N} \sum_{n=1}^{N} \big(y_n - w_0 - w_1 x_{n1}\big) = -\frac{1}{N} \sum_{n=1}^{N} e_n \tag{4}$$

$$\frac{\partial \mathcal{L}(w_0, w_1)}{\partial w_1} = \frac{1}{N} \sum_{i=1}^{N} \big(y_n - w_0 - w_1 x_{n1}\big) x_{n1} = -\frac{1}{N} \sum_{n=1}^{N} e_n x_{n1} \tag{5}$$

Denoting the gradient by $\nabla \mathcal{L}(\boldsymbol{w})$, we can write these operations in vector form as follows,

$$\nabla \mathcal{L}(\boldsymbol{w}) := \begin{bmatrix} \frac{\partial \mathcal{L}(w_0, w_1)}{\partial w_0} & \frac{\partial \mathcal{L}(w_0, w_1)}{\partial w_1} \end{bmatrix} = -\frac{1}{N} \begin{bmatrix} \sum_{n=1}^{N} e_n \\ \sum_{n=1}^{N} e_n x_{n1} \end{bmatrix} = -\frac{1}{N} \tilde{\boldsymbol{X}}^\top \boldsymbol{e} \tag{6}$$

a) Implement a function that computes the gradients. Implement the notebook function `compute_gradient(y, tx, w)` using Equation (6). Verify that the function returns the right values. First, manually compute the gradients for hand-picked values of $\boldsymbol{y}$, $\tilde{\boldsymbol{X}}$, and $\boldsymbol{w}$ and compare them to the output of `compute_gradient`.

b) Fill in the notebook function `gradient_descent(y, tx, initial_w, ...)`. Run the code and visualize the iterations. Also, look at the printed messages that show $\mathcal{L}$ and values of $w_0^{(t)}$ and $w_1^{(t)}$. Take a detailed look at these plots,

- Is the cost being minimized?
- Is the algorithm converging? What can be said about the convergence speed?
- How good are the final values of w_1 and w_0 found?

c) Now let's experiment with the values of the step size and initialization parameters and see how they influences the convergence. In theory, gradient descent converges to the optimum on convex functions, when the value of the step size is chosen appropriately.

- Try the following values of step size: 0.001, 0.01, 0.5, 1, 2, 2.5. What do you observe? Did the procedure converge?
- Try different initializations with fixed step size $\gamma = 0.1$, for instance:
  - $w_0 = 0$, $w_1 = 0$
  - $w_0 = 100$, $w_1 = 10$
  - $w_0 = -1000$, $w_1 = 1000$

  What do you observe? Did the procedure converge?

3 Stochastic Gradient Descent

Consider least squares linear regression as defined above. Let us implement stochastic gradient descent. The update rule for stochastic gradient descent on an objective function $\mathcal{L}(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}_n(\boldsymbol{w})$ at step t is

$$\boldsymbol{w}^{(t+1)} := \boldsymbol{w}^{(t)} - \gamma \nabla \mathcal{L}_n(\boldsymbol{w}^{(t)}) \ . \tag{7}$$

HINT: You can use the function `batch_iter()` in the file of `helpers.py` to generate mini-batch data for stochastic gradient descent.

4 (Bonus) Lasso with SGD and Coordinate Descent

The optimization problem for sparse least squares linear regression (also known as the Lasso) is given by

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^N} \ \|A\boldsymbol{\alpha} - \boldsymbol{b}\|^2 + \lambda \|\boldsymbol{\alpha}\|_1 \tag{8}$$

Where $A \in \mathbb{R}^{D \times N}$ is the given data matrix, and $\boldsymbol{b} \in \mathbb{R}^D$ is the given "measurement" vector. The $\|.\|_1$-norm is the sum of absolute values of coordinates of its argument.

The goal of this exercise is to implement stochastic (sub)gradient descent (SGD) as well as coordinate descent on this objective function, where both algorithms will work based on the $\boldsymbol{\alpha}$ variable vector.

a) Mathematically derive the **SGD** update, when given $\boldsymbol{\alpha}^{(t)}$. To do so, write the objective function (Euclidean norm) as a sum of D terms. If the iterate $\boldsymbol{\alpha}^{(t)}$ happens to be at a non-differentiable point of the $\|.\|_1$-norm, pick an element of the subgradient instead.

b) Mathematically derive the **coordinate update** for one coordinate n (finding the closed-form solution to maximization over just that coordinate), when given $\boldsymbol{\alpha}^{(t)}$.

c) Implement the SGD as well as coordinate descent for this objective function, and compare the performance of the two. Which one is faster? (Plot the objective function vs time or number of iterations). For performance of the implementation, it might be helpful to maintain the vector $r := A\boldsymbol{\alpha} - \boldsymbol{b}$ (called the residual).

5 (Bonus) Support Vector Machines with SGD and Coordinate Descent

The original optimization problem for the Support Vector Machine (SVM) is given by

$$\min_{\boldsymbol{w}\in\mathbb{R}^D} \sum_{n=1}^{N} \ell(y_n \boldsymbol{x}_n^\top \boldsymbol{w}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \tag{9}$$

where $\ell : \mathbb{R} \to \mathbb{R}$, $\ell(z) := \max\{0, 1-z\}$ is the *hinge loss* function. Here for any n, $1 \le n \le N$, the vector $\boldsymbol{x}_n \in \mathbb{R}^D$ is the n^{th} data example, and $y_n \in \{\pm 1\}$ is the corresponding label.

The dual optimization problem for the SVM is given by

$$\max_{\boldsymbol{\alpha}\in\mathbb{R}^N} \boldsymbol{\alpha}^\top \mathbf{1} - \tfrac{1}{2\lambda}\boldsymbol{\alpha}^\top \boldsymbol{Y}\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{Y}\boldsymbol{\alpha} \quad \text{ such that } \quad 0 \le \alpha_n \le 1 \ \forall n \tag{10}$$

where $\boldsymbol{Y} := \mathrm{diag}(\boldsymbol{y})$, and $\boldsymbol{X} \in \mathbb{R}^{D\times N}$ again collects all N data examples as its columns.

a) **SGD for SVM.** Implement stochastic gradient descent (SGD) for the original SVM formulation (9), with respect to the $\boldsymbol{w}$ variable. That is, in every iteration, pick one data example $n \in [N]$ uniformly at random, and perform an update on $\boldsymbol{w}$ based on the (sub)gradient of the n^{th} summand of the objective (9). Then iterate by picking the next n.

b) **Coordinate Descent for SVM.** Derive the coordinate descent algorithm updates for the dual (10) of the SVM formulation, with respect to the $\boldsymbol{\alpha}$ variable. That is in every iteration, pick a coordinate $n \in [N]$ uniformly at random, and optimize the objective (10) with respect to that coordinate alone. Make sure you respect the upper and lower constraints 0 and 1, as the optimum might lie outside this interval. After updating the coordinate α_n, update the corresponding primal vector $\boldsymbol{w}$ such that the first-order correspondence is maintained, that is that always $\boldsymbol{w} = \boldsymbol{w}(\boldsymbol{\alpha}) := \frac{1}{\lambda}\boldsymbol{X}\boldsymbol{Y}\boldsymbol{\alpha}$. Then iterate by picking the next coordinate n.

   1. Mathematically derive the coordinate update for one coordinate n (finding the closed-form solution to maximization over just that coordinate), when given $\boldsymbol{\alpha}$ and corresponding $\boldsymbol{w}$.
   2. Implement the coordinate descent (here ascent) algorithm in Python, and compare to your SGD implementation. Which one is faster? (Compare the training objective values (9) for the $\boldsymbol{w}$ iterates you obtain from each method).