Mathematical Foundations of Deep Neural Networks, M1407.001200
E. Ryu
Fall 2021

Homework 1 Solutions

Problem 1: *Least-squares derivatives.* Let $X_1, \dots, X_N \in \mathbb{R}^p$ and $Y_1, \dots, Y_N \in \mathbb{R}$. Define

$$X = \begin{bmatrix} X_1^\intercal \\ \vdots \\ X_N^\intercal \end{bmatrix} \in \mathbb{R}^{N \times p}, \qquad Y = \begin{bmatrix} Y_1 \\ \vdots \\ Y_N \end{bmatrix} \in \mathbb{R}^N.$$

Let

$$\ell_i(\theta) = \frac{1}{2}(X_i^\intercal \theta - Y_i)^2 \quad \text{for } i = 1, \dots, N, \qquad \mathcal{L}(\theta) = \frac{1}{2}\|X\theta - Y\|^2.$$

Show (a) $\nabla_\theta \ell_i(\theta) = (X_i^\intercal \theta - Y_i) X_i$ and (b) $\nabla_\theta \mathcal{L}(\theta) = X^\intercal(X\theta - Y)$.
Hint. For part (a), start by computing $\frac{\partial}{\partial \theta_j}\ell_i(\theta)$. For part (b), use the fact that

$$Mv = \sum_{i=1}^N M_{:,i} v_i \in \mathbb{R}^p$$

for any $M \in \mathbb{R}^{p \times N}$, $v \in \mathbb{R}^N$, where $M_{:,i}$ is the ith column of M for $i = 1, \dots, N$.

Solution. (a) We write $X_i = [x_{i1}, x_{i2}, \cdots, x_{ip}]^\intercal$. By the chain rule,

$$\frac{\partial}{\partial \theta_j}\ell_i(\theta) = (X_i^\intercal \theta - Y_i)\frac{\partial (X_i^\intercal \theta - Y_i)}{\partial \theta_j} = (X_i^\intercal \theta - Y_i) x_{ij}.$$

Thus, we can conclude

$$\nabla_\theta \ell_i(\theta) = \begin{bmatrix} \frac{\partial}{\partial \theta_1}\ell_i(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_p}\ell_i(\theta) \end{bmatrix} = \begin{bmatrix} (X_i^\intercal \theta - Y_i) x_{i1} \\ \vdots \\ (X_i^\intercal \theta - Y_i) x_{ip} \end{bmatrix} = (X_i^\intercal \theta - Y_i) \begin{bmatrix} x_{i1} \\ \vdots \\ x_{ip} \end{bmatrix} = (X_i^\intercal \theta - Y_i) X_i.$$

(b) Using $\mathcal{L}(\theta) = \sum_{i=1}^N \ell_i(\theta)$ and the result of (a),

$$\nabla_\theta \mathcal{L}(\theta) = \nabla_\theta \sum_{i=1}^N \ell_i(\theta) = \sum_{i=1}^N \nabla_\theta \ell_i(\theta) = \sum_{i=1}^N X_i(X_i^\intercal \theta - Y_i).$$

Simplify the summation as a matrix multiplication.

$$\nabla_\theta \mathcal{L}(\theta) = \sum_{i=1}^N X_i(X_i^\intercal \theta - Y_i) = \begin{bmatrix} X_1 & \cdots & X_N \end{bmatrix} \begin{bmatrix} (X_1^\intercal \theta - Y_1) \\ \vdots \\ (X_N^\intercal \theta - Y_N) \end{bmatrix} = X^\intercal(X\theta - Y).$$

■

Problem 2: *Diverging univariate GD.* Consider the univariate function $f(\theta) = \theta^2/2$. Show that

$$\theta^{k+1} = \theta^k - \alpha f'(\theta^k)$$

with $\theta^0 \neq 0$ diverges if $\alpha > 2$.

Clarification. There is a slight conflict of notation: θ^2 denotes the square of the scalar θ while θ^k denotes the kth iterate of GD.

Solution. Since $f'(\theta) = \theta$, GD is equivalent to

$$\theta^{k+1} = \theta^k - \alpha\theta^k = (1-\alpha)\theta^k.$$

By induction, we can derive $\theta^k = (1-\alpha)^k\theta^0$ and if $\alpha > 2$, then $(1-\alpha)^k$ diverges if $\theta^0 \neq 0$. ■

Problem 3: *Diverging multivariate GD.* Let $X \in \mathbb{R}^{N\times p}$ and $Y \in \mathbb{R}^N$, and consider the optimization problem

$$\underset{\theta\in\mathbb{R}^p}{\text{minimize}} \quad f(\theta)$$

with

$$f(\theta) = \frac{1}{2}\|X\theta - Y\|^2.$$

Show

$$\theta^{k+1} = \theta^k - \alpha\nabla f(\theta^k)$$

with $\alpha > 2/\rho(X^\intercal X)$ diverges for most starting points $\theta^0 \in \mathbb{R}^m$. Here, ρ denotes the spectral radius, i.e., $\rho(X^\intercal X)$ is the largest eigenvalue of the symmetric matrix $X^\intercal X$. For simplicity, you may assume $X^\intercal X$ is invertible.

Hint. Let $\theta^\star = (X^\intercal X)^{-1}X^\intercal Y$ and show that

$$\theta^{k+1} - \theta^\star = \text{Some function of } (\theta^k - \theta^\star).$$

Remark. "Most starting points" can be formalized as "almost everywhere with respect to the Lebesgue measure". If you are unfamiliar with measure theory, you can understand the statement as holding for all starting points except for a lower dimensional set.

Solution. Since $\nabla f(\theta^k) = X^\intercal(X\theta - Y)$, GD is equivalent to

$$\theta^{k+1} = \theta^k - \alpha X^\intercal(X\theta^k - Y) = (I - \alpha X^\intercal X)\theta^k - \alpha X^\intercal Y.$$

Reorganizing, we get

$$\theta^{k+1} - \theta^\star = (I - \alpha X^\intercal X)\left(\theta^k - \theta^\star\right)$$

and

$$\theta^k - \theta^\star = (I - \alpha X^\intercal X)^k\left(\theta^0 - \theta^\star\right).$$

Using spectral theorem, we have $X^\intercal X = P^\intercal\Lambda P$ that $P^\intercal P = I$ and Λ is diagonal matrix. So, $(I - \alpha X^\intercal X)^k = P^\intercal(I - \alpha\Lambda^k)P$. If the top eigenvector component of $(\theta^0 - \theta^\star)$ is nonzero and $\alpha > 2/\rho(X^\intercal X)$, since $(1 - \alpha\rho(X^\intercal X))^k$ diverge, the iteration diverges. Note that the top eigenvector equal to an $(n-1)$-dimensional set. ■

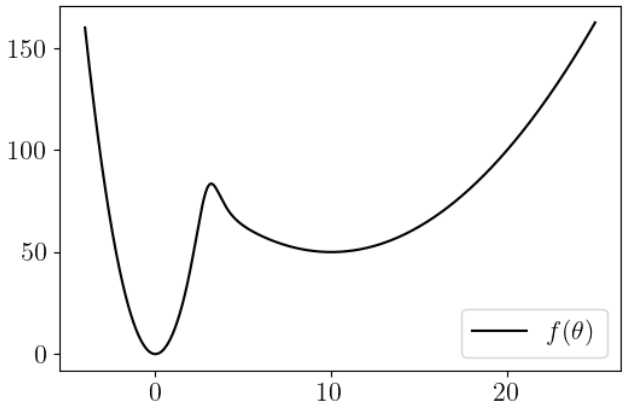
Problem 4: *GD converging to wide local minima.* Consider the optimization problem

$$\underset{\theta \in \mathbb{R}}{\text{minimize}} \quad f(\theta)$$

with

$$f(\theta) = \frac{10\theta^2 + e^{3(\theta-3)}((\theta-10)^2/2 + 50)}{1 + e^{3(\theta-3)}}.$$

Code for evaluating f and f' is implemented in the starter code `wideMinima.py`. We call the global minimum near $\theta = 0$ the *sharp* minimum and the local minimum near $\theta = 10$ the *wide* minimum.



Implement gradient descent and run it with random starting points within the range $[-5, 20]$. Experimentally demonstrate that gradient descent with learning rate $\alpha = 0.01$ converges to either of the two minima, with $\alpha = 0.3$ converges to the wide minimum, and with $\alpha = 4$ does not converge for most starting points.

Remark. The moral of this problem is that the learning rate of GD (and SGD) determines the sharpness of the minima the algorithm converge to. To converge to sharper local minima and thereby achieve a smaller loss, one often progressively reduces the learning rate using "learning rate schedulers". On the other hand, there is some recent work demonstrating that sharp local minima do not generalize well and should be avoided. We will revisit this topic later.

Solution. See the file `wideMinima_sol.py`. ■

Problem 5: *Implementing GD with duck typing.* Consider the optimization problem

$$\underset{x\in\mathbb{R}^n}{\text{minimize}} \quad \ell(Ax-b)+\tfrac{\lambda}{2}\|x\|^2,$$

where $b\in\mathbb{R}^{n-r+1}$ and the linear operator $A\in\mathbb{R}^{(n-r+1)\times n}$ is defined with a given $k\in\mathbb{R}^r$ and

$$A=\begin{bmatrix} k_1 & \cdots & k_r & 0 & \cdots & & & 0\\ 0 & k_1 & \cdots & k_r & 0 & \cdots & & 0\\ 0 & 0 & k_1 & \cdots & k_r & 0 & \cdots & 0\\ \vdots & & & \ddots & & \ddots & & \vdots\\ 0 & & \cdots & 0 & k_1 & \cdots & k_r & 0\\ 0 & & \cdots & 0 & 0 & k_1 & \cdots & k_r \end{bmatrix}.$$

Let $\ell\colon\mathbb{R}^m\to\mathbb{R}$ be the element-wise Huber loss defined as

$$\ell(y)=\sum_{i=1}^{m} h(y_i),$$

where

$$h=\begin{cases} \frac{1}{2}x^2 & \text{for } |x|\le 1\\ |x|-\frac{1}{2} & \text{otherwise.}\end{cases}$$

Code for evaluating ℓ and $\nabla\ell$ is implemented in the starter code `conv1D.py`. We use the following Python implementation of gradient descent

```
for _ in range(10000) :
    x = x - alpha*(A.T@(huber_grad(A@x-b))+lam*x)
```

where `x` and `b` are `numpy` arrays of lengths `n` and `n-r+1`.

The naïve approach of making `A` a regular `numpy` array with

```
from scipy.linalg import circulant
A = circulant(np.concatenate((np.flip(k),np.zeros(n-r))))[r-1:,:]
```

is inefficient because the `0`s of `A` are wasteful when computing the matrix-vector products `A@x` and `A.T@(...)`. Instead, we make `A` an object with methods computing matrix-vector products `A@x` and `A.T@(...)` without directly forming the $(n-r+1)\times n$ matrix.

Download the starter code `conv1D.py`. Implement the `__matmul__` methods so that the above gradient descent code runs without modification. You may not create a $(n-r+1)\times n$ numpy array (nor a $n\times(n-r+1)$ numpy array) in the implementation.

Remark. In machine learning, the operation Ax is called the *convolution* of x with the *receptive field* or *filter* k. In mathematics and signal processing, Ax is called the *cross-correlation* of x with the *kernel* k. (Traditional convolution has the indices of k flipped so that $k_r\cdots k_1$, rather than $k_1\cdots k_r$, appears in A.)

Hint. This problem can be completed by writing two lines of code. More specifically, the

```
return None
```

of `__matmul__` for `Convolution1d` and `TransposedConvolution1d` can each be replaced with

```
return np.asarray([LIST COMPREHENSION])
```

for some list comprehensions.

Solution. See the file `convolution1d_sol.py`. ■