Lecture 30

Understanding Gradient Descent.

(continued)

* Mini Project + New PSet (Friday)

* Drop Date is 11/18 (Wednesday)
GRADIENT DESCENT

\[ X_{k+1} = X_k - \alpha \nabla F(x_k) \]

- **next point**
- **current point**
- **step size**
- **gradient of current point**

Gradient is orthogonal to level sets (steepest descent direction)
DOES EVERY STEPSIZE WORK?

E.g., IN THE UNIVARIATE CASE

\[ F(x) = \frac{1}{2} x^2 \]
\[ \nabla F(x) = x \]

\[ x_{k+1} = x_k - \gamma \nabla F(x_k) \]

Wrong direction!

Slow, oscillates, diverges

1-step convergence

Stepsize \( \gamma \)
Convergence rate (for smooth functions)

\[ \lambda_i \text{ eigenvalues of Hessian (depend on point } x) \]
\[ Q : \text{ condition number } \frac{4}{m} \quad (Q \geq 1) \]

(Assume \( f(x^*) = 0 \), define \( C = \frac{1}{2} \| x - x^* \|^2 \))

CONVEX

\[ 0 \leq \lambda_i \leq L \]

If converges \( 0 < \varepsilon < \frac{2}{L} \)

Stepsize \( \gamma = \frac{1}{L} \)

\( f(x_k) \leq C \cdot \frac{1}{k} \) (slow)

STRONGLY CONVEX

\[ 0 < m \leq \lambda_i \leq L \]

If converges \( 0 < \delta < \frac{2}{L} \)

Stepsize \( \delta = \frac{2}{L + m} \)

\( f(x_k) \leq C \left( \frac{q-1}{q+1} \right)^{2k} \) (fast)
\[ \log f(x_u) \leq \log C + 2k \log \left( \frac{a-1}{a+1} \right) \]

(linear function of \(k\))

\[ \text{log } f(x_u) \]

\[ 2 \log \left( \frac{a-1}{a+1} \right) \]

\[ \text{go to } 0 \]

(\(a\))
Computing gradients

We all know how to compute derivatives, right?

Several approaches:

- "By hand"
  \[ f(x) = x + \frac{b}{x} \]
  \[ f'(x) = 1 - \frac{b}{x^2} \]
  \[ f''(x) = \frac{2b}{x^3} > 0 \]

- Symbolic computation
  e.g. MAPLE, MATHEMATICA.
  \[ D[f, x] \rightarrow 1 - \frac{b}{x^2} \]

- Numerical approximations
  (e.g., finite differences)
  \[ f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \]
  Small \( h \)
Automatic differentiation

\[ f(x, y) = x + \frac{y}{x} \]

"Chain rule on steroids"

General form applies to (almost) any program!

- Many implementations (e.g., Julia, Python, TensorFlow, etc.)

- Different versions (forward vs reverse mode)

- Equivalent/related: "backpropagation" (ML), "adjoint method" (PDE/control)
go to Julia
**Example:** Neural Networks

\[
\begin{bmatrix}
    x_1 \\
    \vdots \\
    x_n \\
\end{bmatrix} \rightarrow \Theta_1 \rightarrow \Theta_2 \rightarrow \cdots \rightarrow \Theta_m \rightarrow \begin{bmatrix}
    y_1 \\
    \vdots \\
    y_m \\
\end{bmatrix}
\]

\[\Theta_1 \ldots \Theta_n\] are matrices (unknown parameters)

\(\sigma\) is a fixed nonlinear function, applied componentwise

\[x \rightarrow \sigma(x) \rightarrow \sigma(\sigma(\cdots))\]

\[\phi_\theta(x) = \sigma\left(\Theta_n \cdots \sigma\left(\Theta_2 \cdot \sigma\left(\Theta_1 \cdot x\right)\right)\right)\]

\[\sigma\left(\begin{bmatrix}
    z_1 \\
    \vdots \\
    z_n \\
\end{bmatrix}\right) = \begin{bmatrix}
    \sigma(z_1) \\
    \vdots \\
    \sigma(z_n) \\
\end{bmatrix}\]

\(\sigma\) is applied componentwise.
PROBLEM: \( Y_i \approx \phi_0(x_i) \)

Given a collection of \( \begin{bmatrix} \frac{1}{2} \\ \frac{3}{4} \end{bmatrix} \) input/output pairs, find "good" matrices \( \Theta_i \):

\[
\begin{bmatrix} \frac{1}{2} \\ \frac{3}{4} \end{bmatrix}
\]

Set up as big optimization problem:

\[
\min_{\Theta_1, \Theta_2, \ldots, \Theta_n} \sum_{(x, y) \in \text{dataset}} \text{loss} (Y, \phi(x))
\]

- NOT CONVEX
- HUGE PROBLEM
- HOW TO COMPUTE GRADIENT?

difficult! (composition of functions, one term per data point)
**GRADIENT**  (scalar function)  \( f: \mathbb{R}^n \rightarrow \mathbb{R} \)

\[ \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \quad \text{column vector} \quad n \times 1 \]

**JACOBIAN**  (vector function)  \( f: \mathbb{R}^n \rightarrow \mathbb{R}^m \)

\[ f: x \mapsto Ax \]

\[ \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \rightarrow \begin{pmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{pmatrix} \]

\[ Jf(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \quad \text{matrix} \quad m \times n \]

**NOTE:** Jacobian of scalar function is transpose of gradient.
Why this definition?

\[ f(x) = AX \quad (A \in \mathbb{R}^{m \times n}) \quad J(ABx) \]

then \[ Jf(x) = A \]

\[ AB \cdot C \cdot D \]

What happens with composition/products?

Think about matrix multiplication:

\[ ABCD = (((AB)C)D) \]

\[ = (A(B(CD))) \]
Similarly, chain rule:

For compositions: \( z(y(x)) \)

\[
\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}
\]

(or, more properly)

\[
\begin{align*}
\left| \frac{dz}{dx} \right| &= \left| \frac{dz}{dy} \right| \cdot \left| \frac{dy}{dx} \right| \\
x &= x_0 \quad y &= y(x_0) \quad x = x_0
\end{align*}
\]
Next Time:

Can we do Better than Gradient Descent??