CIS3990-002: Mathematics of Machine Learning Fall 2023

Lecture: Multivariate Calculus

Date: November 6th, 2023 *Author: Eric Wong*

1 Calculus Basics

An important question in machine learning is "Why does my training algorithm work and how long will it take?" As most machine learning problems can be framed as an optimization problem, theoretical convergence rates in optimization will answer this problem. For example, recall that we typically minimize the empirical risk (often referred to as the objective in optimization):

$$\min_\theta R_{\text{emp}}(f_\theta) = \min_\theta \sum_i \ell(f_\theta(x_i), y_i)$$

and that we often use something called an optimizer to solve this minimization problem, such as stochastic gradient descent. Since stochastic gradient descent has randomness, it is not guaranteed to always improve the objective. So first of all, why does stochastic gradient descent work if it's not always guaranteed to make an improvement? And second, how long does it take for such an optimizer to find a good solution? To answer why it works, we'll show that gradient descent will eventually converge to something close to the right answer. To answer how long, we'll characterize this by the number of steps that the optimizer needs to take to get some distance away from the optimal solution. This will be a result of the form

$$R(f_\theta) - R(f^*) \in O(1/\sqrt{T})$$

for stochastic gradient descent. Compare this with $O(1/T)$ for standard gradient descent. This tells us that the penalty for using random gradients instead of exact gradients is a factor of $\frac{1}{\sqrt{T}}$. In other words, to get ϵ error, SGD requires $\frac{1}{\epsilon^2}$ steps whereas GD requires $\frac{1}{\epsilon}$ steps.

To understand the optimization aspects of machine learning, we'll need to work with calculus thanks to all the gradients and approximations. The proof for SGD will be a combination of the linear algebra and probability from the previous two modules, with the calculus on the current module. Therefore, we'll begin our review of calculus with the univariate case.

- The derivative of f at x is defined as the limit

  $$\frac{\partial f}{\partial x} = \lim_{h \to 0^+} \frac{f(x+h) - f(x)}{h}$$

  This is the direction of steepest ascent.

- Example: $f(x) = x^n$, then

  $$\frac{\partial f}{\partial x} = \lim_{h \to 0^+} \frac{(x+h)^n - f(x)^n}{h} = \lim_{h \to 0^+} \frac{\sum_{i=0}^n \binom{n}{i} x^{n-i} h^i - x^n}{h}$$

  $$= \lim_{h \to 0^+} \sum_{i=1}^n \binom{n}{i} x^{n-i} h^{i-1} = n x^{n-1}$$

- Derivatives are useful for representing a function as an infinite sum of approximations around a single point. A Taylor polynomial of degree n of f at x_0 is

$$T_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

  where $f^{(k)}(x_0)$ is the kth derivative evaluated at x_0.

- For a smooth function $f \in \mathcal{C}^\infty$, a Taylor series of f at x_0 is the infinite Taylor polynomial

$$T_\infty(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

- Differentiation rules. Let f' be the derivative of f. Then,

  1. Product rule: $(f(x)g(x))' = f'(x)g(x) + f(x)g'(x)$
  2. Quotient rule: $\left(\frac{f(x)}{g(x)}\right)' = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$
  3. Sum rule: $(f(x) + g(x))' = f'(x) + g'(x)$
  4. Chain rule: $(g(f(x)))' = (g \circ f)'(x) = g'(f(x))f'(x)$

These are helpful for univariate calculus. However, machine learning problems are typically multivariate (think pixels or words in an sentence). Therefore, we need to comfortable with multivariate calculus.

- Suppose $f : \mathbb{R}^n \to \mathbb{R}$. If $x = (x_1, \ldots, x_n)$ then the partial derivatives are

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x)}{h}$$

  for $i = 1, \ldots, n$. Typically we can collect this into a single row vector called the Jacobian,

$$\nabla_x f = \frac{df}{dx} = \left[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right] \in \mathbb{R}^n$$

  We put it into a row vector because if we have a function $f(x) = (f_1(x), \ldots, f_m(x))$ with multiple outputs, we stack these as rows

$$\nabla_x f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

- Example: Let $f(x_1, x_2) = x_1^2 x_2 + x_1 x_2^3$. Then,

$$\frac{\partial f(x_1, x_2)}{\partial x_1} = 2x_1 x_2 + x_2^3$$

$$\frac{\partial f(x_1, x_2)}{\partial x_2} = x_1^2 + 3x_1 x_2^2$$

- Rules for partial differentiation with one output:
  1. Product rule: $\frac{\partial}{\partial x}(f(x)g(x)) = \frac{\partial f}{\partial x}g(x) + f(x)\frac{\partial g}{\partial x}$
  2. Sum rule: $\frac{\partial}{\partial x}(f(x) + g(x)) = \frac{\partial f}{\partial x} + \frac{\partial g}{\partial x}$
  3. Chain rule: $\frac{\partial}{\partial x}(g \circ f)(x) = \frac{\partial}{\partial x}(g(f(x))) = \frac{\partial g}{\partial f(x)}\frac{\partial f}{\partial x}$.
- In the last example, we assumed that f has one output. What if the intermediate function has more than one output?
- Example: Consider $f : \mathbb{R}^2 \to \mathbb{R}$ and $x_1(t), x_2(t)$ are both functions of t. Then

$$\frac{df}{dt} = \begin{bmatrix} \frac{df}{dx_1}, \frac{df}{dx_2} \end{bmatrix} \begin{bmatrix} \frac{dx_1}{dt} \\ \frac{dx_2}{dt} \end{bmatrix}$$

- In other words,

$$\frac{d}{dt}f(x(t)) = \sum_i \frac{df}{dx_i}\frac{dx_i}{dt}$$

The above was done for scalar valued functions. We can generalize all of the above to multiple outputs, i.e. vector valued functions.

- Suppose $f : \mathbb{R}^n \to \mathbb{R}^m$, where $f(x) = \begin{bmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{bmatrix} \in \mathbb{R}^m$. Then, $\frac{df}{dx_i} = \begin{bmatrix} \frac{f_1}{dx_i} \\ \vdots \\ \frac{f_m}{dx_i} \end{bmatrix}$.
- We can stack this altogether for all input derivatives to get

$$\frac{df}{dx} = \begin{bmatrix} \frac{f_1}{dx_1} \cdots \frac{f_1}{dx_n} \\ \vdots \\ \frac{f_m}{dx_1} \cdots \frac{f_m}{dx_n} \end{bmatrix}$$

  This matrix is called the *Jacobian*. When f maps onto a scalar, this is the special case from before (a row vector).

Therefore, we can do derivatives of many outputs (vectors) with respect to many inputs (vectors). Lastly, we can take this one step further and do derivatives of tensor outputs with respect to tensor inputs.

- Suppose we have a transformation $f : \mathbb{R}^{m \times n} \to \mathbb{R}^{p \times q}$, i.e. $A = f(B)$. Then, $J = \frac{dA}{dB} \in \mathbb{R}^{m \times n \times p \times q}$ where $J_{ijkl} = \frac{dA_{ij}}{dB_{kl}}$
- Recall that the set of $m \times n$ matrices is isomorphic to the set of mn length vectors, i.e. $\mathbb{R}^{mtimesn}$ and $\mathbb{R}^{mn}$ are isomorphic. Since the gradient is a linear operator, we can treat gradients of matrices (or tensors) as equivalent to gradients of vectors.
- For example, we can reshape A and B to be vectors of length mn and pq, and then compute $J \in \mathbb{R}^{mn \times pq}$. Then reshape this back to the tensor form.

- Example: let $f(R) = R^\top R = K \in \mathbb{R}^{N\times N}$ where $R \in \mathbb{R}^{m\times N}$. What is $\frac{dK}{dR}$? We know this is in $\mathbb{R}^{N\times N\times M\times N}$, lets start with just one output.

$$\frac{dK_{pq}}{dR} = \frac{d}{dR} r_p^\top r_q = \frac{d}{dR}\sum_{m=1}^{M} R_{mp}R_{mq}$$

  Therefore, for R_{ij} we have

$$\frac{dK_{pq}}{R_{ij}} = \frac{d}{dR_{ij}}\sum_{m=1}^{M} R_{mp}R_{mq} = \begin{cases} R_{iq} \text{ if } j = p, p \neq q \\ R_{ip} \text{ if } j = q, p \neq q \\ 2R_{iq} \text{ if } j = p, p = q \\ 0 \text{ otherwise} \end{cases}$$

- Section 5.5 has useful identities for computing gradients.

$$\frac{\partial}{\partial \boldsymbol{X}} \boldsymbol{f}(\boldsymbol{X})^\top = \left(\frac{\partial \boldsymbol{f}(\boldsymbol{X})}{\partial \boldsymbol{X}}\right)^\top \tag{5.99}$$

$$\frac{\partial}{\partial \boldsymbol{X}} \text{tr}(\boldsymbol{f}(\boldsymbol{X})) = \text{tr}\left(\frac{\partial \boldsymbol{f}(\boldsymbol{X})}{\partial \boldsymbol{X}}\right) \tag{5.100}$$

$$\frac{\partial}{\partial \boldsymbol{X}} \det(\boldsymbol{f}(\boldsymbol{X})) = \det(\boldsymbol{f}(\boldsymbol{X}))\text{tr}\left(\boldsymbol{f}(\boldsymbol{X})^{-1}\frac{\partial \boldsymbol{f}(\boldsymbol{X})}{\partial \boldsymbol{X}}\right) \tag{5.101}$$

$$\frac{\partial}{\partial \boldsymbol{X}} \boldsymbol{f}(\boldsymbol{X})^{-1} = -\boldsymbol{f}(\boldsymbol{X})^{-1}\frac{\partial \boldsymbol{f}(\boldsymbol{X})}{\partial \boldsymbol{X}}\boldsymbol{f}(\boldsymbol{X})^{-1} \tag{5.102}$$

$$\frac{\partial \boldsymbol{a}^\top \boldsymbol{X}^{-1}\boldsymbol{b}}{\partial \boldsymbol{X}} = -(\boldsymbol{X}^{-1})^\top \boldsymbol{a}\boldsymbol{b}^\top (\boldsymbol{X}^{-1})^\top \tag{5.103}$$

$$\frac{\partial \boldsymbol{x}^\top \boldsymbol{a}}{\partial \boldsymbol{x}} = \boldsymbol{a}^\top \tag{5.104}$$

$$\frac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{a}^\top \tag{5.105}$$

$$\frac{\partial \boldsymbol{a}^\top \boldsymbol{X}\boldsymbol{b}}{\partial \boldsymbol{X}} = \boldsymbol{a}\boldsymbol{b}^\top \tag{5.106}$$

$$\frac{\partial \boldsymbol{x}^\top \boldsymbol{B}\boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{x}^\top(\boldsymbol{B} + \boldsymbol{B}^\top) \tag{5.107}$$

$$\frac{\partial}{\partial \boldsymbol{s}}(\boldsymbol{x} - \boldsymbol{A}\boldsymbol{s})^\top \boldsymbol{W}(\boldsymbol{x} - \boldsymbol{A}\boldsymbol{s}) = -2(\boldsymbol{x} - \boldsymbol{A}\boldsymbol{s})^\top \boldsymbol{W}\boldsymbol{A} \quad \text{for symmetric } \boldsymbol{W} \tag{5.108}$$

Figure 1: Identities for computing gradients

- Chain rule revisited: recall that if $y = f(x)$ for $f : \mathbb{R}^N \to \mathbb{R}$ and $x = g(t)$ for $g : \mathbb{R} \to \mathbb{R}^N$, then

$$\frac{\partial}{\partial t} f(g(t) = \sum_i \frac{\partial f}{\partial x_i}\frac{\partial x_i}{\partial t} = \frac{\partial f}{\partial x} \cdot \frac{\partial x}{\partial t}$$

  where the former is a row vector (one output) and the latter is a column vector (N outputs).

- This can now be generalized to multiple inputs and multiple outputs by simply replicating the formula over multiple outputs of f and multiple inputs of g. If $f : \mathbb{R}^M \to \mathbb{N}$ and $g : \mathbb{R}^L \to \mathbb{R}^M$

then

$$\frac{\partial f}{\partial x} \cdot \frac{\partial x}{\partial t}$$

where $\frac{\partial f}{\partial x} \in \mathbb{R}^{N \times M}$ and $\frac{\partial g}{\partial t} \in \mathbb{R}^{M \times L}$

The big place where derivatives are used is in backpropagation and automatic differentiation. By repeatedly applying the chain rule, you can calculate the derivative of any formula. This used to minimize

$$\min_{\theta} \sum_{i} \ell(f_\theta(x_i), y_i)$$

where we repeat the chain rule over and over again until we get to θ.

- Suppose we have a chain of operations,

  $$y = f(x) = (f_K \odot f_{K-1} \odot \cdots \odot f_1)(x) = f_K(f_{K-1}(\cdots(f_1(x))\cdots))$$

  and $\ell(f(x), y) = \|y - f(x)\|_2^2$, where θ_i is the parameters of the ith function f_i. Then,

  $$\frac{\partial L}{\partial \theta_{K-1}} = \frac{\partial L}{\partial f_K} \frac{\partial f_K}{\partial \theta_{K-1}}$$

  $$\frac{\partial L}{\partial \theta_{K-2}} = \frac{\partial L}{\partial f_K} \frac{\partial f_K}{\partial f_{K-1}} \frac{\partial K-1}{\partial \theta_{K-2}}$$

  and so on. This is the concept driving automatic differentiation.

- Backward mode: We will recursively compute $\frac{\partial y}{\partial v_i}$ for all intermediate nodes v_i, where $v_i = (f_i \odot \cdots \odot f_1)(x)$ i.e.

  $$\frac{\partial y}{\partial v_i} = \sum_{j:\text{parent}(v_i)} \frac{\partial f}{\partial v_j} \frac{\partial v_j}{\partial v_i}$$

- Forward mode: We will recursively compute $\frac{\partial v_i}{\partial x}$ for all intermediate nodes v_i, i.e.

  $$\frac{\partial v_i}{\partial x} = \sum_{j:\text{child}(v_i)} \frac{\partial v_i}{\partial v_j} \frac{\partial v_j}{\partial x}$$

Up to this point, we've mostly focused on first-order derivatives. Sometimes we want to use higher order derivatives. This depends on the optimization algorithm.

Suppose we want to minimize $\min_\theta f(\theta)$. Whereas gradient descent only used first-order derivatives, i.e.

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

Other methods like Newton's method for optimization use higher order derivatives, like

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla^2 f(\theta^{(t)})^{-1} \nabla f(\theta^{(t)})$$

The second derivative term here $\nabla^2 f(\theta^{(t)})$ is called the Hessian. Let's look at higher order derivatives in more detail.

- First consider a function $f : \mathbb{R}^2 \to \mathbb{R}$ of two variables, $f(x, y)$.
- Derivatives are *linear operators*. Therefore the derivative operator behaves very similarly to matrices. Furthermore, the order doesn't matter.
- Examples of higher order derivatives:
  1. $\frac{\partial^2 f}{\partial x^2} = \frac{\partial}{\partial x}\frac{\partial f}{\partial x}$
  2. $\frac{\partial^n f}{\partial x^n} = \frac{\partial}{\partial x} \cdots \frac{\partial f}{\partial x}$
  3. $\frac{\partial^2 f}{\partial y \partial x} = \frac{\partial}{\partial y}\frac{\partial f}{\partial x} = \frac{\partial}{\partial x}\frac{\partial f}{\partial y} = \frac{\partial^2 f}{\partial x \partial y}$
- For $f(x, y)$, it can be convenient to write down all the second order derivatives as

  $$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}$$

  This is called the Hessian matrix and is sometimes denoted as $H = \nabla^2_{x,y} f(x, y)$.
- We can do this more generally for functions of more than 2 variables, $f : \mathbb{R}^N \to \mathbb{R}$ for $f(x) = f(x_1, \ldots, x_N)$:

  $$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_N} & \cdots & \frac{\partial^2 f}{\partial x_N^2} \end{bmatrix}$$

  Or equivalently, $H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$
- Let's generalize even further: what if there are multiple outputs for $f : \mathbb{R}^N \to \mathbb{R}^M$?
- Same as the Jacobian, we'll adopt an "outputs first" notation. Recall that the Jacobian $J_{ij} = (\nabla f)_{ij} = \frac{\partial f_i}{\partial x_j}$, where the first dimension corresponded to the output and the second dimension corresponded to the partial derivatives.
- Then, the Hessian of f will be an $M \times N \times N$ tensor, where $H_{ijk} = (\nabla^2 f_i)_{jk}$ where H_i is the Hessian of the ith output.

Let's now return to the Taylor series. The Taylor series was a useful tool for approximating a function as a series of polynomials. For example, the linear approximation of f around x_0 was

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

How does this all change for functions of more than one variable?

- The gradient is often used for a local linear approximation that looks similar to the uni-variate case.

  $$f(x) \approx f(x_0) + \nabla f(x_0)(x - x_0)$$

  Remember that $\nabla f(x_0)$ is a row vector (outputs first).

- Recall that the full univariate Taylor series was

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

- The multivariate Taylor series is similar:

$$f(x) = \sum_{k=0}^{\infty} \frac{D^k f(x_0)}{k!}\delta^k$$

  where you can take D^k to be the kth derivative of f and $\delta = x - x_0$

- δ^k is the outer product of δ done k times, resulting in a tensor of size $D \times D \times \cdots \times D$ where this is done k times (a kth order array).

- For $k = 2$, this is

$$\delta^2 = \delta \otimes \delta = \delta\delta^\top, \quad \delta^2[i, j] = \delta[i]\delta[j]$$

- For $k = 3$, this is

$$\delta^3 = \delta \otimes \delta \otimes \delta \quad \delta^3[i, j, k] = \delta[i]\delta[j]\delta[k]$$

- In general, this is

$$\delta^k[i_1, \ldots, i_k] = \delta[i_1] \cdots \delta[i_k]$$

- Lastly (this is a fairly big abuse of notation from the textbook) we can collapse the terms with the usual dot product (sometimes called the Frobenius inner product):

$$D^k f(x_0)\delta^k = \sum_{i_1} \cdots \sum_{i_k} D^k f(x_0)[i_1, \ldots, i_k]\delta^k[i_1, \ldots, i_k]$$

$$= \sum_{i_1} \cdots \sum_{i_k} D^k f(x_0)[i_1, \ldots, i_k]\delta[i_1] \cdots \delta[i_k]$$

  As a recap, $D^k f(x_0)$ and δ^k are both kth order tensors where each dimension has size D. Note that the input dimension D is different from the derivative operator D^k.

- Examples:

  1. $k = 0$ we have $D^0 f(x_0)\delta^0 = f(x_0) \in \mathbb{R}$
  2. $k = 1$ we have $D^1 f(x_0)\delta^1 = \nabla f(x_0)\delta \in \mathbb{R}$
  3. $k = 2$ we have $D^2 f(x_0)\delta^2 = \nabla^2 f(x_0)\delta^2 = \text{trace}(H\delta\delta^\top) = \delta^\top H\delta \in \mathbb{R}$
  4. $k = 3$ we have $D^3 f(x_0)\delta^3 = \sum_i \sum_j \sum_k D^3 f(x_0)[i, j, k]\delta[i]\delta[j]\delta[k] \in \mathbb{R}$

What are multivariate Taylor series used for?

- Mainly to create a locally linear approximation of f around $x_0 0$, i.e.

$$f(x) \approx f(x_0) + \nabla f(x_0)(x - x_0)$$

  which approximates the function f at x_0 with a straight line

- In iterative optimization methods (like gradient descent), to approximate local regions using the local linear approximation.
- To minimize $f(x_0) + \nabla f(x_0)(x - x_0)$, we simply take a step in the direction of $-\nabla f(x_0)$ (gradient methods, 1st order optimization) constrained within some radius α (the step size).
- We have to minimize within a radius since a linear function is unbounded otherwise.
- For ℓ_2 bounded steps (i.e. $\|x - x_0\|_2 \leq \alpha$ this is equivalent to gradient descent.
- FOr other norms, this is called steepest descent with respect to the ℓ_p norm:

$$\min_{\|x-x_0\|_p \leq \alpha} f(x_0) + \nabla f(x_0)(x - x_0)$$

- To minimize $f(x_0) + \nabla f(x_0)(x - x_0) + (x - x_0)^\top \nabla^2 f(x_0)(x - x_0)$ we can find the root of this equation since it is a quadratic:

$$\frac{\partial}{\partial x}[f(x_0) + \nabla f(x_0)(x - x_0) + (x - x_0)^\top \nabla^2 f(x_0)(x - x_0)]$$

$$= \nabla f(x_0)^\top + \nabla^2 f(x_0)(x - x_0) = 0 \Rightarrow x = x_0 - \nabla^2 f(x_0)^{-1} \nabla f(x_0)^\top$$

- In calculating integrals, i.e.

$$\mathbb{E}[f(x)] = \int f(x)p(x)dx$$

  can be very complicated even if the distribution p is known i.e. Gaussian. Instead, we can approximate $f(x)$ with a Taylor series around the mean and integrate that instead.
- Why do we use first-order methods for things like neural network? One reason is because the number of parameters is so large, computing the Hessian is computationally infeasible.