CIS3333: Mathematics of Machine Learning Fall 2025

Lecture: Practical Aspects of Gradient Descent

Date: November 19th, 2025 *Author: Surbhi Goel*

In the last two lectures, we proved that GD converges at $O(1/T)$ for smooth, convex functions and SGD converges at $O(1/\sqrt{T})$ for convex functions. These guarantees assume we eventually reach the optimum—but they say nothing about *how painful the journey is.* In practice, optimization can be slow even when the theory promises convergence, and we often face challenges like varying curvature and noise.

Why does vanilla GD/SGD struggle in practice, and what can we do about it?

Today, we use a single concrete example—a 2D diagonal quadratic—to understand a practical challenge: **ill-conditioning** (curvature mismatch).

1 Challenge: Ill-Conditioning

Consider the 2D diagonal quadratic

$$f(x_1, x_2) = \frac{1}{2}\big(\lambda_1 x_1^2 + \lambda_2 x_2^2\big), \quad 0 < \lambda_1 < \lambda_2, \quad \nabla f(x_1, x_2) = \begin{bmatrix} \lambda_1 x_1 \\ \lambda_2 x_2 \end{bmatrix}.$$

The condition number is $\kappa = \lambda_2/\lambda_1$. When κ is large (e.g., 10,000), we have a severe **curvature mismatch**:

- To avoid divergence in the steep direction (x_2), we must use a small step size $\eta \approx 1/\lambda_2$.
- But in the flat direction (x_1), this step size is tiny compared to the scale of the problem, leading to painfully slow convergence.

Specifically, the contraction factor is $(1 - 1/\kappa)$. If κ = 10,000, we need roughly 10,000 steps to reduce the error by a factor of e.

2 Solution A: Regularization

One common way to fix ill-conditioning is **Regularization**. By adding a penalty term $\frac{\lambda}{2}\|x\|_2^2$ to the objective, we effectively add a positive constant to the eigenvalues of the Hessian. This prevents any curvature from being too small (flat), which improves the condition number κ.

The trade-off is that we are now minimizing a different function (biased towards 0), but this often helps with overfitting in machine learning.

3 Solution B: Momentum

Another solution is to change the algorithm. **Momentum** accumulates velocity from past gradients:

$$\begin{aligned} v_{t+1} &= \beta v_t + \nabla f(x_t), \\ x_{t+1} &= x_t - \eta v_{t+1}, \end{aligned}$$

where $\beta \in [0, 1)$ is the momentum parameter (in practice often around 0.9). When $\beta = 0$, this reduces to standard gradient descent.

Let us unroll the recursion, starting from $v_0 = 0$:

$$\begin{aligned} v_1 &= \nabla f(x_0), \\ v_2 &= \beta \nabla f(x_0) + \nabla f(x_1), \\ v_3 &= \beta^2 \nabla f(x_0) + \beta \nabla f(x_1) + \nabla f(x_2), \quad \text{etc.} \end{aligned}$$

In general,

$$v_t = \sum_{k=0}^{t-1} \beta^k \, \nabla f(x_{t-1-k}).$$

So v_t is an **exponentially weighted moving average** of past gradients:

- recent gradients have larger weight,
- older gradients have weight β^k, which decays geometrically.

This gives a useful picture:

- if gradients keep pointing in roughly the same direction, they *reinforce* each other in v_t;
- if gradients keep changing sign in some direction, they *cancel out* in v_t.

Toy Example 1: Constant Gradient (Flat Direction). To see how momentum helps with ill-conditioning, consider a 1D toy where the gradient is constant $g > 0$ at every step. This models coordinate 1 of our 2D quadratic when we're far from the origin: the gradient $\lambda_1 x_1$ barely changes because λ_1 is small.

With momentum, the velocity evolves as

$$v_{t+1} = \beta v_t + g, \qquad v_0 = 0.$$

Unrolling the recursion:

$$v_1 = g, \quad v_2 = \beta g + g = (1 + \beta) g, \quad v_3 = \beta(1 + \beta) g + g = (1 + \beta + \beta^2) g, \ \ldots$$

In general,

$$v_t = g(1 + \beta + \beta^2 + \cdots + \beta^{t-1}) = g \cdot \frac{1 - \beta^t}{1 - \beta}.$$

The velocity *builds up* over time because the gradient keeps reinforcing itself. After many steps ($\beta^t \approx 0$), the velocity saturates at

$$v_t \approx \frac{g}{1 - \beta}.$$

Effective step size. The step taken at time t is

$$x_{t+1} - x_t = -\eta v_{t+1} \approx -\eta \cdot \frac{g}{1-\beta}.$$

So momentum behaves like vanilla GD with a *larger* effective stepsize:

$$\eta_{\text{eff}} \approx \frac{\eta}{1-\beta}.$$

Example: With $\eta = 0.01$ and $\beta = 0.9$, we get $\eta_{\text{eff}} \approx 0.1$, a $10\times$ boost!

Toy Example 2: Alternating Gradient (Steep Direction). Now consider the opposite extreme: the gradient flips sign each step. This models coordinate 2 of our 2D quadratic when the stepsize is too large—we overshoot the minimum and oscillate back and forth. Suppose

$$\nabla f(x_t) = g_t = (-1)^t g$$

for some fixed $g > 0$ (gradient alternates $+g, -g, +g, -g, \dots$).

With momentum, the velocity evolves as:

$$\begin{aligned}
v_1 &= g, \\
v_2 &= \beta g - g = (\beta - 1)g, \\
v_3 &= \beta(\beta - 1)g + g = (\beta^2 - \beta + 1)g, \\
v_4 &= \beta(\beta^2 - \beta + 1)g - g = (\beta^3 - \beta^2 + \beta - 1)g, \text{ etc.}
\end{aligned}$$

When β is close to 1 (e.g., $\beta = 0.9$), the alternating terms nearly cancel:

- $v_2 = -0.1g$ (almost zero),
- $v_3 = 0.91g$ (smaller than g),
- $v_4 = -0.181g$, etc.

The velocity v_t stays much smaller than the raw gradient magnitude g.

Damping oscillations. Compare vanilla GD vs. momentum in this oscillating regime:

- **Vanilla GD:** Takes full steps of size ηg each time, alternating direction. The iterates zig-zag wildly across the minimum.
- **Momentum:** The velocity v_t is much smaller than g because consecutive gradients cancel. Steps are *smaller* in this direction, reducing oscillations.

This is the **damping** effect: momentum suppresses motion in directions where the gradient keeps changing sign.

4 Solution C: Adaptive Methods

A third solution is to use a different step size for each coordinate. Ideally, if we knew the curvature λ_i along each coordinate, we would set $\eta_i \propto 1/\lambda_i$. Since we don't know λ_i, adaptive methods estimate it from the data.

The algorithm **RMSProp** maintains a running average of *squared* gradients to estimate the magnitude of the gradient in each direction:

$$s_{t+1} = \beta s_t + (1-\beta)(\nabla f(x_t) \odot \nabla f(x_t)),$$

where $\odot$ denotes the element-wise product. This s_t acts as a proxy for the "steepness" of the function along each coordinate.

The update rule then scales the step size inversely by the root of this sum:

$$x_{t+1} = x_t - \frac{\eta}{\sqrt{s_{t+1}} + \epsilon} \odot \nabla f(x_t),$$

where ϵ is a small constant for numerical stability.

This achieves our goal in the context of our 2D quadratic:

- **Steep direction (x_2):** Large curvature λ_2 leads to large gradients $\Rightarrow$ large s_t $\Rightarrow$ small effective step size (preventing divergence).
- **Flat direction (x_1):** Small curvature λ_1 leads to small gradients $\Rightarrow$ small s_t $\Rightarrow$ large effective step size (speeding up convergence).

Adam combines this adaptive scaling (from RMSProp) with momentum, making it robust to ill-conditioning without manual per-parameter tuning.