SGD, Momentum, Nesterov

Lecture 4 · ES 667: Deep Learning

Prof. Nipun Batra
IIT Gandhinagar · Aug 2026

Learning outcomes

By the end of this lecture you will be able to:

1. Identify **ravines** and **saddles** in high-dimensional loss.
2. Show why **vanilla SGD** oscillates across ravines.
3. Derive **momentum** as EMA of gradients.
4. Explain **Nesterov's lookahead** and its convergence rate payoff.
5. Pick **β** appropriately (0.9 default; when to adjust).
6. Diagnose an optimizer failure from training curves.

Recap · where we are

- **Deep networks** are trainable with ResNets + He init + ReLU.
- **PyTorch recipe** — forward, loss, zero_grad, backward, step.
- **Debugging ladder** — overfit one batch, LR finder, error analysis.

> REFERENCE
>
> Today maps to **UDL Ch 6** · fitting models (SGD, momentum, acceleration).

One piece we glossed over: **the optimizer**. Today we open that box.

Four questions for today

1. What does the loss landscape *look like*, really?
2. Why does vanilla SGD oscillate on it?
3. How does momentum fix the oscillation?
4. What does Nesterov's *lookahead* add on top?

Pop quiz · what's killing this run?

A 50-layer ResNet trains fine for 200 steps, then loss starts **oscillating** between 1.4 and 1.9 forever — never diverging, never improving.

POP QUIZ

(a) Vanishing gradients.
(b) Step size too large for a narrow ravine.
(c) Bad data shuffling.
(d) Saddle point.

Stop and decide. We'll come back to this once you've seen the ravine geometry — your gut answer will probably change.

This is **the most common failure mode** of vanilla SGD on real loss surfaces. By the end of today you'll diagnose it from the curves alone.

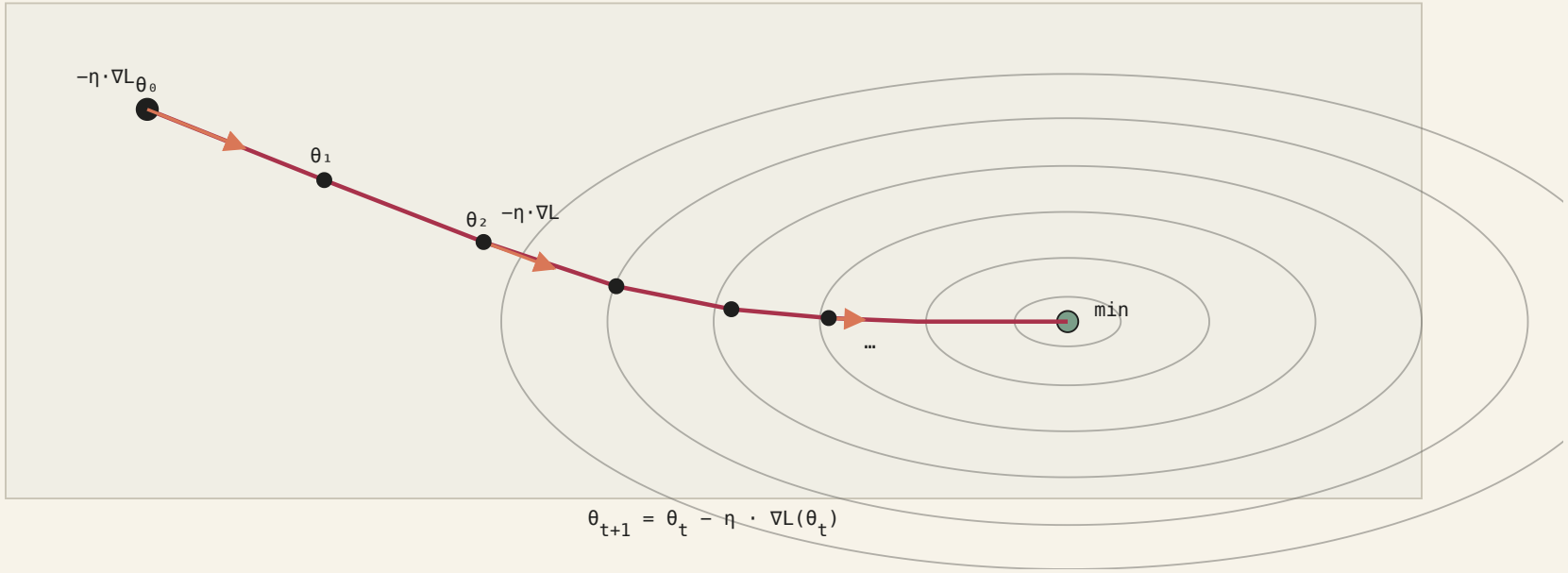
PART 1

The loss landscape

What makes neural-net optimization hard

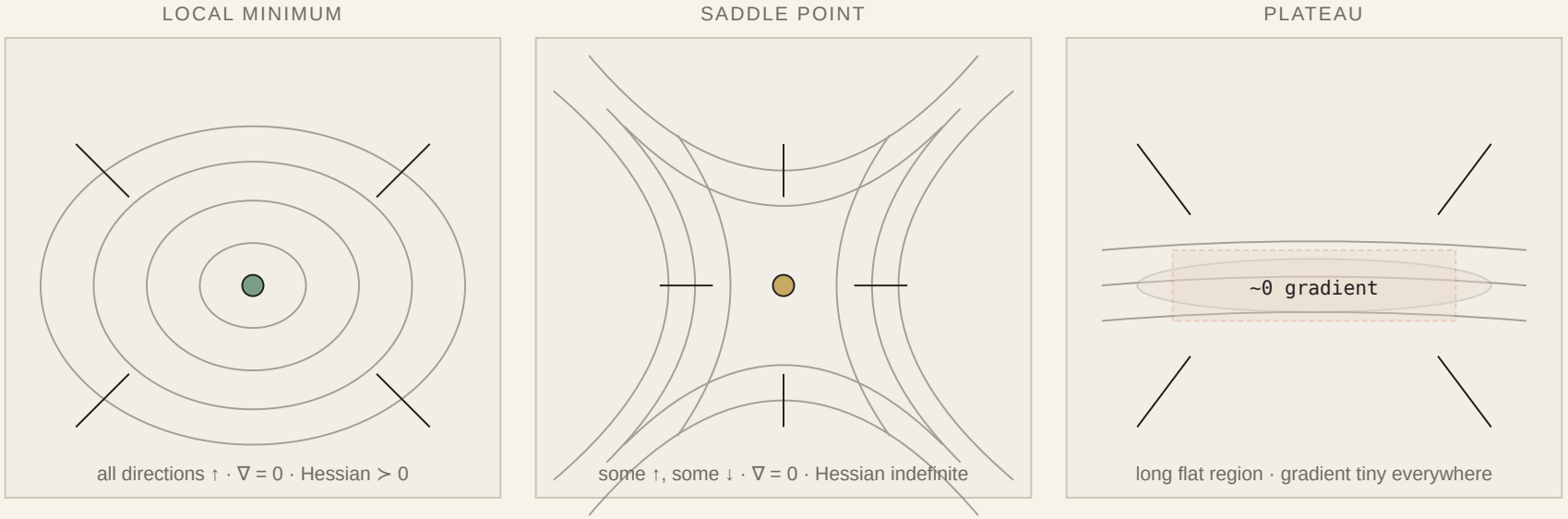
Gradient descent — picture

Gradient descent — each step goes against the gradient, step size = η · ‖∇‖



Three kinds of critical points

Three kinds of critical points — the landscape is mostly saddles in high dimension



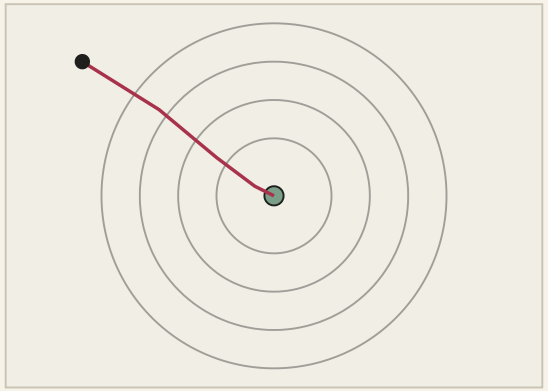
IN HIGH DIMENSION

Bengio et al. 2014 · **saddles outnumber minima exponentially**. Momentum + noise help escape them. Most of SGD's time is spent crossing saddles, not getting stuck in minima.

The ravine problem — κ elongates the basin

Condition number $\kappa = \lambda_{max} / \lambda_{min}$ · bigger κ = more oscillation

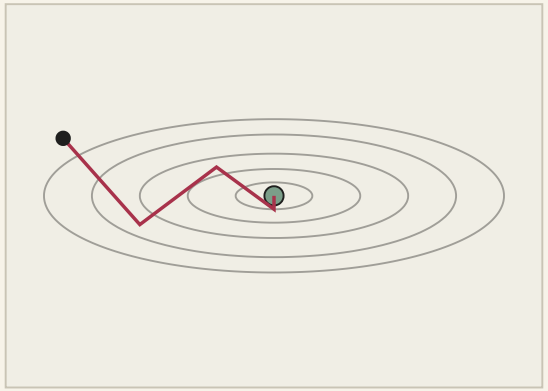
K = 1 (ISOTROPIC)



gradient always points at minimum

`SGD converges in O(log 1/ε) steps`

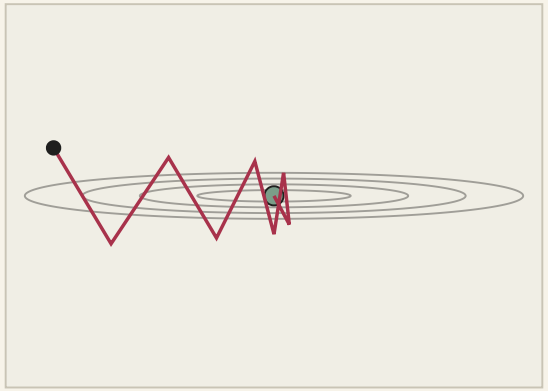
K = 10 (MILD)



a few zig-zags · still reasonable

`steps ~ √κ ≈ 3`

K = 100 (DEEP-NET TYPICAL)



SGD oscillates heavily across the narrow axis

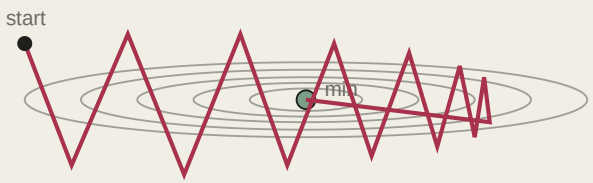
`steps ~ √κ ≈ 10 → momentum helps!`

*In deep nets κ can exceed 10^6. This is why **vanilla SGD is almost never used** — momentum is essential.*

Why vanilla SGD oscillates on ravines

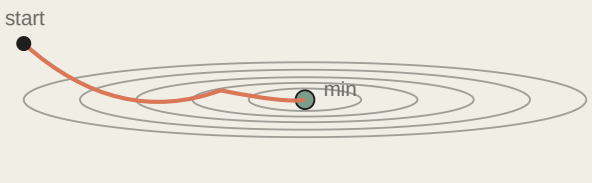
The ravine — why vanilla SGD oscillates (and momentum doesn't)

SGD · VANILLA



zig-zags across the narrow direction, crawls along the long one

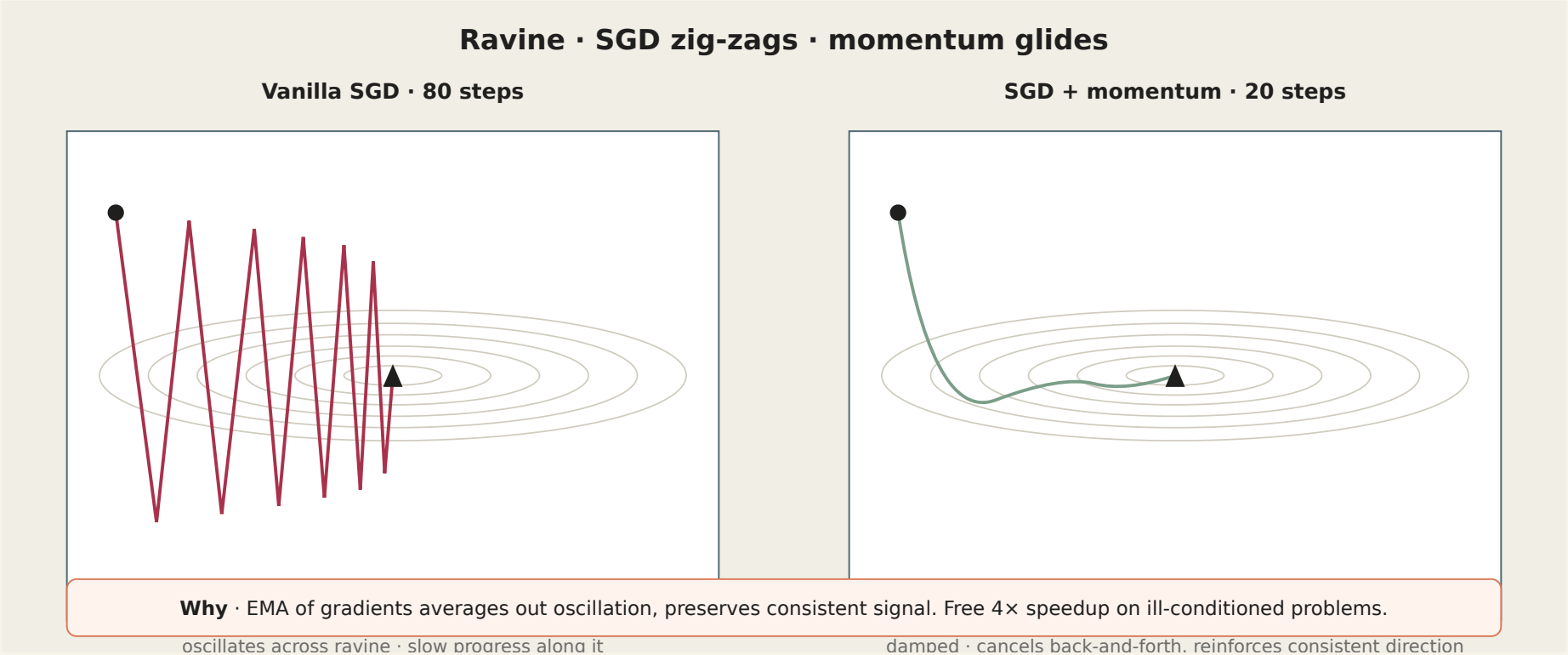
SGD + MOMENTUM



EMA cancels the back-and-forth, momentum builds along the long axis

Hessian condition number κ = λ_max / λ_min sets how much worse this gets — vanilla SGD scales with √κ, momentum with **just κ^(1/4)**.

Ravine · zig-zag vs glide



What makes a valley hard to navigate?

INTUITION

Analogy · hiking. A round bowl is easy — every direction goes downhill. A steep, narrow **canyon** is tricky. The walls are very steep, but the path forward (along the floor) is almost flat.

- Big step → slam into the canyon wall.
- Tiny step (avoiding the wall) → crawl along the floor.

The "steepness ratio" between the walls and the floor is the **condition number** κ. Large κ → narrow ravine → SGD struggles.

The condition number · let's compute it

Loss: $\mathcal{L}(\theta) = \frac{1}{2}(10\theta_1^2 + \theta_2^2)$. Hessian eigenvalues $\lambda_1 = 10,\ \lambda_2 = 1$ → **condition number** $\kappa = 10$.

1. **Gradient.** $\nabla\mathcal{L} = [10\theta_1,\ \theta_2]$.
2. **GD update.** $\theta_t = \theta_{t-1} - \eta\,\nabla\mathcal{L}(\theta_{t-1})$.
3. **Per-coordinate.**
   - $\theta_{t,1} = (1 - 10\eta)\,\theta_{t-1,1}$
   - $\theta_{t,2} = (1 - \eta)\,\theta_{t-1,2}$
4. **Stability constraint.** $|1 - 10\eta| < 1 \Rightarrow \eta < 0.2$.
   Pick the largest stable LR: $\eta = 0.15$.

- θ_1 shrinks by $|1 - 1.5| = 0.5$ — actually **flips sign** (overshoots).
- θ_2 shrinks by only 0.85 — **crawls.**

The high-curvature direction forces a tiny LR; the low-curvature direction then converges painfully slowly.

Worked numeric · SGD in a ravine

Start $\theta_0 = [10, 1]$, $\eta = 0.15$.

STEP	θ	$\nabla\mathcal{L}$	NEXT θ
0 → 1	$[10, 1]$	$[100, 1]$	$[10, 1] - 0.15[100, 1] = [-5,\ 0.85]$
1 → 2	$[-5, 0.85]$	$[-50, 0.85]$	$[-5, 0.85] - 0.15[-50, 0.85] = [2.5,\ 0.7225]$
2 → 3	$[2.5, 0.7225]$	$[25, 0.7225]$	$[2.5, 0.7225] - 0.15[25, 0.7225] = [-1.25,\ 0.6141]$

θ_1 **zig-zags wildly**: $10 \to -5 \to 2.5 \to -1.25$.
θ_2 **crawls**: $1 \to 0.85 \to 0.72 \to 0.61$.

In deep nets, κ is often 10^3–10^6. This is why momentum, adaptive LR, and normalization all help — they rescale so κ matters less.

What kinds of "flat spots" exist?

A **critical point** is anywhere $\nabla \mathcal{L} = 0$. In 1D: valley bottom (min) or hilltop (max).
In 2D and higher: a **third option** — the saddle.

INTUITION

Analogy · Pringles chip / horse saddle. At the centre, the gradient is zero. But it's *not* a minimum. Along the horse's spine, the surface curves **up**. Across its back, the surface curves **down**. Mixed curvature → **saddle point.**

To classify a critical point, look at **curvature** in every direction. The **Hessian** H stores all second derivatives; its **eigenvalues** give curvature along the principal directions.

Why saddles dominate in high dimensions

In D dimensions there are D curvature directions:

- **Local min** — all eigenvalues > 0 (curvature UP everywhere).
- **Local max** — all < 0 (curvature DOWN everywhere).
- **Saddle** — mix of $+$ and $-$.

Toy probability: assume each eigenvalue is randomly $+$ or $-$ with prob $1/2$.

$$\Pr[\text{all } D \text{ positive}] = 0.5^D$$

For $D = 10^6$ parameters: probability of a *true* local minimum is $0.5^{1{,}000{,}000}$ — essentially zero.

KEY IDEA

Almost every critical point in a deep net is a saddle, not a minimum. The challenge isn't escaping valleys — it's navigating vast, flat saddle regions. Momentum's memory saves you here: it keeps you moving in a consistent direction through the flat plateau.

Mini-batch noise is not always bad

Gradient from a batch is a *noisy estimate* of the full gradient.

- **Bad:** adds variance to each step.
- **Good:** helps escape saddle points and shallow local minima.
- **Good (more):** implicit regularization from noise is part of why SGD generalizes.

> INTUITION
>
> Larger batch → less noise → worse generalization in practice. "Linear-scaling" rule: if you 2× the batch, 2× the learning rate.

PART 2

Momentum

The single most important change to SGD

Momentum · the heavy-ball analogy

KEY IDEA

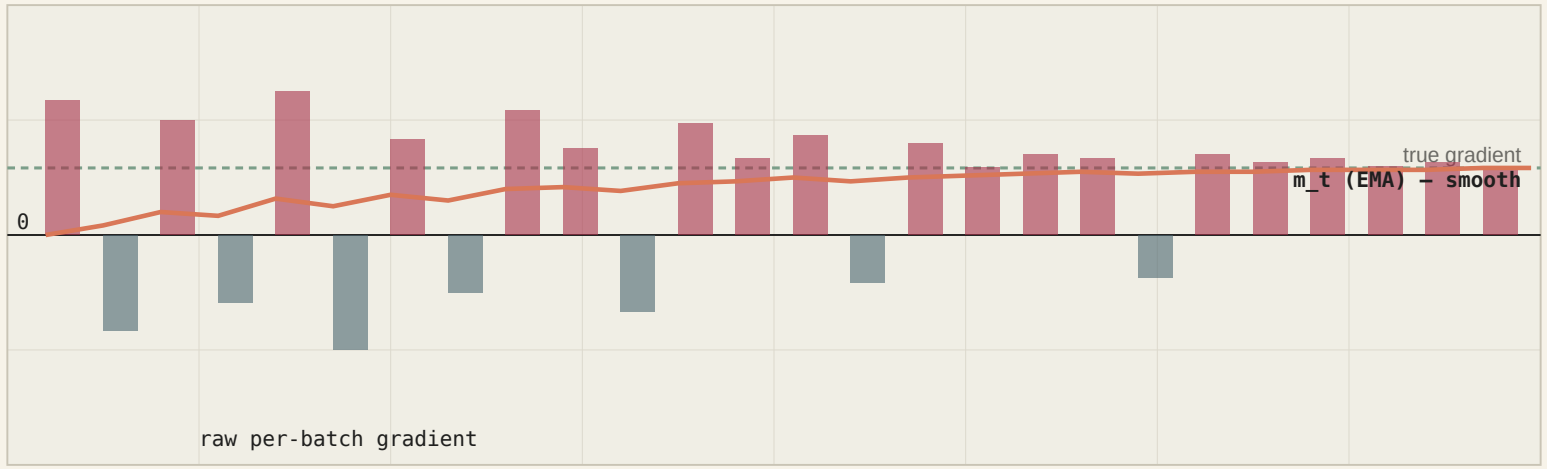
Vanilla SGD is a **short-sighted hiker** · only looks at the slope under their feet. In a narrow canyon they zig-zag wildly.

Momentum turns the hiker into a **heavy ball rolling down the hill**. The ball's inertia smooths out the zig-zags and carries it through small bumps and flat spots.

Algorithmically · keep an exponentially-weighted average of past gradients · use that as the update direction. The next slide turns this analogy into one line of math.

Momentum = EMA of gradients

Momentum = exponential moving average of gradients — averaging out noise



$m_t = \beta \cdot m_{t-1} + (1-\beta) \cdot g_t$ $\beta = 0.9$ · effective memory ≈ 10 steps

The physical intuition

Replace **position updates** with **velocity updates**. A ball rolling down the valley:

- accumulates speed in consistent directions
- averages out back-and-forth from noise

Formally — keep an exponential moving average of past gradients.

Momentum · numerical trace

Let gradients in a ravine look like: $g_t = [\pm 1, 0.1]$ (flipping sign on θ_1 every step, small consistent push on θ_2).

DERIVATION

With $\beta = 0.9$, EMA settles to:

- $v_{t,1}$ · average of ± 1 oscillations → **near zero**
- $v_{t,2}$ · average of 0.1 → **0.1** (preserved)

Vanilla SGD zig-zags ± 1 on θ_1. Momentum cancels that out — direction 2 gets all the step budget. **Zig-zag in, drift out.**

The same EMA mechanism shows up in Adam (L5), batch-norm running stats, and target networks in RL. One primitive, many uses.

Momentum = one more hyperparameter?

Yes — but a forgiving one.

β	EFFECTIVE MEMORY	BEHAVIOR
0.5	2 steps	barely smooths
0.9	10 steps	**sensible default**
0.95	20 steps	slower to respond to curvature changes
0.99	100 steps	heavy; needs gradient clipping

INTUITION

Most practitioners set $\beta = 0.9$ once and never touch it again. The knob you tune is η.

From hiker to heavy ball

Vanilla SGD only cares about the slope **right now**. A heavy ball has **inertia** — its motion today is a mix of *where it was already going* and the *new push from the slope*.

INTUITION

Analogy · pushing a bowling ball. Push it once → it rolls. Push it again in the same direction → it speeds up. Push it sideways → it changes direction, but it doesn't stop and turn on a dime. This **memory of past motion** is what we add to SGD.

Momentum · build the update step-by-step

1. **Define velocity $\mathbf{v}$** — a vector that remembers past gradients.
2. **Velocity update.** Mix old velocity with the new gradient using $\beta \in (0, 1)$:

$$\mathbf{v}_t = \underbrace{\beta\, \mathbf{v}_{t-1}}_{\text{inertia · keep most of old velocity}} + \underbrace{(1-\beta)\, \nabla \mathcal{L}(\theta_{t-1})}_{\text{new info · nudge with current gradient}}$$

3. **Position update.** Step using the smoothed velocity, not the raw gradient:

$$\theta_t = \theta_{t-1} - \eta\, \mathbf{v}_t$$

This is an **Exponential Moving Average (EMA)**. With $\beta = 0.9$: keep 90% of the old velocity, mix in 10% of the new gradient. Effective memory $\frac{1}{1-\beta} = 10$ steps.

PyTorch's `SGD(..., momentum=0.9)` uses an equivalent form.

Worked numeric · momentum smooths the ravine

Ravine gradients: $g_t = [\pm 1.0,\ 0.1]$ — first component flips sign every step, second is constant.
$\beta = 0.9,\ \mathbf{v}_0 = [0, 0]$.

t	g_t	$\mathbf{v}_t = 0.9\,\mathbf{v}_{t-1} + 0.1\,g_t$
1	$[-1.0, 0.1]$	$[0,0]\cdot 0.9 + [-1, 0.1]\cdot 0.1 = [-0.100,\ 0.0100]$
2	$[+1.0, 0.1]$	$[-0.090, 0.009] + [0.1, 0.01] = [+0.010,\ 0.0190]$
3	$[-1.0, 0.1]$	$[0.009, 0.0171] + [-0.1, 0.01] = [-0.091,\ 0.0271]$

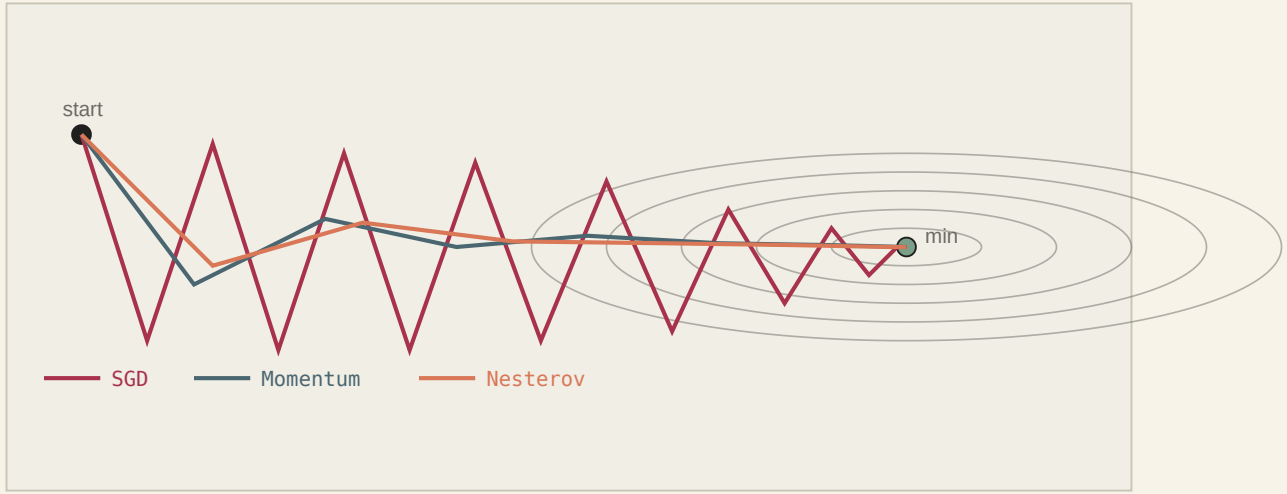
Observation.

- $v_{t,1}$ oscillates near zero ($-0.10 \to 0.01 \to -0.09$): zig-zags **cancel out**.
- $v_{t,2}$ steadily grows ($0.010 \to 0.019 \to 0.027$): consistent push **accumulates**.

Momentum **damps oscillation, amplifies persistence**.

What momentum fixes

The SGD family — three trajectories on the same landscape



Same landscape, same LR. SGD crawls; momentum accelerates; Nesterov turns faster near the bowl.

IN PRACTICE

Interactive: race SGD, momentum, Adam on a 2D quadratic — optimizer-race.

A concrete example · MNIST

Same model, same LR, after 10 epochs:

OPTIMIZER	TRAIN LOSS	VAL ACCURACY
SGD	0.42	92.1%
SGD + momentum	0.13	97.6%

Momentum is a **free 5 points** on MNIST. The single highest-value change to SGD.

Momentum in PyTorch · one line

```
# vanilla SGD
opt = torch.optim.SGD(model.parameters(), lr=0.01)

# SGD + momentum  — the sensible default
opt = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

IN PRACTICE

Never use vanilla SGD without momentum for a deep network. It is a Lego brick, not a finished optimizer.

PART 3

Nesterov accelerated gradient

Evaluate the gradient one step ahead

Nesterov · driving with a longer-range view

KEY IDEA

Standard momentum is like driving by looking at the road right in front of your bumper · you steer based on what's directly under you.

Nesterov is like looking *down the road*. You first imagine where momentum is taking you, look at the slope **at that future point**, and steer based on that.

The result · less overshoot near valley walls · cleaner approach to the minimum · provably better convergence rate (in the convex case).

Classical vs Nesterov

A smarter heavy ball

Standard momentum is a bit reckless. It computes the gradient **at the current spot**, then commits to a big velocity-driven step. It's like a driver looking only at the road **right under the car**.

INTUITION

Analogy · Nesterov is a smarter driver who looks ahead.

1. First, make a "guess" move based only on old velocity → land at a **lookahead point**.
2. From the lookahead, compute the gradient.
3. Use **that** gradient (the slope where you're about to be) to make the actual step.

If your velocity was about to drive you into a wall, the lookahead gradient already points back — correcting the course **before** you fully commit.

Nesterov · the update in three steps

1. **Project a lookahead point** — where would old velocity take us?

$$\theta_{\text{look}} = \theta_{t-1} - \eta\beta\,\mathbf{v}_{t-1}$$

2. **Compute gradient at the lookahead** (not at θ_{t-1}):

$$\mathbf{g}_{\text{look}} = \nabla\mathcal{L}(\theta_{\text{look}})$$

3. **Standard momentum update** with the smarter gradient:

$$\mathbf{v}_t = \beta\,\mathbf{v}_{t-1} + (1-\beta)\,\mathbf{g}_{\text{look}} \qquad \theta_t = \theta_{t-1} - \eta\,\mathbf{v}_t$$

One change: *where* the gradient is measured.

Worked numeric · Nesterov's correction

1D toy: $\mathcal{L}(\theta) = \theta^2,\ \nabla\mathcal{L} = 2\theta$. Start $\theta_0 = 2,\ v_0 = 2,\ \eta = 0.1,\ \beta = 0.9$.

Standard momentum

- $g_0 = 2 \cdot 2 = 4$
- $v_1 = 0.9 \cdot 2 + 0.1 \cdot 4 = 2.20$
- $\theta_1 = 2 - 0.1 \cdot 2.20 = \mathbf{1.780}$

Nesterov

- $\theta_{\text{look}} = 2 - 0.1 \cdot 0.9 \cdot 2 = 1.82$
- $g_{\text{look}} = 2 \cdot 1.82 = 3.64$ (less steep!)
- $v_1 = 0.9 \cdot 2 + 0.1 \cdot 3.64 = 2.164$
- $\theta_1 = 2 - 0.1 \cdot 2.164 = \mathbf{1.7836}$

The Nesterov gradient was smaller — it "saw" the valley flattening ahead → slightly more conservative step → less overshoot. Tiny difference per step, but compounds over training.

Why it helps

If the lookahead overshoots, the gradient at that point **points back** — correction kicks in *before* you commit to the full step.

Less overshoot near curvature changes, slightly faster convergence.

> DERIVATION
>
> **Theoretical payoff** (convex, smooth case): optimal convergence rate $O(1/t^2)$ vs GD's $O(1/t)$.

In practice on deep nets, the gain is modest but free.

A geometric way to see Nesterov

Classical momentum

1. Take step $\eta\beta v_{t-1}$ (momentum part)
2. Add a correction based on gradient *at start*

If the gradient at the start was misleading, you've committed before knowing.

Nesterov

1. Tentatively move $\eta\beta v_{t-1}$ to a *lookahead* point
2. Measure the gradient *there*
3. Use that gradient for the real step

Information from the landscape you're about to visit, not the one you're leaving.

That's it · the same update, just measured at a smarter location.

Nesterov in PyTorch · one flag

```
opt = torch.optim.SGD(model.parameters(),
                      lr=0.01,
                      momentum=0.9,
                      nesterov=True)    # ← the flag
```

That is the entire cost of using it.

PART 4

Practical recommendations

What to actually use

Current 2026 practice

USE-CASE	OPTIMIZER	REASON
CNN from scratch	**SGD + momentum + Nesterov**	best test accuracy on vision
Fine-tuning anything	AdamW (L5)	robust across LR
Transformer training	AdamW + warmup + cosine	field default
Debugging a new model	Adam first	faster to iterate

INTUITION

Image researchers often prefer SGD-Momentum for final runs — it tends to find flatter minima that generalize slightly better. Everyone else uses AdamW.

A common mistake

WATCH OUT

Q. Student sets `momentum=0.99` because "more is better." Loss diverges. Why?

Higher β = longer memory. If curvature changes abruptly (early training), a heavy memory keeps you pushing in stale directions *after* the gradient has reversed → overshoot.

Rule of thumb: keep $\beta \in [0.9, 0.95]$. Go higher only with strong gradient clipping + warmup.

Momentum's compounding effect

What happens when the gradient points the same way, step after step?

INTUITION

Analogy · pushing a child on a swing. First push — they get some velocity. Next time they swing by — push again. The new push **adds to the existing velocity**. They go higher and higher. Momentum does the same: each consistent gradient builds the velocity, leading to far bigger steps than the gradient alone.

Let's derive **how big** that compounding gets.

Deriving the effective learning rate

Assume gradient $\mathbf{g}$ is constant for many steps. Use the simpler form $\mathbf{v}_t = \beta\,\mathbf{v}_{t-1} + \mathbf{g}$ (same steady-state behaviour). Unroll, with $\mathbf{v}_0 = 0$:

- $t = 1$: $\mathbf{v}_1 = \mathbf{g}$
- $t = 2$: $\mathbf{v}_2 = \beta\,\mathbf{g} + \mathbf{g}$
- $t = 3$: $\mathbf{v}_3 = \beta^2\,\mathbf{g} + \beta\,\mathbf{g} + \mathbf{g}$
- $t = \infty$: $\mathbf{v}_\infty = (1 + \beta + \beta^2 + \cdots)\,\mathbf{g}$

  A **geometric series** with $\beta < 1$:

$$\sum_{i=0}^{\infty} \beta^i = \frac{1}{1-\beta} \quad \Rightarrow \quad \mathbf{v}_\infty = \frac{1}{1-\beta}\,\mathbf{g}$$

So the parameter update at terminal velocity is:

$$\theta_t = \theta_{t-1} - \eta\,\mathbf{v}_\infty = \theta_{t-1} - \underbrace{\frac{\eta}{1-\beta}}_{\eta_{\text{eff}}}\,\mathbf{g}$$

The "effective LR" in numbers

How much does β amplify η?

β	$1/(1-\beta)$	EFFECT
0.0	$1\times$	no momentum
0.5	$2\times$	light
0.9	$\mathbf{10}\times$	**standard default**
0.95	$20\times$	heavy
0.99	$100\times$	very heavy

WATCH OUT

Key takeaway. A small bump from $\beta = 0.9$ to $\beta = 0.99$ multiplies your effective LR by $10\times$. Your previously-stable run will diverge. **When you raise momentum, lower η to compensate.**

IN PRACTICE

Practical recipe: fix $\beta = 0.9$; use the LR finder to pick η. Revisit β only if training is unstable.

Debugging optimizer failures

Common symptoms and fixes:

SYMPTOM	LIKELY CAUSE	FIX
Loss → NaN after step 1	LR too high, fp16 overflow	halve η, enable gradient clipping
Loss oscillates (±)	ravine + momentum too low	raise β to 0.9
Loss plateaus for hundreds of steps	stuck near saddle	raise β or switch to Adam
Loss drops then climbs	overfitting (not optimizer)	add weight decay, lower η
Training slower than Keras example	no momentum	add `momentum=0.9`

INTUITION

Most "my network doesn't train" bugs are optimizer-level. The debug ladder from L3 + this table catches ~90% of them in practice.

Putting it all together · the L04 master sentence

DERIVATION

Vanilla SGD is steepest descent · momentum is a low-pass filter on gradients · Nesterov is momentum that peeks ahead. All three live on the same loss landscape · they differ only in *how they smooth the gradient signal across steps*.

UPDATE	EQUATION	CURES
SGD	$\theta_{t+1} = \theta_t - \eta\, g_t$	nothing — pure 1st-order
Momentum	$v_{t+1} = \beta v_t + g_t,\, \theta_{t+1} = \theta_t - \eta\, v_{t+1}$	ravine zig-zag, saddles
Nesterov	g_t evaluated at $\theta_t - \eta\beta v_t$	overshoot near a minimum

Effective LR under momentum · $\eta_{\text{eff}} = \eta/(1-\beta)$. Raising β from 0.9 to 0.99 multiplies it by 10× — the most common cause of "I bumped momentum and everything diverged."

Pop quiz · revisit

The 50-layer ResNet that oscillated forever? The answer is **(b) ravine + step too large**.

KEY IDEA

Vanishing gradients (a) would *flatten* loss, not oscillate. Bad shuffling (c) would show step-to-step jitter, not a stable cycle. Saddles (d) plateau, not oscillate.

Fix · raise β to 0.9 (momentum smooths the cycle), or halve η.

Practice problems

DERIVATION

P1. A quadratic $L = \frac{1}{2}(\lambda_1 x_1^2 + \lambda_2 x_2^2)$ has $\lambda_1 = 100, \lambda_2 = 1$. Compute the largest stable LR for SGD. Why does this LR make the x_2 direction crawl?

P2. Show that the momentum recurrence $v_{t+1} = \beta v_t + g_t$ with constant g converges to $v_\infty = g/(1-\beta)$. Use this to derive the effective LR.

P3. Explain in one sentence why momentum helps escape **saddles** but does **not** help escape **strict local minima**.

P4. Run two trajectories on a Rosenbrock-like ravine · SGD with $\eta = 0.01$ vs SGD-momentum with $\eta = 0.01, \beta = 0.9$. Predict which oscillates more and why.

P5. Show that Nesterov's update can be rewritten as classical momentum *plus* a gradient correction term. State the correction.

P6. A practitioner raises momentum from 0.9 to 0.99 and the run NaNs. Without changing β, what one change rescues the run?

Summary · Lecture 4 — summary

- **Loss landscapes** — ravines, saddles, ill-conditioning. High-dim is mostly saddles.
- **Vanilla SGD** serves one direction at a time → oscillates across narrow valleys.
- **Momentum** = EMA of gradients; damps zig-zag, reinforces consistent directions.
- **Nesterov** evaluates the gradient at the lookahead point — a free small speed-up.
- **In practice** · SGD+momentum(+Nesterov) for vision, AdamW for everything else.

Read before Lecture 5

Prince — Ch 6 §6.4–6.6. Free at udlbook.github.io.

Next lecture

Adam, AdamW, and learning-rate schedules — per-parameter adaptive LR, bias correction, decoupled weight decay, warmup + cosine.

NOTEBOOK

Notebook 4 · `04-optimizer-race.ipynb` — implement SGD, momentum, Nesterov from scratch; animate trajectories on a 2D quadratic and Rosenbrock.