10-701: Introduction to Machine Learning Lecture 6 – Optimization for ML

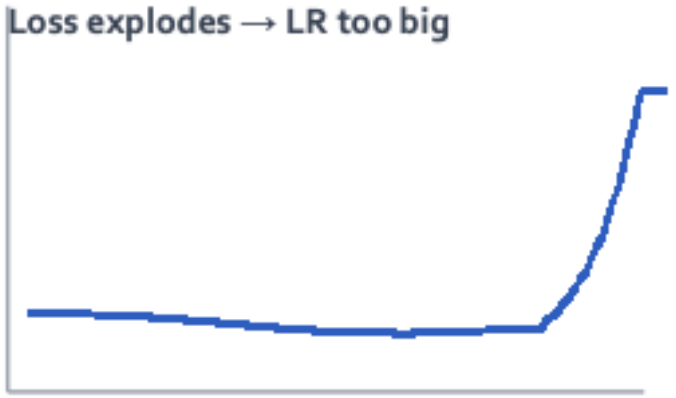
Pradeep Ravikumar

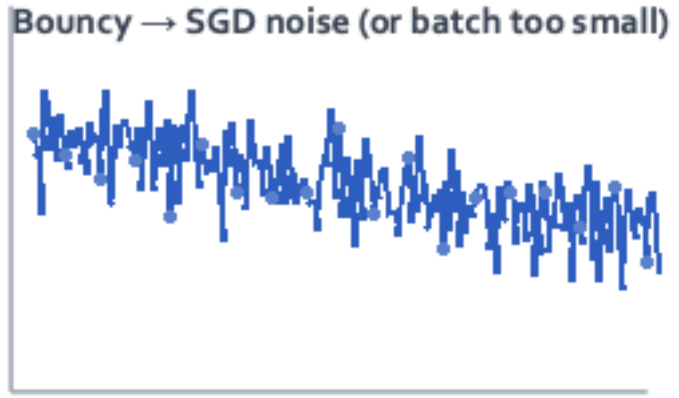
Spring 2026

- Announcements:
  - HW2 has been released and is due on Tue, Feb. 10
- Additional Readings:
  - Bottou, Stochastic Gradient Descent Tricks
  - Goodfellow, Bengio, Courville, Deep Learning, optimization chapter

Optimization is mostly: reading your loss curve

We'll learn to diagnose + fix the usual training dramas.



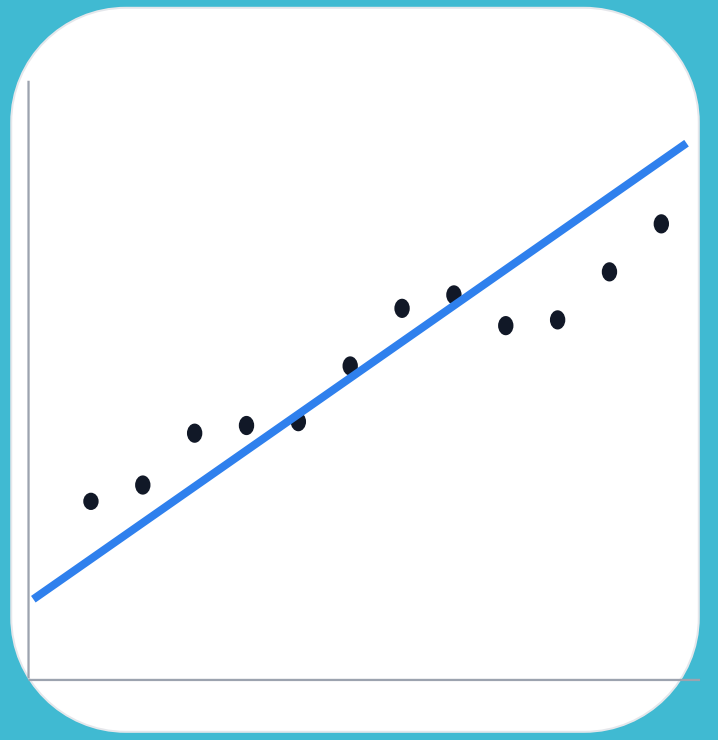


Today's goal: turn "my model is sad 😭" into a 3-step fix.

We'll use one running example (linear regression) to understand
GD → SGD → Momentum → Adam/AdamW

Running example: linear regression learns a line

Same objective, different optimizers. Let's watch them behave.



1. Define a model and model parameters
   1. Assume $y = \boldsymbol{w}^T \boldsymbol{x}$
   2. Parameters: $\boldsymbol{w} = [w_0, w_1, \ldots, w_D]$
2. Write down an objective function
   1. Minimize the mean squared error

$$\ell_{\mathcal{D}}(\boldsymbol{w}) = \frac{1}{N} \sum_{n=1}^{N} \left(\boldsymbol{w}^T \boldsymbol{x}^{(n)} - y^{(n)} \right)^2$$

3. Optimize the objective w.r.t. the model parameters

Minimizing the Squared Error

$$\ell_{\mathcal{D}}(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N}\left(\boldsymbol{w}^T\boldsymbol{x}^{(n)} - y^{(n)}\right)^2 = \frac{1}{N}\sum_{n=1}^{N}\left(\boldsymbol{x}^{(n)^T}\boldsymbol{w} - y^{(n)}\right)^2$$

$$= \frac{1}{N}\|X\boldsymbol{w} - \boldsymbol{y}\|_2^2 \text{ where } \|\boldsymbol{z}\|_2 = \sqrt{\sum_{d=1}^{D} z_d^2} = \sqrt{\boldsymbol{z}^T\boldsymbol{z}}$$

$$= \frac{1}{N}(X\boldsymbol{w} - \boldsymbol{y})^T(X\boldsymbol{w} - \boldsymbol{y})$$

$$= \frac{1}{N}(\boldsymbol{w}^T X^T X\boldsymbol{w} - 2\boldsymbol{w}^T X^T\boldsymbol{y} + \boldsymbol{y}^T\boldsymbol{y})$$

$$\nabla_{\boldsymbol{w}}\ell_{\mathcal{D}}(\widehat{\boldsymbol{w}}) = \frac{1}{N}(2X^T X\widehat{\boldsymbol{w}} - 2X^T\boldsymbol{y}) = 0$$

$$\rightarrow X^T X\widehat{\boldsymbol{w}} = X^T\boldsymbol{y}$$

$$\rightarrow \widehat{\boldsymbol{w}} = (X^T X)^{-1}X^T\boldsymbol{y}$$

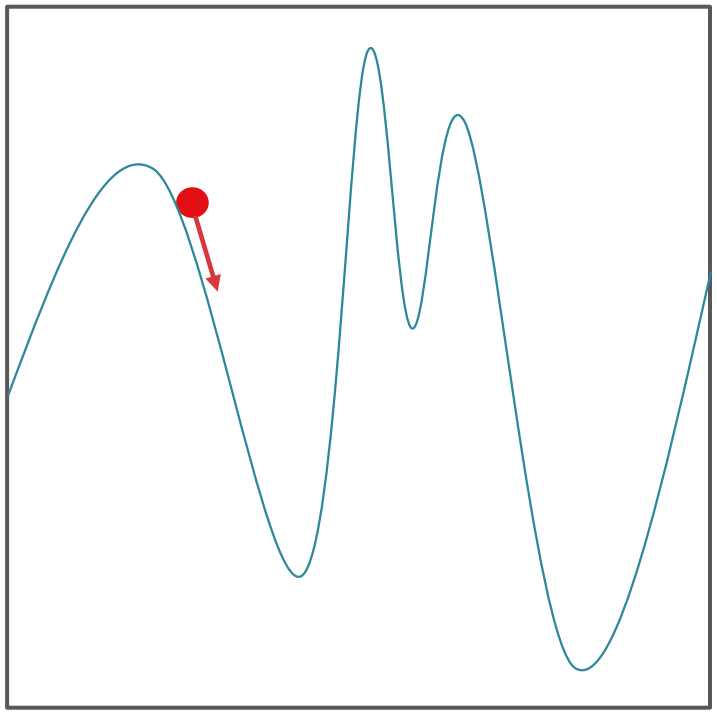
Closed Form Solution

$$\hat{\boldsymbol{w}} = (X^T X)^{-1} X^T \boldsymbol{y}$$

1. Is $X^T X$ invertible?
   - When $N \gg D + 1$, $X^T X$ is (almost always) full rank and therefore, invertible
   - If $X^T X$ is not invertible (occurs when one of the features is a linear combination of the others) then there are infinitely many solutions.
2. If so, how computationally expensive is inverting $X^T X$?
   - $X^T X \in \mathbb{R}^{D+1 \times D+1}$ so inverting $X^T X$ takes $O(D^3)$ time…
     - Computing $X^T X$ takes $O(ND^2)$ time
   - What alternative optimization method can we use to minimize the mean squared error?

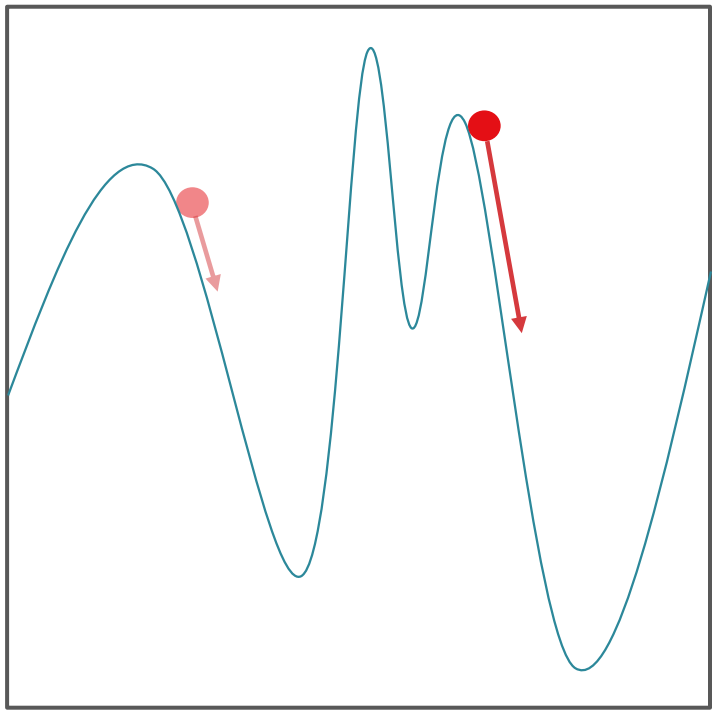
Gradient Descent: Intuition

- An iterative method for minimizing functions
- Requires the gradient to exist everywhere



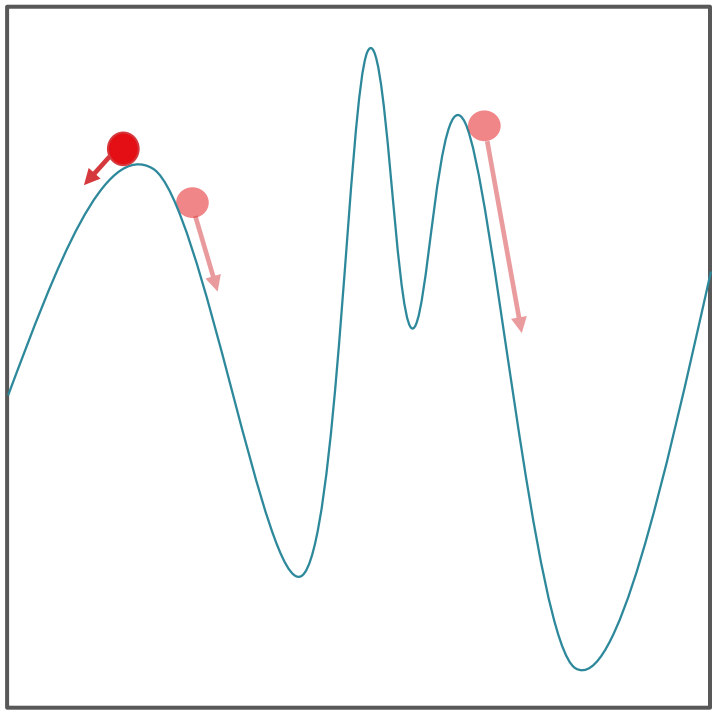
Gradient Descent: Intuition

- An iterative method for minimizing functions
- Requires the gradient to exist everywhere



Gradient Descent: Intuition

- An iterative method for minimizing functions
- Requires the gradient to exist everywhere



Gradient Descent

- Suppose the current weight vector is $\boldsymbol{w}^{(t)}$
- Move some distance, η, in the "most downhill" direction, $\hat{\boldsymbol{v}}$:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \eta\hat{\boldsymbol{v}}$$

Gradient Descent: Step Direction

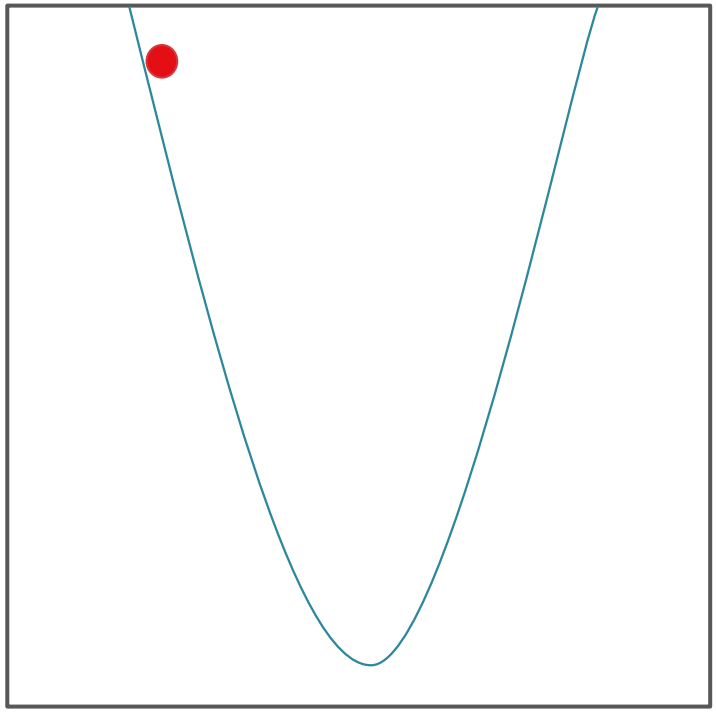
- Suppose the current weight vector is $\boldsymbol{w}^{(t)}$
- Move some distance, η, in the "most downhill" direction, $\widehat{\boldsymbol{v}}$:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \eta\widehat{\boldsymbol{v}}$$

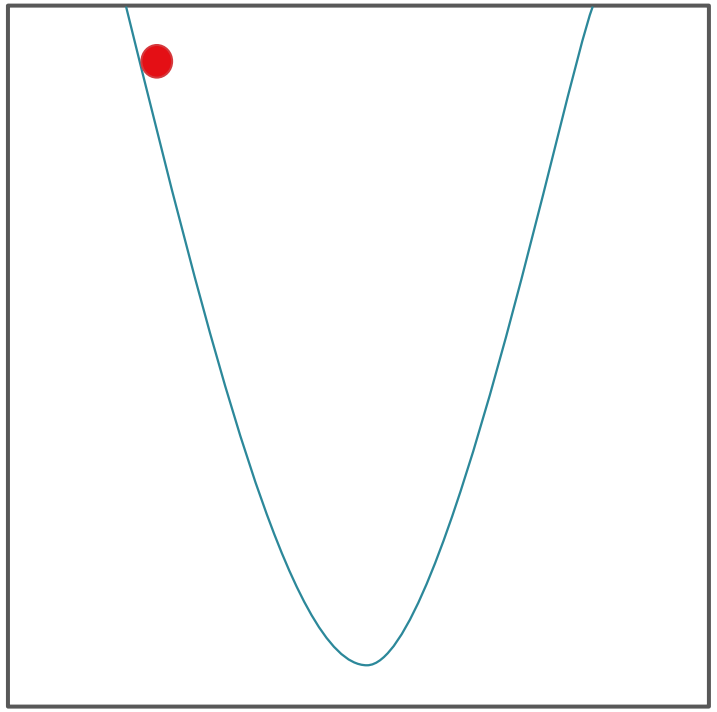
- The gradient points in the direction of steepest *increase* …
- … so $\widehat{\boldsymbol{v}}$ should point in the opposite direction:

$$\widehat{\boldsymbol{v}}^{(t)} = -\frac{\nabla_{\boldsymbol{w}}\ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)}{\left\|\nabla_{\boldsymbol{w}}\ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)\right\|}$$

Gradient Descent: Step Size

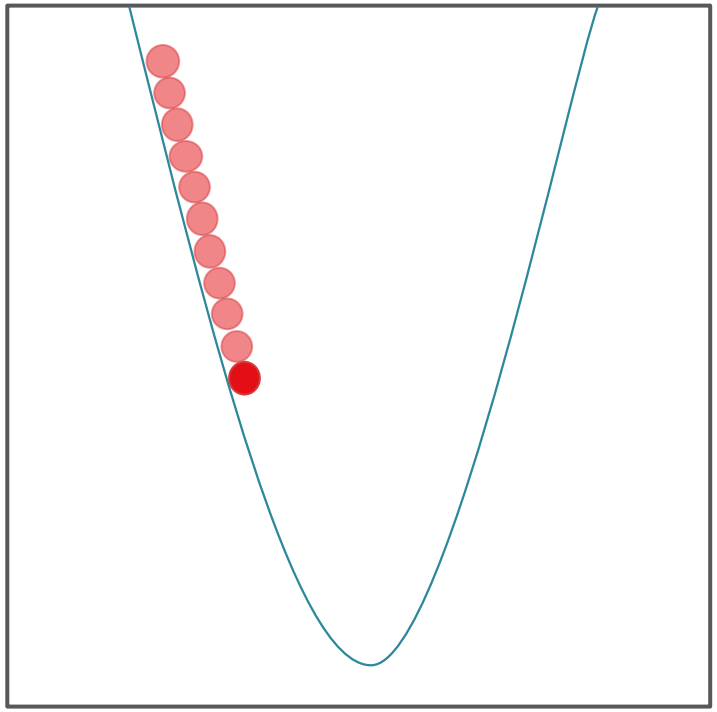


Small η

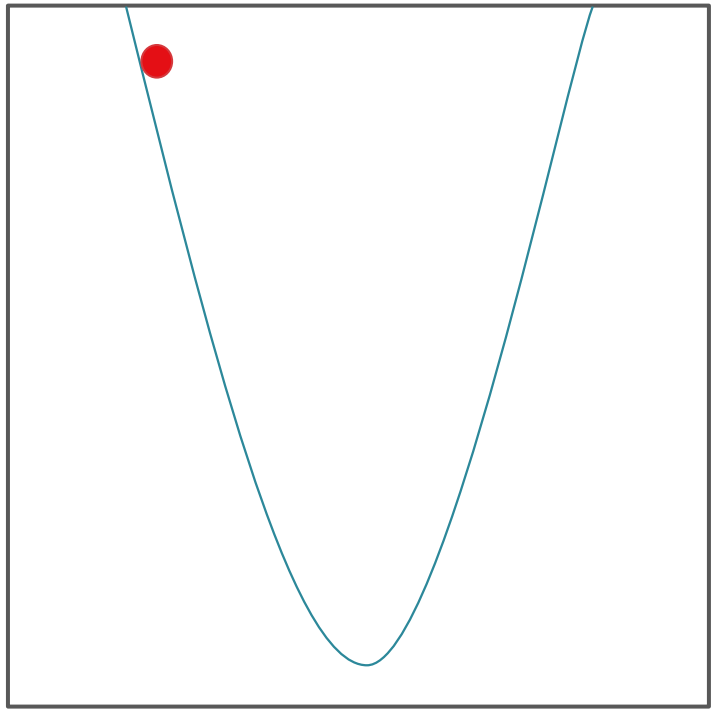


Large η

Gradient Descent: Step Size

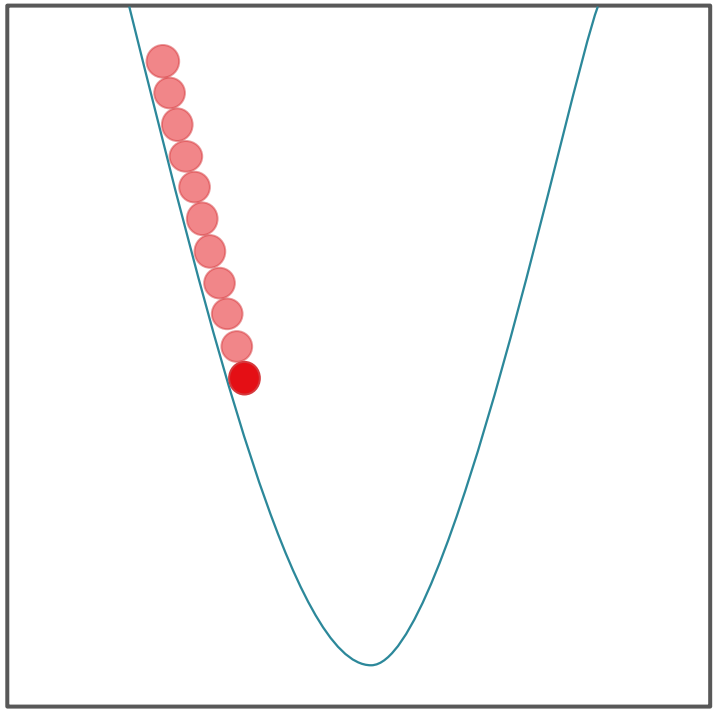


Small η

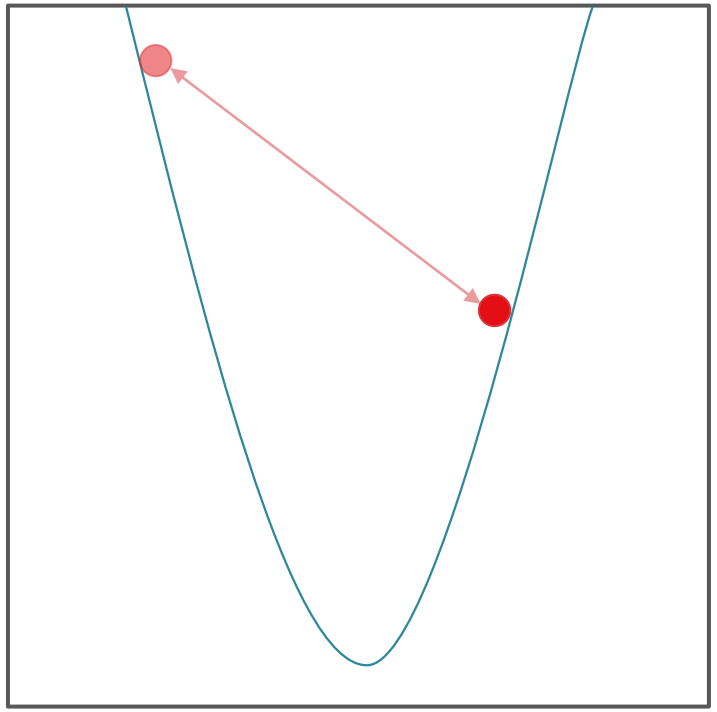


Large η

Gradient Descent: Step Size



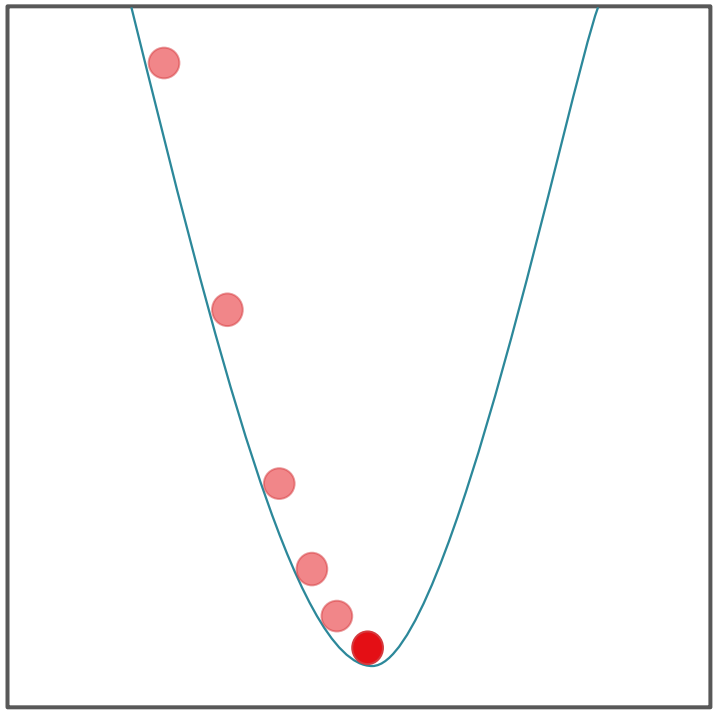
Small η



Large η

Gradient Descent: Step Size

- Use a variable $\eta^{(t)}$ instead of a fixed η!



- Set $\eta^{(t)} = \eta^{(0)} \left\| \nabla_{\boldsymbol{w}} \ell_{\mathcal{D}} \left(\boldsymbol{w}^{(t)} \right) \right\|$
- $\left\| \nabla_{\boldsymbol{w}} \ell_{\mathcal{D}} \left(\boldsymbol{w}^{(t)} \right) \right\|$ decreases as $\ell_{\mathcal{D}}$ approaches its minimum
  $\rightarrow \eta^{(t)}$ (hopefully) decreases over time

Gradient Descent

- $\widehat{\boldsymbol{v}}^{(t)} = -\frac{\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)}{\left\|\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)\right\|}$

- $\eta^{(t)} = \eta^{(0)}\left\|\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)\right\|$

- $$\begin{aligned}\boldsymbol{w}^{(t+1)} &= \boldsymbol{w}^{(t)} + \eta^{(t)}\widehat{\boldsymbol{v}}^{(t)} \\ &= \boldsymbol{w}^{(t)} + \left(\eta^{(0)}\left\|\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)\right\|\right)\left(-\frac{\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)}{\left\|\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)\right\|}\right) \\ &= \boldsymbol{w}^{(t)} - \eta^{(0)}\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)\end{aligned}$$

Gradient Descent

- Input: $\mathcal{D} = \left\{\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}, \eta$

1. Initialize $\boldsymbol{w}^{(0)}$ to all zeros and set $t = 0$
2. While TERMINATION CRITERION is not satisfied
   a. Compute the gradient:
      $\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}} \left(\boldsymbol{w}^{(t)}\right)$
   b. Update $\boldsymbol{w}$: $\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \eta \nabla_{\boldsymbol{w}} \ell_{\mathcal{D}} \left(\boldsymbol{w}^{(t)}\right)$
   c. Increment t: $t \leftarrow t + 1$

- Output: $\boldsymbol{w}^{(t)}$

Gradient Descent

- Input: $\mathcal{D} = \left\{\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}, \eta, \epsilon$

1. Initialize $\boldsymbol{w}^{(0)}$ to all zeros and set $t = 0$
2. While $\left\|\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)\right\| > \epsilon$
   a. Compute the gradient: $\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)$
   b. Update $\boldsymbol{w}$: $\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \eta \nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)$
   c. Increment t: $t \leftarrow t + 1$

- Output: $\boldsymbol{w}^{(t)}$

Gradient Descent

- Input: $\mathcal{D} = \left\{\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}, \eta, T$

1. Initialize $\boldsymbol{w}^{(0)}$ to all zeros and set $t = 0$
2. While $t < T$
   a. Compute the gradient:
      $\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)$
   b. Update $\boldsymbol{w}$: $\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \eta \nabla_{\boldsymbol{w}} \ell_{\mathcal{D}}\left(\boldsymbol{w}^{(t)}\right)$
   c. Increment t: $t \leftarrow t + 1$

- Output: $\boldsymbol{w}^{(t)}$

Why Gradient Descent for linear regression?

- Input: $\mathcal{D} = \left\{\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)\right\}_{i=1}^{N}, \eta, T$

1. Initialize $\boldsymbol{w}^{(0)}$ to all zeros and set $t = 0$
2. While TERMINATION CRITERION is not satisfied
   a. Compute the gradient:
      $\nabla_{\boldsymbol{w}} \ell_{\mathcal{D}} \left(\boldsymbol{w}^{(t)}\right)$
   b. Update $\boldsymbol{w}$: $\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \eta \nabla_{\boldsymbol{w}} \ell_{\mathcal{D}} \left(\boldsymbol{w}^{(t)}\right)$
   c. Increment t: $t \leftarrow t + 1$

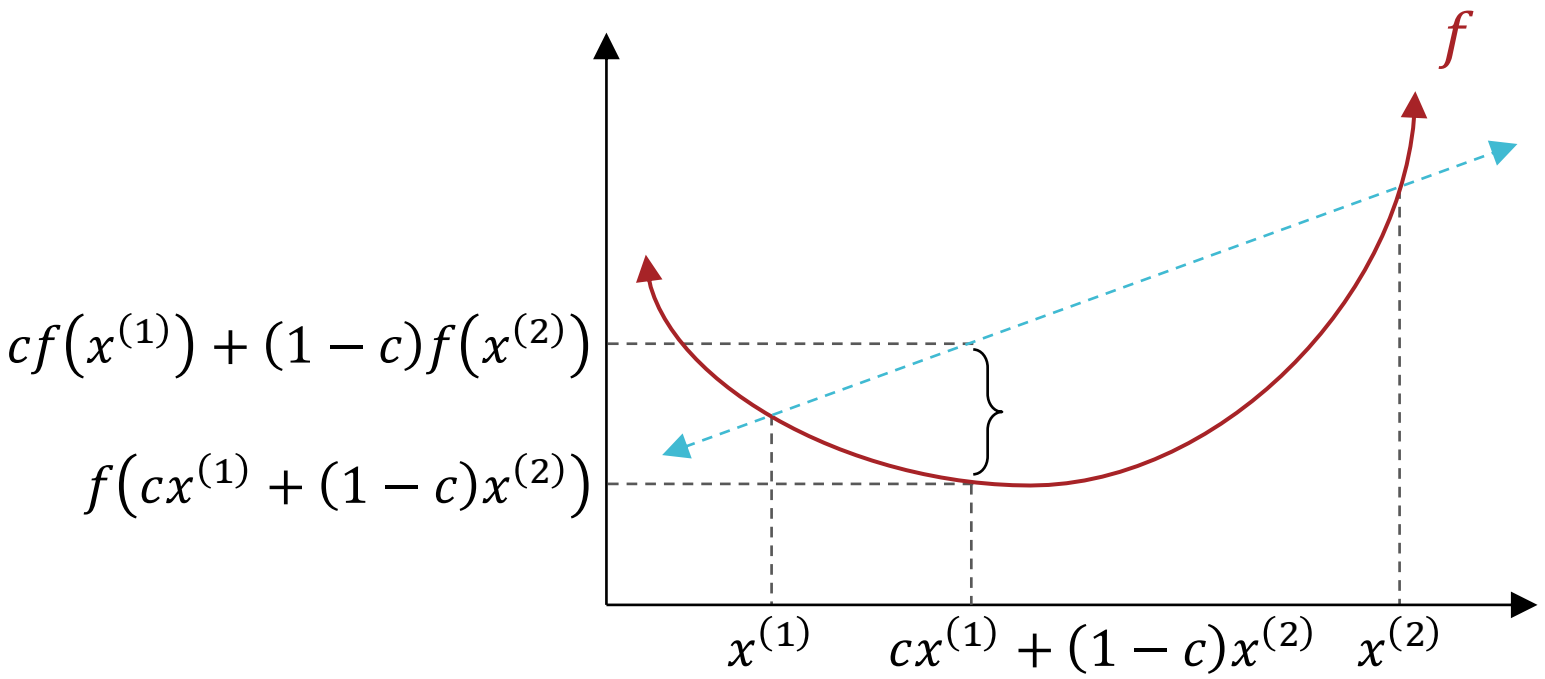
- Output: $\boldsymbol{w}^{(t)}$

Convexity

- A function $f: \mathbb{R}^D \rightarrow \mathbb{R}$ is convex if

  $\forall \boldsymbol{x}^{(1)} \in \mathbb{R}^D, \boldsymbol{x}^{(2)} \in \mathbb{R}^D$ and $0 \leq c \leq 1$

$$f\left(c\boldsymbol{x}^{(1)} + (1-c)\boldsymbol{x}^{(2)}\right) \leq cf\left(\boldsymbol{x}^{(1)}\right) + (1-c)f\left(\boldsymbol{x}^{(2)}\right)$$

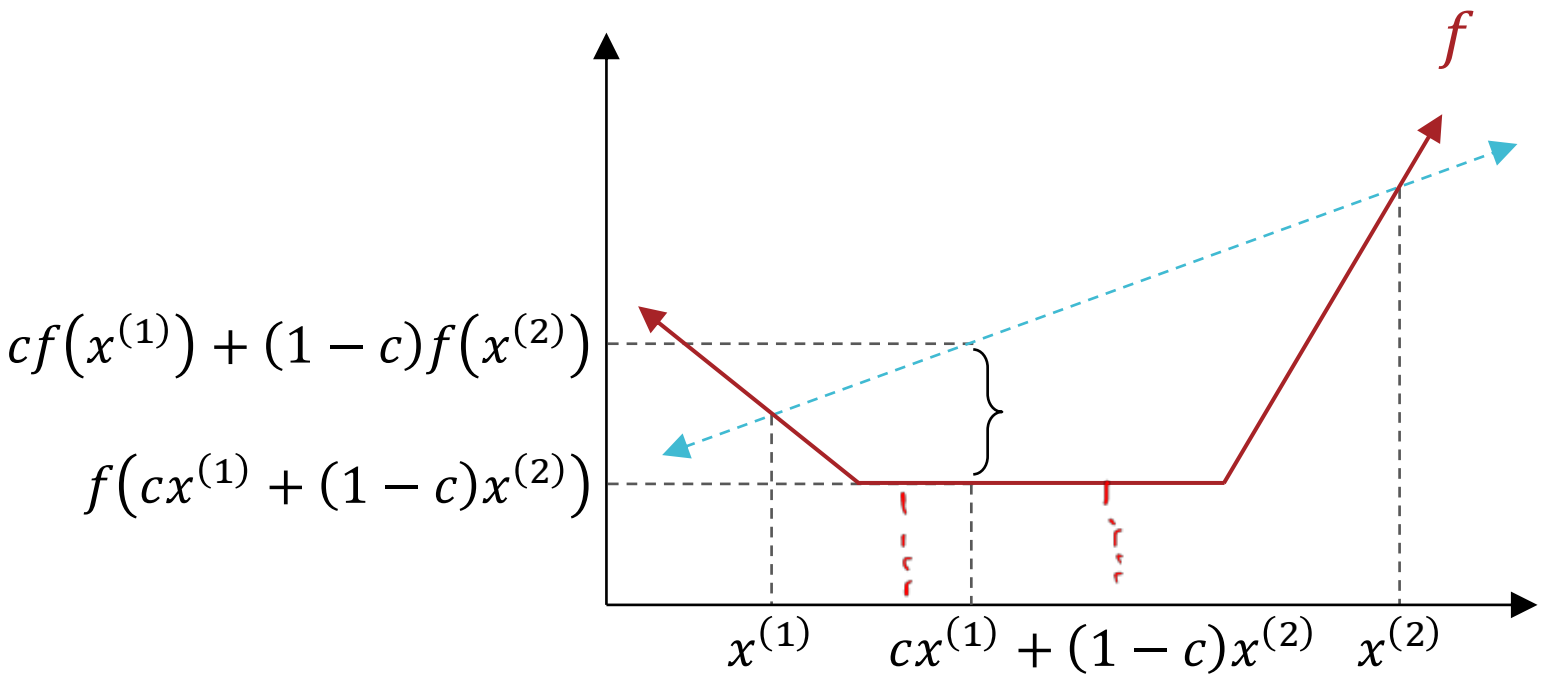


Convexity

- A function $f: \mathbb{R}^D \rightarrow \mathbb{R}$ is convex if

  $\forall\, \boldsymbol{x}^{(1)} \in \mathbb{R}^D, \boldsymbol{x}^{(2)} \in \mathbb{R}^D$ and $0 \leq c \leq 1$

$$f\left(c\boldsymbol{x}^{(1)} + (1-c)\boldsymbol{x}^{(2)}\right) \leq cf\left(\boldsymbol{x}^{(1)}\right) + (1-c)f\left(\boldsymbol{x}^{(2)}\right)$$

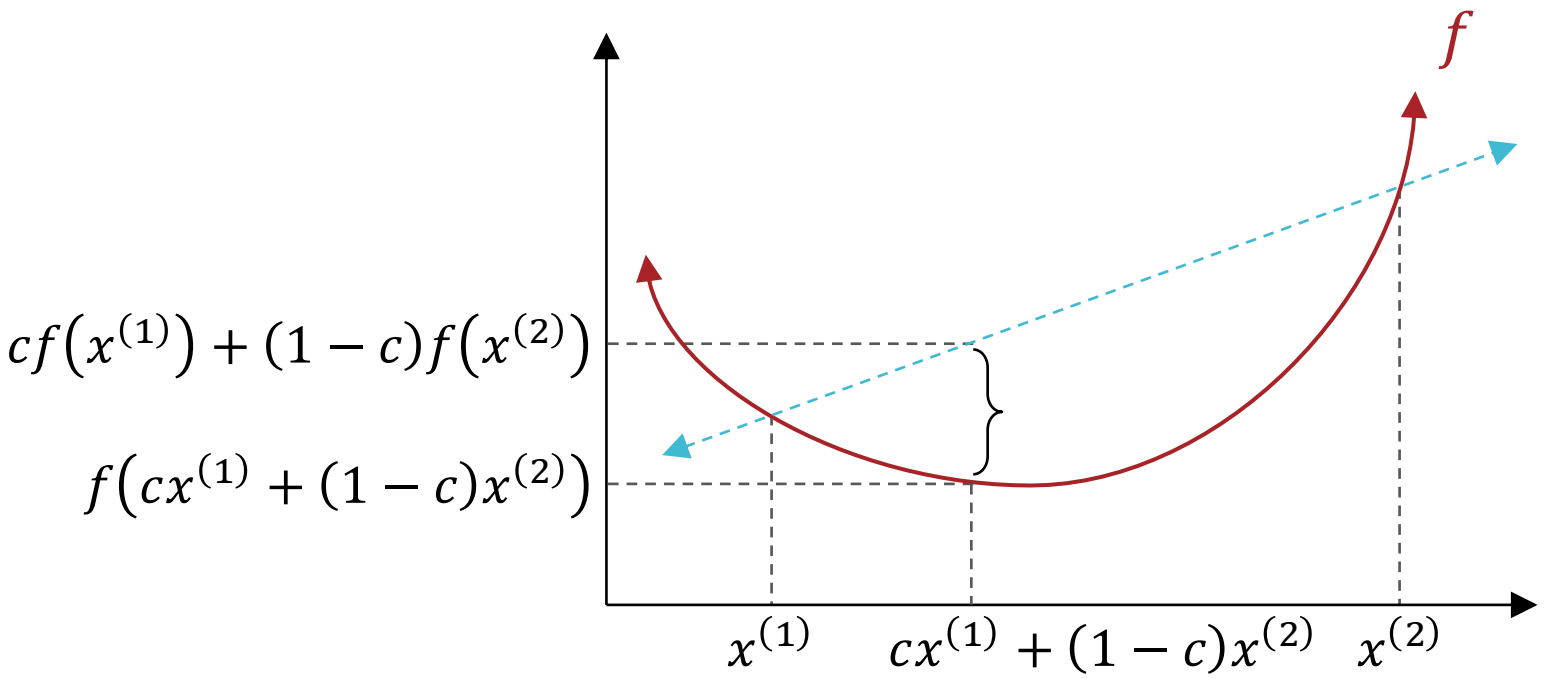


Convexity

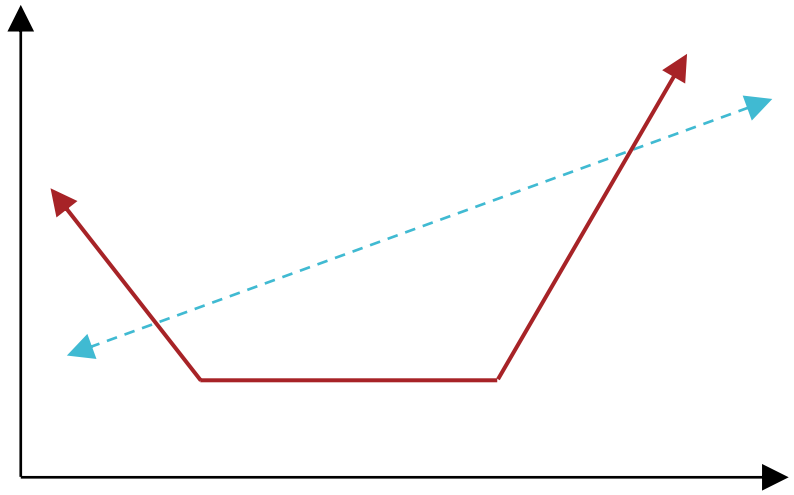
- A function $f: \mathbb{R}^D \rightarrow \mathbb{R}$ is *strictly* convex if

  $\forall\, \boldsymbol{x}^{(1)} \in \mathbb{R}^D, \boldsymbol{x}^{(2)} \in \mathbb{R}^D$ and $0 < c < 1$

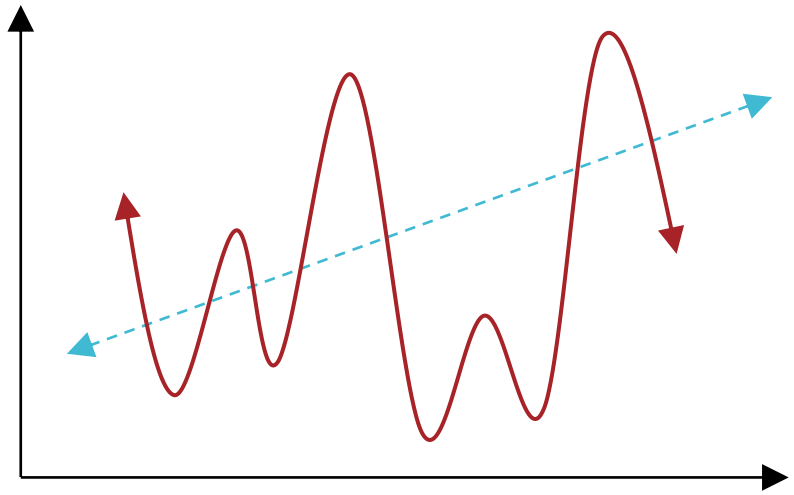
  $$f\left(c\boldsymbol{x}^{(1)} + (1-c)\boldsymbol{x}^{(2)}\right) < cf\left(\boldsymbol{x}^{(1)}\right) + (1-c)f\left(\boldsymbol{x}^{(2)}\right)$$



Convexity

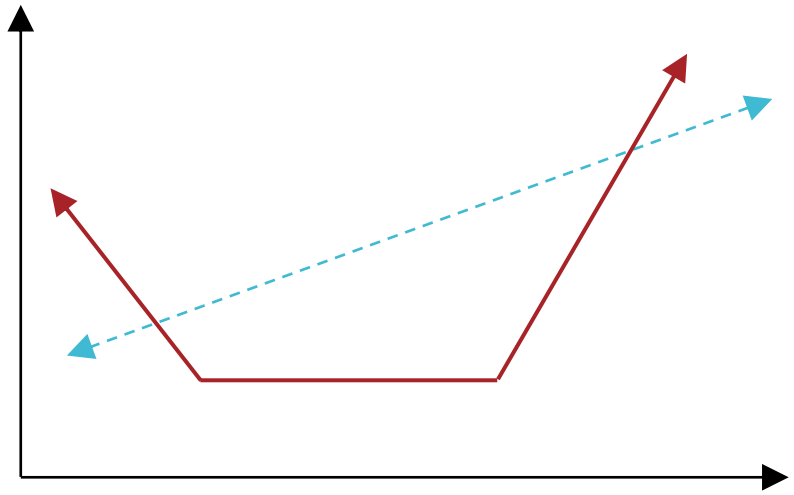


Convex functions



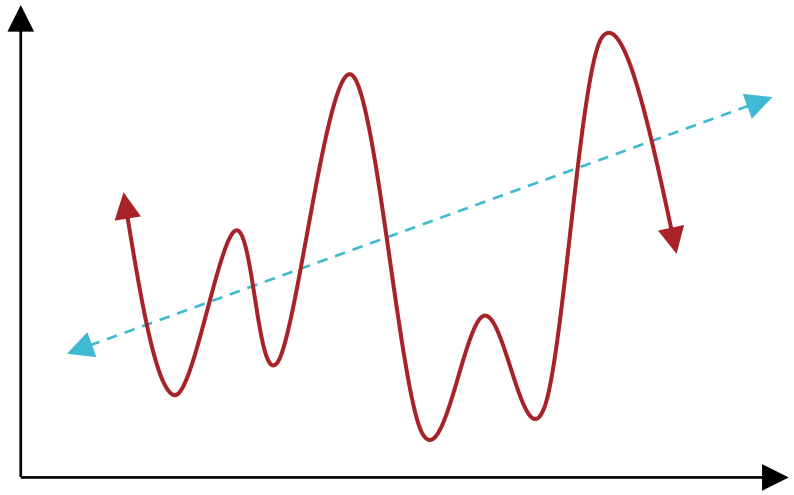
Non-convex functions

Convexity



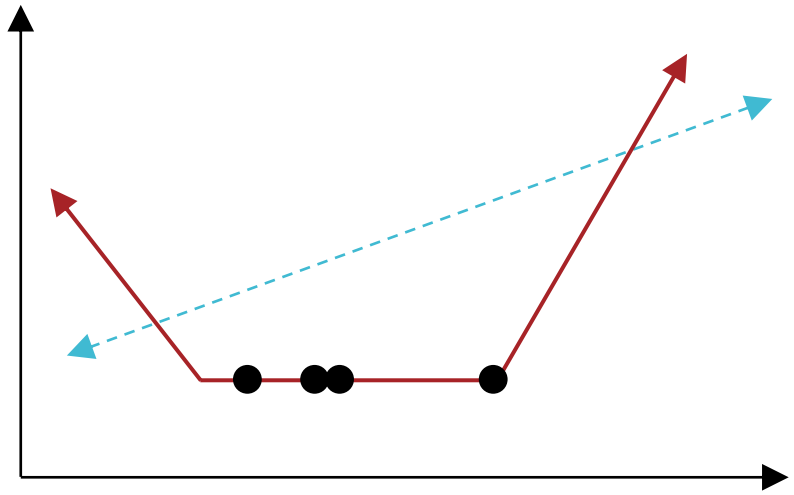
Given a function $f: \mathbb{R}^D \rightarrow \mathbb{R}$

- $\boldsymbol{x}^*$ is a *global* minimum iff $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}) \; \forall \; \boldsymbol{x} \in \mathbb{R}^D$



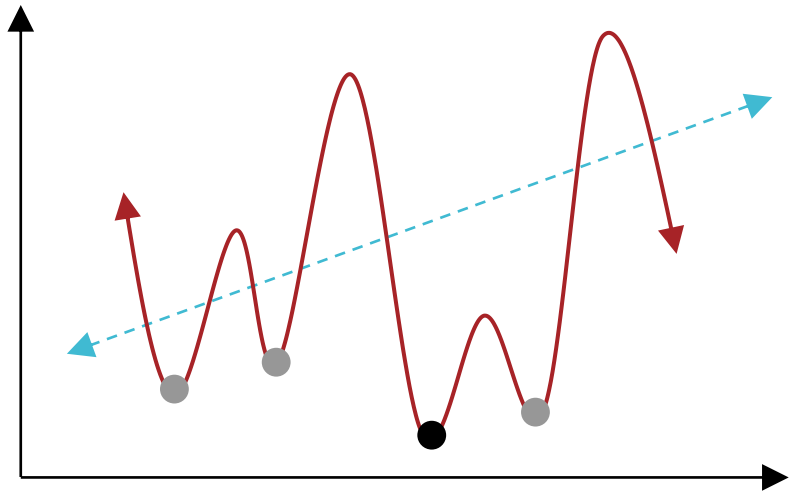
- $\boldsymbol{x}^*$ is a *local* minimum iff $\exists \; \epsilon$ s.t. $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}) \; \forall \; \boldsymbol{x}$ s.t. $\|\boldsymbol{x} - \boldsymbol{x}^*\|_2 < \epsilon$

Convexity



Convex functions:

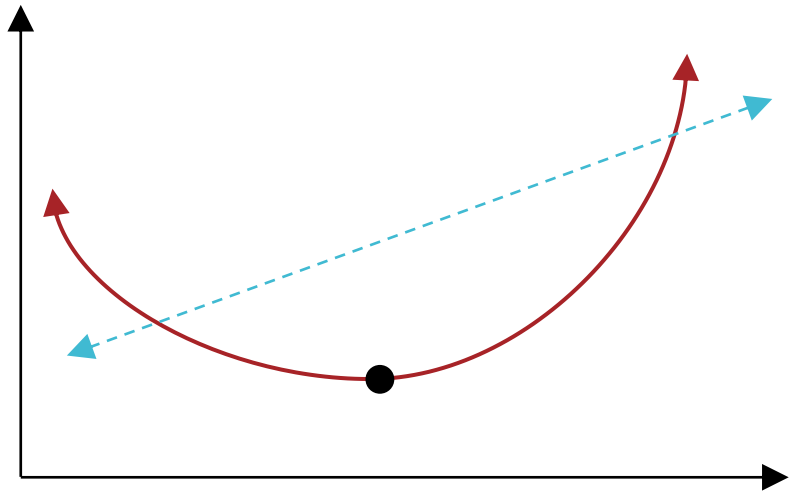
Each local minimum is a global minimum!



Non-convex functions:

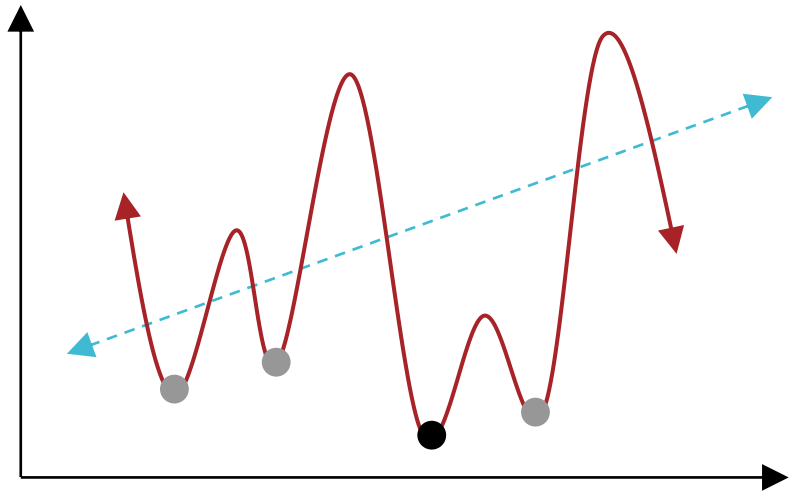
A local minimum may or may not be a global minimum…

Convexity



Strictly convex functions:

There exists a unique global minimum!

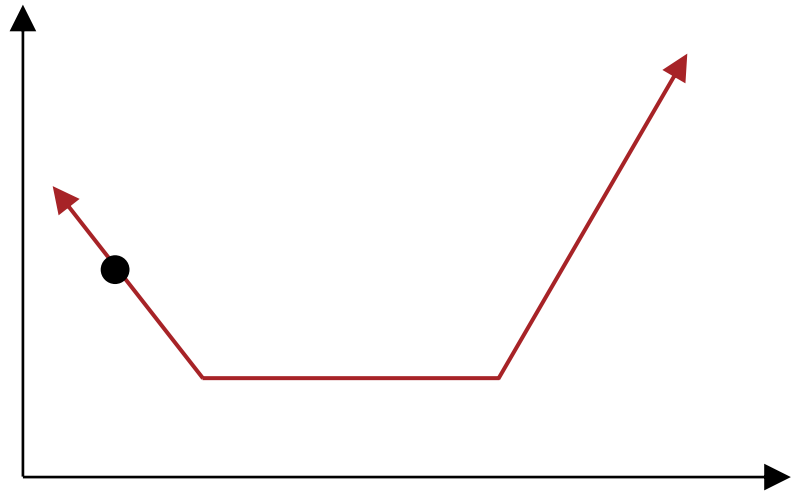


Non-convex functions:

A local minimum may or may not be a global minimum…

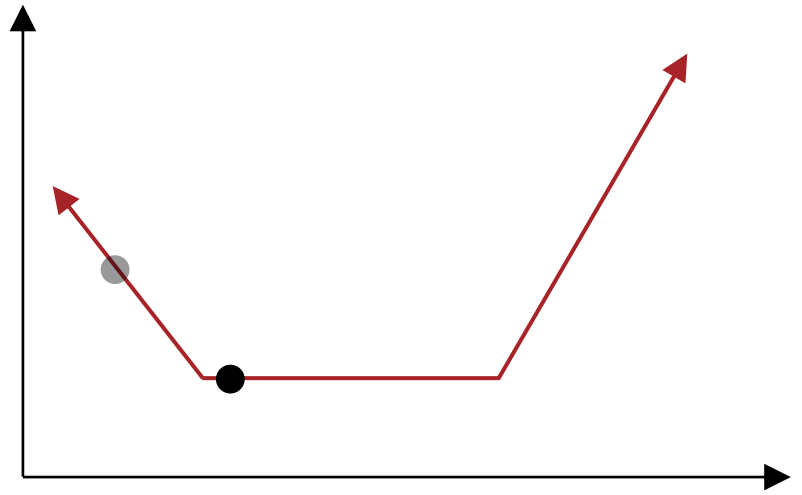
Gradient Descent & Convexity

- Gradient descent is a local optimization algorithm – it will converge to a local minimum (if it converges)
  - Works great if the objective function is convex!



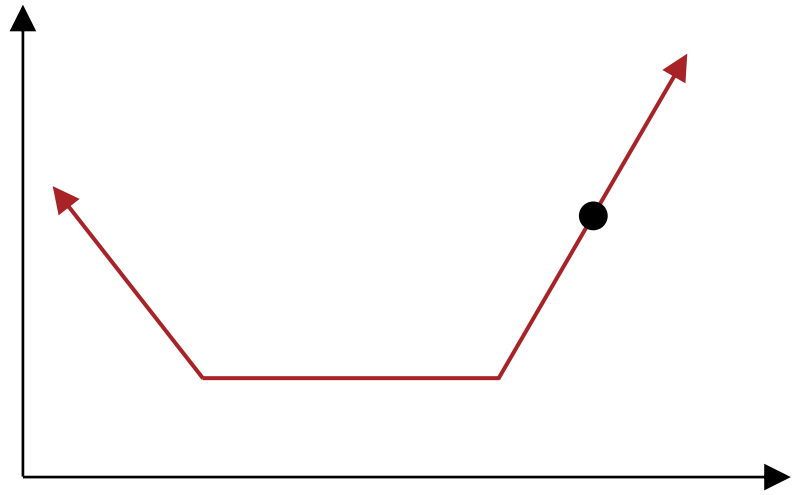
Gradient Descent & Convexity

- Gradient descent is a local optimization algorithm – it will converge to a local minimum (if it converges)
  - Works great if the objective function is convex!



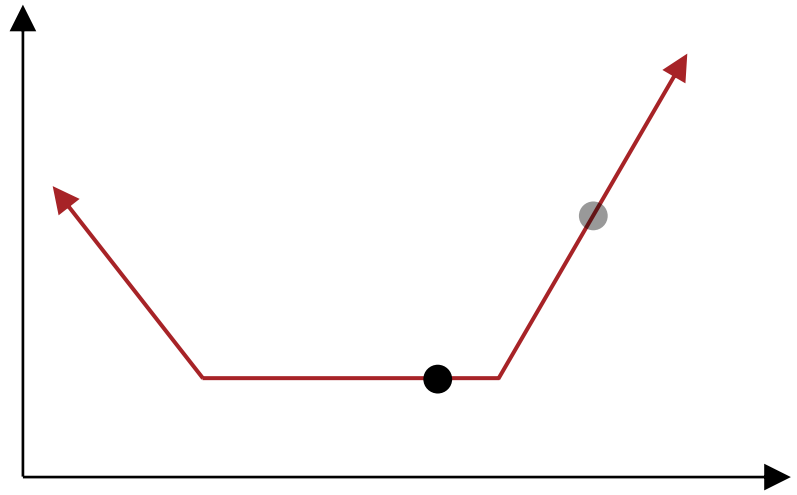
Gradient Descent & Convexity

- Gradient descent is a local optimization algorithm – it will converge to a local minimum (if it converges)
  - Works great if the objective function is convex!



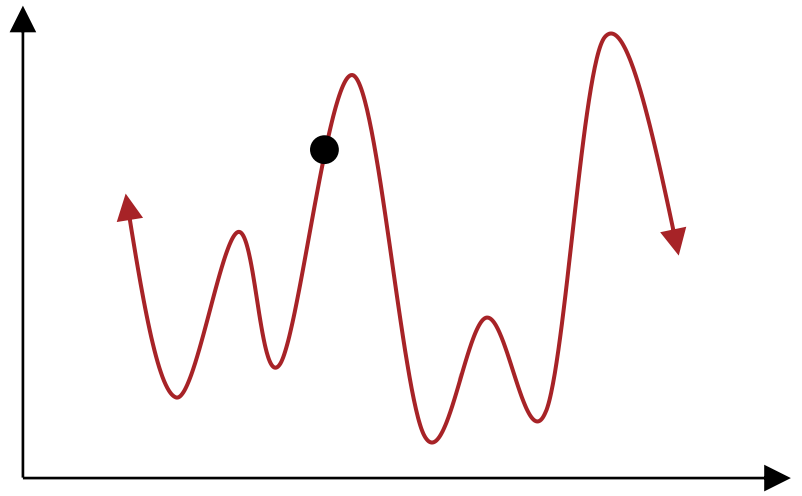
Gradient Descent & Convexity

- Gradient descent is a local optimization algorithm – it will converge to a local minimum (if it converges)
  - Works great if the objective function is convex!



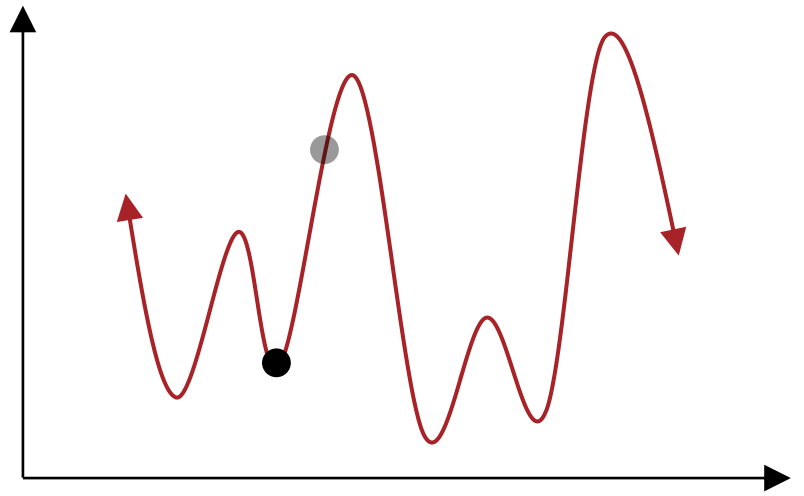
Gradient Descent & Convexity

- Gradient descent is a local optimization algorithm – it will converge to a local minimum (if it converges)
  - Not ideal if the objective function is non-convex…



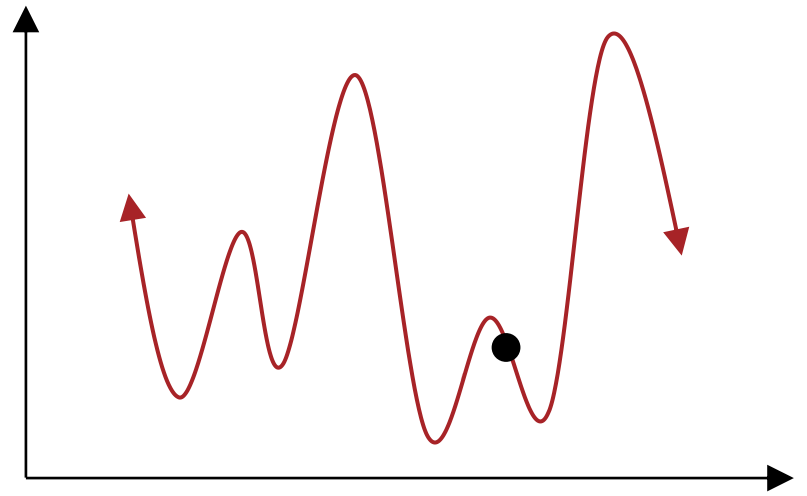
Gradient Descent & Convexity

- Gradient descent is a local optimization algorithm – it will converge to a local minimum (if it converges)
  - Not ideal if the objective function is non-convex…



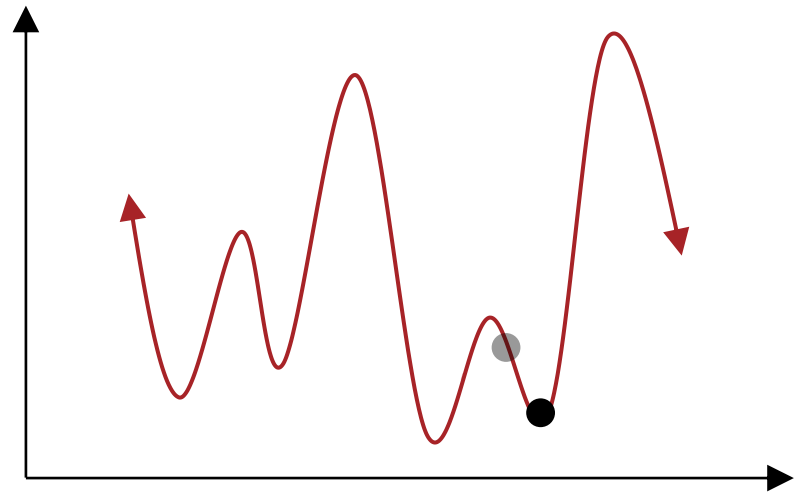
Gradient Descent & Convexity

- Gradient descent is a local optimization algorithm – it will converge to a local minimum (if it converges)
  - Not ideal if the objective function is non-convex…



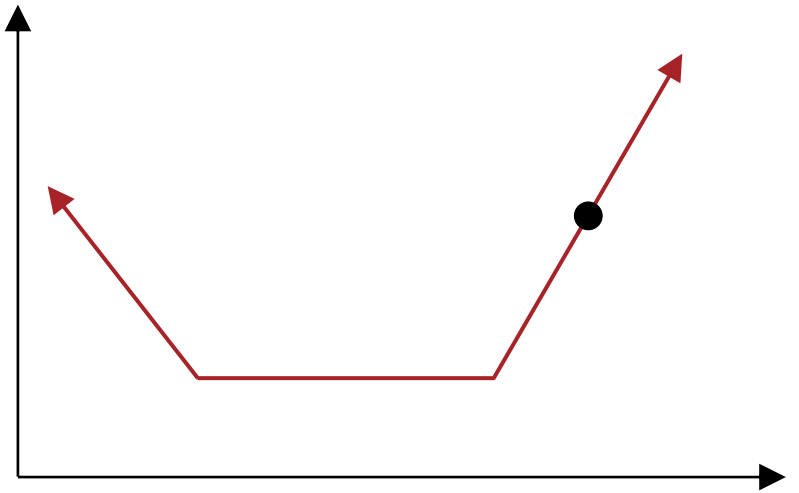
Gradient Descent & Convexity

- Gradient descent is a local optimization algorithm – it will converge to a local minimum (if it converges)
  - Not ideal if the objective function is non-convex…



The squared error for linear regression is convex (but not strictly convex)!

- Gradient descent is a local optimization algorithm – it will converge to a local minimum (if it converges)
  - Works great if the objective function is convex!



$H_{\boldsymbol{w}}\ell_{\mathcal{D}}(\boldsymbol{w}) = \frac{2}{N}X^T X$ which is positive *semi*-definite

Stochastic Gradient Descent

- Many ML objectives decompose over data:
  - $L(w) = (1/N) \Sigma_i \ell_i(w)$
- Full-batch GD computes $\nabla L(w)$ using all N examples each step
- SGD uses a stochastic gradient from one example (or a mini-batch)
  - → cheaper updates, works for large-scale / online learning

SGD: Update Rule

- Sample an index $i_t \in \{1,\ldots,N\}$ (or shuffle and iterate)
- $g_t = \nabla \ell_{i_t}(w_t)$ (unbiased: $\mathbb{E}[g_t] = \nabla L(w_t)$)
- Update: $w_{t+1} = w_t - \eta_t g_t$
- One epoch ≈ N updates (or N/B for mini-batch size B)

Mini-batch SGD

- Use batch size B to reduce gradient noise:
  - $g_t = (1/B) \Sigma_{\{j \in B_t\}} \nabla \ell_j(w_t)$
- B = 1 → classic SGD
- B = N → full-batch gradient descent
- Tradeoff: larger B ⇒ lower variance, higher compute per step, better parallelism

GD vs SGD: same goal, very different vibes 😌 vs 🎲

Full gradient = calm & expensive. Stochastic = cheap & jittery.

Gradient Descent (full batch) 😌

Uses ALL data each step
Direction is stable
Great when N is small / objective is well-behaved
But one step can be expensive

Stochastic / Mini-batch SGD 🎲

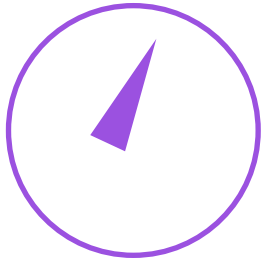
Uses 1 example or a mini-batch
Direction is noisy (sometimes uphill!)
Much cheaper per update → scales
Noise can help in non-convex landscapes

SGD has a few big knobs

If you remember only one: tune the learning rate first.

Learning rate η

Too big → diverge. Too small → nap time.

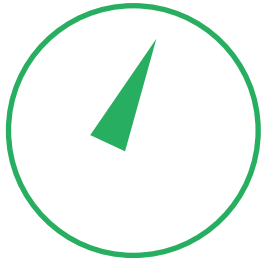


Momentum β

Averages directions → less zig-zag.

Batch size B

Noise vs throughput. Bigger B → smoother.



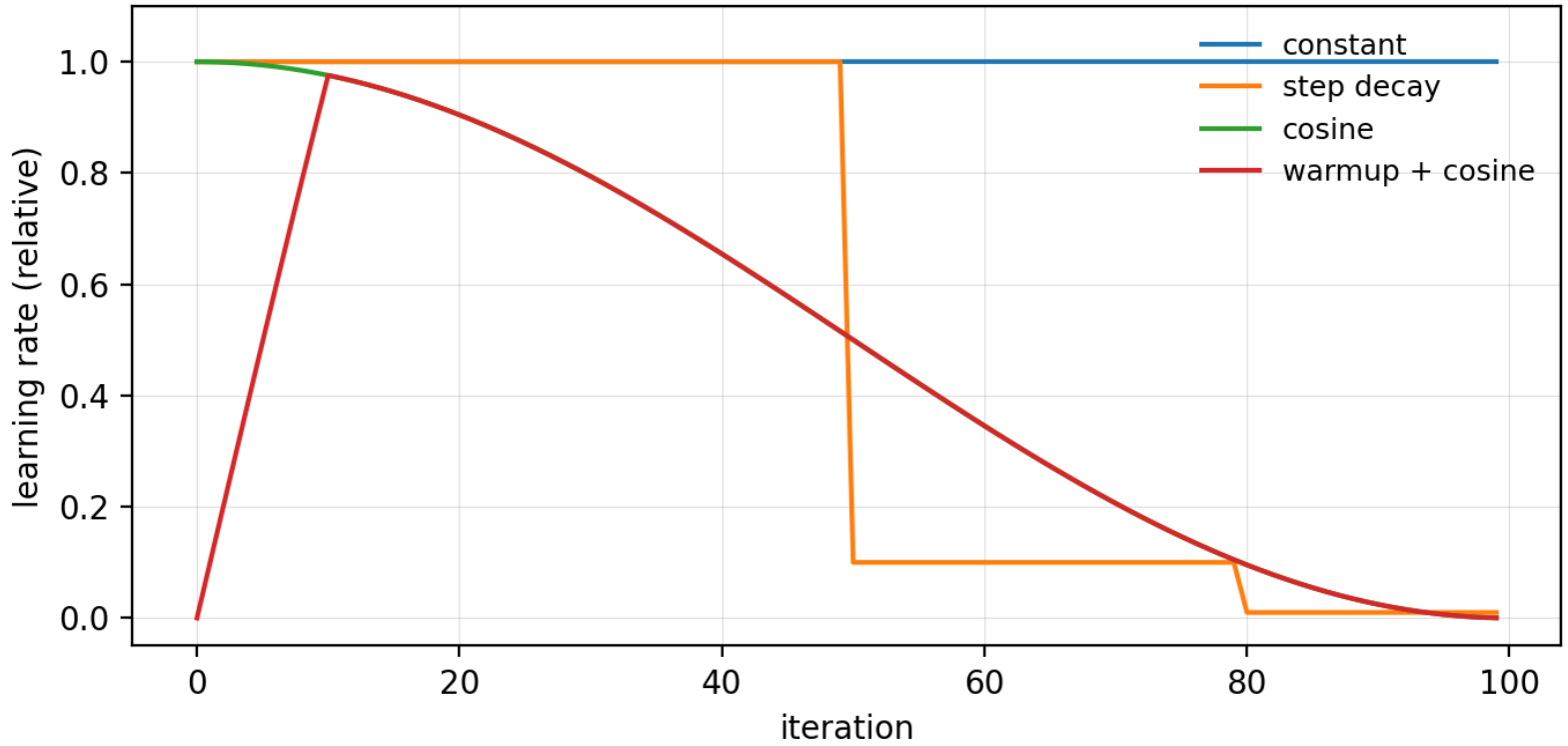
Schedule

Decay to "settle"; warmup for huge batches.

Bonus knob: weight decay (regularization), tune separately from η.

Learning Rate Schedules

- Decaying η_t is key for stable convergence with stochastic gradients
- Common choices: step decay, cosine decay, one-cycle; optional warmup



Early stopping on validation can replace aggressive decay

Practical Checklist

- Shuffle data each epoch; use mini-batches
- Standardize features
- Good initialization (helps conditioning)
- Monitor training vs validation; use early stopping
- Gradient clipping if exploding gradients
- If SGD is finicky: try momentum or adaptive optimizers (Adam/AdamW)

Optimization Mythbusters 

Things that *sound* true until SGD humbles us.

Myth: "Loss must go down every step."

Reality: SGD is noisy — some steps go up, overall trend goes down.

Myth: "Bigger batch is always better."

Reality: It's a tradeoff: smoother gradients vs fewer updates.

Myth: "Adam always wins."

Reality: Great default, but SGD+momentum can win with good schedules.

Myth: "Learning rate is just a detail."

Reality: LR is *the* knob. Tuning it beats fancy tricks.

Is SGD gradient "correct" on average?

Yes: it points the right way *in expectation*.

If $J(\theta) = (1/n) \Sigma_i \ell_i(\theta)$ and we sample i_t uniformly,

then $\mathbb{E}[g_t \mid \theta_t] = \mathbb{E}[\nabla \ell_{i_t}(\theta_t) \mid \theta_t] = \nabla J(\theta_t)$. (Unbiased!)

But… unbiased ≠ monotone.

SGD can take "bad" steps — that's normal. We manage it with batch size, momentum, and LR schedules.

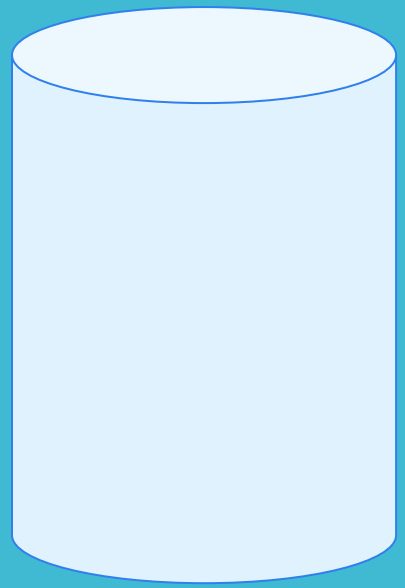
Stochastic Optimization : Deeper Dive

SGD for linear regression

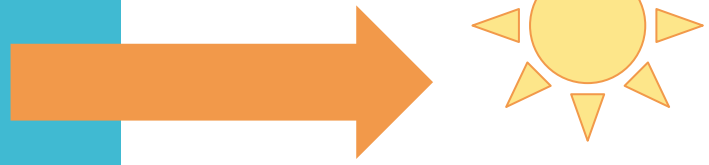
- Noise intuition
- Momentum
- Adam/AdamW

Why stochastic updates? Because the dataset is *huge*

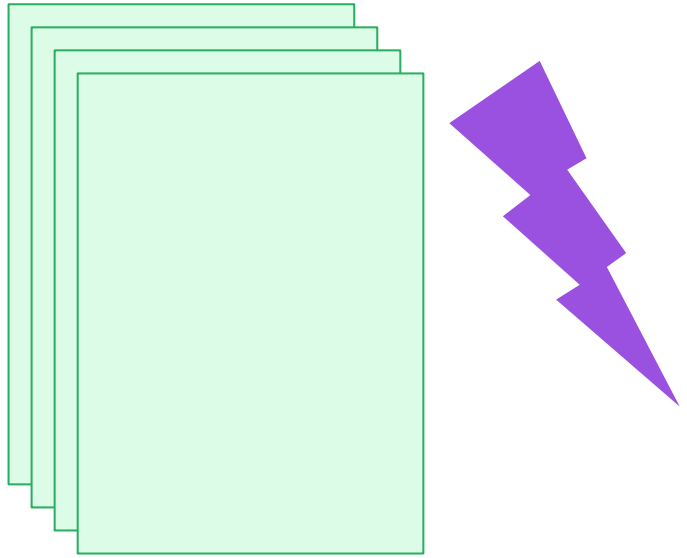
Full GD wants to read everything before taking one step.



N = 10^9 rows



Full GD:
1 update / epoch



SGD / mini-batch:
Many cheap updates → fast feedback + scalability

SGD = GD + noise

Think: "right direction on average, jittery in the moment."

$$g_t = \nabla \ell_{i_t}(w_t) \approx \nabla L(w_t) + \text{noise}$$

signal (full gradient direction)

noise (mini-batch randomness)

That's why the loss can go up on some steps — and it's still learning.

SGD for linear regression: objective

- Data: $\{(x_i, y_i)\}_{i=1}^N, \quad x_i \in \mathbb{R}^d, y_i \in \mathbb{R}$
- Model: $\hat{y}_i = w^T x_i$ (or $\hat{y}_i = w^T x_i + b$)
- Empirical risk (MSE):

$$L(w) = (1/N) \sum_{i=1}^N (w^T x_i - y_i)^2$$

SGD for linear regression: per-example gradient

- Per-example loss:
  $\ell_i(w) = (w^T x_i - y_i)^2$
- Gradient (vector):
  $\nabla \ell_i(w) = 2 (w^T x_i - y_i) x_i$
- Include bias by augmenting:
  $\tilde{x}_i = [x_i; 1], \quad \tilde{w} = [w; b].$

SGD update rule (linear regression)

Sample $i_t \in \{1,\ldots,N\}$

$g_t = 2 (w_t^T x_{i_t} - y_{i_t}) x_{i_t}$

Update:

$w_{t+1} = w_t - \eta_t g_t$

One epoch $\approx$ N updates (or N/B updates with mini-batch size B).

Mini-batch SGD for linear regression

Pick a mini-batch B_t of size B

$$g_t = (1/B) \sum_{j \in B_t} 2 (w_t^T x_j - y_j) x_j$$

Update:

$$w_{t+1} = w_t - \eta_t g_t$$

As $B \uparrow$: variance $\downarrow$, but compute per step $\uparrow$ (often better GPU utilization).

Pseudocode: mini-batch SGD (linear regression)

```
initialize w
repeat for epochs:
    shuffle data
    for each mini-batch B:
        pred = X_B w
        err  = pred – y_B
        g    = (2/B) X_B^T err
        w    = w – η g
```

Cost + vibe: GD vs SGD vs mini-batch

All optimize the same objective; they just spend compute differently.

GD

Updates/epoch: 1
Cost/update: O(Nd)
Stable direction
Best when N is small

SGD (B=1)

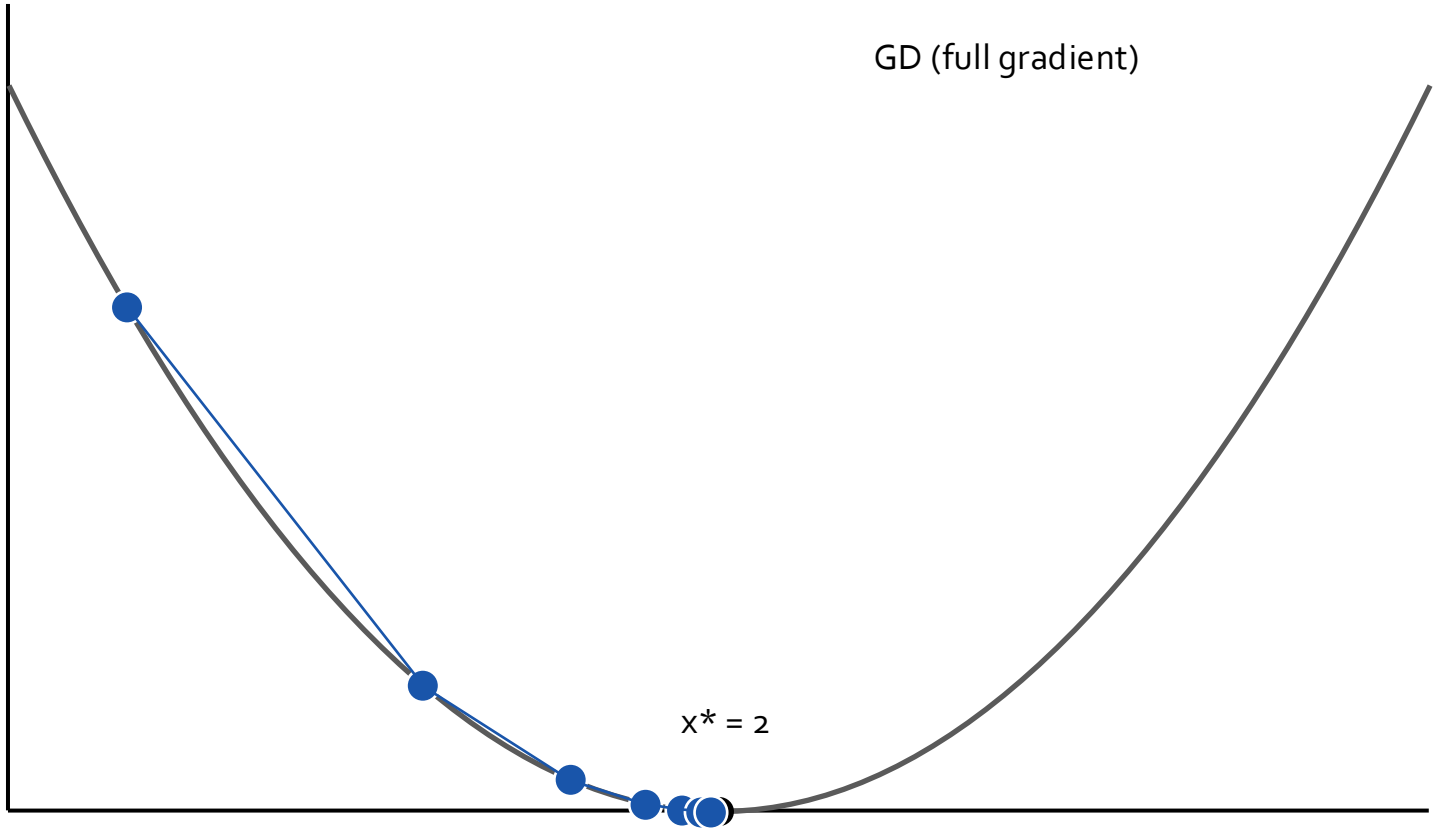
Updates/epoch: N
Cost/update: O(d)
Noisy steps
Fast feedback

Mini-batch

Updates/epoch: N/B
Cost/update: O(Bd)
Less noise
Great for GPUs

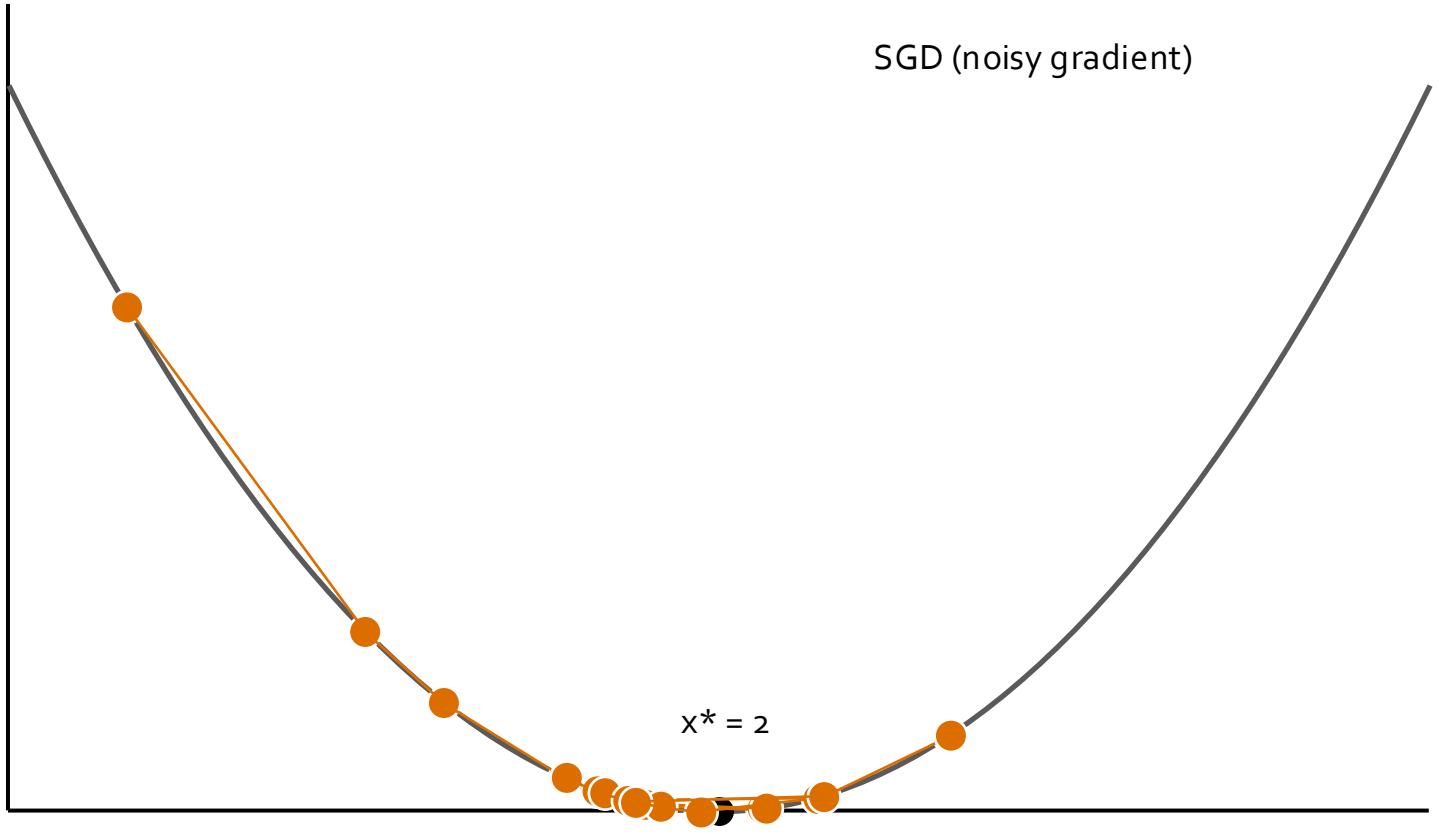
Toy quadratic: GD iterates (deterministic)

Objective: $f(x)=(x-2)^2$. Same starting point, fixed η.



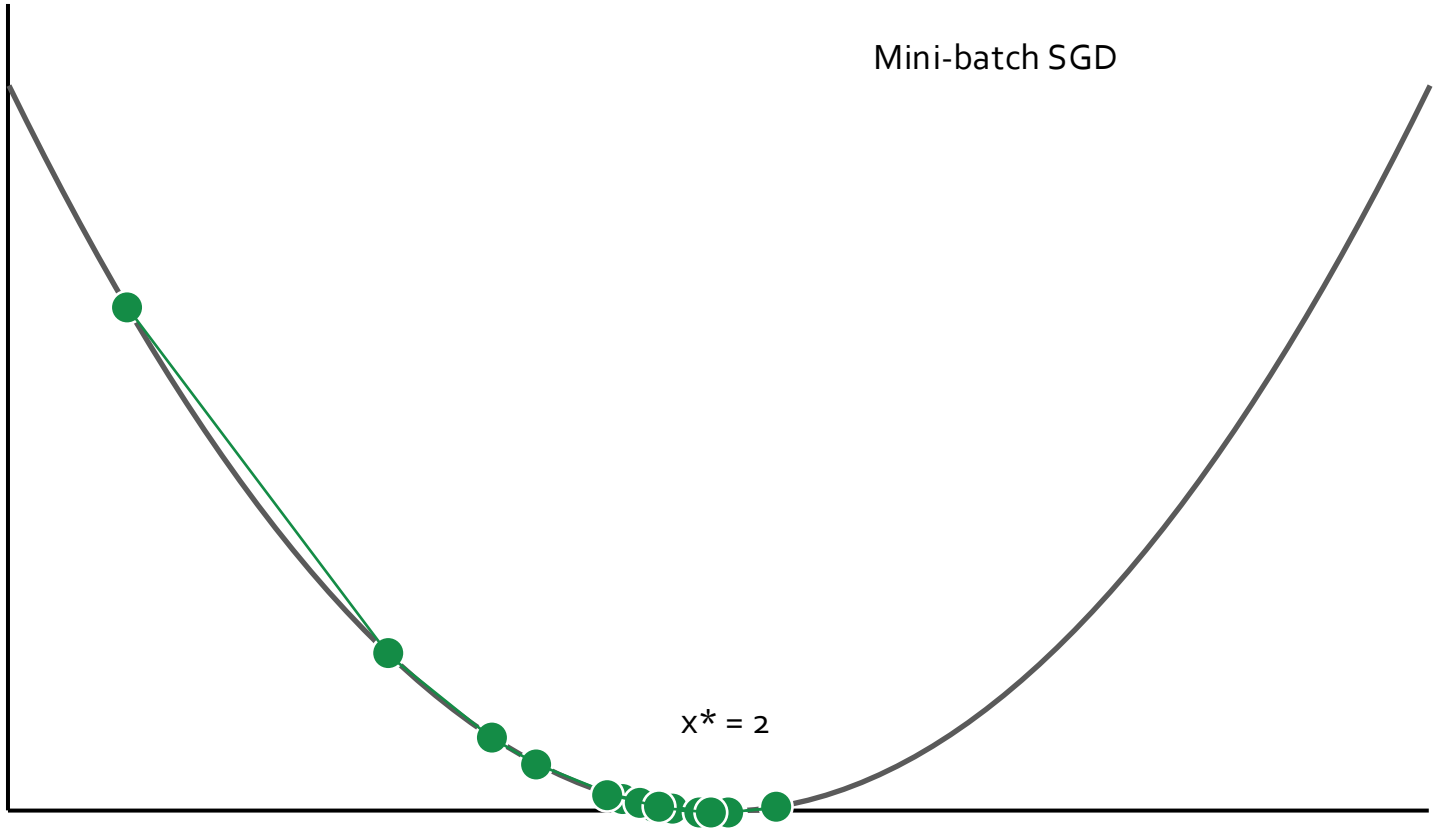
Toy quadratic: SGD iterates (noisy)

Replace true gradient with a noisy (unbiased) estimate.



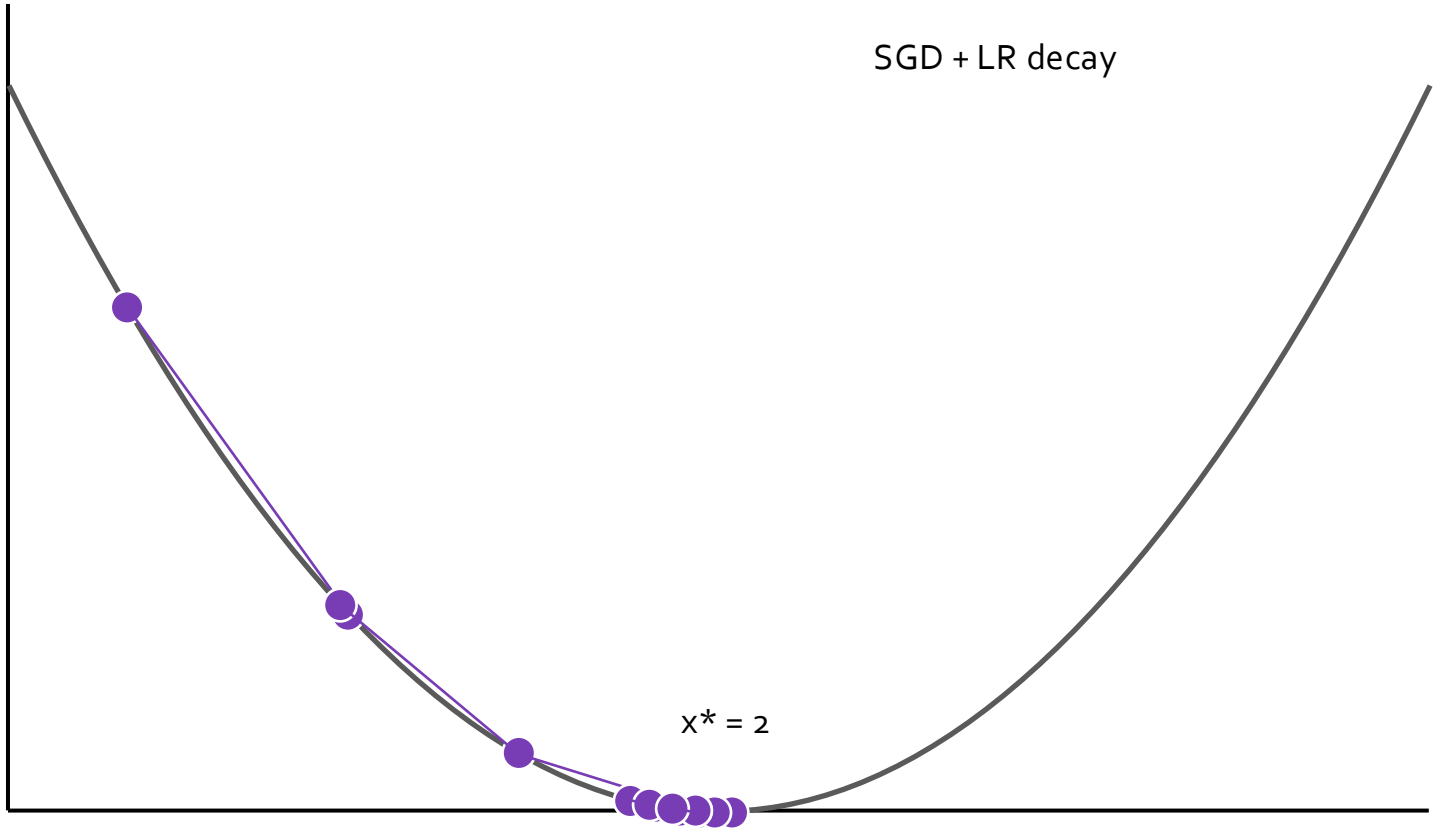
Toy quadratic: mini-batch reduces noise

Averaging gradients shrinks variance → smoother trajectory.



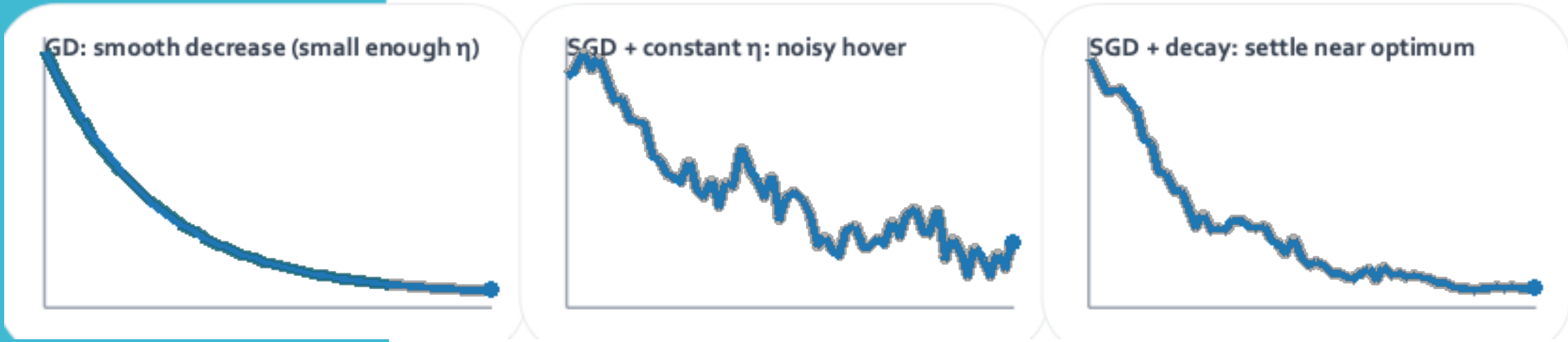
Toy quadratic: learning-rate decay

Decay η_t to shrink step sizes and "settle" near the minimum.



How does it "end"? Three common endings

The difference is mostly: learning-rate schedule + noise.



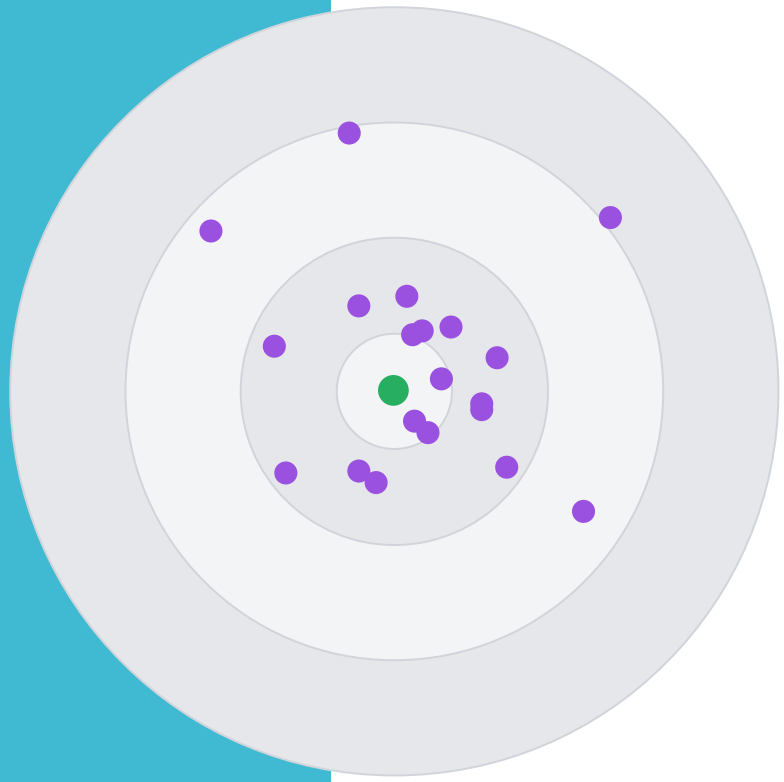
Stochastic gradients are unbiased, but variance slows the last mile.

Constant $\eta \rightarrow$ you bounce in a neighborhood (a "noise floor").

Decaying η (or bigger batches) helps you refine to a point.

Noise floor: why SGD “jitters” near the minimum 🎯

Near optimum, true gradient shrinks — but mini-batch noise doesn’t.



Fixes (pick one):

- decay η
- increase batch size
- add momentum
- average weights / EMA

Gradient noise: why batch size matters

- SGD trades computation for variance
- Batch size B reduces gradient variance roughly like 1/B
- But larger B means fewer parameter updates per epoch
- In practice: there's a "sweet spot" where extra B mostly buys parallelism

Sampling & shuffling: practical details

- Common practice: shuffle once per epoch and iterate through data
- This is not exactly i.i.d. sampling, but works well and is cache-friendly
- If data are ordered (e.g., by class/time), not shuffling can bias training dynamics
- For streaming data, sampling with replacement is natural

Batch size tradeoff: noise vs throughput ⚖️

Small batch = noisy but many updates. Large batch = smooth but fewer updates.

Small B (e.g., 32)

More noise
More parameter updates
Often better "exploration"

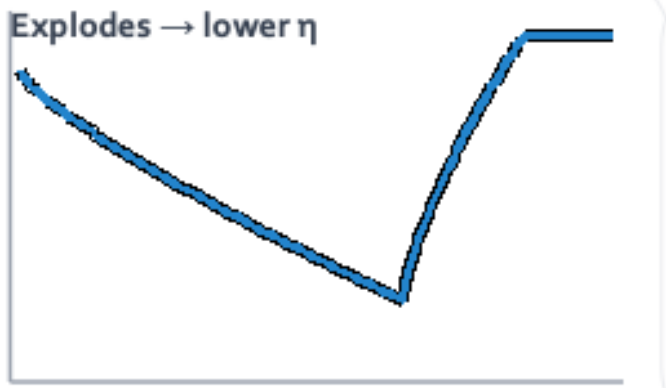
Large B (e.g., 4096)

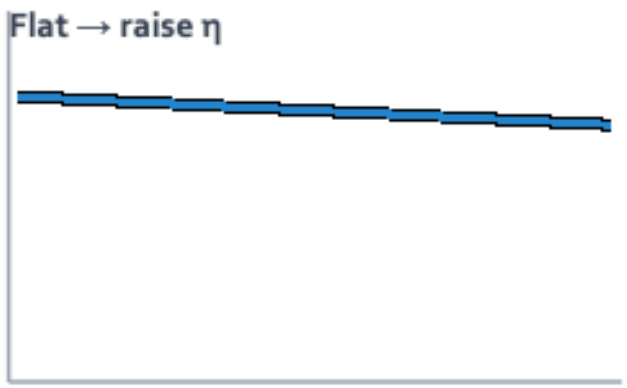
Less noise
Fewer updates per epoch
Great GPU throughput (may need warmup)

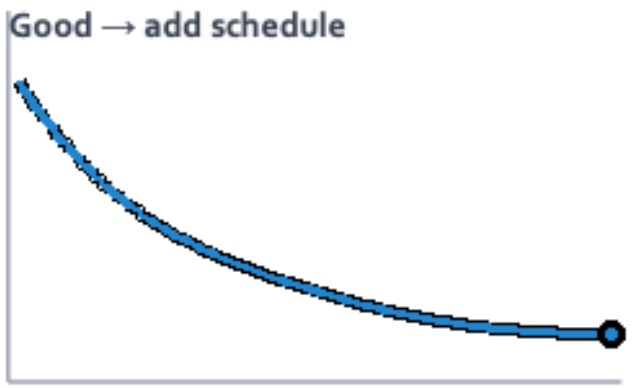
Rule of thumb: pick B to saturate hardware, then tune η (and schedule).

Learning rate tuning: the 3 curve test

Do a quick log sweep; watch for these shapes.







Practical mini-playbook:

- Start with η sweep (×10 up/down).
- If using huge batches: warmup helps.
- If you want crisp final loss: decay η.

Momentum: intuition

- SGD steps can “zig-zag” when gradients change direction across dimensions
- Momentum keeps a running velocity v_t that smooths directions
- Effect: faster progress along consistent directions + less sensitivity to noise
- Think: a ball rolling down a valley with friction

SGD + Momentum

Velocity update:

$$v_{t+1} = \beta v_t + g_t$$

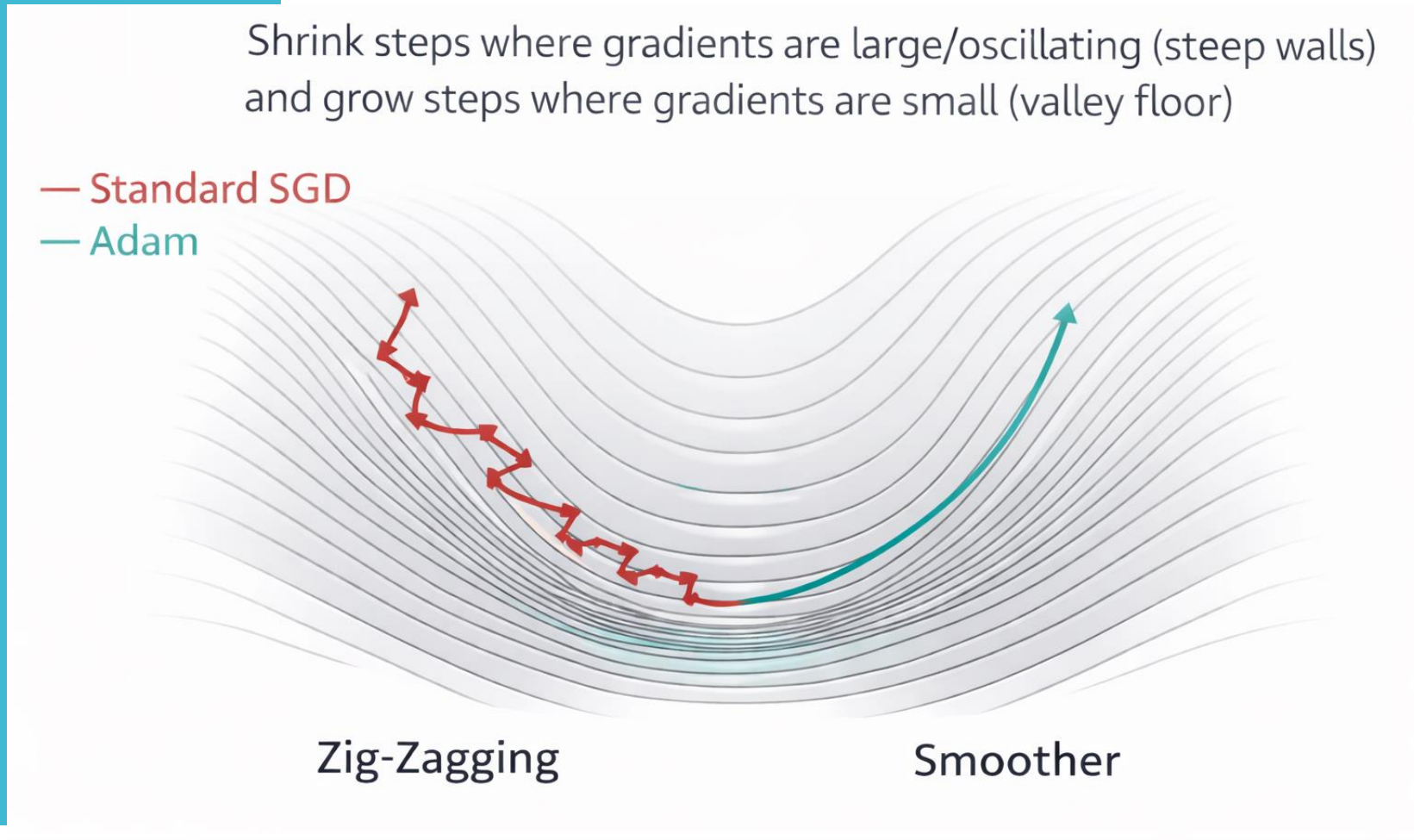
Parameter update:

$$w_{t+1} = w_t - \eta v_{t+1}$$

Typical β: 0.9 (or 0.99).
Often pairs well with mini-batches.

Adaptive optimizers: a ravine problem

Same η in all directions can be painful when curvature differs.



Adaptive optimizers: big picture

- Problem: different parameters may have very different gradient scales
- Idea: keep a per-parameter learning rate using past gradients
- Benefit: less manual tuning in many settings
- Caveat: can generalize worse than SGD+momentum in some deep learning tasks

AdaGrad (per-coordinate learning rates)

Accumulate squared gradients:

$$s_{t+1} = s_t + g_t \odot g_t$$

Update:

$$w_{t+1} = w_t - \eta \; g_t / (\sqrt{s_{t+1}} + \varepsilon)$$

Good for sparse features;
step sizes can shrink too aggressively.

RMSProp (EMA of squared gradients)

Exponential moving average:

$$s_{t+1} = \rho\, s_t + (1-\rho)\,(g_t \odot g_t)$$

Update:

$$w_{t+1} = w_t - \eta\ \ g_t \,/\, (\mathrm{sqrt}(s_{t+1}) + \varepsilon)$$

Fixes AdaGrad’s “learning rate goes to zero” problem.

Adam: momentum + RMSProp

First moment (mean):

$$m_{t+1} = \beta_1 m_t + (1-\beta_1) g_t$$

Second moment:

$$v_{t+1} = \beta_2 v_t + (1-\beta_2) (g_t \odot g_t)$$

Bias-correct:

$$\hat{m}_{t+1} = m_{t+1} / (1-\beta_1^{t+1})$$

$$\hat{v}_{t+1} = v_{t+1} / (1-\beta_2^{t+1})$$

because we start at 0, the EMA underestimates the true first/second moments

Update:

$$w_{t+1} = w_t - \eta \ \hat{m}_{t+1} / (\text{sqrt}(\hat{v}_{t+1}) + \varepsilon)$$

Adam: update rule (with bias correction)

```
initialize m=0, v=0, t=0
repeat:
    t = t + 1
    g = stochastic_gradient(w)
    m = β1 m + (1−β1) g
    v = β2 v + (1−β2) (g ⊙ g)
    m_hat = m / (1−β1^t)
    v_hat = v / (1−β2^t)
    w = w − η * m_hat / (sqrt(v_hat) + ε)
```

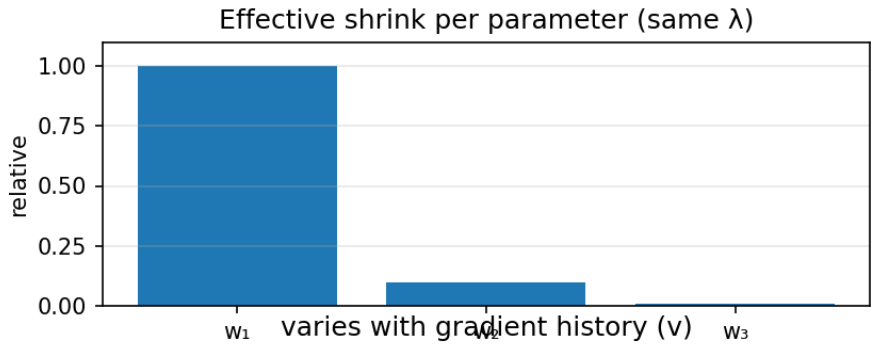
AdamW: fixing the regularization pitfall in Adam

Why “L2 regularization” ≠ “weight decay” for adaptive optimizers

Pitfall: Adam + L2 in the loss

$$g \leftarrow g + \lambda\,\theta$$
$$\theta \leftarrow \theta - \eta\,\frac{\hat{m}}{\sqrt{\hat{v}} + \varepsilon}$$

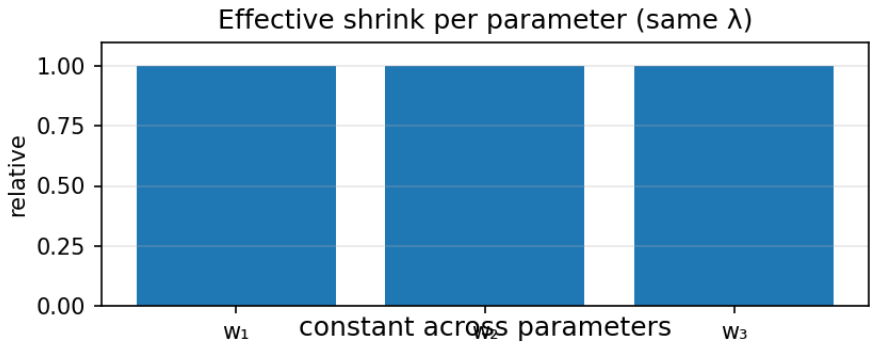
- The decay term is scaled by 1/√v (per-parameter)
- Some weights shrink a lot, others barely shrink
- λ stops being a clean global regularization knob



Fix: AdamW (decoupled weight decay)

$$\theta \leftarrow \theta - \eta\,\frac{\hat{m}}{\sqrt{\hat{v}} + \varepsilon}$$
$$\theta \leftarrow (1 - \eta\lambda)\,\theta$$

- Decay is applied directly to weights (not normalized)
- Every parameter gets the same shrink factor
- λ behaves like true weight decay again



Takeaway: AdamW keeps adaptivity for the gradient step, but makes regularization a simple, uniform weight shrink.

Which optimizer should I start with?

A friendly default decision rule for intro ML.

Convex / linear models

Closed-form if possible
Otherwise GD / SGD
Main issue: conditioning

Deep nets (fast to get going)

AdamW default
Use warmup + decay
Great early progress

Chasing best generalization

Try SGD + momentum
Needs LR schedule
Often wins late

Rule of thumb: tune η first, then (batch, momentum, decay). Fancy optimizers don't save a bad learning rate.

Debugging training: a tiny flowchart 🧯

Most failures are just LR, data scaling, or shuffling.

What does the loss curve look like?

Explodes 🚨
- ↓ learning rate
- check feature scaling
- try gradient clipping

Flat 😴
- ↑ learning rate
- normalize features
- better initialization

Noisy 🎲
- ↑ batch size
- add momentum
- decay η

Overfits ❓
- weight decay
- early stopping
- more data/augmentations