CS498: Algorithmic Engineering

Lecture 10: Convexity, Gradient Descent, & PyTorch.

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 05 – 02/19/2026

Outline

Where We Are

Part I (Weeks 1–5): Linear and Integer Programming.

- LPs, duality, simplex, branch-and-bound, modeling.
- Everything was *constrained* optimization over a **polytope**.

Part II (Weeks 6–8/9): Continuous Optimization.

- Convex optimization, Gradient methods, Non-convex optimization, neural networks, constrained nonlinear optimization.
- We need new tools.

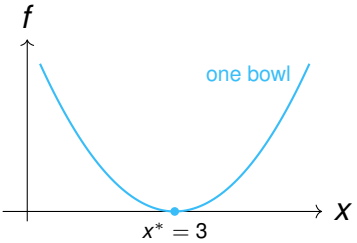
Motivating Example: Which problem is “easy”?

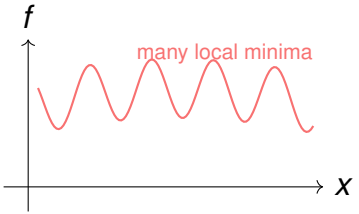
Problem A:

$$\min_{x \in \mathbb{R}} (x - 3)^2$$

Problem B:

$$\min_{0 \le x \le 6} \cos(5x) - 0.1x^2$$





Why does Problem A feel easy? It is **convex**: any local minimum is automatically a global minimum.

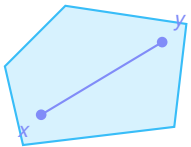
Convex Sets: Definition

Definition

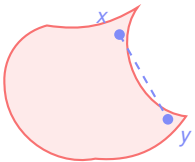
A set $C \subseteq \mathbb{R}^n$ is **convex** if for every $x, y \in C$ and $\theta \in [0, 1]$:

$$\theta x + (1 - \theta) y \ \in \ C.$$

In words: the **line segment** connecting any two points in C stays inside C.



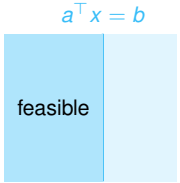
Convex



Non-convex

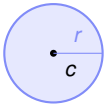
Examples of Convex Sets

Halfspace: $\{x : a^\top x \leq b\}$



Polyhedron: $\{x : Ax \leq b\}$
(intersection of halfspaces)

Ball: $\{x : \|x - c\|_2 \leq r\}$



PSD cone: $\mathbb{S}^n_+ = \{X : X \succeq 0\}$
(positive semidefinite matrices)

Connection to Part I

LP feasible regions $\{x : Ax \leq b, x \geq 0\}$ are **polyhedra**, convex sets we already know!

Key Property: Intersection Preserves Convexity

Theorem

If C_1 and C_2 are convex, then $C_1 \cap C_2$ is convex.

Proof sketch. Take any $x, y \in C_1 \cap C_2$ and $\theta \in [0, 1]$.

- Since $x, y \in C_1$ and C_1 convex: $\theta x + (1 - \theta)y \in C_1$.
- Since $x, y \in C_2$ and C_2 convex: $\theta x + (1 - \theta)y \in C_2$.
- Therefore $\theta x + (1 - \theta)y \in C_1 \cap C_2$.

□

Why this matters

A polyhedron is the intersection of halfspaces.
Each halfspace is convex. So every LP feasible region is convex.
We have been doing convex optimization all along in Part I.

Convex Functions: Definition

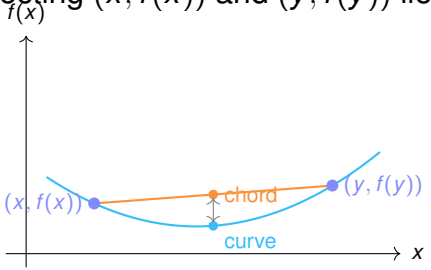
Definition

$f : \mathbb{R}^n \to \mathbb{R}$ is **convex** if for all x, y and $\theta \in [0, 1]$:

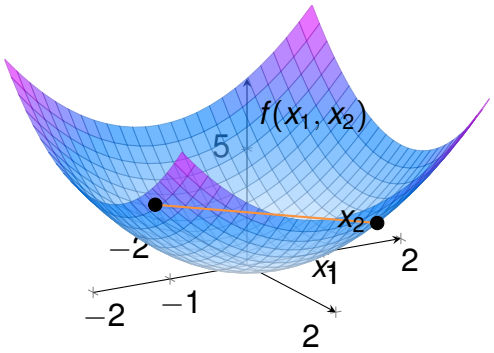
$$f(\theta x + (1 - \theta)y) \leq \theta f(x) + (1 - \theta)f(y).$$

f is **concave** if and only if $-f$ is convex.

In words: the **chord** connecting $(x, f(x))$ and $(y, f(y))$ lies **above** the function.



Convex Functions in $\mathbb{R}^2$



Convex Functions: Examples

Convex:

- $f(x) = x^2$ (bowl)
- $f(x) = e^x$ (exponential)
- $f(x) = |x|$ (absolute value)
- $f(\mathbf{x}) = \|\mathbf{x}\|$ (norm)
- $f(\mathbf{x}) = \mathbf{c}^\top \mathbf{x}$ (linear! both convex and concave)
- $f(\mathbf{X}) = -\log\det \mathbf{X}$ (for PSD $\mathbf{X}$)

Not convex:

- $f(x) = -x^2$ (concave)
- $f(x) = \sin(x)$ (oscillates)
- $f(x) = x^3$ (inflection point at $x = 0$)
- $f(x, y) = xy$ (saddle)

Note

Linear functions are both convex and concave.

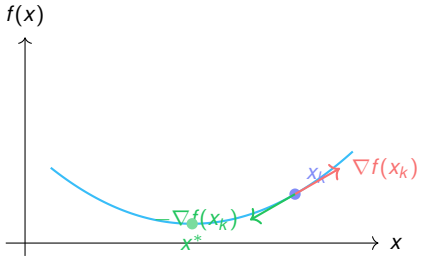
The Gradient: A Reminder

For $f : \mathbb{R}^n \to \mathbb{R}$, the gradient at x is:

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)^\top \in \mathbb{R}^n.$$

Two key geometric facts:

- $\nabla f(x)$ points in the direction of **steepest ascent**.
- $-\nabla f(x)$ points in the direction of **steepest descent**.

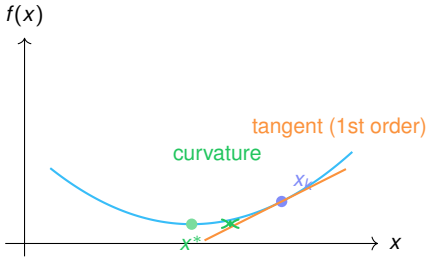


The Hessian: A Reminder

For $f : \mathbb{R}^n \to \mathbb{R}$, the Hessian at x is:

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \in \mathbb{R}^{n \times n}.$$

$\nabla^2 f(x)$ measures the **local curvature** of f.



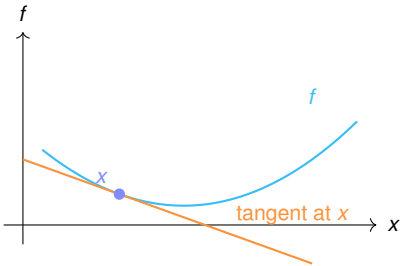
Equivalent Characterizations of Convexity

When f is differentiable, convexity has clean equivalents.

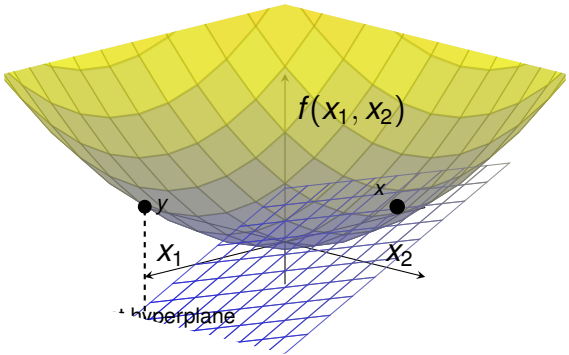
First-order condition (gradient inequality):

$$f(y) \geq f(x) + \nabla f(x)^\top (y - x) \qquad \forall x, y.$$

In words: the **tangent plane at x is a global lower bound** on f.



Equivalent Characterizations of Convexity



Equivalent Characterizations (continued)

Second-order condition (Hessian):

$$f \text{ is convex} \iff \nabla^2 f(x) \succeq 0 \quad \forall x.$$

(The Hessian is **positive semidefinite** everywhere.)

Examples:

- $f(x) = x^2$: $f''(x) = 2 > 0$. Convex.
- $f(x_1, x_2) = e^{x_1} + x_2^2 + 2x_1x_2$:

$$\nabla^2 f(x) = \begin{bmatrix} e^{x_1} & 2 \\ 2 & 2 \end{bmatrix}.$$

  Since $\det(\nabla^2 f) = 2e^{x_1} - 4 \geq 0 \iff x_1 \geq \ln 2$,, we have $\nabla^2 f(x) \succeq 0$ for all x with $x_1 \geq \ln 2$. Convex on $\{x_1 \geq \ln 2\}$.
- $f(x) = -x^2$: $f''(x) = -2 < 0$. Not convex.
- $f(x) = e^x$: $f''(x) = e^x > 0$. Convex.

Operations that Preserve Convexity

You can build complex convex functions from simple convex functions.

- **Nonneg. linear combination:** $\alpha f + \beta g$ is convex if $\alpha, \beta \geq 0$ and f, g are convex.
- **Composition with affine:** $f(A\mathbf{x} + b)$ is convex if f is convex.
- **Pointwise max:** $\max(f(x), g(x))$ is convex if f, g are convex.
- **Norms:** all norms $f(\mathbf{x}) = \|\mathbf{x}\|_p$ for $p \geq 1$ are convex.

Quick check: is $f(x) = \max(x^2,\ 3x + 1)$ convex?

- x^2: convex. $3x + 1$: convex (linear). max of two convex functions: **convex.**

Quick check: is $f(x) = \|Ax - b\|^2 + \lambda\|x\|^2$ convex?

- $\|Ax - b\|^2$ convex. $\lambda\|x\|^2$ convex ($\lambda \geq 0$). Sum: **convex.**
- (This is ridge regression. We'll come back here soon.)

The Core Theorem: Local = Global

Theorem (Fundamental Property of Convex Optimization)

Let f be convex and C be a convex set. Then any **local minimum** of

$$\min_{x \in C} f(x)$$

is also a **global minimum**.

Proof sketch (by contradiction). Suppose x^* is a local minimum but not global. Then $\exists y$ with $f(y) < f(x^*)$.
Consider the segment $x_\theta = \theta x^* + (1-\theta)y \in C$ for $\theta \in [0,1]$.
By convexity:

$$f(x_\theta) \leq \theta f(x^*) + (1-\theta) f(y) < f(x^*)$$

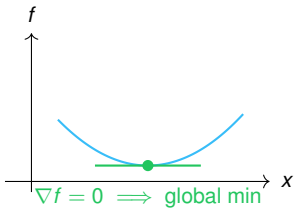
for θ close to 1. This contradicts x^* being a local minimum. $\square$

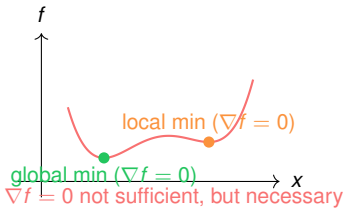
First-Order Optimality for Unconstrained Convex Problems

For unconstrained problems $\min_x f(x)$ with f convex and differentiable:

$$x^* \text{ is optimal} \iff \nabla f(x^*) = 0.$$

For convex f: the gradient condition is both necessary **and sufficient**. This is **not** true in general.





First-Order Optimality in Higher Dimensions

Convex example (2 variables):

$$f(x_1, x_2) = x_1^2 + 2x_2^2$$

$$\nabla f(x) = \begin{bmatrix} 2x_1 \\ 4x_2 \end{bmatrix} \quad \Rightarrow \quad \nabla f(x) = 0 \iff x = (0, 0)$$

Convex example ($\mathbb{R}^n$):

$$f(x) = \|Ax-b\|^2 = (Ax-b)^T(Ax-b) = (x^TA^T-b^T)(Ax-b) = x^TA^TAx-2b^TA^Tx+b^Tb$$

$$\nabla f(x) = 2A^\top(Ax - b)$$

$$\nabla f(x^*) = 0 \iff A^\top Ax^* = A^\top b$$

First-Order Optimality in Higher Dimensions

Nonconvex example (2 variables):

$$f(x_1, x_2) = x_1^4 - x_1^2 + x_2^2$$

$$\nabla f = \begin{bmatrix} 4x_1^3 - 2x_1 \\ 2x_2 \end{bmatrix}$$

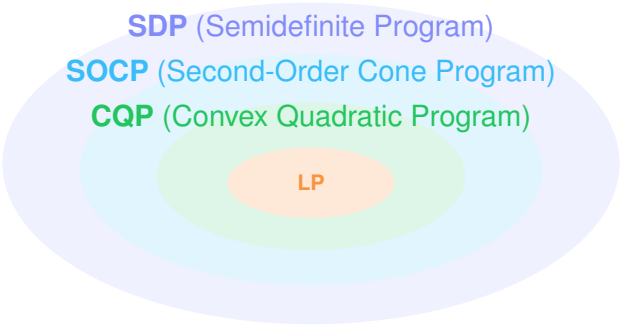
Stationary points:

$$x_2 = 0, \quad x_1 = 0 \text{ or } \pm \frac{1}{\sqrt{2}}$$

At $(0, 0)$: saddle point.
$\nabla f = 0$ is not sufficient in general.

The Convex Program Hierarchy

Convexity comes in a **hierarchy** of increasing modeling power.



Each layer is **strictly more expressive** than the one inside.
Each layer remains **tractable**: polynomial-time solvable by Ellipsoid Method.

Setup: What Are We Solving?

As we said, for convex f, every local min is global, and $\nabla f(x^*) = 0$ is sufficient.
Question: how do we actually *find* x^*?

Setting

$$\min_{x \in \mathbb{R}^n} f(x)$$

where f is **convex** and **differentiable**. We have access to $f(x)$ and $\nabla f(x)$ at any point x.

We will not assume we can solve $\nabla f(x^*) = 0$ in closed form.

Gradient Descent: The Algorithm

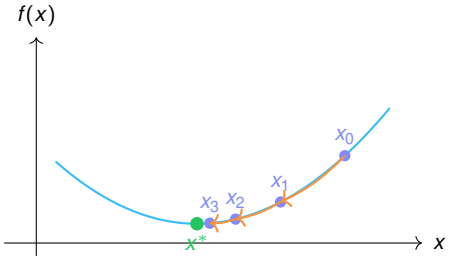
Gradient Descent

Start at some x_0. Repeat:

$$x_{k+1} = x_k - \alpha \nabla f(x_k)$$

where $\alpha > 0$ is the **step size** (also called *learning rate*).

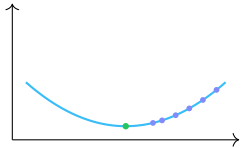
Geometric interpretation: at each step, move in the direction that decreases f most rapidly.



Step Size: The Critical Hyperparameter

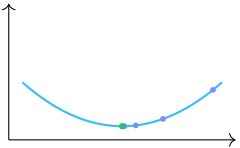
The step size α completely controls behavior.

Too small α



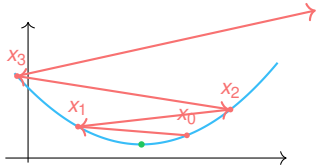
Crawls. Slow.

Just right α



Converges fast.

Too large α



Diverges (oscillations).

Step Size: The Theory

Assume f has L-**Lipschitz gradient**:

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\| \quad \forall x, y.$$

Theorem

If $\alpha \leq \frac{1}{L}$, then gradient descent makes progress at every step:

$$f(x_{k+1}) \leq f(x_k) - \frac{\alpha}{2}\|\nabla f(x_k)\|^2.$$

Example, $f(x) = 4x^2$, then $L = 8$ is safe.

Scaling Changes the Lipschitz Constant

Suppose f has L-Lipschitz gradient:

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|.$$

Now define a scaled function:

$$g(x) = c\, f(x), \qquad c > 0.$$

Then

$$\nabla g(x) = c\nabla f(x).$$

Therefore,

$$\|\nabla g(x) - \nabla g(y)\| = c\|\nabla f(x) - \nabla f(y)\| \leq cL\|x - y\|.$$

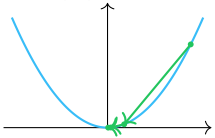
Key Fact

The new Lipschitz constant is $L_g = cL$. **Scaling the function scales the curvature**

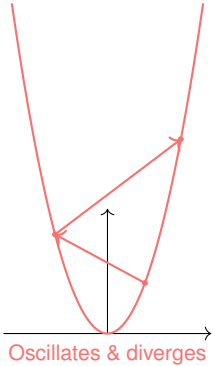
One Step Size Can Fail After Scaling

Same step size in both panels: $\alpha = 0.4$

$f(x) = x^2$



$g(x) = 3x^2$



Line Search (Simple Idea)

Goal: choose a step size $\alpha > 0$ that decreases f.

Basic idea:

- Start with a large step (e.g., $\alpha = 1$).
- If $f(x_k - \alpha \nabla f(x_k)) < f(x_k)$, accept it.
- Otherwise, shrink the step (e.g., $\alpha \leftarrow \beta\alpha$, $\beta \in (0, 1)$).
- Repeat until the function decreases.

This guarantees we move downhill without knowing the Lipschitz constant.
More advanced rules (e.g., Armijo, Wolfe) give stronger guarantees, but the core idea is just: **try a step and shrink until it decreases.**

Convergence Rates

How fast does gradient descent reach a solution?

Theorem: Smooth convex functions (Lipschitz gradient)

$$f(x_k) - f(x^*) \leq \frac{\|x_0 - x^*\|^2}{2\alpha k}.$$

Example, for $f(x) = 4x^2$, and $\alpha = 1/8$, and $x_0 = 0.5$, then after $K = 1000$ iterations, $f(x_k) \leq 0.001$.

But actually, $x_1 = x_0 - \frac{1}{8} \cdot 8x_0 = 0$. We find $x_1 = x^* = 0$ in one iteration. The bound here is an upper bound!

Application: Linear Regression Problem

We observe data:

$$(x_i, y_i), \quad i = 1, \ldots, m$$

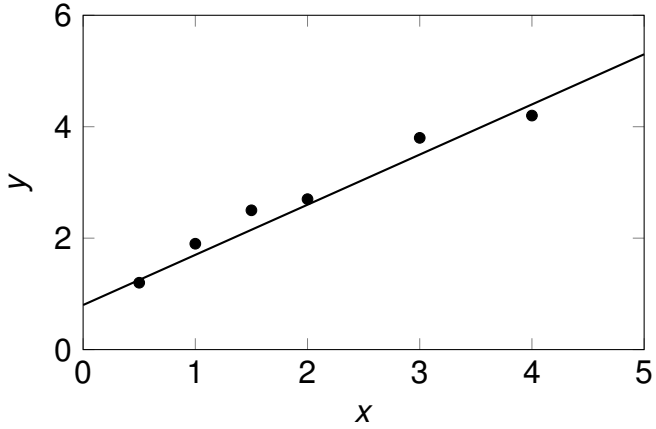
Goal: fit a line

$$y \approx \beta_0 + \beta_1 x.$$

We cannot usually fit all points exactly.
So we minimize squared residuals:

$$\min_{\beta_0, \beta_1} \sum_{i=1}^{m} \left(y_i - (\beta_0 + \beta_1 x_i)\right)^2.$$

Least Squares Fit



Minimize $\sum_i$(vertical errors)^2.

Matrix Form

Define

$$A = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}.$$

Then regression becomes

$$\min_{\beta} \|A\beta - y\|^2.$$

This is the general least squares problem.

Application: Least Squares

Setup: given data matrix $A \in \mathbb{R}^{m \times n}$, targets $b \in \mathbb{R}^m$.

$$\min_{x \in \mathbb{R}^n} f(x) = \|Ax - b\|^2.$$

Is f convex?

$$\nabla^2 f = 2A^\top A \succeq 0 \quad \checkmark$$

Gradient:

$$\nabla f(x) = 2A^\top (Ax - b).$$

Gradient descent step:

$$x_{k+1} = x_k - \alpha \cdot 2A^\top (Ax_k - b).$$

Note: Least squares has a closed form soluton of $x^* = (A^T A)^{-1} A^T b$. However, computing it takes $O(n^3)$ which can be slow for large n.

Gradient Descent from Scratch

```python
import numpy as np

def gradient_descent(A, b, alpha=0.01, n_iters=500):
    """
    Least Squares regression via gradient descent.
    Minimizes ||Ax - b||^2
    """
    m, n = A.shape
    x = np.zeros(n)                # x_0 = 0

    for k in range(n_iters):
        grad = 2 * A.T @ (A @ x - b)  # nabla f(x)
        x = x - alpha * grad                        # x_{k+1} = x_k - alpha * grad
    return x
np.random.seed(0)
m, n = 200, 50
A = np.random.randn(m, n)
b = np.random.randn(m)
x_gd = gradient_descent(A, b, alpha=1e-3, n_iters=1000)
```

The Problem with Hand-Derived Gradients

We previously computed ∇f by hand:

$$f(x) = \|Ax - b\|^2 \quad \Rightarrow \quad \nabla f(x) = 2A^\top(Ax - b).$$

That was easy. Now consider a 3-layer neural network loss function:

$$f(W_1, W_2, W_3) = \frac{1}{m}\sum_{i=1}^{m} \ell(\sigma(W_3\,\sigma(W_2\,\sigma(W_1 x_i + b_1) + b_2) + b_3),\ y_i)\,.$$

Where $\ell(x, y) = (x - y)^2$ and $\sigma(x) = 1/(1 + e^{-x})$.

Deriving gradients by hand is

- **Tedious.** Pages of chain rule.
- **Error-prone.** One sign error $\Rightarrow$ nothing works, hard to debug.

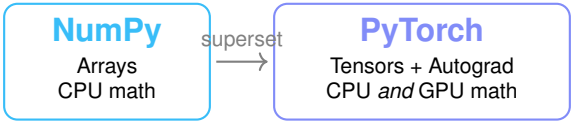
The fix: Automatic Differentiation ("autograd")

Write your loss as code. The framework computes ∇f for you, exactly.

What Is PyTorch?

PyTorch is a Python library for numerical computation with two superpowers:

1. **Tensors** — like NumPy arrays, but can live on GPUs and track gradients.
2. **Autograd** — automatic differentiation engine. You write the forward computation; PyTorch gives you gradients via the chain rule.



Why do we care?

Gradient descent needs $\nabla f(x)$. PyTorch computes it automatically, so we can focus on *modeling* instead of *calculus*.

PyTorch Tensors: NumPy → PyTorch

```
#!pip install torch
import numpy as np
import torch

# NumPy way
a_np = np.array([1.0, 2.0, 3.0])
b_np = np.array([4.0, 5.0, 6.0])
c_np = a_np @ b_np                      # dot product: 32.0

# PyTorch way (almost identical syntax!)
a_pt = torch.tensor([1.0, 2.0, 3.0])
b_pt = torch.tensor([4.0, 5.0, 6.0])
c_pt = a_pt @ b_pt                      # dot product: tensor(32.)

# Convert between them freely
a_pt = torch.from_numpy(a_np)           # numpy -> torch
a_np = a_pt.numpy()                     # torch -> numpy
```

Key point

If you know NumPy, you already know 90% of PyTorch's tensor API.
Same operations: `@`, `+`, `*`, `.T`, `reshape`, `sum`, indexing, slicing, ...

The Magic Switch: `requires_grad=True`

The key difference from NumPy: PyTorch tensors can **track their history**.

```
# This tensor will record every operation done to it
x = torch.tensor([2.0, 3.0], requires_grad=True)

# Any computation involving x builds a hidden "computation graph"
y = x[0]**2 + x[1]**2       # y = 4 + 9 = 13

# One call: compute ALL gradients via chain rule
y.backward()

# Result: dy/dx = [2*x[0], 2*x[1]] = [4, 6]
print(x.grad)                # tensor([4., 6.])
```

Gradient Descent: From Scratch in PyTorch

Let us solve ridge regression: $\min_x \|Ax - b\|^2 + \lambda\|x\|^2$.

```
import torch
torch.manual_seed(0)
m, n = 200, 50
A = torch.randn(m, n)
b = torch.randn(m)
lam = 1.0
x = torch.zeros(n, requires_grad=True)   # x_0 = 0
alpha = 1e-3

for k in range(100):
    # Forward: compute loss (PyTorch records the graph)
    loss = (A @ x - b) @ (A @ x - b) + lam * (x @ x)

    # Backward: compute gradients
    loss.backward() #Now x.grad has the gradient of loss with respect to x

    # Update: gradient descent step
    with torch.no_grad():           # don't track this arithmetic
        x -= alpha * x.grad
        x.grad.zero_()              # CRITICAL: clear gradients for next iteration
print(f"Final loss: {loss:.4f}")
```

Two Details That Trip Everyone Up

- Why do we need `with torch.no_grad()`?
- Why do we need `x.grad.zero_()`?

These two lines prevent:

1. Tracking *the update itself* (not what we want)
2. Accidentally accumulating gradients across iterations

Gradients Accumulate by Default

```python
import torch

x = torch.tensor(2.0, requires_grad=True)

f1 = x**2
f1.backward()
print(x.grad)   # ?
```

For $f(x) = x^2$, we expect:

$$\nabla f(x) = 2x = 4$$

So `x.grad = 4`.

Second Backward Without Zeroing

```
# Second backward pass (without zeroing!)
f2 = x**2
f2.backward()
print(x.grad)   # ?
```

Expected gradient at $x = 2$:

$$2x = 4$$

But PyTorch ADDS it:

$$x.grad = 4 + 4 = 8$$

Key Point

.backward() adds to .grad.

That's why we must reset gradients every iteration with x.grad.zero_().

What If We Forget `with torch.no_grad():`?

```
x = torch.tensor(2.0, requires_grad=True)
f = x**2
f.backward()
#with torch.no_grad(): <-- OOPS
x = x - 0.1 * x.grad
```

Now autograd tracks the update step itself.

Why this is a problem:

- $x_1 = x_0 - \alpha \cdot 2x_0$ becomes part of the graph
- Future gradients are wrong (they differentiate through the updates)

The update $x \leftarrow x - \alpha \nabla f(x)$ is **bookkeeping**, not part of the model or function.

Rule

Always use `with torch.no_grad():` for parameter updates.

The 3-Line Pattern You Always Write

```
loss.backward()              # compute gradients
with torch.no_grad():
    x -= alpha * x.grad      # update parameters
    x.grad.zero_()           # reset gradients
```

Memorize this pattern. You will write it hundreds of times.