CS498: Algorithmic Engineering

Lecture 11: PyTorch, Ridge Regression & Constrained Optimization Preview.

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 06 – 02/24/2026

Outline

Recap: What We Saw Last Time

Last lecture we introduced **PyTorch**: tensors + autograd.
We saw one example:

```
x = torch.tensor([2.0, 3.0], requires_grad=True)
y = x[0]**2 + x[1]**2       # y = 13
y.backward()
print(x.grad)                # tensor([4., 6.])
```

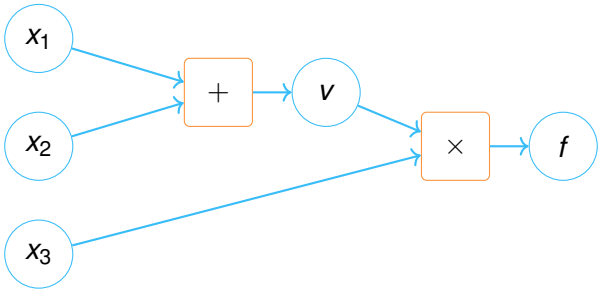
Today: how does this actually work inside? And how do we use it for real optimization?

Computational Graphs: The Big Idea

Every computation can be drawn as a **directed graph**:

- Nodes = intermediate values.
- Edges = operations.

Example: $f(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$



Chain Rule = The Engine of Backprop

Suppose $f = g(h(x))$. Chain rule:

$$\frac{df}{dx} = \frac{dg}{dh} \cdot \frac{dh}{dx}.$$

For a graph with intermediate node v:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial v} \cdot \frac{\partial v}{\partial x}.$$

Key insight

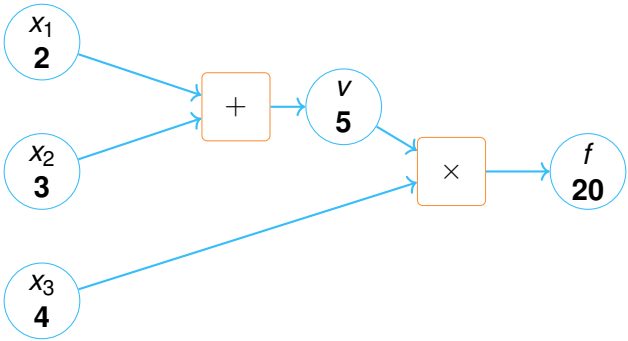
Each node only needs to know:

1. **Its local derivative** (e.g., $\frac{\partial v}{\partial x}$ for its own operation).
2. **The upstream gradient** $\frac{\partial f}{\partial v}$ arriving from the next node.

Multiply them. Pass the result upstream. That is **all of backpropagation**.

Concrete Example: Forward Pass (Values)

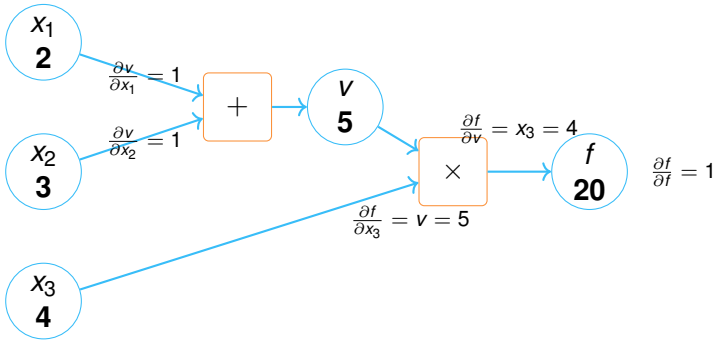
$f(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$ with $x_1 = 2,\ x_2 = 3,\ x_3 = 4.$



- $v = x_1 + x_2 = 5$
- $f = v \cdot x_3 = 20$

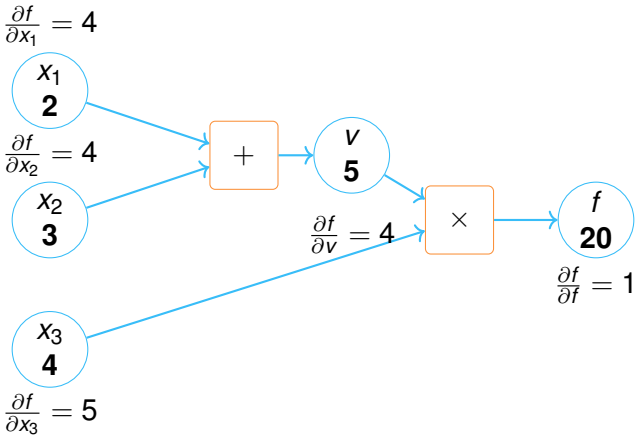
Concrete Example: Backward Pass (Local Derivatives)

Start from $\frac{\partial f}{\partial f} = 1$ and move backward.



Key idea: Each node contributes a small local derivative.

Concrete Example: Backward Pass (Gradients)



$$\frac{\partial f}{\partial x_1} = \frac{\partial f}{\partial v} \cdot \frac{\partial v}{\partial x_1} = 4 \qquad \frac{\partial f}{\partial x_2} = \frac{\partial f}{\partial v} \cdot \frac{\partial v}{\partial x_2} = 4 \qquad \frac{\partial f}{\partial x_3} = 5$$

Each Operation Stores Its Own Local Rule

The key design principle: **every operation knows how to differentiate itself**.

Operation	Forward	Local derivatives
Addition	$v = a + b$	$\frac{\partial v}{\partial a} = 1, \quad \frac{\partial v}{\partial b} = 1$
Multiply	$v = a \cdot b$	$\frac{\partial v}{\partial a} = b, \quad \frac{\partial v}{\partial b} = a$
Square	$v = a^2$	$\frac{\partial v}{\partial a} = 2a$
Exp	$v = e^a$	$\frac{\partial v}{\partial a} = e^a$
MatMul	$v = Wa$	$\frac{\partial v}{\partial a} = W$

To compute the gradient of any composite function, PyTorch just **chains** these local rules. That is all autograd does.

How PyTorch Builds the Graph

Every tensor remembers **how it was created**:

```
import torch

x = torch.tensor(3.0, requires_grad=True)

v = x ** 2          # v = 9.0
print(v.grad_fn)    # <PowBackward0>  -- "I was made by **2"

w = v + 1            # w = 10.0
print(w.grad_fn)    # <AddBackward0>  -- "I was made by +"

f = torch.log(w)     # f = log(10)
print(f.grad_fn)    # <LogBackward0>  -- "I was made by log"
```

What `f.backward()` does

Starting from f, it follows the chain of `.grad_fn` pointers backwards, applying each operation's local derivative rule, multiplying via the chain rule, and storing the final result in `x.grad`.

Adding a New Operation

What if PyTorch doesn't know an operation? You can teach it.

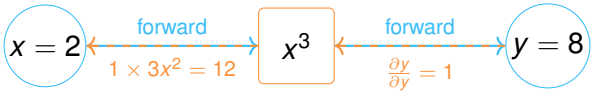
You define two things:

1. **Forward:** how to compute the output from the input.
2. **Backward:** the local derivative (how to pass gradients back).

```
class MyCube(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)   # stash x for the backward pass
        return x ** 3              # forward: f(x) = x^3

    @staticmethod
    def backward(ctx, grad_output):
        x, = ctx.saved_tensors
        return grad_output * 3 * x**2   # f/x = f/MyCube · MyCube/x =
x = torch.tensor(2.0, requires_grad=True)
y = MyCube.apply(x)   # y = 8.0
y.backward()
print(x.grad)          # tensor(12.)  = 3 * 4
```

Adding a New Op: What Just Happened



The contract

Forward: you tell PyTorch what $f(x)$ returns.
Backward: you tell PyTorch the local derivative $f'(x)$.
PyTorch handles the rest: chaining, accumulation, storage.

In practice, you rarely need custom ops: PyTorch already knows hundreds of operations. But it's good to know the mechanism.

Warning: 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`.

Warning: 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_().

Recall: The Linear Regression Problem

We observe data:

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

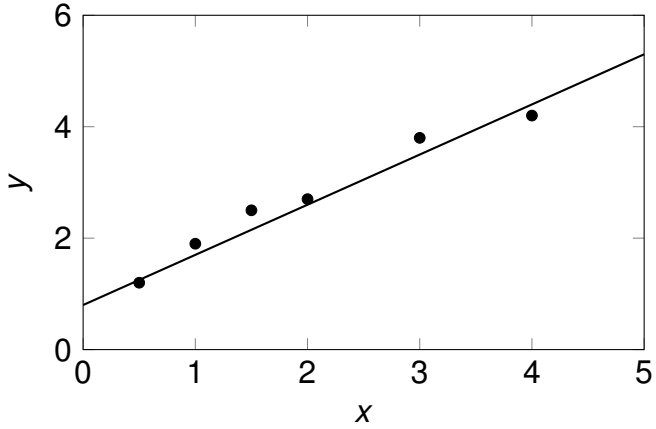
Goal: fit a line

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

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

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

Recall: Least Squares Fit



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

Recall: Matrix Form and Gradient

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

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

Gradient (from last lecture):

$$\nabla f(\beta) = 2A^\top (A\beta - y).$$

Last time

We implemented GD for this problem using NumPy, computing ∇f by hand. **Now:** let PyTorch compute the gradient automatically.

Vanilla Autograd: Write the Update Yourself

```python
import torch

x = torch.zeros(n, requires_grad=True)
alpha = 1e-3   # step size

for k in range(1000):
    if x.grad is not None:
        x.grad.zero_()          # clear old gradients

    loss = (A @ x - b) @ (A @ x - b)
    loss.backward()             # compute gradient

    # gradient descent update
    x.data -= alpha * x.grad
```

What's happening

`loss.backward()` computes ∇_x`loss` and stores it in `x.grad`.
`x.data -= alpha * x.grad` performs the update manually.

`torch.optim`: Don't Write the Loop Yourself

PyTorch packages the update step into **optimizers**:

```
import torch
import torch.optim as optim

x = torch.zeros(n, requires_grad=True)
optimizer = optim.SGD([x], lr=1e-3, momentum=0.0, weight_decay=0.0)    # basic gradient descent

for k in range(1000):
    optimizer.zero_grad()                # zero gradients

    loss = (A @ x - b) @ (A @ x - b)
    loss.backward()                      # compute gradients
    optimizer.step()                     # x -= lr * x.grad  (handled for you)
```

What changed

`optimizer.zero_grad()` replaces `x.grad.zero_()`.

`optimizer.step()` replaces the manual `x -= alpha * x.grad`.

Same math. Cleaner code. And you can swap in fancier optimizers (`Adam`, `RMSProp`, `LBFGS`) by changing one line.

Least Squares: What Can Go Wrong?

Recall the closed-form solution:

$$\hat{\beta} = (A^\top A)^{-1} A^\top y.$$

Let's start with the simplest case: **one variable**.

$$\min_\beta \sum_i (y_i - \beta x_i)^2 \quad \Rightarrow \quad \hat{\beta} = \frac{\sum_i x_i y_i}{\sum_i x_i^2}.$$

$(A^\top A)^{-1}$ is just $\frac{1}{\sum x_i^2}$ here: it's **division**.

What if $\sum x_i^2$ is very small (features close to zero)?

$$\hat{\beta} = \frac{\text{something}}{\text{tiny number}} = \textbf{huge.}$$

A tiny change in y gets amplified into a massive change in $\hat{\beta}$.

The Same Problem in Higher Dimensions

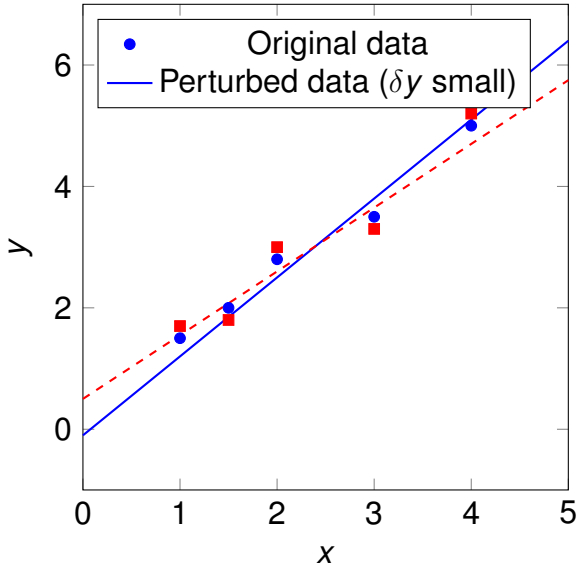
In n dimensions, $(A^\top A)^{-1}$ involves dividing by the **eigenvalues** of $A^\top A$.
If $A^\top A$ has eigenvalues $\sigma_1^2 \geq \sigma_2^2 \geq \cdots \geq \sigma_n^2$:

$(A^\top A)^{-1}$ divides by each σ_i^2. (think about diagonalization)

If the smallest eigenvalue $\sigma_n^2 \approx 0$ (columns of A nearly collinear):

$\hat{\beta}$ blows up in the direction of that eigenvector.

The Same Problem in Higher Dimensions



The Fix: Stop Dividing by Tiny Numbers

1D intuition:

$$\hat{\beta} = \frac{\sum x_i y_i}{\sum x_i^2} \quad \longrightarrow \quad \hat{\beta}_{\text{ridge}} = \frac{\sum x_i y_i}{\sum x_i^2 + \lambda}.$$

Even if $\sum x_i^2$ is tiny, we're now dividing by tiny $+ \lambda$, which is at least λ.

In n dimensions: ridge regression replaces $A^\top A$ with $A^\top A + \lambda I$:

$$\hat{\beta}_{\text{ridge}} = (A^\top A + \lambda I)^{-1} A^\top y.$$

The eigenvalues of $A^\top A + \lambda I$ are $\sigma_i^2 + \lambda$.

Key insight

λ puts a **floor** under every eigenvalue. We never divide by anything smaller than λ. No blow-up. Stable $\hat{\beta}$.

Ridge Regression: Gradient and GD Step

Penalize values of x that "blow up" or are large.

$$\min_{x \in \mathbb{R}^n} f_\lambda(x) = \|Ax - b\|^2 + \lambda \|x\|^2$$

Gradient:

$$\nabla f_\lambda(x) = 2A^\top (Ax - b) + 2\lambda x = 2(A^T A + \lambda I)x - 2A^T b.$$

$$\nabla f_\lambda(x) = 0 \iff (A^T A + \lambda I)x = A^T b \iff x = (A^T A + \lambda I)^{-1} A^T b.$$

Gradient descent step:

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

But with PyTorch, we don't need this formula at all.

Ridge Regression in PyTorch

```python
import torch
import torch.optim as optim

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)
optimizer = optim.SGD([x], lr=1e-3)

for k in range(1000):
    optimizer.zero_grad()
    loss = (A @ x - b) @ (A @ x - b) + lam * (x @ x)   # ridge loss
    loss.backward()
    optimizer.step()

print(f"Final loss: {loss.item():.4f}")
```

Observation

Compared to least squares, we added **ten characters**: `+lam*(x@x)`.

Effect of λ

$\lambda = 0$	λ **just right**	λ **too large**
Ordinary least squares. No regularization.	Stable $\hat{\beta}$. Still fits the data well.	Shrinks $\hat{\beta} \to 0$. Ignores the data.
Can be unstable when $A^\top A$ is ill-conditioned.	Best of both worlds.	Underfits.

Bigger picture

This idea: **penalizing complexity to get a better solution** is one of the most important ideas in machine learning. It will return when we train neural networks in a few weeks.

Everything So Far: Unconstrained

So far today, we solved:

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

No constraints. The gradient descent step is always valid.
But real problems almost always have constraints.

Example

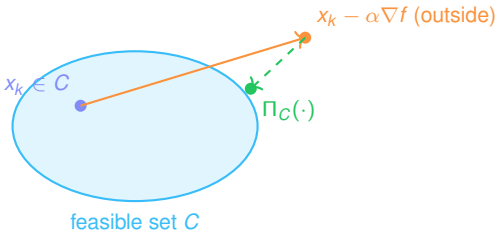
Part I was *entirely* about constrained optimization (LP, IP).
We now have gradient tools (gradient descent, PyTorch). How do we put them together?

Approach 1: Projected Gradient Descent

Idea: take a gradient step, then **project** back onto the feasible set C.

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

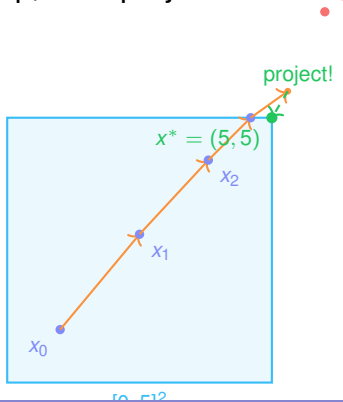
where $\Pi_C(z) = \arg\min_{x \in C} \|x - z\|$ is the **Euclidean projection** onto C.



Example: Minimizing over a Box

$$\min_{x \in [0,5]^2} f(x) = (x_1 - 6)^2 + (x_2 - 7)^2.$$

Unconstrained optimum: $x^* = (6, 7)$ — outside the box!
Projected GD: gradient step, then project.



Projection onto a Box

Constraint:

$$x \in [\mathbf{l}, \mathbf{u}] = [l_1, u_1] \times \cdots \times [l_n, u_n].$$

Projection problem:

$$\Pi(z) = \arg\min_{x \in [\mathbf{l}, \mathbf{u}]} \|x - z\|^2 = \arg\min_{x \in [\mathbf{l}, \mathbf{u}]} \sum_{i=1}^{n} (x_i - z_i)^2.$$

Key fact: Each coordinate projects independently:

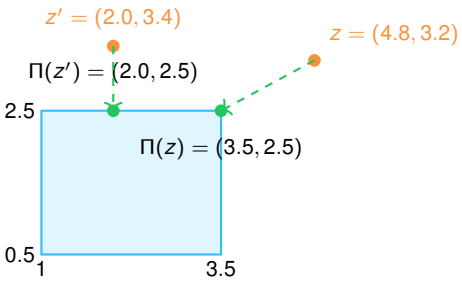
$$\Pi(z)_i = \mathrm{clip}(z_i, l_i, u_i) = \min(u_i,\ \max(l_i,\ z_i)).$$

Cost: $O(n)$
PyTorch: `x.clamp(min=l, max=u)`

Geometric Example

Example box:

$$[l, u]^2 = [1, 3.5] \times [0.5, 2.5]$$



$$\Pi(4.8, 3.2) = (3.5, 2.5), \qquad \Pi(2.0, 3.4) = (2.0, 2.5)$$

Example: Box Projection in PyTorch

```
import torch

x = torch.tensor([1.0, 1.0], requires_grad=True)
alpha = 0.1

for k in range(100):
    loss = (x[0] - 6)**2 + (x[1] - 7)**2

    loss.backward()

    with torch.no_grad():
        x -= alpha * x.grad           # gradient step
        x.clamp_(min=0.0, max=5.0)    # project onto [0, 5]^2
        x.grad.zero_()

print(x)   # tensor([5., 5.])
```

Observation

One extra line: `x.clamp_(min=0.0, max=5.0)`. That's the entire projection step.

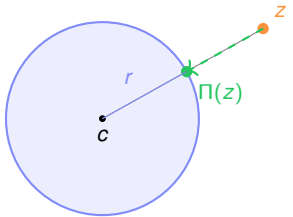
Projection onto a Ball

Constraint: $\|x - c\| \leq r$ (Euclidean ball).

Projection: if inside, stay. If outside, walk from c toward z, stop at radius r.

$$\Pi(z) = \begin{cases} z & \text{if } \|z - c\| \leq r, \\ c + r \cdot \dfrac{z - c}{\|z - c\|} & \text{if } \|z - c\| > r. \end{cases}$$

Why? The closest point on a sphere to z lies along the ray from c through z.

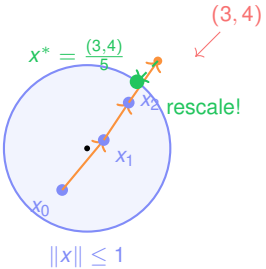


Cost: $O(n)$. Compute $\|z - c\|$, rescale if needed.

Example: Minimizing over a Ball

$$\min_{\|x\| \le 1} \; f(x) = (x_1 - 3)^2 + (x_2 - 4)^2.$$

Unconstrained optimum: $(3, 4)$ — far outside the unit ball.
Projected GD: gradient step, then rescale to $\|x\| \le 1$ if needed.



Example: Ball Projection in PyTorch

```python
import torch

x = torch.tensor([-0.3, -0.5], requires_grad=True)
r = 1.0
alpha = 0.1

for k in range(100):
    loss = (x[0] - 3)**2 + (x[1] - 4)**2

    loss.backward()

    with torch.no_grad():
        x -= alpha * x.grad                          # gradient step
        norm = torch.norm(x)
        if norm > r:
            x *= r / norm                             # project onto ball
        x.grad.zero_()

print(x)   # tensor([0.6000, 0.8000])
```

Same pattern: gradient step, then project. Two extra lines.

When Projection is Hard

Box and ball projections are cheap and have closed forms.

But consider **many coupled constraints**:

$$C = \{x : Ax \leq b,\ Cx = d,\ x \geq 0\}$$

Projection requires solving a **quadratic program** at every step:

$$\Pi_C(z) = \arg \min_{Ax \leq b,\, Cx = d,\, x \geq 0} \|x - z\|^2.$$

Nonlinear constraints $g(x) \leq 0$: projection may not even have a tractable formulation.

The problem

When the feasible set is complex, projection is too expensive to call at every iteration.
We need a fundamentally different approach to handle constraints.

Approach 2: Penalty Methods

Idea: remove the constraints, but **punish violations** in the objective.

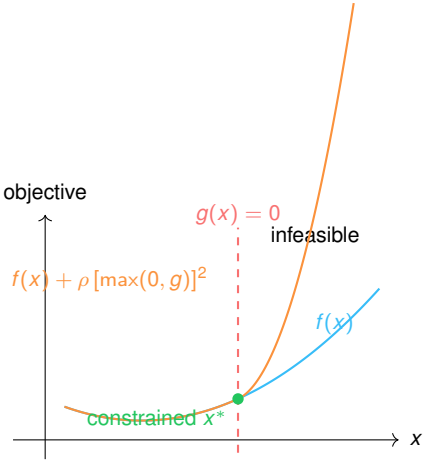
Original problem:

$$\min_x f(x) \quad \text{subject to} \quad g(x) \leq 0.$$

Penalty reformulation:

$$\min_x f(x) + \rho \cdot \left[\max(0,\ g(x))\right]^2, \qquad \rho > 0.$$

Approach 2: Penalty Methods



Penalty Methods: The Tradeoff

As $\rho \to \infty$, the penalty solution approaches the true constrained solution.

Pros:

- Turns a constrained problem into an unconstrained one.
- We can use gradient descent (or PyTorch) directly.

Cons:

- Need large ρ for accuracy $\Rightarrow$ objective becomes ill-conditioned.
- Gradient descent slows down as ρ grows (steep penalty, tiny steps).

Can we do better?

Is there a principled way to handle constraints that doesn't require $\rho \to \infty$?

Teaser: Lagrangian Duality

Instead of a brute-force penalty, introduce a **Lagrange multiplier** $\mu \geq 0$:

$$\mathcal{L}(x, \mu) = f(x) + \mu\, g(x).$$

This looks like the penalty method, but μ is not a fixed large number, it is a **variable** that we optimize over.

Key idea: instead of cranking up a penalty, we find the *right price* μ^* for violating the constraint.

Next lecture

- Lagrangian duality and KKT conditions.
- Connects Part I (LP duality) with Part II (gradient methods).
- A principled framework for constrained nonlinear optimization.