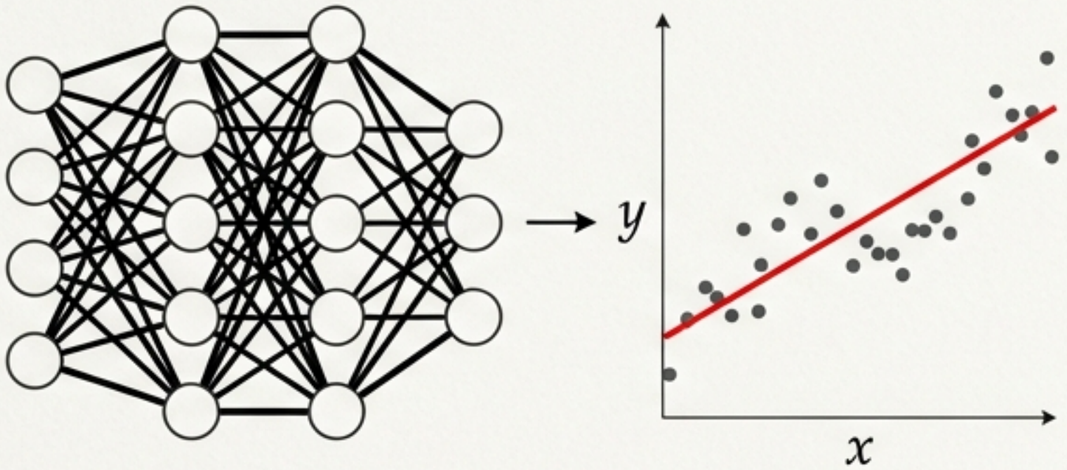
Gaussian Processes: The Art of Uncertainty

A probabilistic approach to modeling distributions over functions.

Moving from parameters to functions

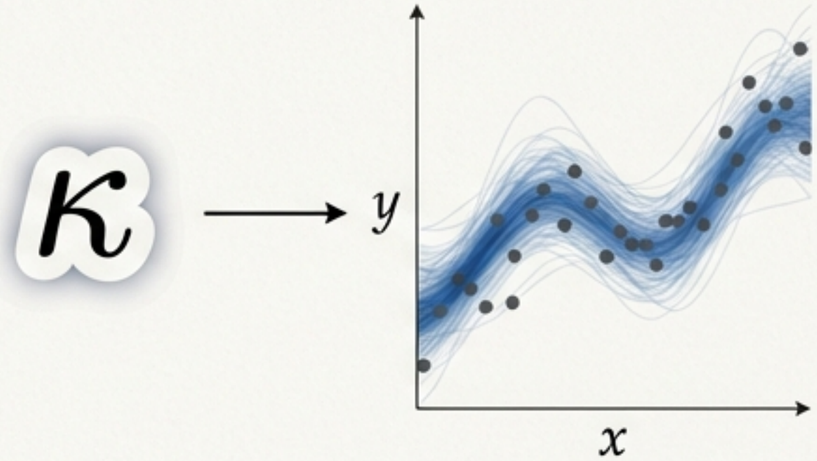
Standard ML (Neural Networks)



Distribution over Parameters (Weights)

Result: A single, deterministic curve.

Gaussian Processes



Distribution over Functions

Result: Infinite possible curves, weighted by probability.

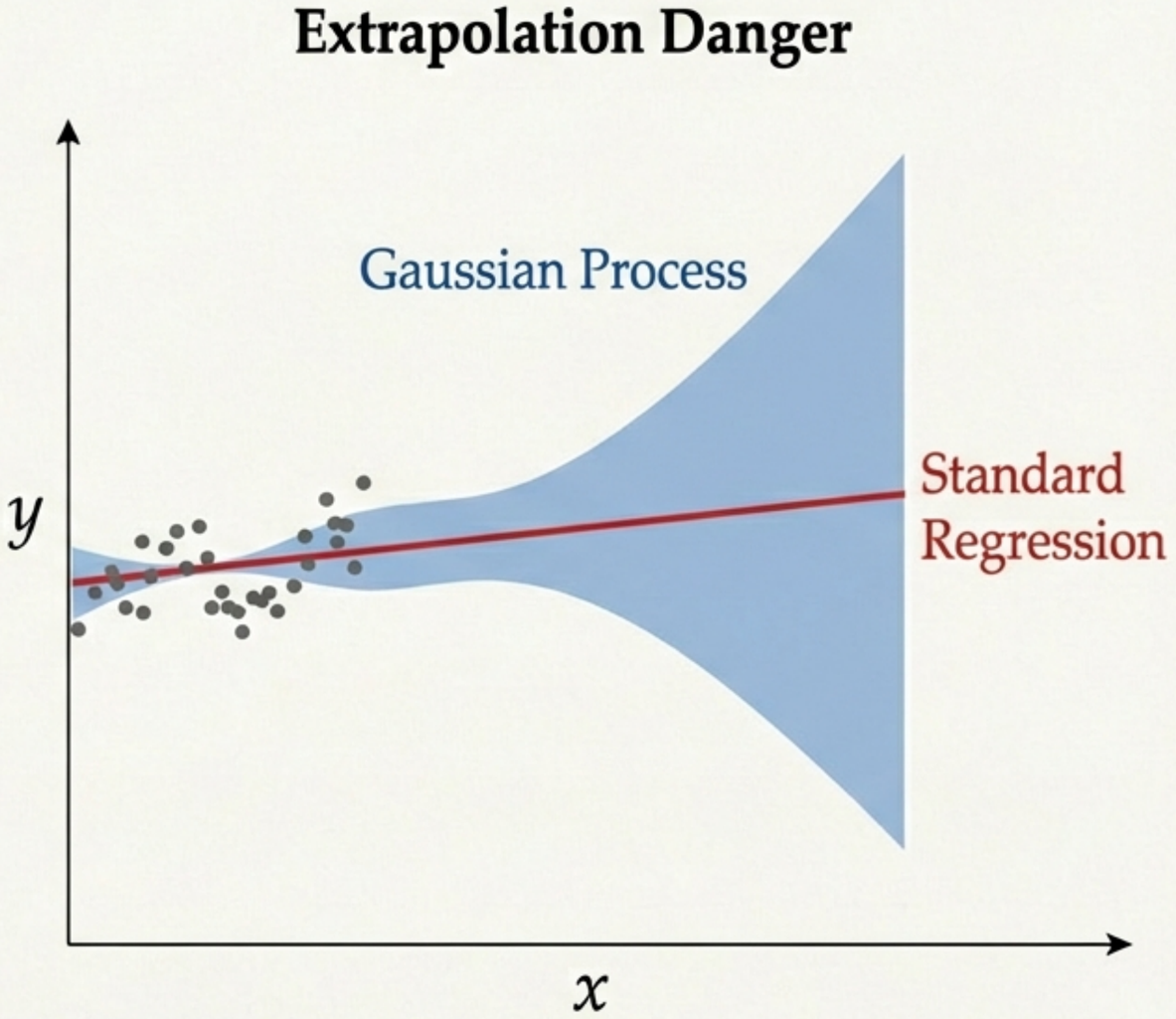
Neural networks → deterministic function; Gaussian Process → distribution over functions.

When “I don’t know” is the most valuable prediction.

The Problem: In real-world scientific modeling, data is often sparse, noise is high, and decisions are expensive.

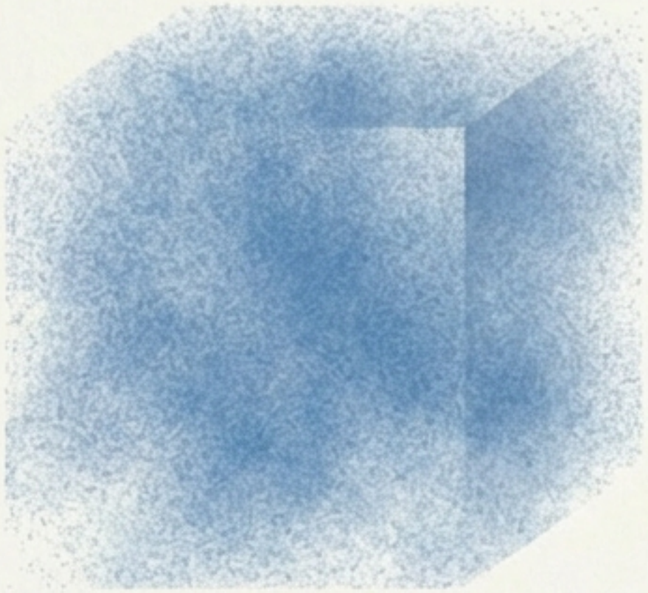
The GP Superpower: Unlike standard neural nets, GPs explicitly model their own ignorance. Uncertainty increases as we move away from observed data.

Key Applications: Bayesian Optimization, Active Learning, and modeling small-to-medium datasets where data efficiency is paramount.



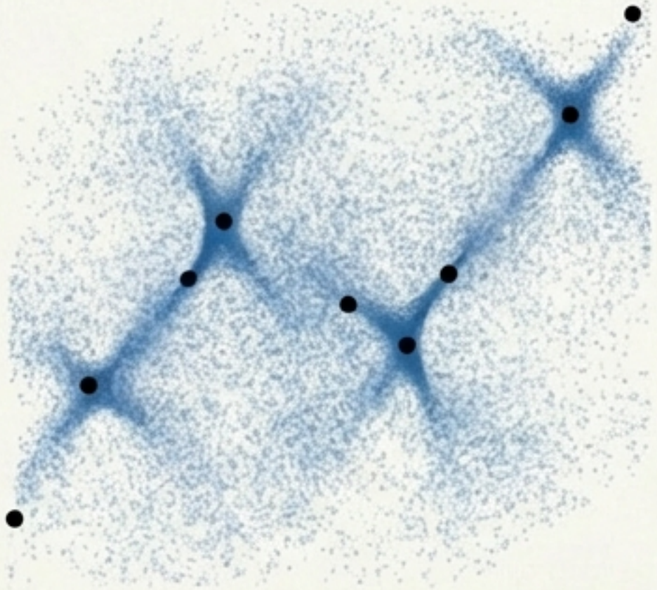
Sculpting probability from a block of clay.

The Prior
(The Block)



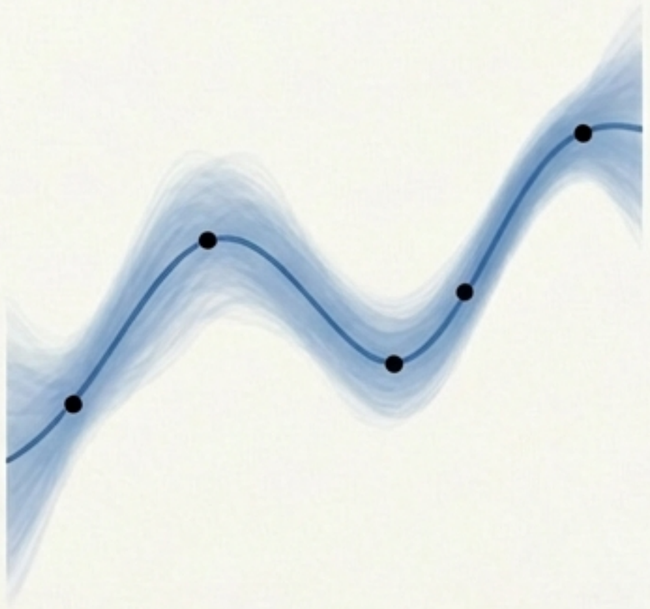
Assumptions before data.
“Functions are smooth.”

The Observation
(The Hands)



Data points act as constraints.
We condition the prior.

The Posterior
(The Sculpture)

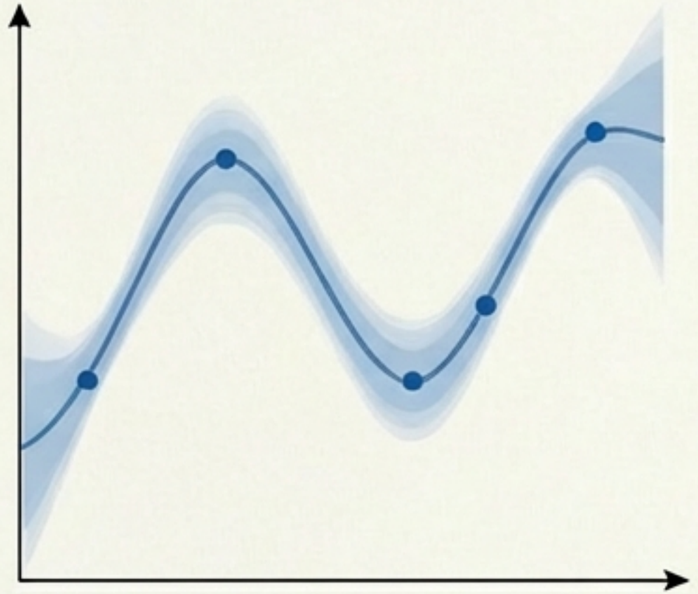


The distribution collapses.
Result: Prediction + Doubt.

The Kernel is the model.

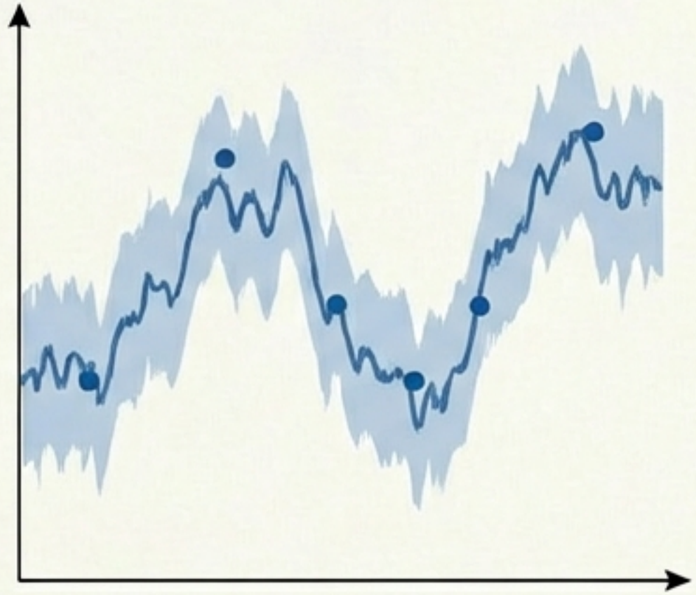
GPs don’t “discover” structure unless the kernel allows it.

RBF (Squared Exponential)



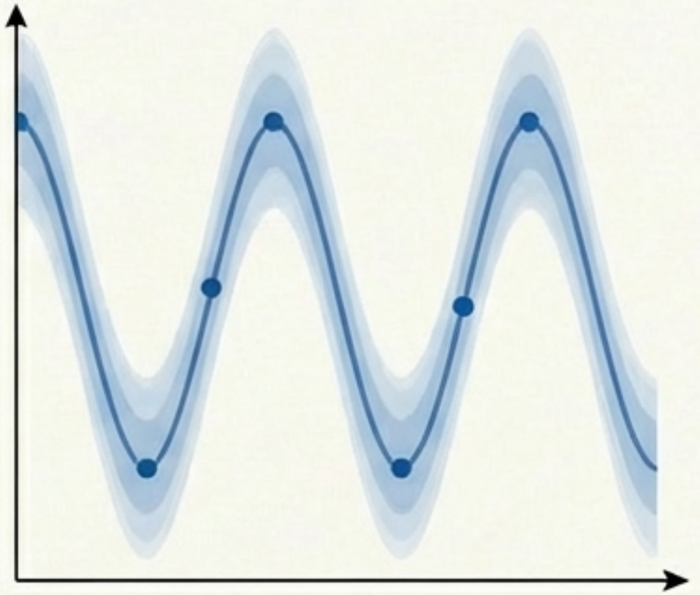
Assumes infinite smoothness.
Great for interpolation.

Matérn Kernel



Controls roughness.
Realistic for physical systems.

Periodic Kernel



Encodes seasonality and
repeating cycles.

Choosing the wrong kernel is equivalent to choosing the wrong model.

The Mathematical Definition.

$$f(x) \sim \mathrm{GP}(m(x), k(x, x'))$$

$m(x)$

The Mean Function.

Usually assumed to be 0 for simplicity, representing a neutral baseline.

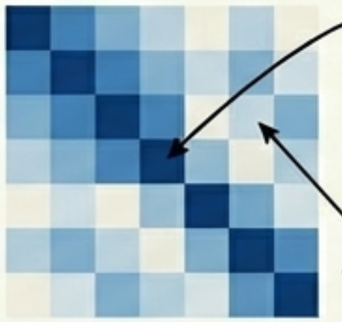
$k(x, x')$

The Covariance (Kernel).

This function defines the similarity between any two points in the input space.

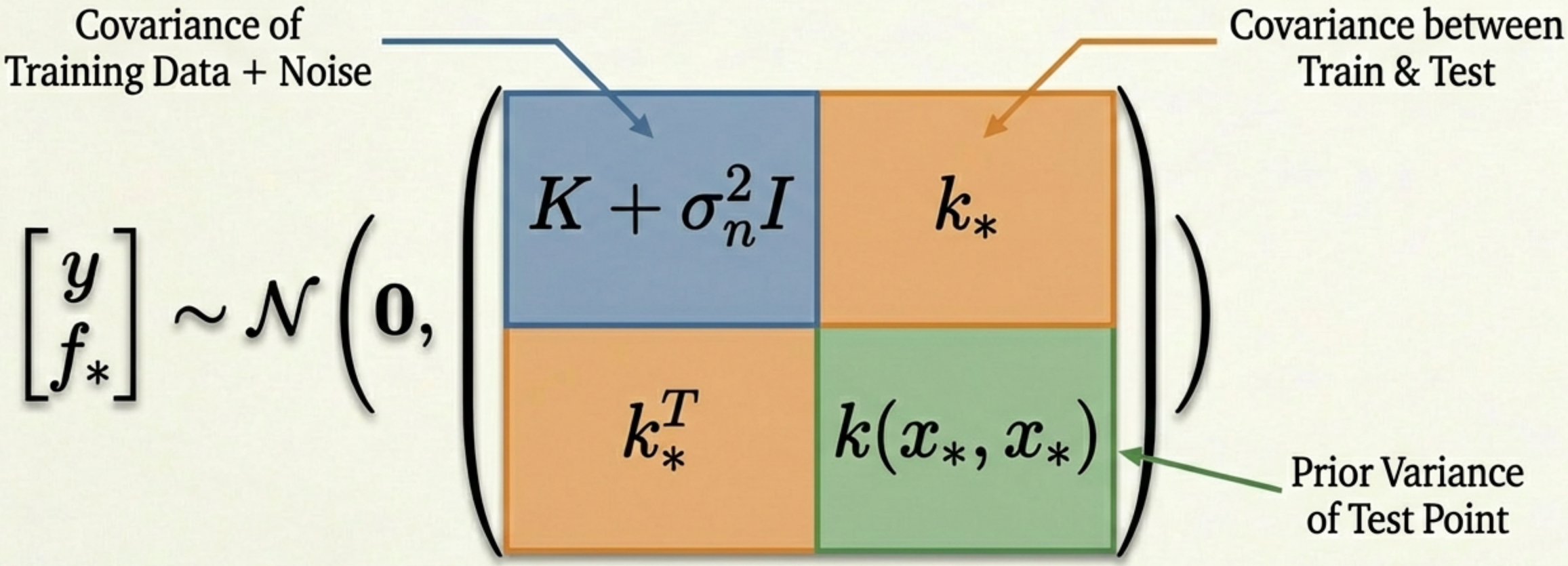
Matrix K

When applied to data, the kernel produces a matrix.



Conditioning the Joint Distribution.

We do not “fit” weights. We construct a Joint Distribution containing both the observed data (Training) and the unknown points (Test).



The Prediction Equations.

Calculating the mean and variance for a new point x^*.

1. **Posterior Mean** (The Prediction)

$$\mu^* = k_*^T (K + \sigma_n^2 I)^{-1} y$$

Interpretation: A similarity-weighted sum of observed values. It projects the training targets onto the new point based on closeness.

---

2. **Posterior Variance** (The Uncertainty)

$$\sigma^{*2} = k(x^*, x^*) - k_*^T (K + \sigma_n^2 I)^{-1} k_*$$

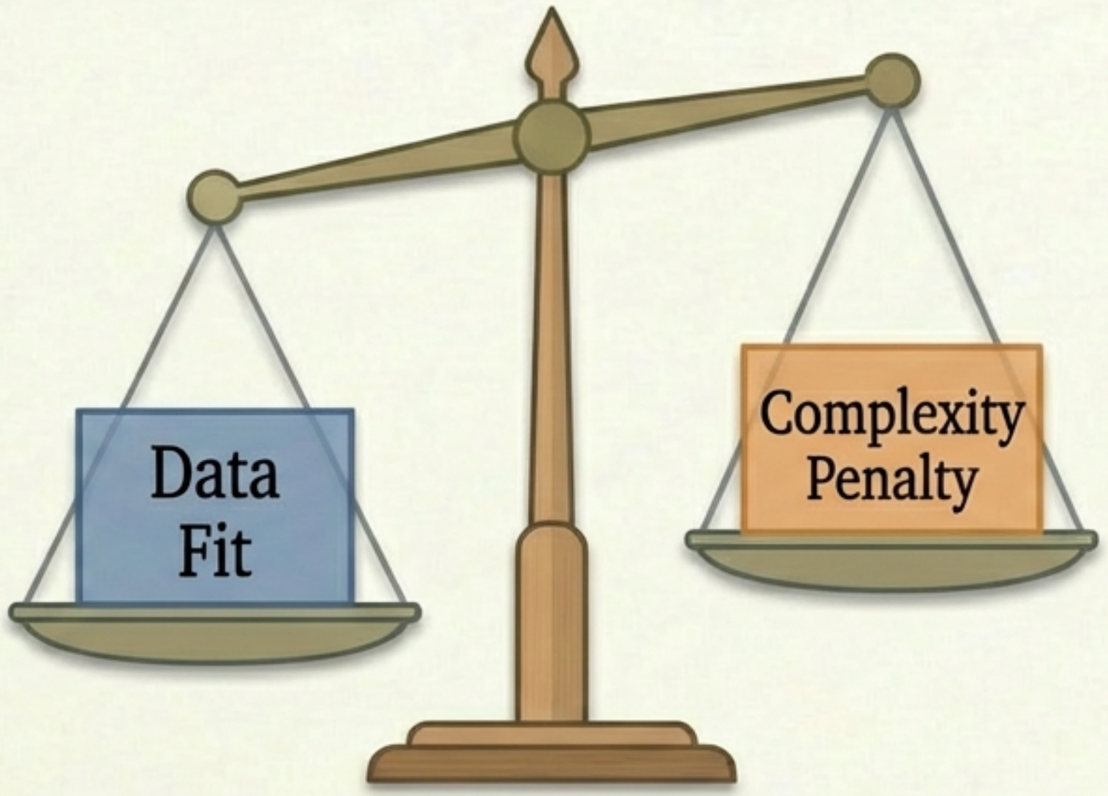
Interpretation: The prior uncertainty minus the information gained from data.

Notice that variance depends only on inputs X, not targets y.

Learning Hyperparameters & Occam's Razor.

We learn kernel parameters (e.g., length scale) by maximizing the **Log Marginal Likelihood:**

$$\log p(y|X)$$



Term 1: **Data Fit.** Rewards the model for explaining the data well.
Term 2: **Complexity Penalty.** Penalizes the model for being too flexible.

"This acts as an automatic **Occam's razor**, balancing flexibility with simplicity."

Implementation: The High-Level Abstraction.

Using `sklearn.gaussian_process`.

```
kernel = 1.0 * RBF(length_scale=1.0) +
    WhiteKernel(noise_level=0.1)

gp = GaussianProcessRegressor(
    kernel=kernel,
    n_restarts_optimizer=5,
    normalize_y=True

gp.fit(X, y)
```

Models observation noise (σ_n^2).

Essential for numerical stability.

Prevents getting stuck in local optima.

Under the Hood: Pure NumPy.

The math implies Linear Algebra.

Math Concepts

1. Compute Cholesky factor L of covariance matrix.
2. Solve linear system.
3. Compute posterior.

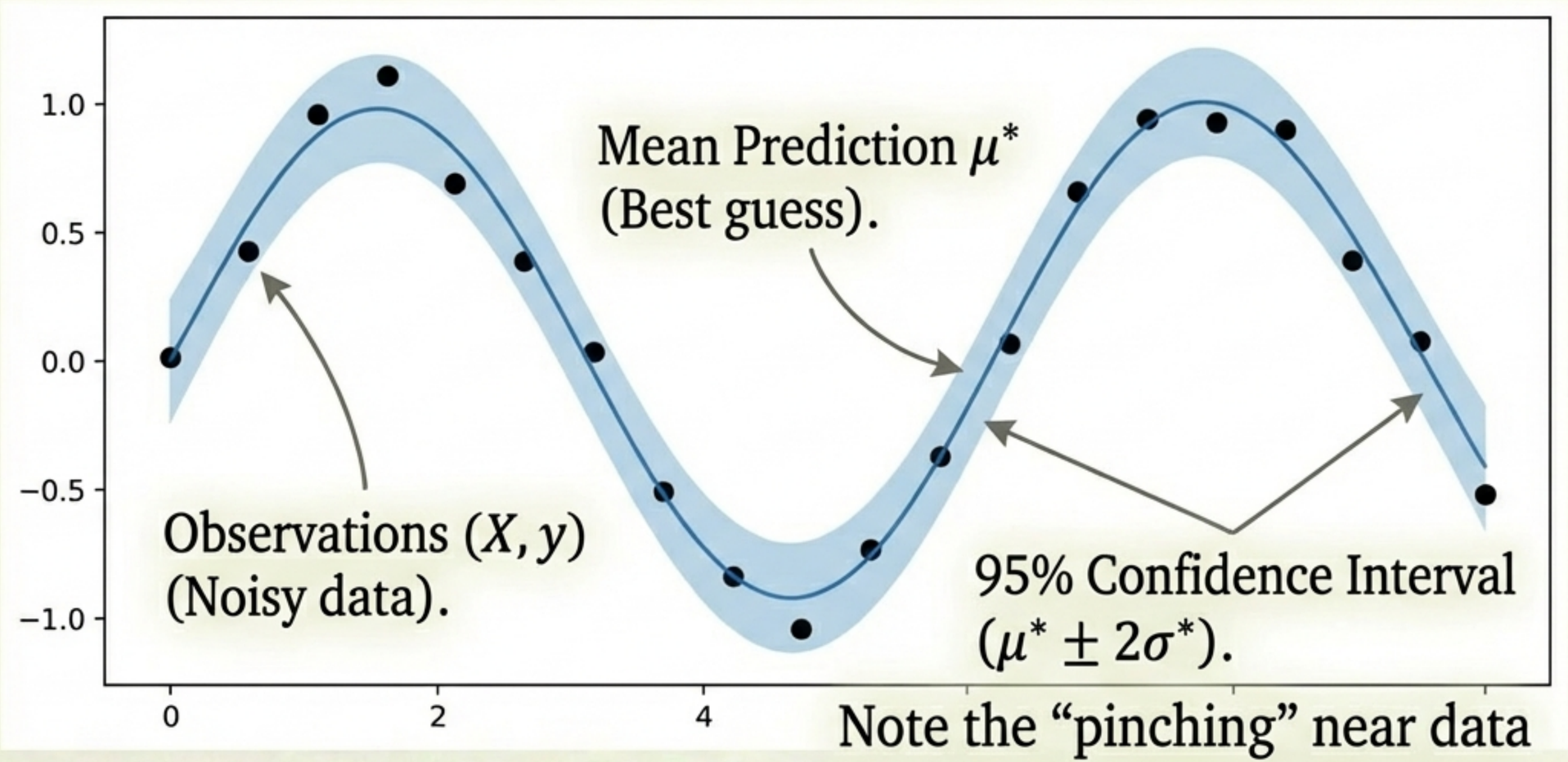
Code Implementation

```
# Cholesky decomposition (efficient & stable)
L = np.linalg.cholesky(K + noise * np.eye(N))

# Solve for alpha without inverting
alpha = np.linalg.solve(L.T, np.linalg.solve(L, y_train))

# Compute Posterior Mean and Variance
mu = K_s.T @ alpha
v = np.linalg.solve(L, K_s)
cov = K_ss - v.T @ v
```

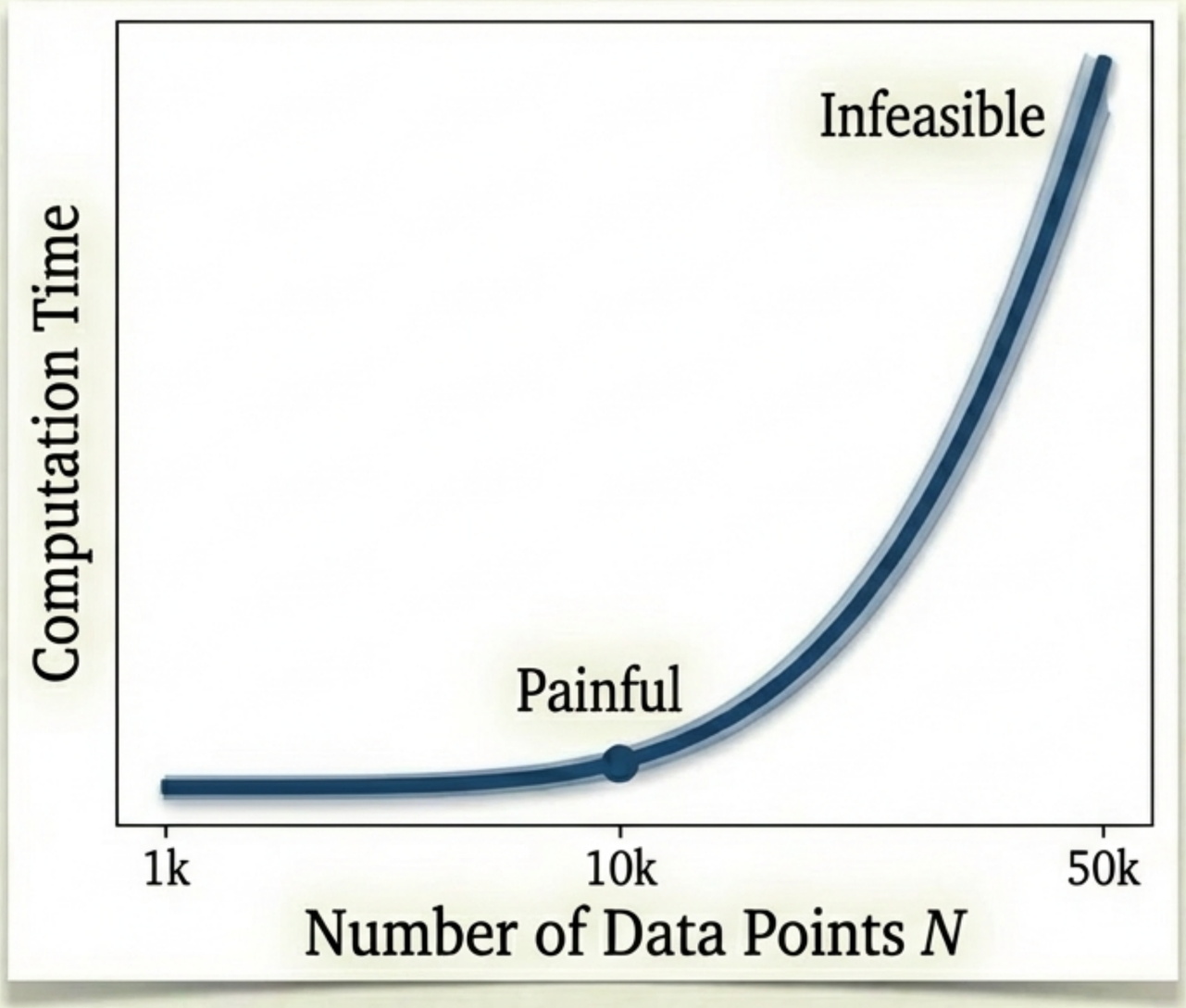
Key Takeaway: We use `np.linalg.cholesky` instead of `inv()` for $O(n^3)$ efficiency and stability.

Visualizing the Posterior.

How to read the results of a Gaussian Process.



The Computational Reality.



The Bottleneck:
Matrix inversion (via Cholesky) scales cubically.
Time Complexity: $O(n^3)$
Memory Complexity: $O(n^2)$

The Workarounds:
For large data, we must use approximations like Sparse GPs or Inducing Points (Variational Inference).

Choosing the Right Tool.

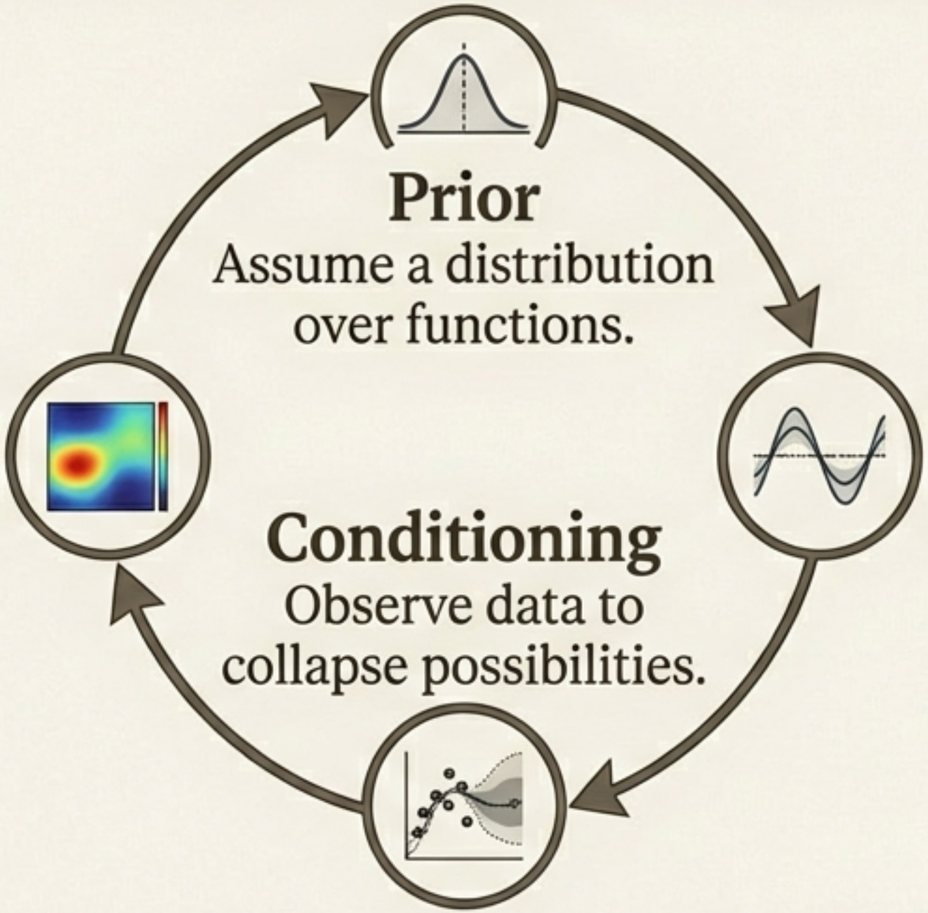
Use Gaussian Processes When...

- Data is limited (Small-to-medium datasets).
- Uncertainty estimates are critical (e.g., medical, financial).
- Decisions are expensive (Bayesian Optimization).
- Interpretability of assumptions is required.

Avoid Gaussian Processes When...

- Millions of training samples exist.
- Inputs are high-dimensional raw data (images, audio, text).
- Real-time inference at scale is required.

Summary: Exact Bayesian Inference.



Prior
Assume a distribution over functions.

Kernel
Encode beliefs (smoothness, periodicity).

Conditioning
Observe data to collapse possibilities.

Posterior
Result is a calibrated probability field.

A GP is not ‘better’ or ‘worse’ than Deep Learning; it solves a different class of problems where data is scarce and certainty is a luxury.