Learning Functions from Data

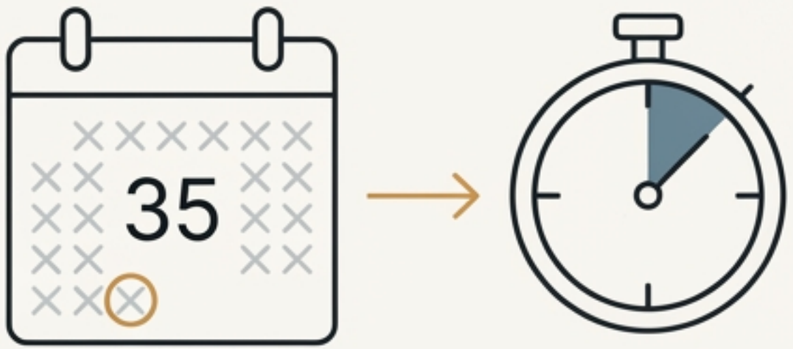
From Neurons to Emulators

> What I cannot create, I do not understand.
>
> — Richard Feynman (found on his blackboard at the time of his death, 1988)

The Computational Bottleneck: A 35-Day Problem

Scientific inference often requires thousands of model evaluations. For complex simulations like N-body dynamics, this is computationally prohibitive.

$$\underbrace{2000}_{\text{MCMC samples}} \times \underbrace{50}_{\text{leapfrog steps}} \times \underbrace{30\ \text{sec}}_{\text{per sim}} = 35\ \text{days}$$



That's for a *single* inference run.
This is unacceptable.

The Solution: Emulators as Learned Compression

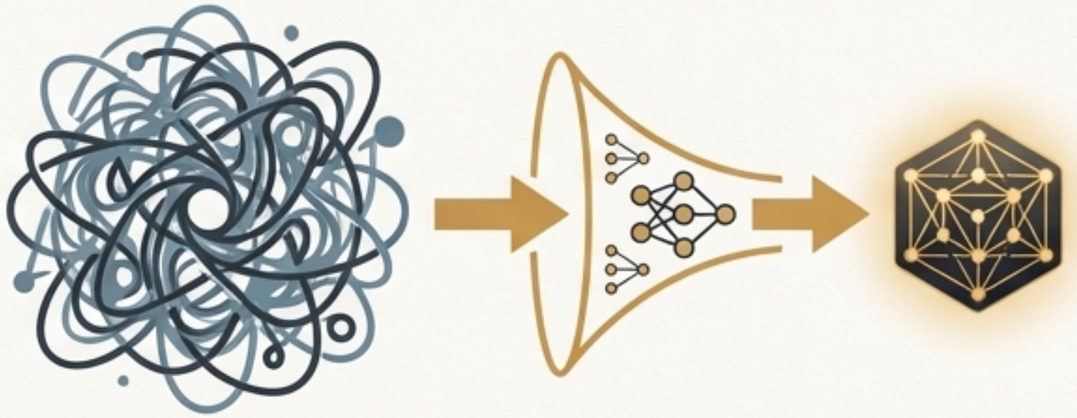
Definition:

An **Emulator** (or **surrogate model**) is a fast approximation of an expensive computation, trained on input-output pairs from the original model.

> **Connection to Module 1: Models as Compression**
>
> Summary statistics compress simulation outputs. An emulator compresses the *entire input-output relationship* of the simulator into a set of neural network weights. It's a learned compression that automatically discovers a low-dimensional representation of a complex physical process.

The Payoff:

- **Train:** on ~100 N-body simulation runs.
- **Predict:** in microseconds instead of seconds.
- **Result:** Inference takes minutes, not months.



The Leap from Engineered Features to Learned Features

1. Level 1: Linear Regression

Simple but limited.

$$\hat{y} = w^T x + b$$

2. Level 2: Basis Regression

More flexible, but requires human guesswork.

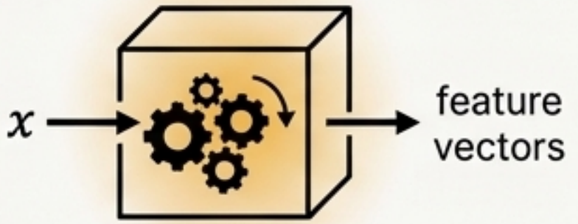
$$\hat{y} = w_0 + w_1 x + w_2 x^2 + \cdots$$

The challenge: You must manually *choose* features like Q_0^2, $\log a$, or $Q_0 \times a$.

3. Level 3: Neural Networks

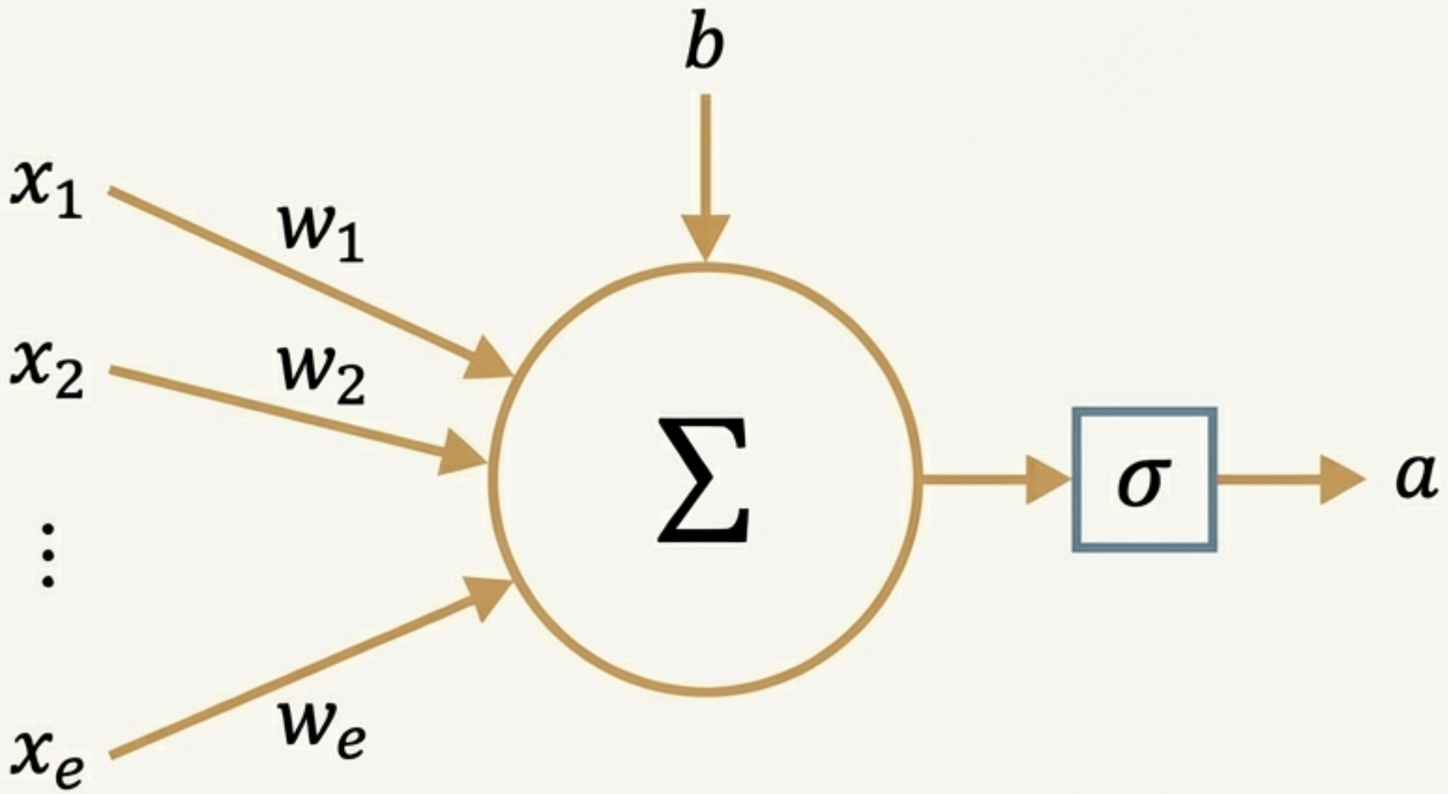
The key insight—**what if we *learned* the features?**

$$\hat{y} = w_{\text{out}}^T \cdot \underbrace{\sigma(W_{\text{hidden}} x + b_{\text{hidden}})}_{\text{Learned Features}} + b_{\text{out}}$$



Traditional Statistics	Neural Networks
Human chooses features → Algorithm finds weights.	Algorithm finds features *and* weights.

The Anatomy of a Computational Neuron



Weights (w): Control the influence of each input.

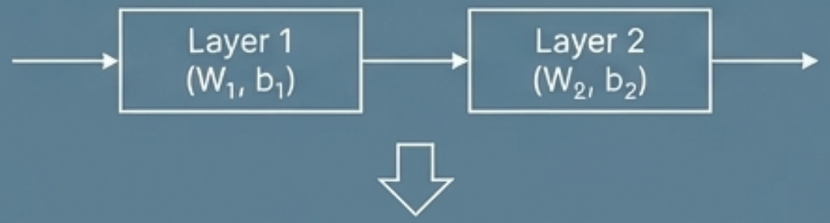
Bias (b): Shifts the activation threshold.

Pre-activation (z): The linear sum, $z = \mathbf{w}^T\mathbf{x} + b$.

Activation Function (σ): Introduces essential **nonlinearity**.

$$a = \sigma\left(\sum_{j=1}^{d} w_j x_j + b\right) = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

Why Nonlinearity is Essential



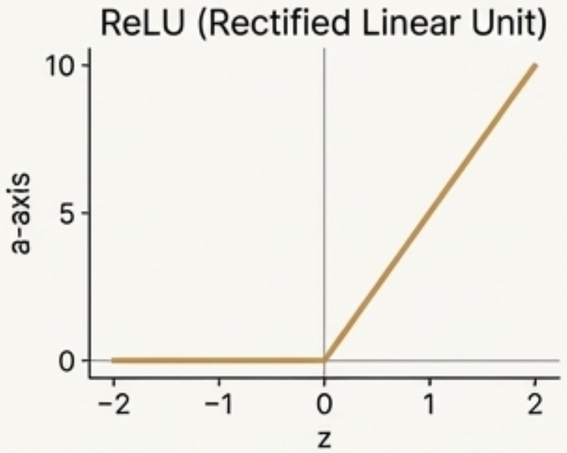
Without activation functions, a deep stack of layers...

$$\mathbf{y} = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

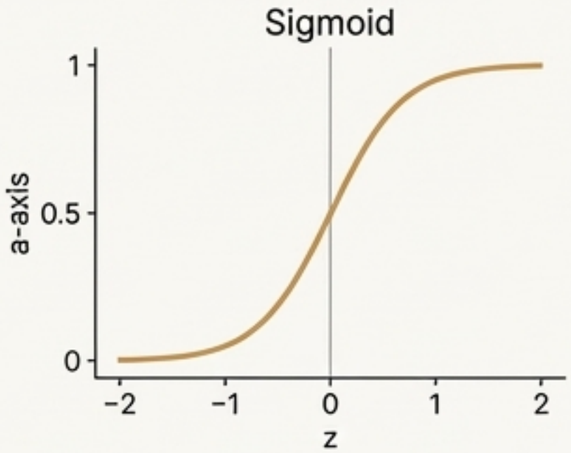
...collapses into a single, equivalent linear transformation:

$$\mathbf{y} = \underbrace{(\mathbf{W}_2\mathbf{W}_1)}_{\text{one matrix}}\mathbf{x} + \underbrace{(\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2)}_{\text{one bias}}$$

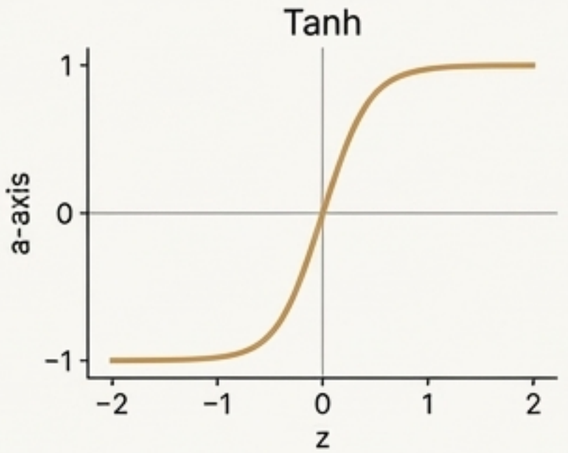
Deep networks would be no more powerful than shallow ones.



ReLU: $\max(0, z)$.
The modern default.



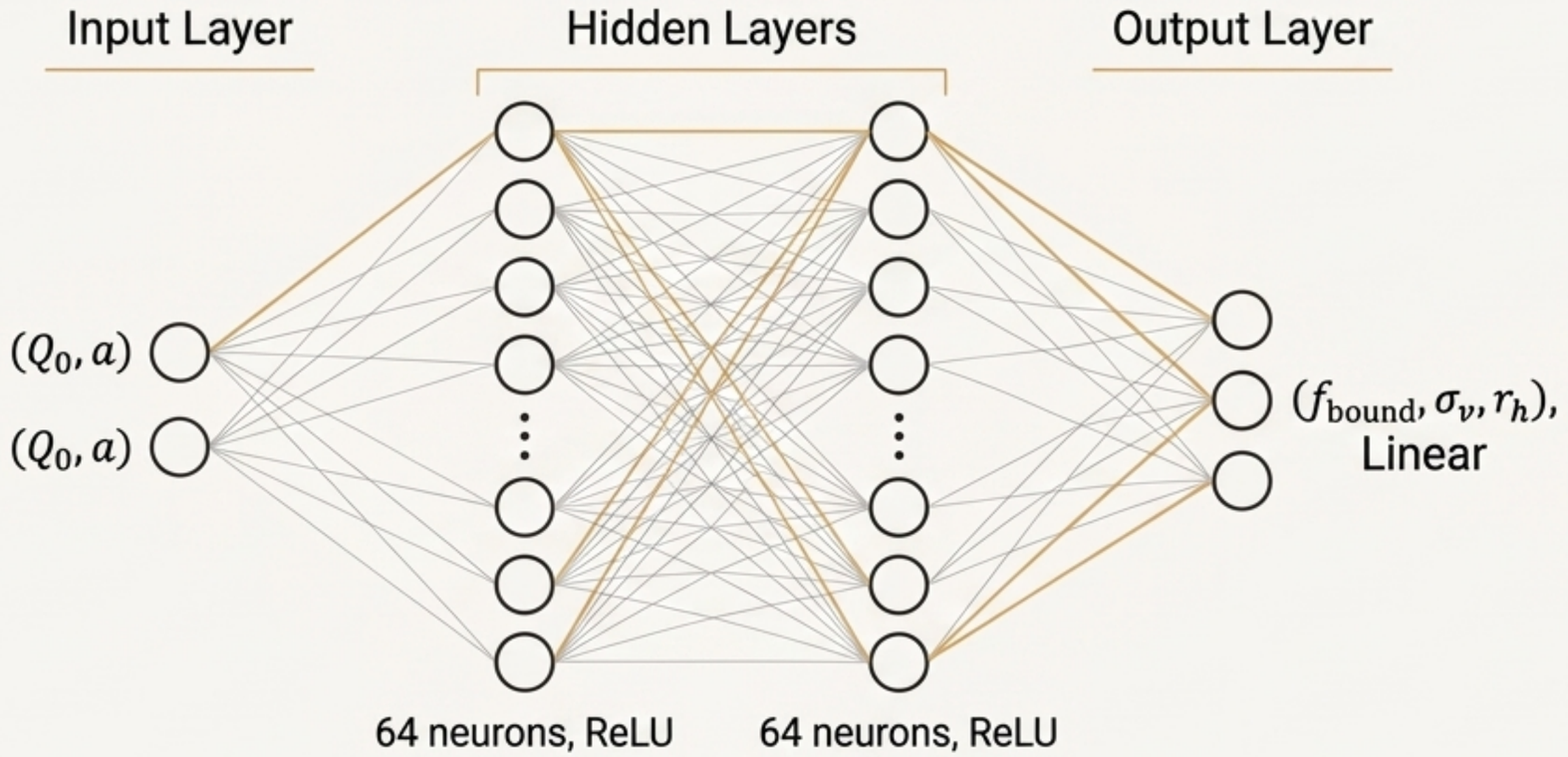
Sigmoid: $1/(1 + e^{-z})$.
Squashes to (0, 1).



Tanh: $\tanh(z)$.
Squashes to (-1, 1).

Recommendation for Emulators:
Use ReLU for hidden layers. It's simple, efficient, and works remarkably well.

Assembling Neurons into a Network



Our Emulator Architecture

Input: 2 features (Q_0, a)
Hidden 1: 64 neurons, ReLU
Hidden 2: 64 neurons, ReLU
Output: 3 predictions $(f_{\text{bound}}, \sigma_v, r_h)$, Linear

Counting the Knobs (Learnable Parameters)

Layer 1 (2→64):
64 × 2 + 64 = 192

Layer 2 (64→64):
64 × 64 + 64 = 4,160

Layer 3 (64→3):
3 × 64 + 3 = 195

Total: 4,547 parameters

The Overfitting Paradox: When More is Different

Alarm bells should ring. We have **4,547 parameters** but only **~100 training examples.** That's ~45x more parameters than data points. Isn't this a recipe for extreme overfitting?

🔗 Connection to Module 1: Degrees of Freedom

Why It Doesn't (Always) Doom Neural Networks

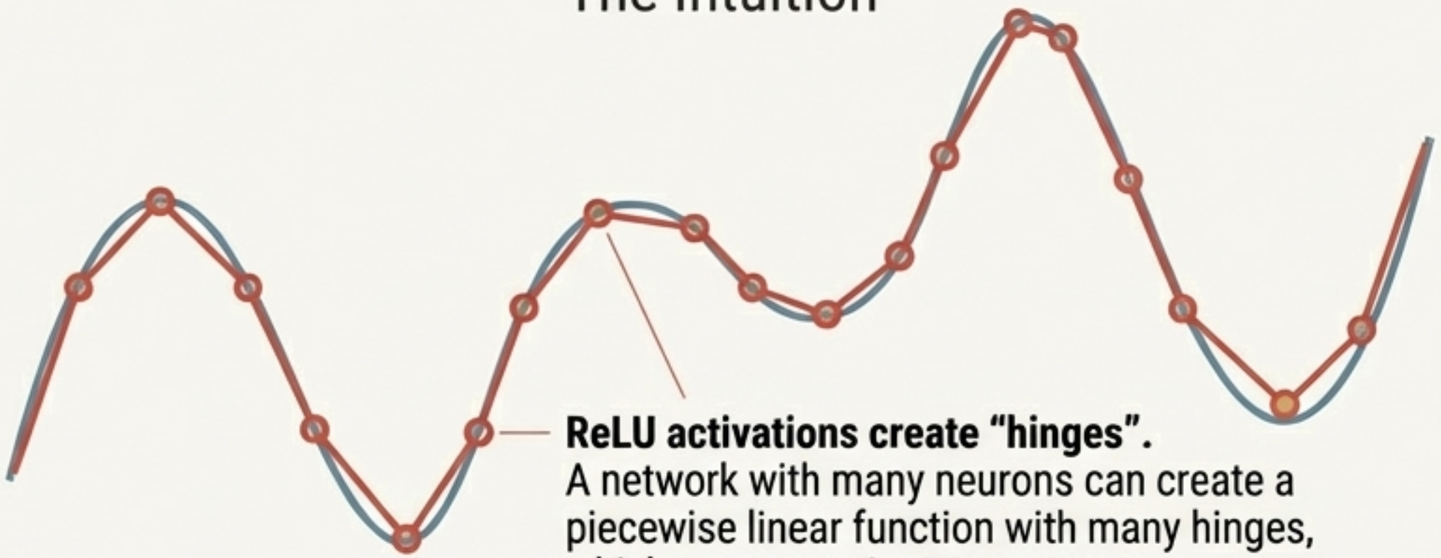
1. **Implicit Regularization**: Gradient descent with early stopping tends to find "simpler" solutions.
2. **Parameter Sharing**: Many different weight configurations produce functionally similar models.
3. **Validation**: We actively monitor performance on unseen data to detect and stop overfitting.

Overfitting is a **real risk**. We manage it with careful validation and quantify the remaining uncertainty using ensembles.

The Theoretical Guarantee: Universal Approximation

A neural network with a single hidden layer can approximate **any continuous function** to **arbitrary accuracy**, provided it has *sufficiently many neurons*.

The Intuition



ReLU activations create "hinges".
A network with many neurons can create a piecewise linear function with many hinges, which can approximate any curve.

The Fine Print

It doesn't say *how many* neurons you need.

It doesn't tell you *how to find* the right weights (training is hard).

It doesn't guarantee generalization from a *finite* training set.

It says nothing about behavior *outside* the training data region.

Universal Approximation ≠ Universal Learning
The theorem is about **expressiveness**, not **learnability**.

Training is a Search for the Best Parameters

1. Define a Loss Function

Measure “how wrong” the predictions are.

Standard choice: Mean Squared Error (MSE),

$$\mathcal{L} = \frac{1}{N} \sum \|\hat{\mathbf{y}}_i - \mathbf{y}_i\|^2$$

🔗 **Connection to Module 5:** Minimizing MSE is equivalent to Maximizing a Gaussian Likelihood.

2. Use Gradient Descent

Iteratively step towards the minimum of the loss landscape.

Update Rule:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}$$

We use **Adam**, an adaptive optimizer that adjusts the learning rate (η) per parameter.

3. Compute Gradients via Backpropagation

The engine that finds the direction of steepest descent.

🔗 **Connection to Module 6:** Backpropagation is just reverse-mode automatic differentiation (what `jax.grad` does) applied to a neural network. JAX handles this automatically.

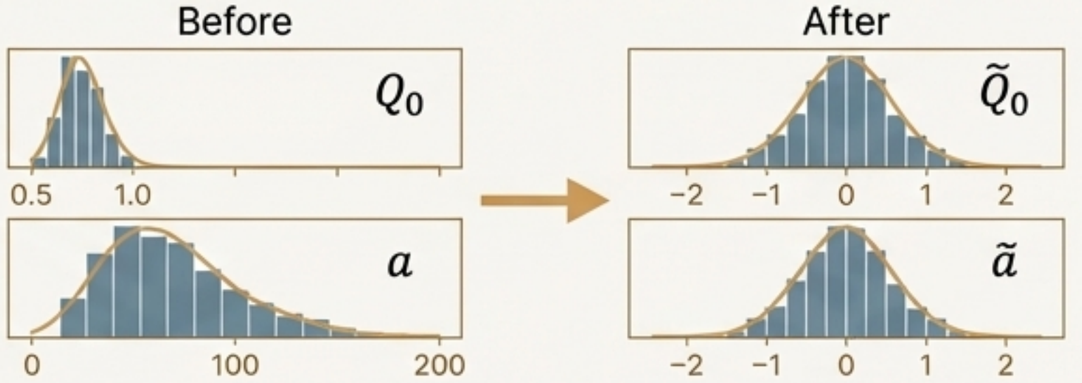
The Art of Training: Normalization is Not Optional

The Problem

- Neural networks are sensitive to the scale of inputs and outputs.
  - **Input Problem:**
    If input a (50-200) is much larger than Q_0 (0.5-1.5), learning will be unstable.
  - **Output Problem:**
    If output r_h has a much larger range than f_{bound}, its error will dominate the MSE loss, and the network will ignore f_{bound}.

The Solution: Standardization

Transform each feature (inputs and outputs) to have **zero mean** and **unit variance**.



$$\tilde{x} = \frac{x - \mu_x}{\sigma_x}$$

The Cardinal Rule of Normalization

- Compute μ and σ from the **training set only**.
- Apply the **same** transformation to your training, validation, and test data.
- **Never** use test set statistics. This is data leakage.

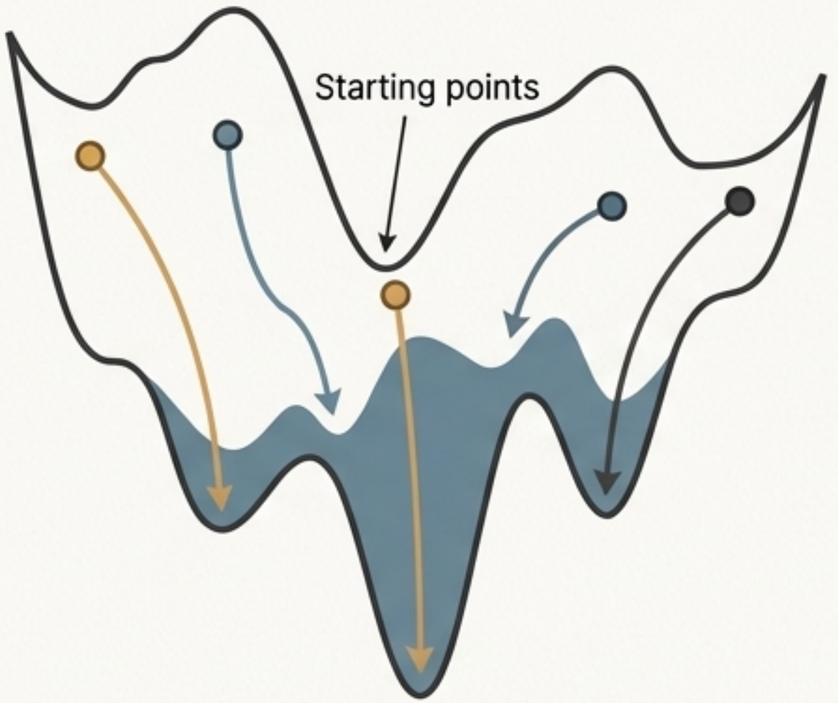
Mastering Uncertainty with Ensembles

The Problem with Single Networks

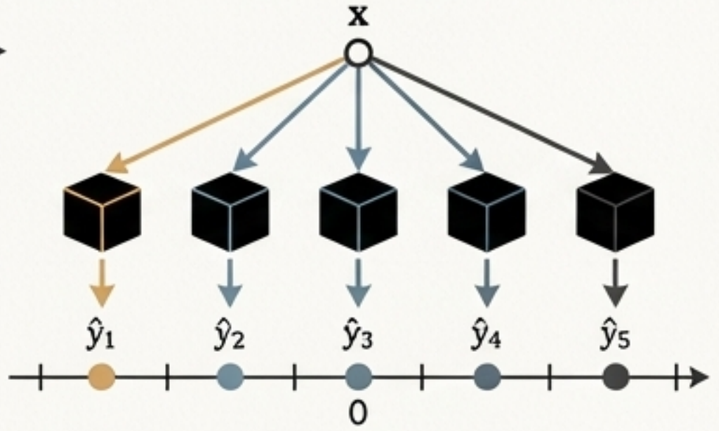
They provide point predictions and can be confidently wrong, especially when extrapolating.

Science demands uncertainty.

The Deep Ensemble Solution



1. Train M separate networks (e.g., M=5) on the same data, but with different random weight initializations.



Calculating Uncertainty

Mean Prediction: $\bar{y}(\mathbf{x}) = \frac{1}{M}\sum \hat{y}_m(\mathbf{x})$

Epistemic Uncertainty: $\sigma_y(\mathbf{x}) = \text{std}(\hat{y}_1, \ldots, \hat{y}_M)$

Interpretation
The ensemble spread, σ_y, quantifies uncertainty due to limited training data. It will be low in well-sampled regions and high at the edges or in gaps.

Tools of the Trade: JAX, Equinox & Optax

A functional, high-performance toolkit for modern machine learning.

Equinox: Neural Networks as PyTrees.

- Models are just data structures (PyTrees) that JAX knows how to handle.
- This enables seamless use of `jax.grad`, `jax.jit`, `jax.vmap`.
- Models are callable (`output = model(input)`) and have explicit state.

Optax: Gradient Processing and Optimization.

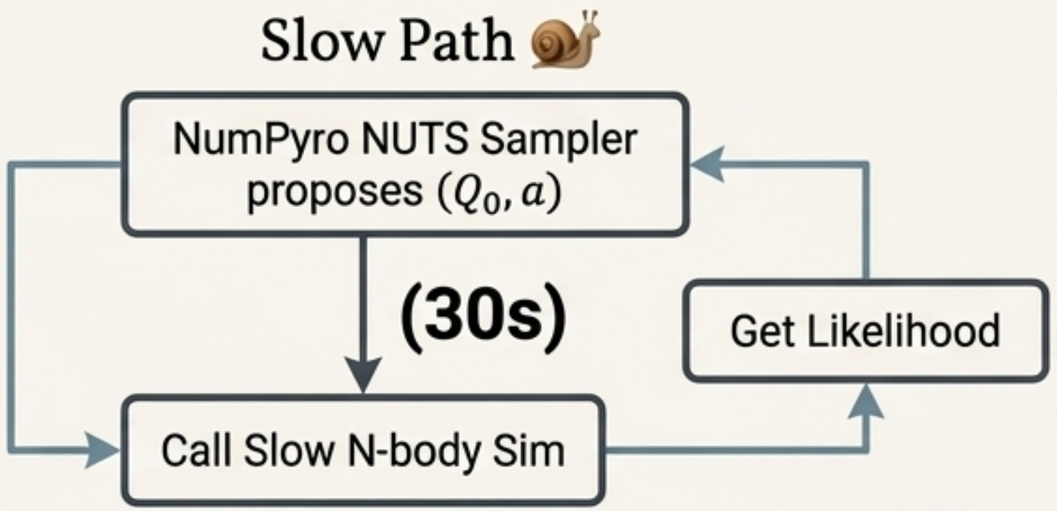
- Provides standard optimizers like `optax.adam`.

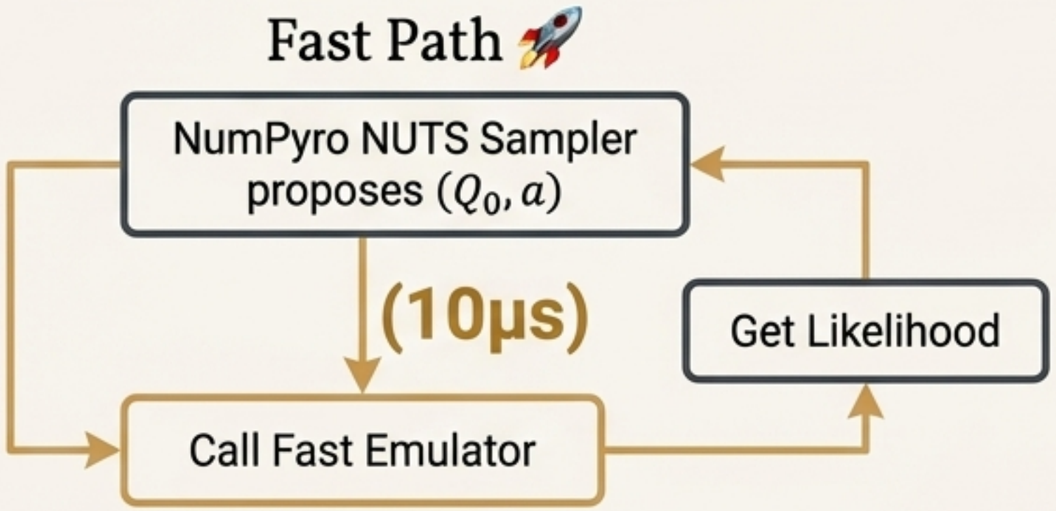
```
import equinox as eqx

class MyEmulator(eqx.Module):
  layers: list

  def __init__(self, key):
    # ... initialize layers using jax.random.s
    # Example: self.layers = [eqx.nn.Linear(..

  def __call__(self, x):
    # ... apply layers and activations ...
    # Example: for layer in self.layers: x = l
    return x
```

Learning from documentation is a core professional skill. The Equinox and Optax **"Getting Started" guides** are your primary resources.

The Payoff: From Emulator to Inference

Given observed cluster properties, infer the initial conditions (Q_0, a).





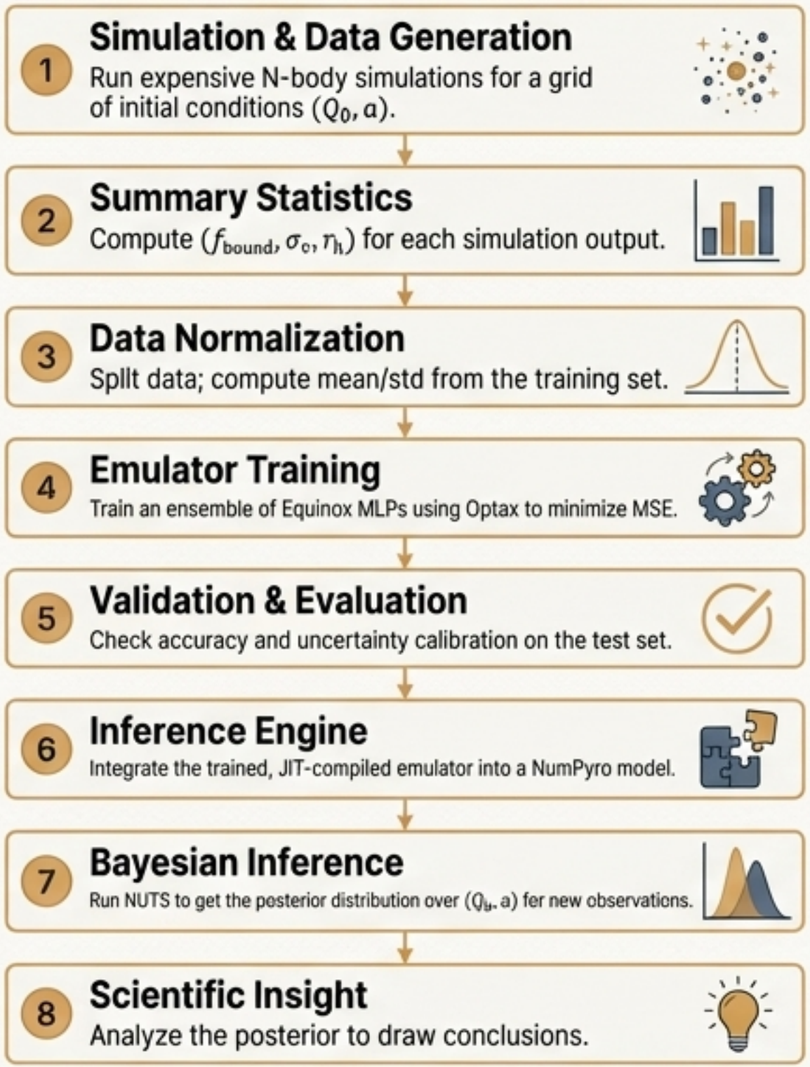
The NumPyro Model Structure

1. Define priors on parameters:
   `Q0 = numpyro.sample("Q0", dist.Uniform(...))`
2. Call the emulator to get the mean prediction:
   `mean_pred = emulator(Q0, a)`
3. Define the likelihood using the prediction and an observation error:
   `numpyro.sample("obs", dist.Normal(mean_pred, sigma_obs), obs=data)`

Revisiting the Payoff

Total inference time:
Minutes, not months.

Synthesis: The Complete Scientific Workflow



This workflow—expensive simulations to fast surrogates to Bayesian inference—is how frontier research operates. Now go build something that learns. 🚀