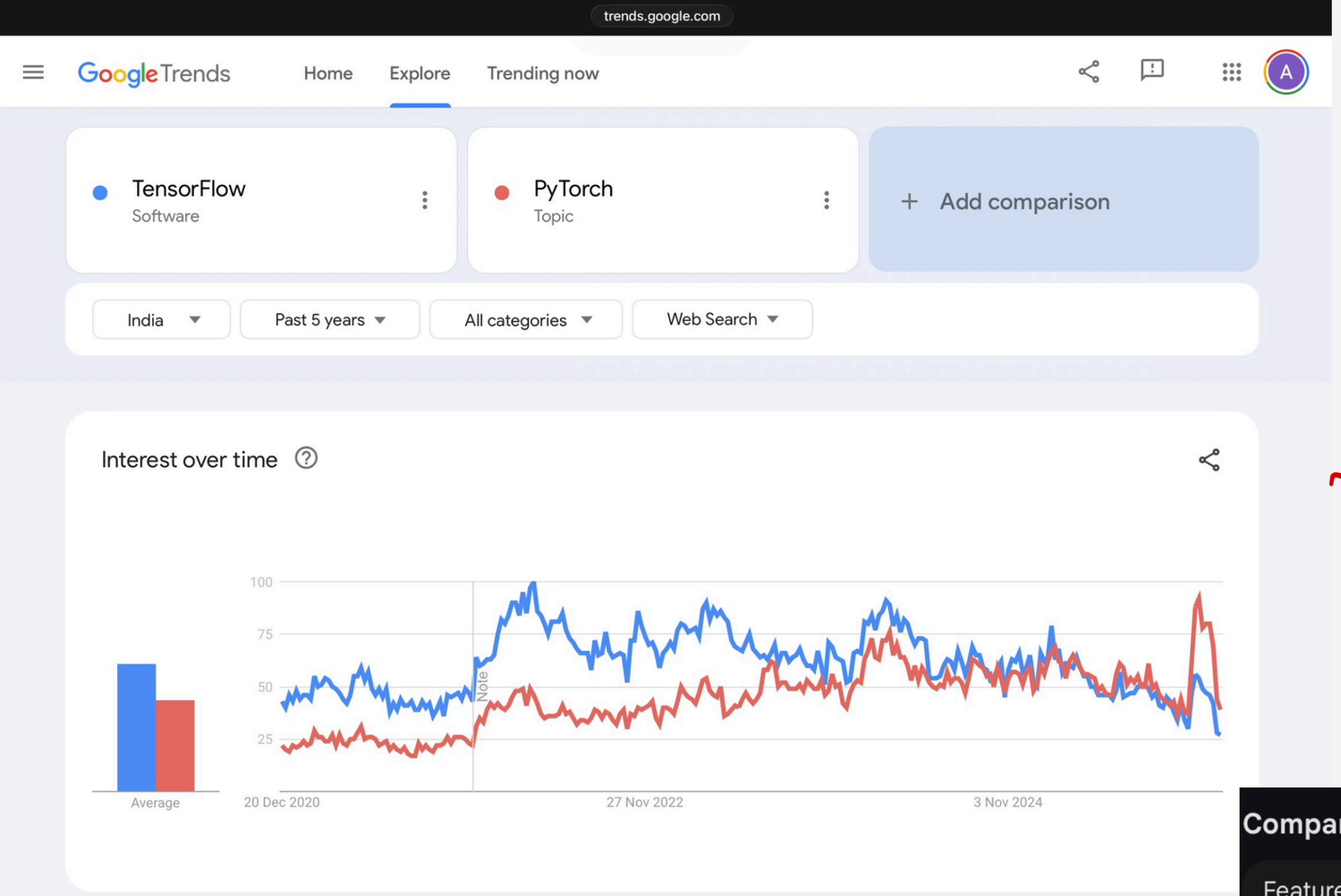
The Core of PyTorch: Tensors, Data, and Gradients

A storyteller's guide to the engine of modern deep learning.

Lec-9



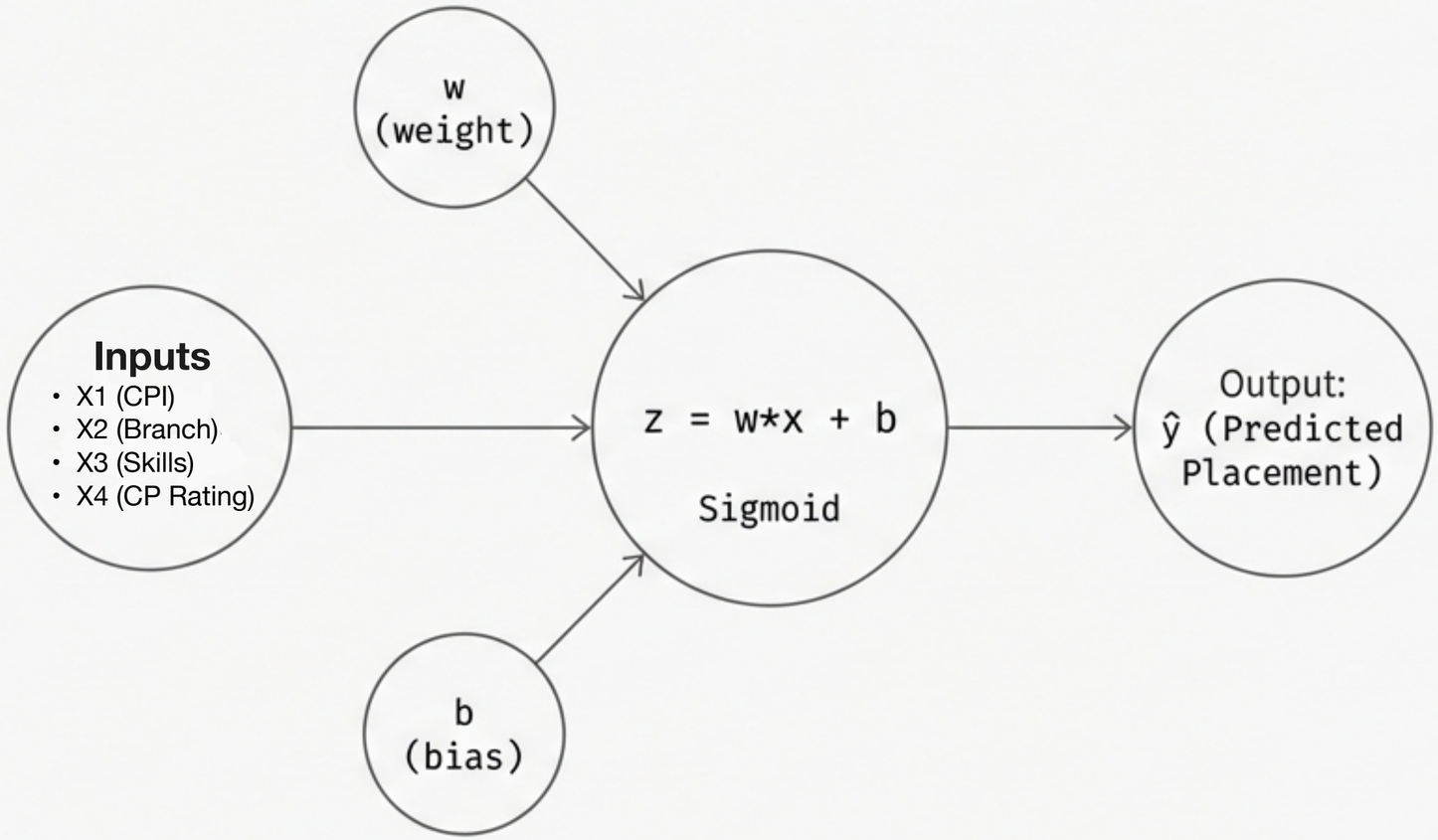
PyTorch vs TensorFlow

Comparison: At a Glance

Feature	PyTorch (The Winner)	TensorFlow (The Challenger)
Philosophy	"Python-first" (Dynamic)	"Graph-first" (Static/Hybrid)
Research Adoption	**~92%** of 2024/25 papers	~8% and shrinking
GenAI Support	Default for LLMs & Diffusers	Secondary support
Debugging	Easy (Standard Python tools)	Difficult (Complex stack traces)
Leading Edge	High-performance (PyTorch 2.x)	Mature but slower to iterate

The Goal: Teaching a Machine to Learn

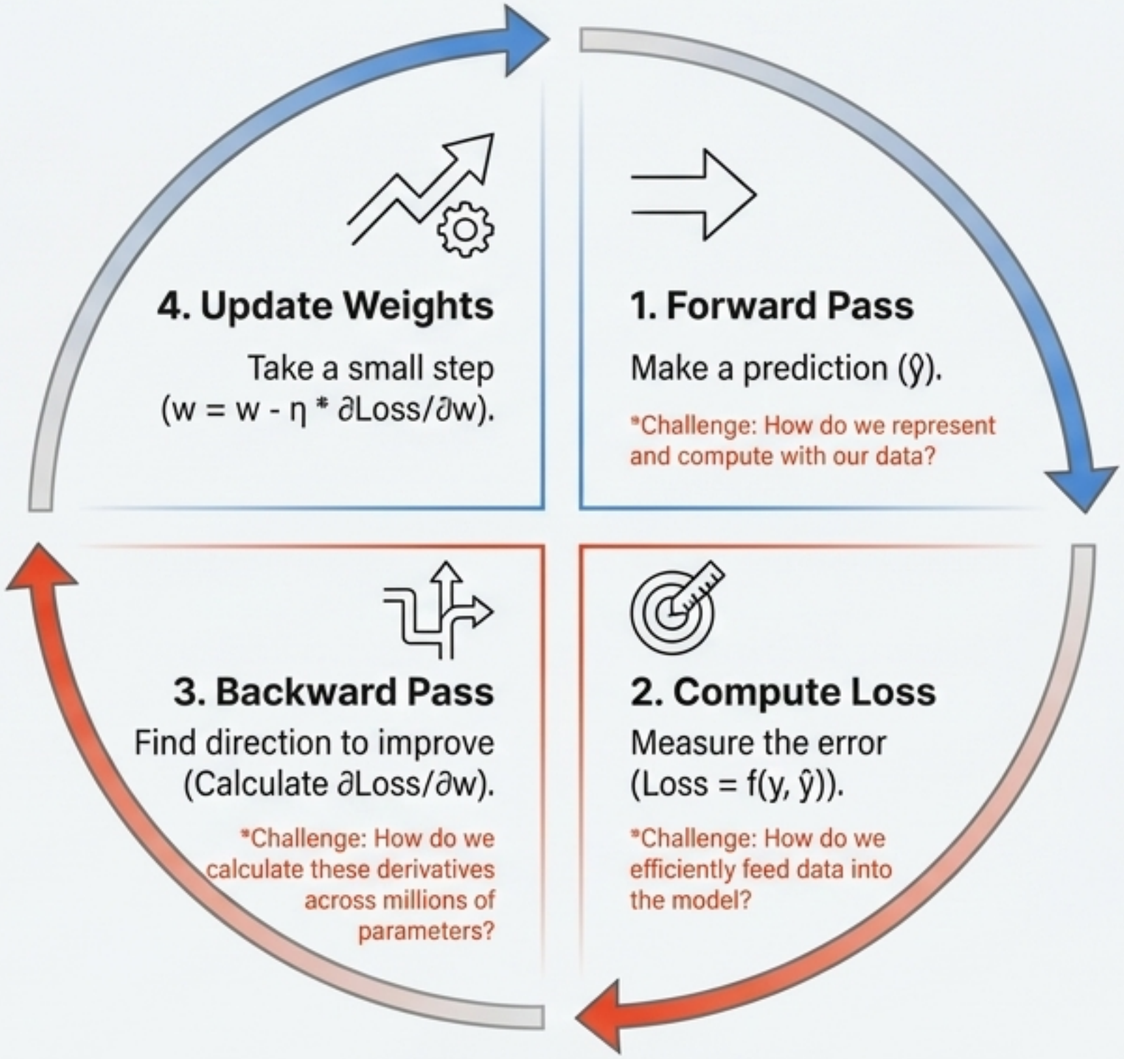
Every neural network, from the simplest to the most complex, faces the same fundamental challenge: how does it learn from data to make correct predictions? This process involves making a guess, measuring the error, and systematically adjusting its internal parameters to improve.



But how does the network *learn* the right values for w and b?

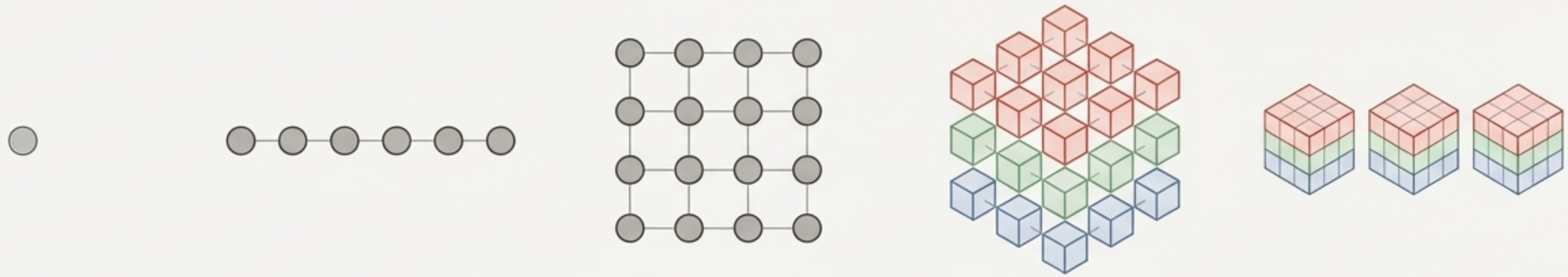
The Learning Process is a Four-Step Loop

Training a neural network is an iterative cycle. We repeatedly guide the model towards better performance by following four distinct steps. Each step presents a unique computational challenge.



A Tensor is a Generalisation of Vectors and Matrices

A **tensor** is a multi-dimensional array. It's the foundational data structure in PyTorch, capable of representing data of any dimensionality.



0D Tensor (Scalar)

A single number.

Example: A loss value.

1D Tensor (Vector)

A sequence of numbers.

Example: A word embedding.

2D Tensor (Matrix)

A grid of numbers.

Example: A grayscale image.

3D Tensor

A cube of numbers.

Example: An RGB (colour) image.

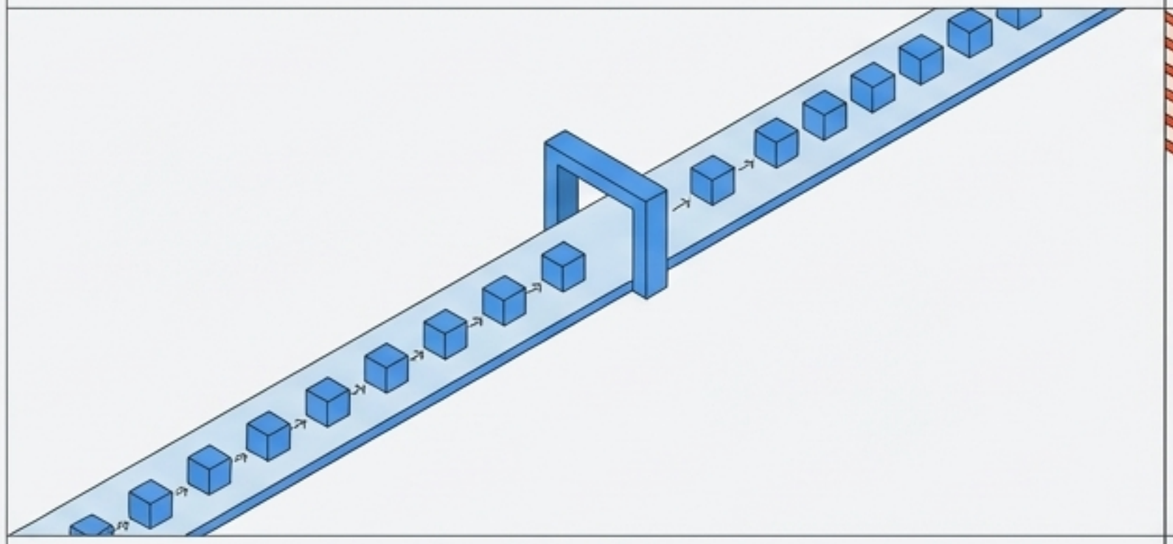
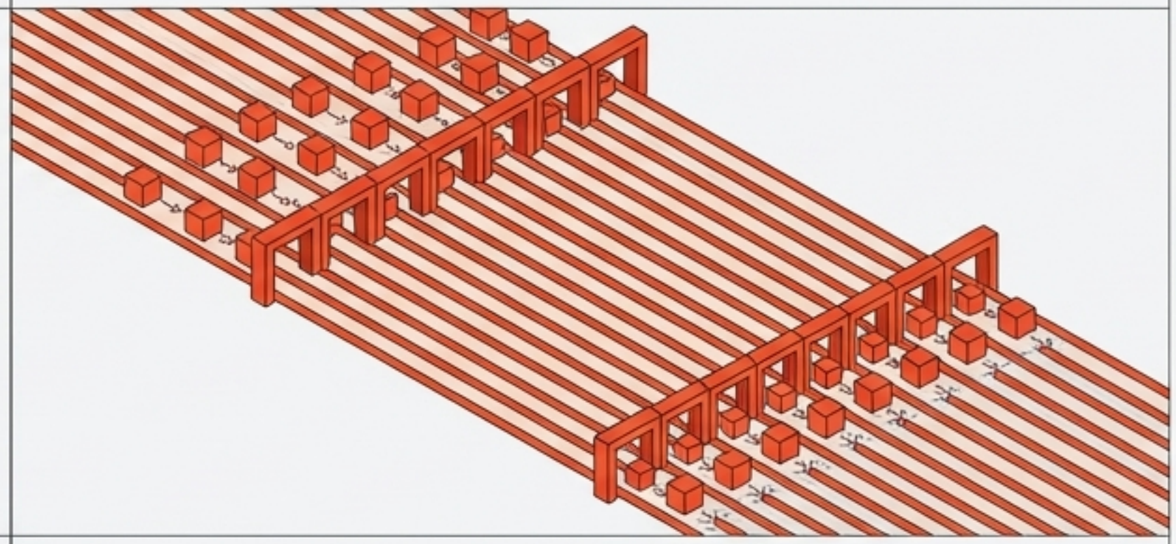
4D Tensor

A series of cubes.

Example: A mini-batch of RGB images.

The Tensor Superpower: Massively Parallel Computation

Tensors are more than just multi-dimensional arrays; they are designed for hardware acceleration. Unlike standard Python arrays, tensor operations can be executed in parallel on Graphics Processing Units (GPUs), dramatically speeding up the matrix calculations that are central to deep learning.

CPU: Sequential Processing	GPU: Parallel Processing
	
One task at a time	Hundreds of tasks simultaneously

A 10,000 x 10,000 matrix multiplication can be **~30x faster** on a GPU.

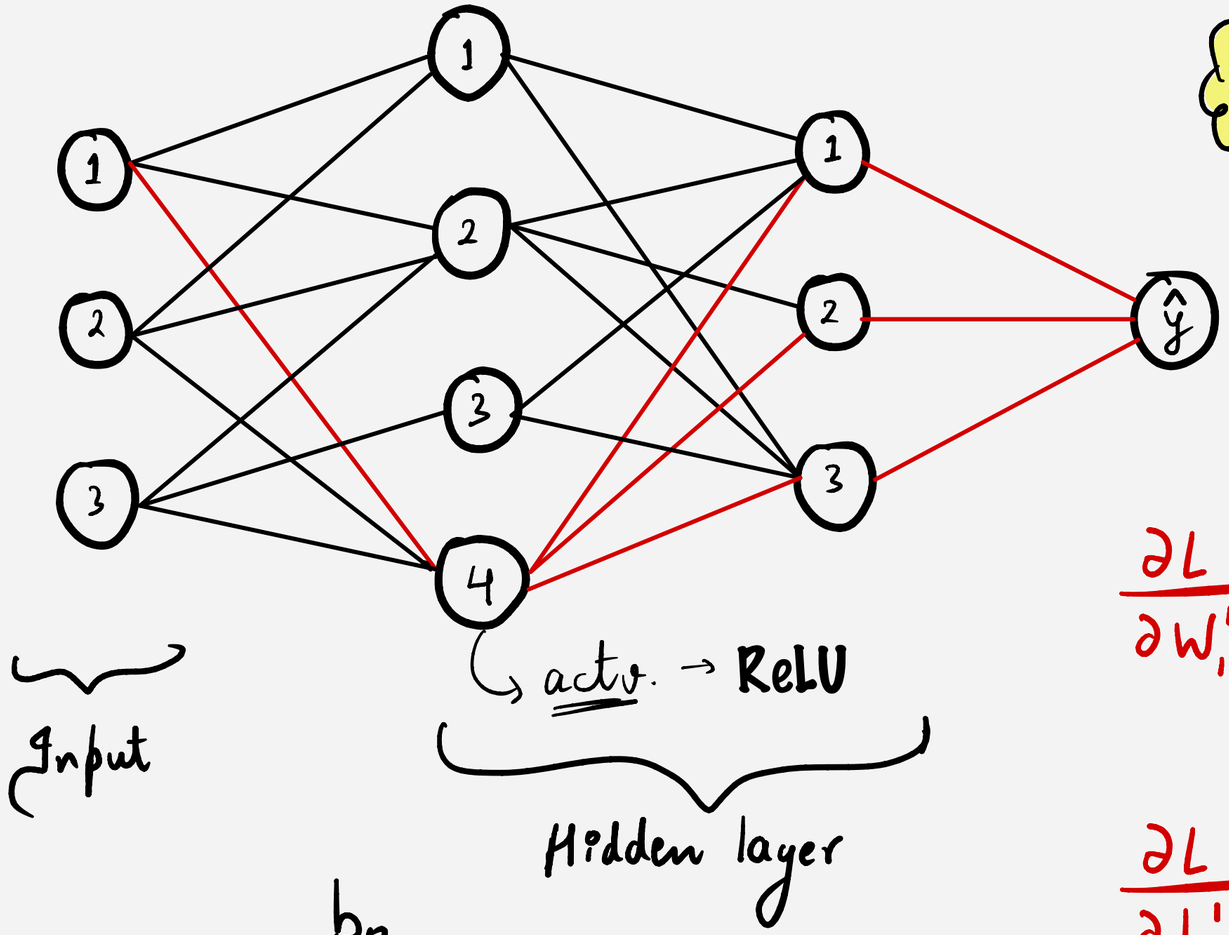
Problem : The Impossible Calculus of Learning

The backward pass requires us to calculate the gradient of the loss function with respect to every single model parameter. For a simple nested function, this is tedious. For a deep neural network with millions of parameters, doing this manually is practically impossible.

Manageable

$$\frac{d}{dx}\left(\sin(x^2)\right) = \cos(x^2) \cdot 2x$$

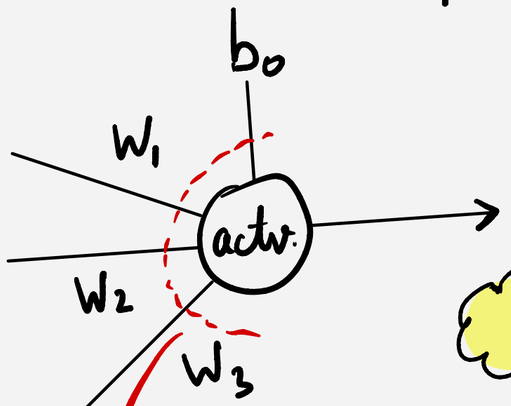
Impossible by Hand



W^k_{nm} b^k_n

$$\frac{\partial L}{\partial W^1_{1,4}} = \overbrace{\sum_k \left(\frac{\partial L}{\partial Z^2_k} \cdot \frac{\partial Z^2_k}{\partial A^1_4} \right)}^{\text{Sum of all paths from node 4 of layer 1}} \cdot \overbrace{\frac{\partial A^1_4}{\partial Z^1_4}}^{\text{ReLU}} \cdot \frac{\partial Z^1_4}{\partial W^1_{1,4}}$$

$$\frac{\partial L}{\partial b^1_4} = \underbrace{\sum_k \left(\frac{\partial L}{\partial Z^2_k} \cdot \frac{\partial Z^2_k}{\partial A^1_4} \right)}_{\text{Sum of all paths from node 4 of layer 1}} \cdot \underbrace{\frac{\partial A^1_4}{\partial Z^1_4}}_{\text{ReLU}} \cdot \frac{\partial Z^1_4}{\partial b^1_4}$$



$$A = \text{actv.}(\underbrace{W^T X + b}_{Z})$$

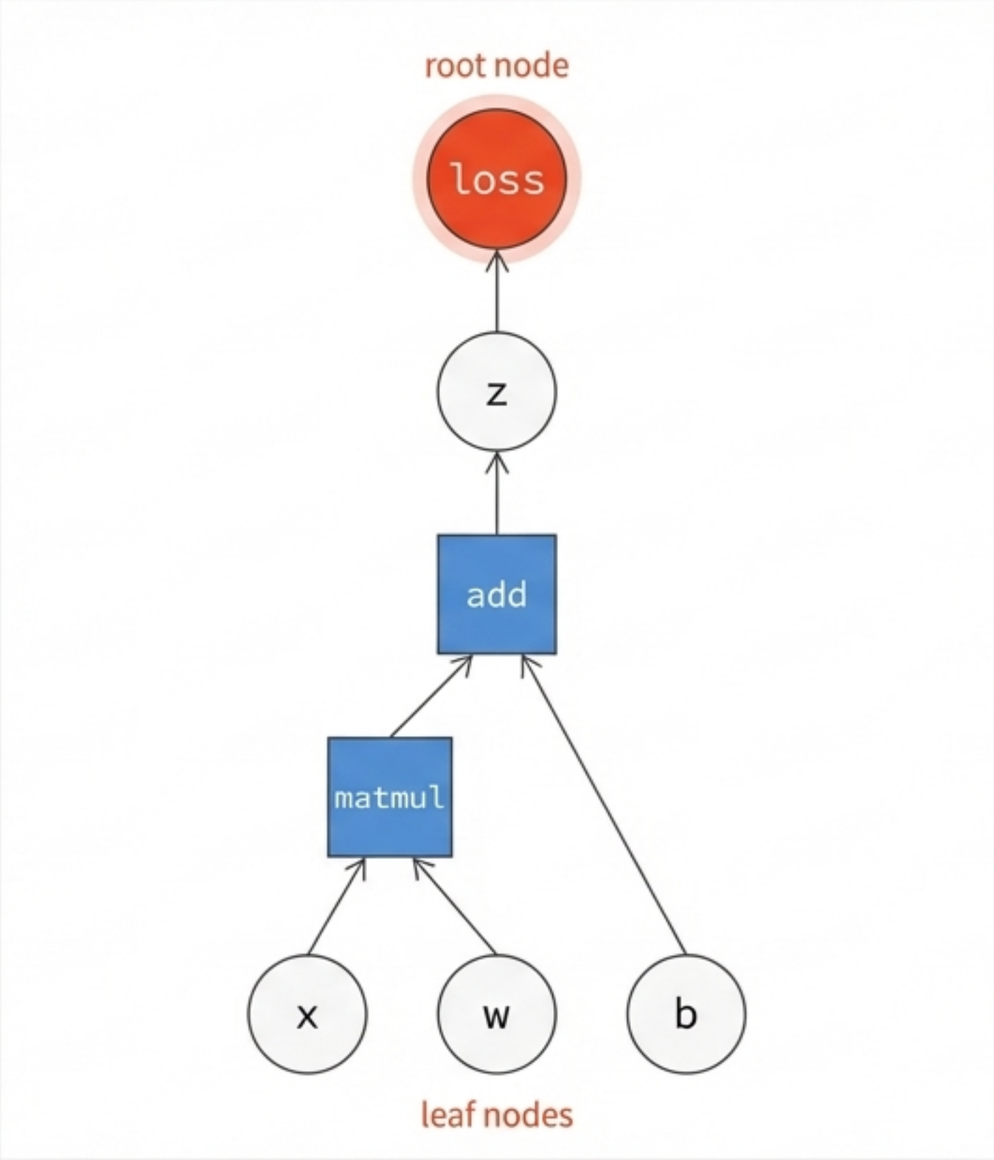
$$\sum_i W_i X_i + b_0 = Z$$

Solution: PyTorch's Automatic Differentiation Engine

PyTorch solves this with `torch.autograd`. It's a built-in engine that automatically computes gradients for any computational graph.

Core Idea

During the forward pass, PyTorch dynamically builds a Directed Acyclic Graph (DAG). This graph records all operations and tensors, creating a map that can be traced backwards from the final loss to every parameter to compute gradients using the chain rule.



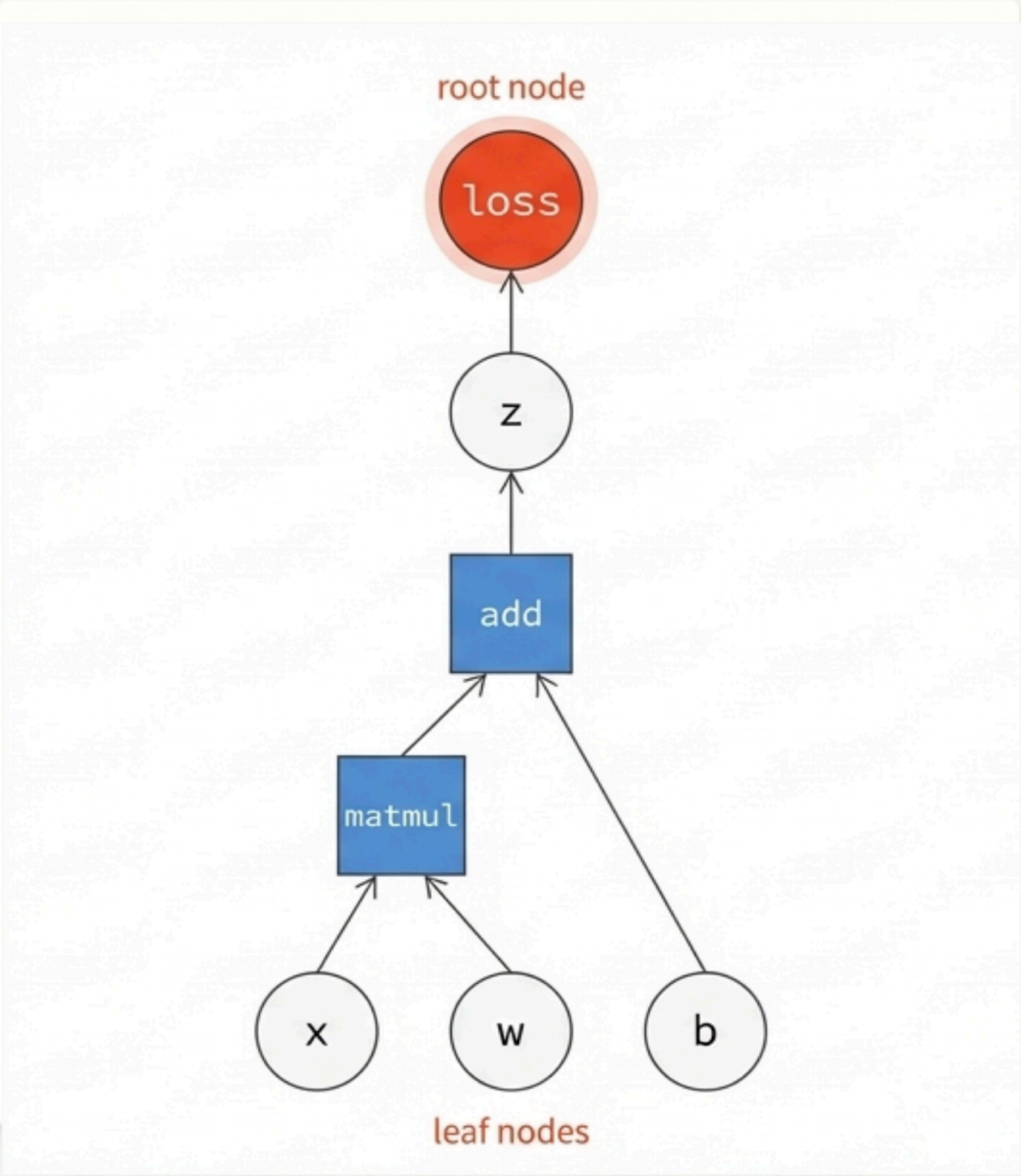
The Magic is in One Command: `.backward()`

You don't need to manually implement the chain rule. After calculating the loss, a single call to `loss.backward()` triggers `autograd` to compute, accumulate, and propagate gradients back through the network to the leaf nodes.

```
# Tensors w and b have requires_grad=True
z = torch.matmul(x, w) + b
loss = loss_fn(z, y)

# The magic happens here!
loss.backward()

# Gradients are now available
print(w.grad)
print(b.grad)
```



Controlling the Gradient Engine

You have fine-grained control over the `autograd` engine.

Starting the Engine: `requires_grad=True`

To tell PyTorch to track operations on a tensor and its subsequent outputs, set its `requires_grad`
its subsequent outputs, set its `requires_grad`
property to `True`. This is essential for the model's trainable parameters like weights and biases.

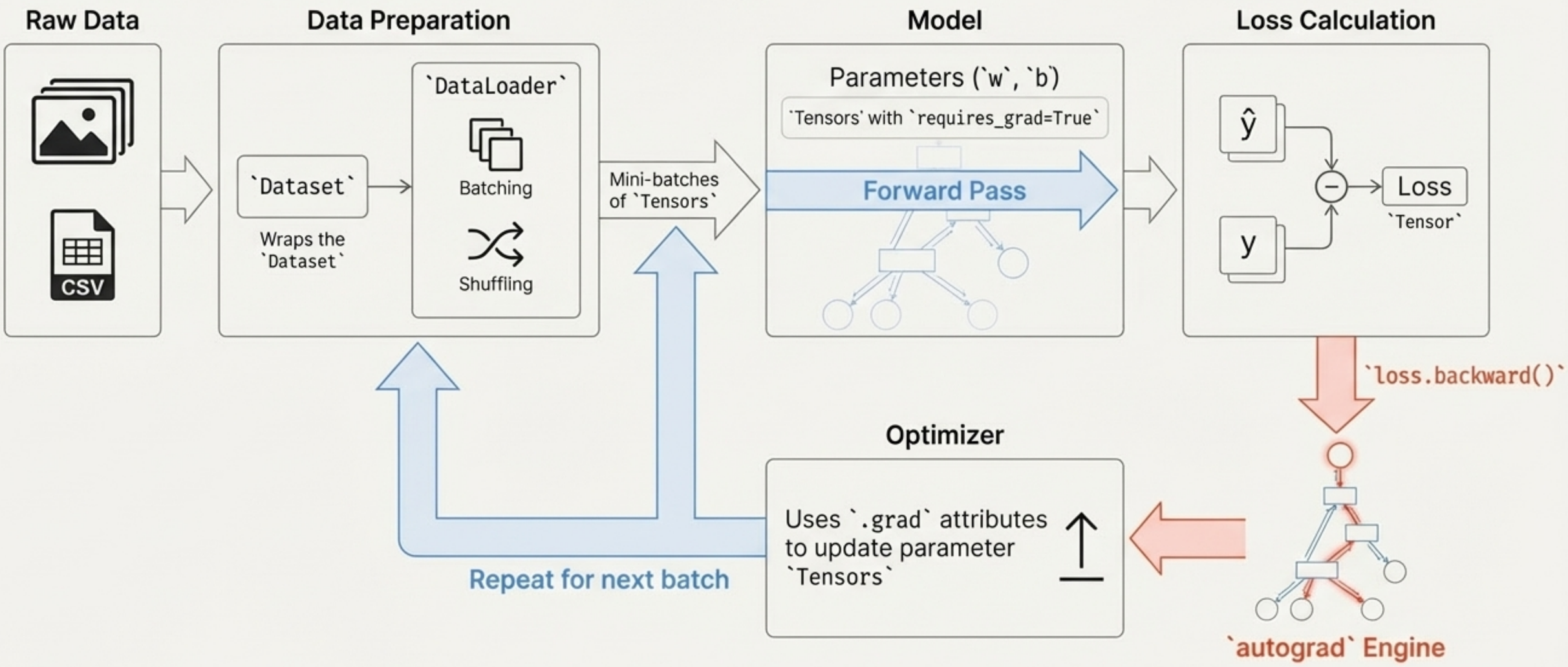
```
# Track gradients for this tensor
w = torch.randn(5, 3, requires_grad=True)
```

Pausing the Engine: `torch.no_grad()`

During inference or evaluation, you can disable gradient tracking to speed up computations and save memory. Operations inside this context manager are not recorded by `autograd`.

```
with torch.no_grad():
    # Operations here are not tracked
    z = torch.matmul(x, w) + b
```

The Complete PyTorch Workflow in Harmony



You already know what to do !!