FOUNDATION TIER

MODULE 06

Building Autograd

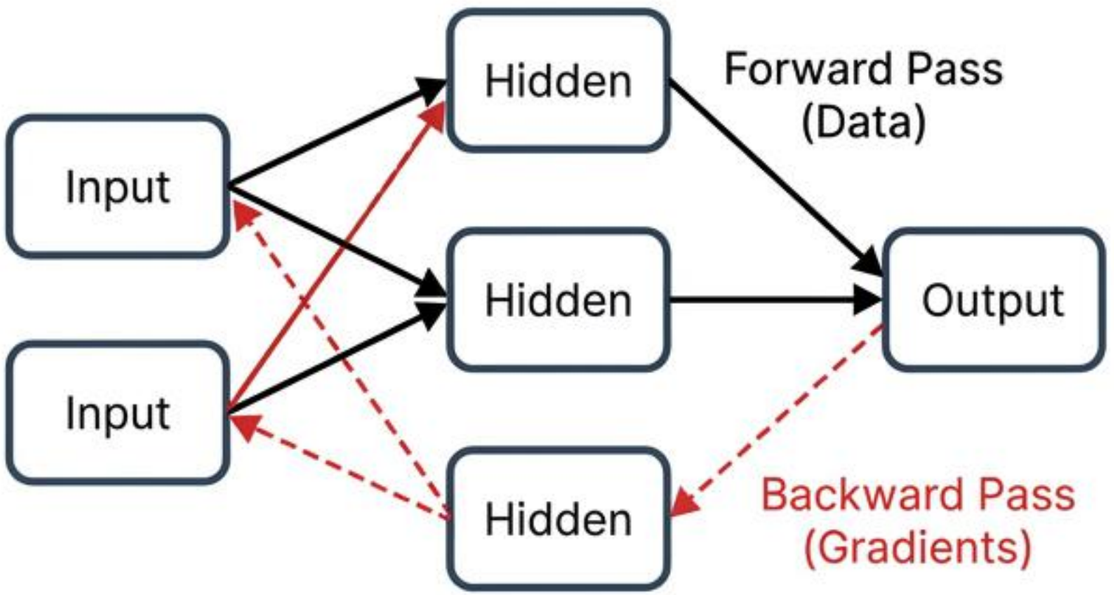
Automatic differentiation from first principles

TinyTorch

MODULE 06

AUTOGRAD

The Gradient Engine

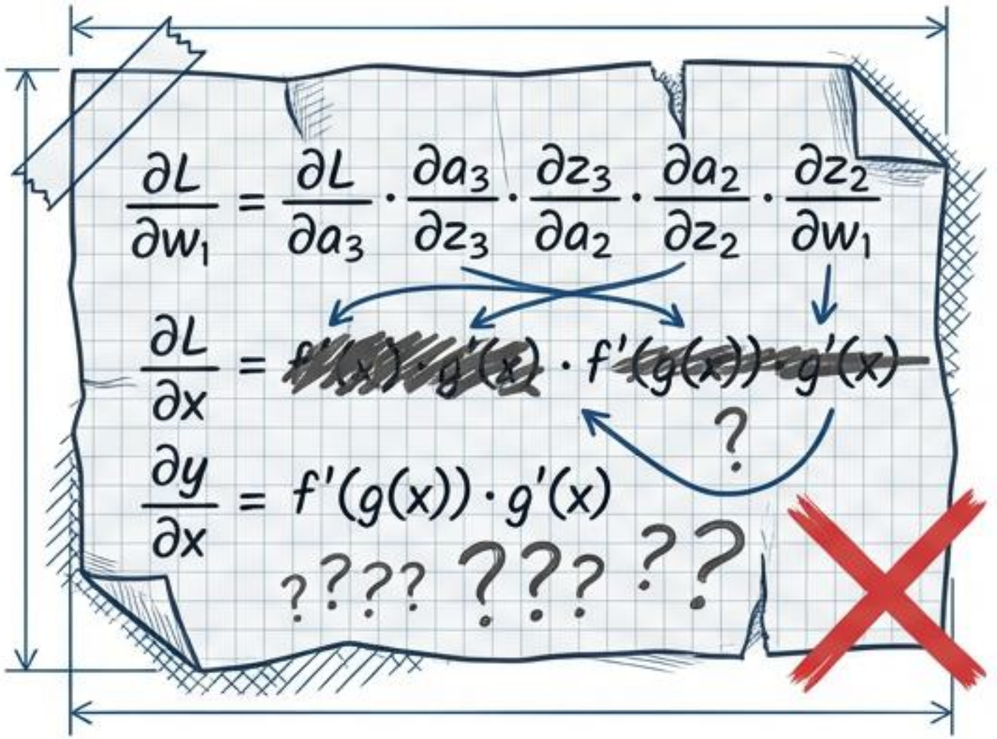


Goal: Build the reverse-mode automatic differentiation system from scratch.

Role: The bridge between Computation (Tensors) and Learning (Optimizers).

The Problem: Derivatives at Scale

Manual Differentiation



Error-prone. Brittle. Impossible for 100+ layers.

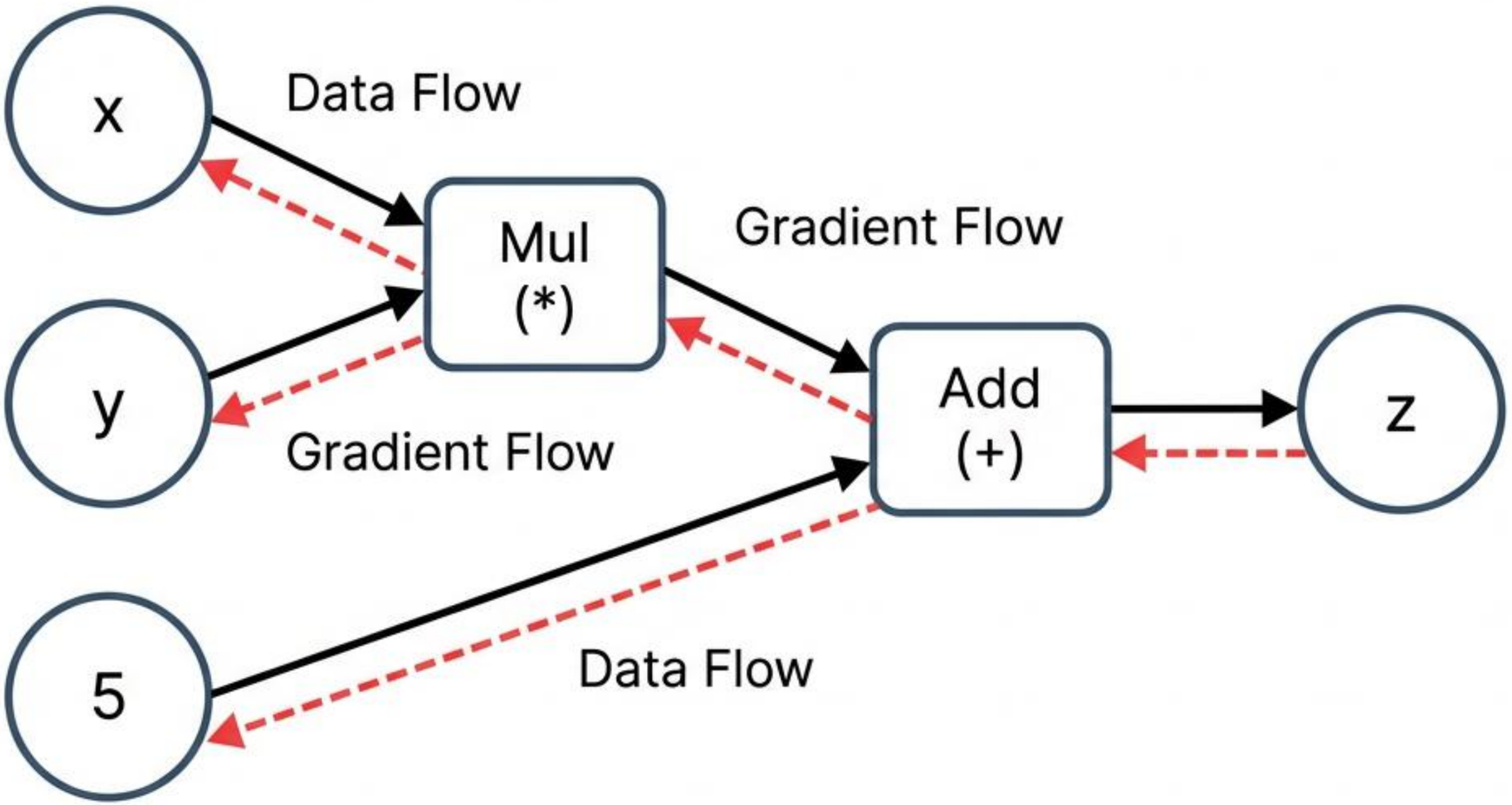
Automatic Differentiation

```
# The entire backward pass
loss.backward()
```

Exact. Generalized. Scales to billions of parameters.

Approach: “Define-by-Run” — The graph is built dynamically as the code executes.

The Abstraction: Computation Graphs

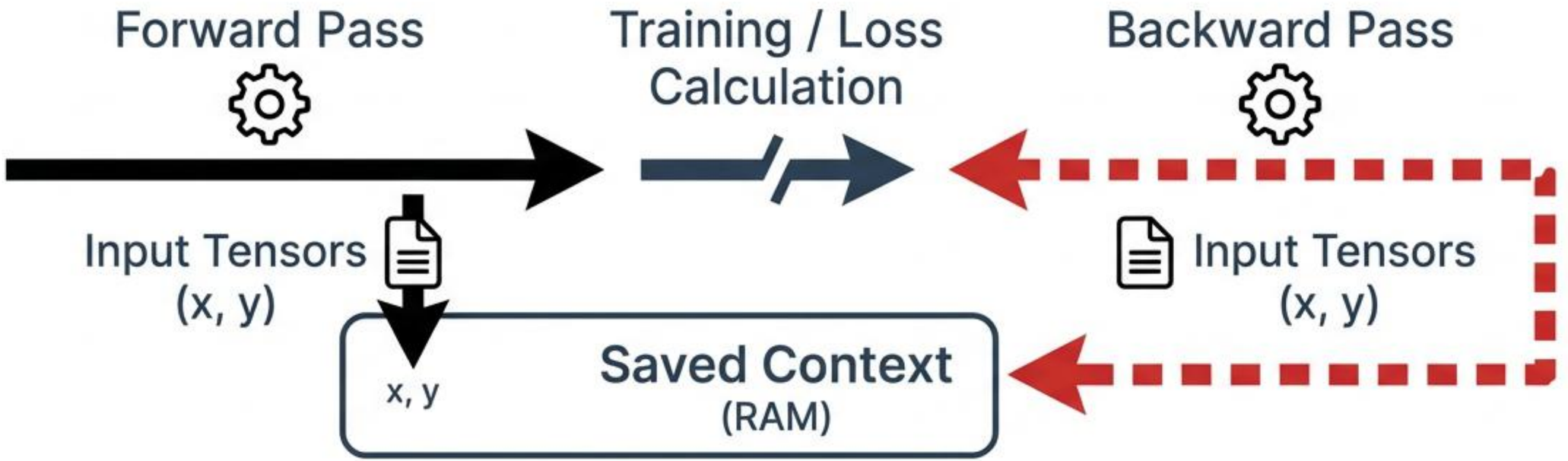


Nodes:
Tensors (Data) & Functions (Operations)

Edges:
Dependencies

Invariant:
Every forward operation records its history to enable the backward pass.

Systems Constraint: The Memory Trade-off



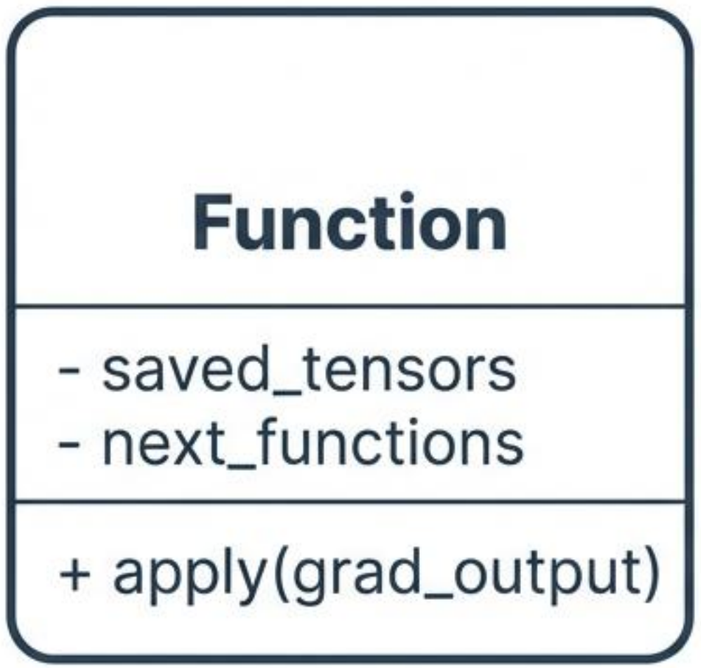
The Math Constraint: To compute $d(x \cdot y)/dx = y$, we need the value of ‘y’.

The Cost: Intermediate values must be cached in memory until the backward pass is complete.

Reality: Training memory usage ≈ 2x-3x Inference memory usage.

The Node Abstraction: Function Class

Concept



Implementation

Captures inputs for backward pass

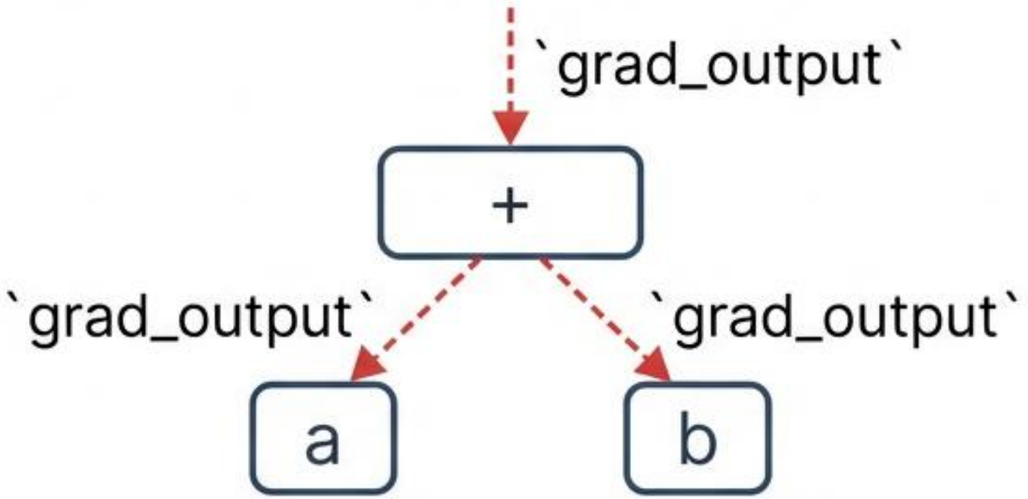
Defines the local Chain Rule logic

```python
class Function:
    def __init__(self, *tensors):
        # The Memory Cost
        self.saved_tensors = tensors
        # The Graph Structure
        self.next_functions = []

    def apply(self, grad_output):
        """Compute gradients for inputs."""
        raise NotImplementedError()
```

Implementation: AddBackward

Math/Visual

If $z = a + b$, then $\frac{\partial z}{\partial a} = 1$ and $\frac{\partial z}{\partial b} = 1$



Code

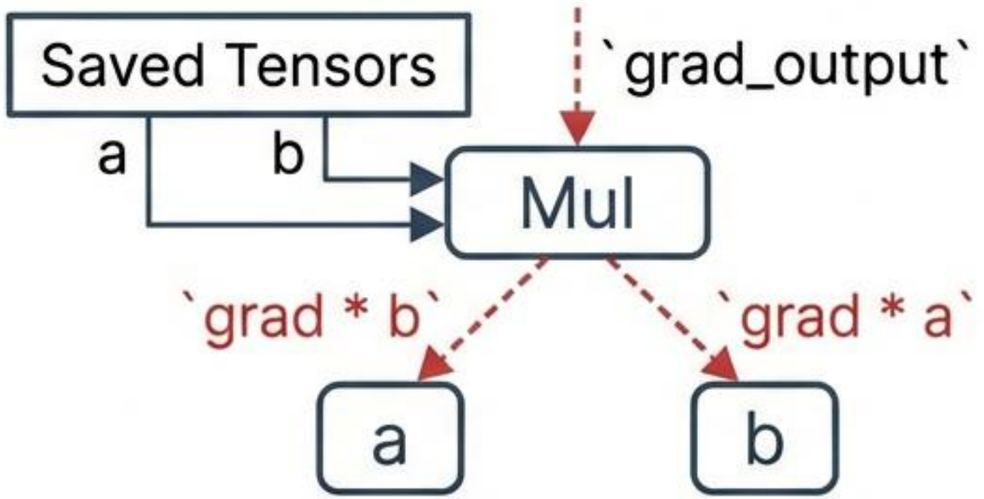
```
class AddBackward(Function):
    def apply(self, grad_output):
        # Gradient flows equally to both inputs
        # 1 * grad_output
        return grad_output, grad_output
```

Simple distribution. Broadcasting handled internally.

Implementation: MulBackward

Math/Visual

If $z = a \cdot b$, then $\frac{\partial z}{\partial a} = b$ and $\frac{\partial z}{\partial b} = a$



Code

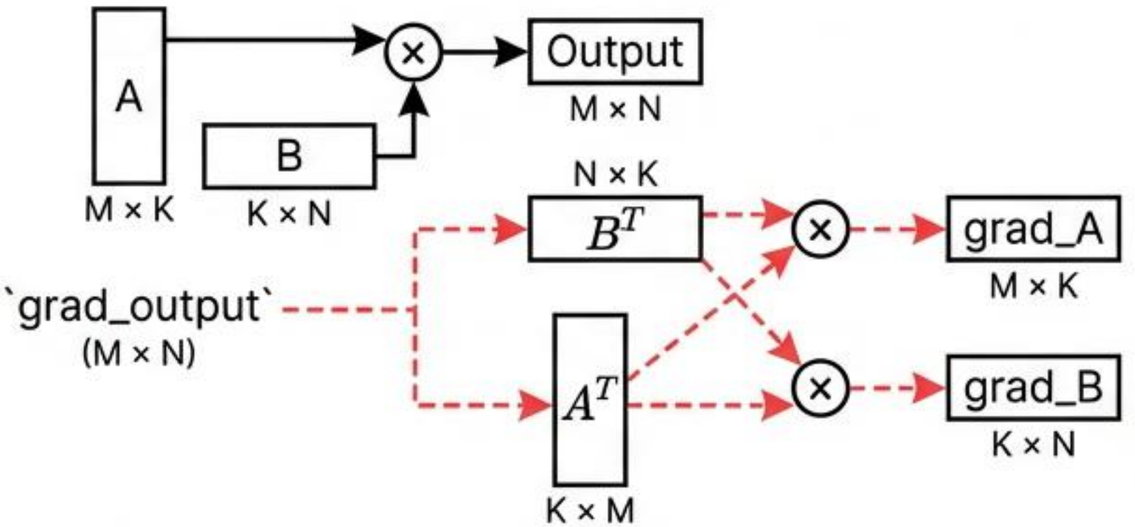
```
class MulBackward(Function):
    def apply(self, grad_output):
        # Retrieve from memory
        # Scale by the 'other' input
        grad_b = grad_output * a.data
```

Gradients are scaled by the value of the other input.

Implementation: MatmulBackward

Matrix Calculus requires Transpose (T)

Backward: Show Output Gradient being multiplied by B^T (B flipped on side) to match shape of A.



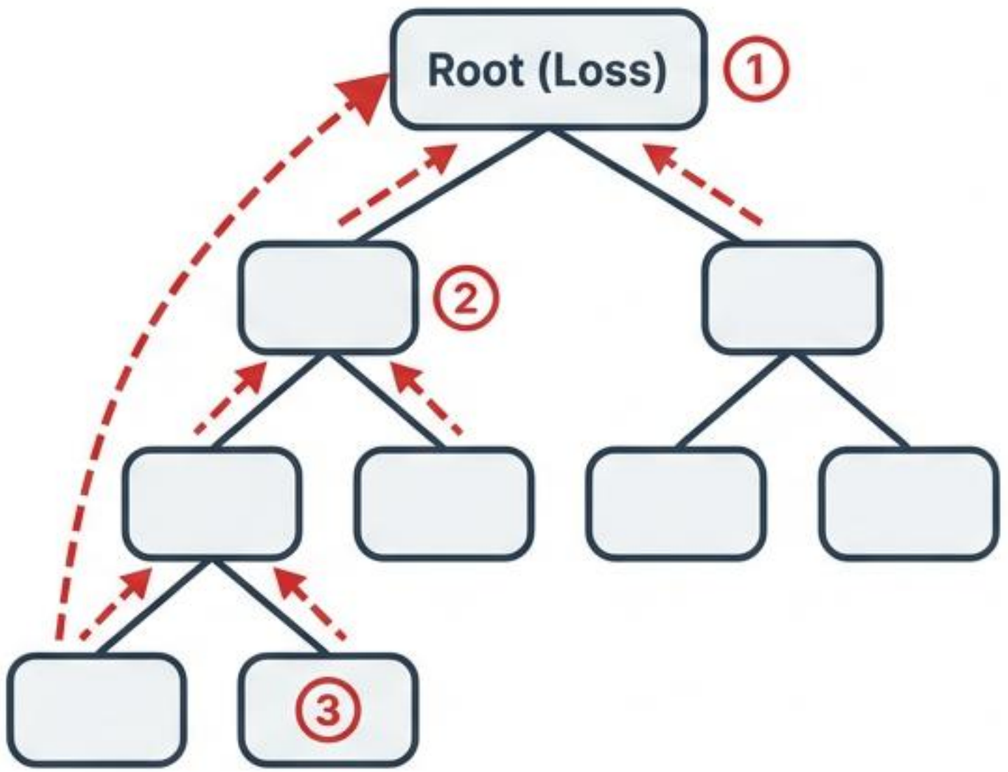
Code

```python
class MatmulBackward(Function):
    def apply(self, grad_output):
        a, b = self.saved_tensors

        # Align shapes by transposing the partner matrix
        grad_a = np.matmul(grad_output, b.data.T)
        grad_b = np.matmul(a.data.T, grad_output)

        return grad_a, grad_b
```

The heavy lifter of Linear Layers and Attention.

The Engine: Recursive Backward

Concept: Reverse-Mode Differentiation



Traverse upwards from Loss via Depth-First Search.

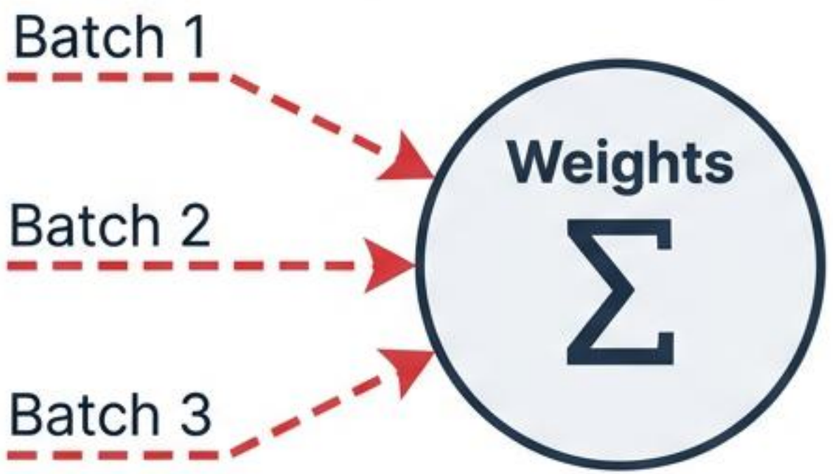
Implementation: Code

```
def backward(self, gradient=None):
  if self._grad_fn is not None:
    # 1. Compute local gradients
    grads = self._grad_fn.apply(gradient)

    # 2. Recurse to parents
    for tensor, grad in
  zip(self._grad_fn.saved_tensors, grads):
      tensor.backward(grad)
```

State Management: Gradient Accumulation

We sum gradients (`+=`), we do not overwrite them.

Concept



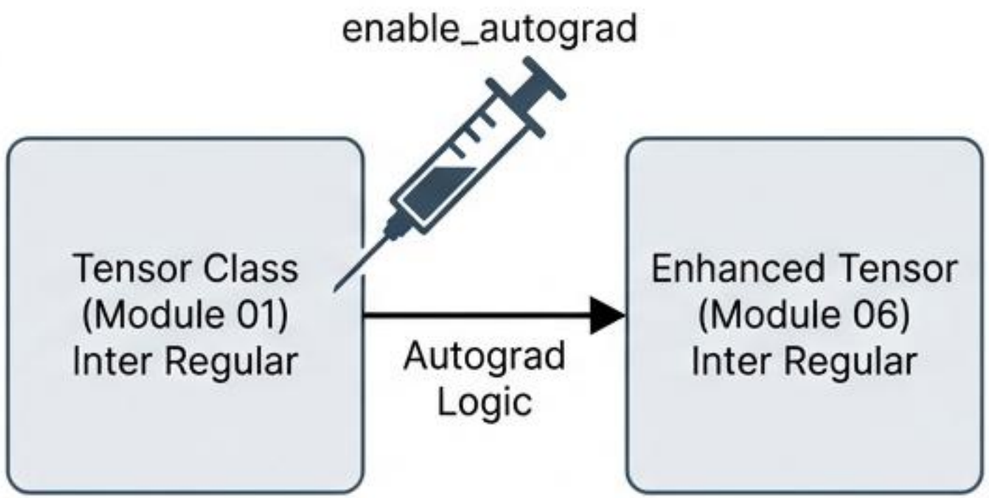
Code

```python
# In Tensor.backward()
if self.grad is None:
    self.grad = np.zeros_like(self.data)

self.grad += gradient # <--- Accumulate!
```

System Insight
Why? Enables mini-batching. **Risk:** Must call `zero_grad()` or history never clears.

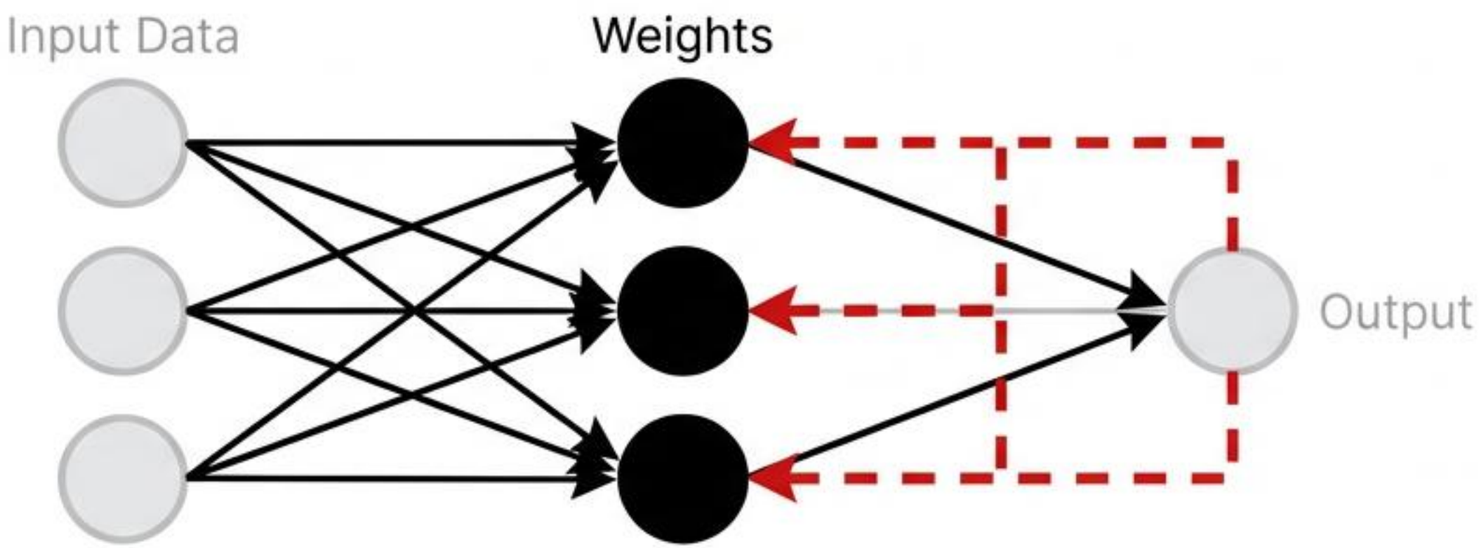
Integration: Monkey-Patching

```
def enable_autograd():
    # Inject methods into the existing class
    Tensor.__add__ = tracked_add
    Tensor.backward = backward

def tracked_add(self, other):
    result = _original_add(self, other)

    # The graph builds itself as a side-effect
    if self.requires_grad:
        result._grad_fn = AddBackward(self, other)

    return result
```



Optimization: The `requires_grad` Flag

```
# Data usually doesn't need gradients
x = Tensor(data, requires_grad=False)

# Weights MUST have gradients
W = Tensor(weights, requires_grad=True)
```



In a typical batch, >90% of tensors (input data) do not need gradient updates. This saves massive memory.

Safety: The "In-Place" Trap

❌ WRONG: Modifying data underneath the graph

```
# ❌ WRONG: Modifying data underneath the graph
x = Tensor([1.0], requires_grad=True)
y = x * 2
x.data[0] = 999
# Corrupts history stored for y!
```

The backward pass will read 999, not 1.0.

✅ RIGHT: Create new tensors

```
# ✅ RIGHT: Create new tensors
x = Tensor(x.data - lr * x.grad)
```

Inputs must be immutable while the graph is alive.

Synthesis: The Concept Map

`tensor.py`	The User Interface (Data Holder)	
`autograd.py (Function)`	The Node Logic (History & Memory)	
`autograd.py (Backward)`	The Math (Chain Rule & Recursion)	
`enable_autograd()`	The Wiring (Monkey-Patching)	

Summary & What's Next

Takeaways

✅ **Graph Construction:** Dynamic, built during forward pass ('Define-by-Run').

✅ **Memory Cost:** `Function` nodes store inputs needed for derivatives.

✅ **Execution:** `backward()` recursively traverses the graph applying the Chain Rule.

Next: Module 07

Optimizers

SGD, Adam, and Parameter Updates.

We have the gradients... now how do we use them to learn?