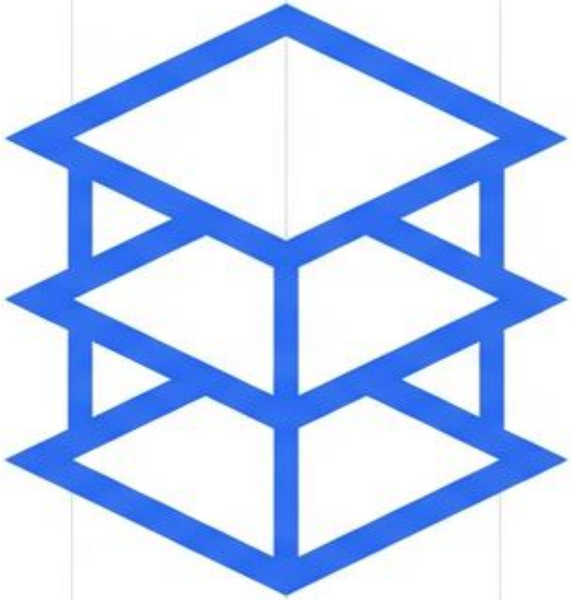
FOUNDATION TIER

MODULE 03

Neural Building Blocks

Constructing the fundamental layers that form neural networks



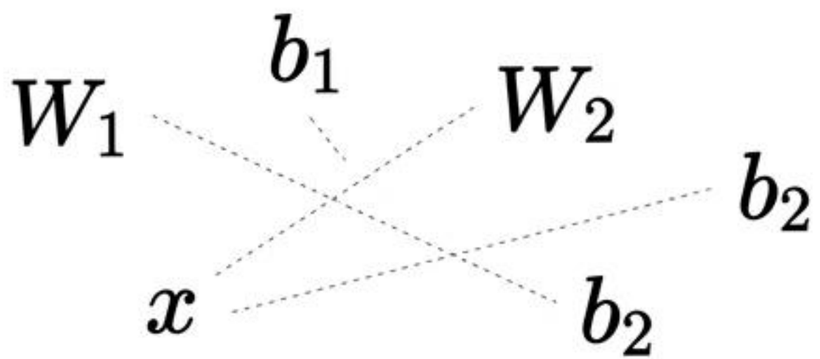
Module 03: Layers

From Tensor Arithmetic to Neural Building Blocks

TinyTorch Curriculum | Foundation Tier

Why We Need an Abstraction

The Problem: Loose Wires

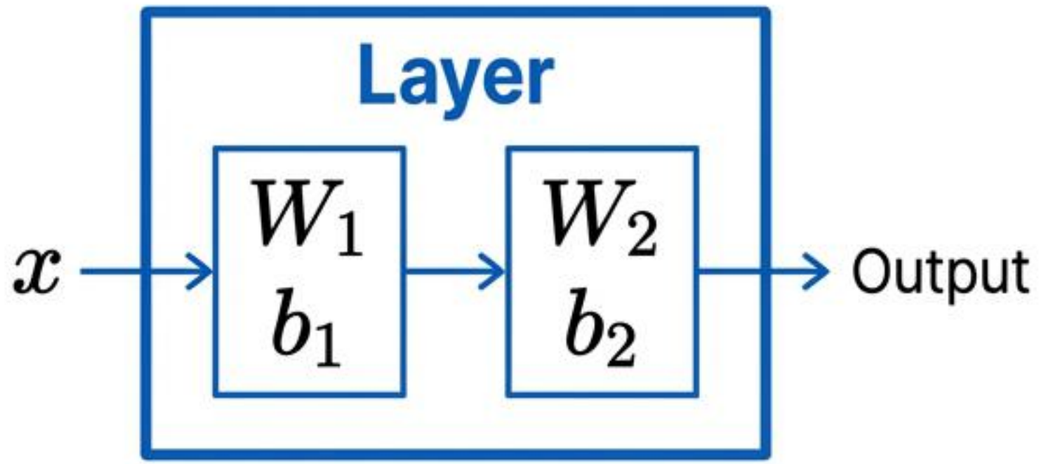


- Manual management of state.
- Complex parameter tracking.
- Prone to shape errors.

Code Snippet

```
y = x @ w + b
```

The Solution: Integrated Chips



- Encapsulates computation (Forward) and state (Parameters).
- Provides a consistent interface.

The Base Class Contract

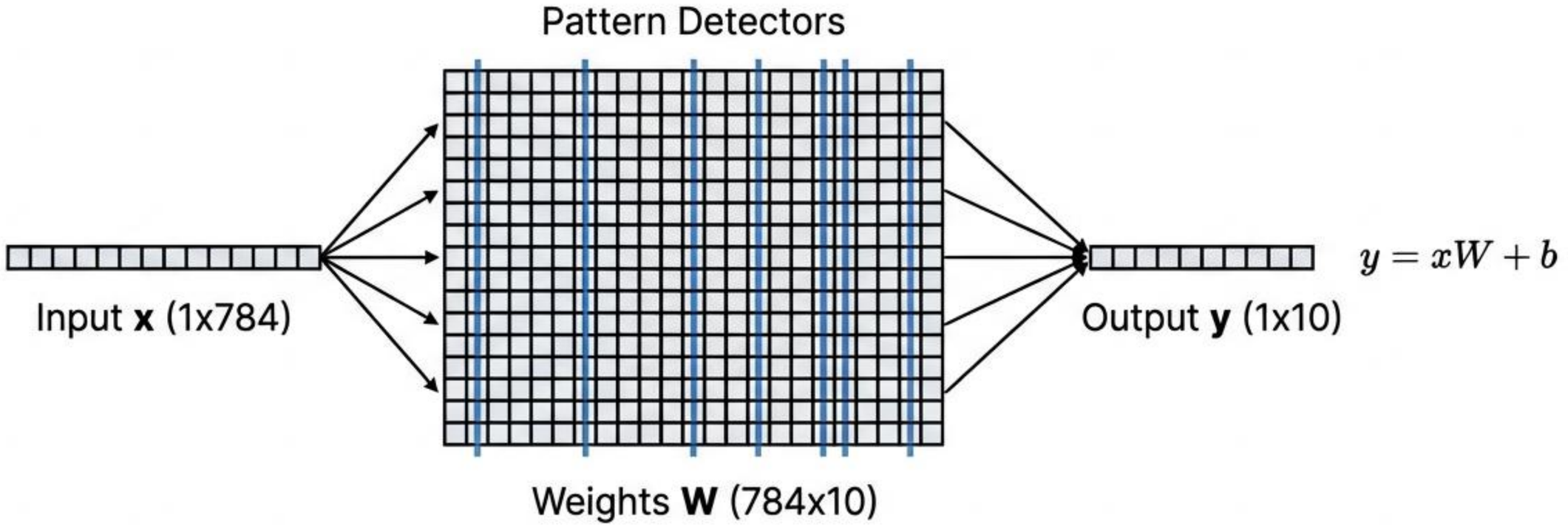
```
class Layer:
    def forward(self, x):
        """Compute layer output"""
        raise NotImplementedError

    def parameters(self):
        """Return list of trainable parameters"""
        return []

    def __call__(self, x, *args, **kwargs):
        return self.forward(x, *args, **kwargs)
```

Consistency: Every layer looks the same to the optimizer.

Callability: Enables intuitive `layer(x)` syntax.

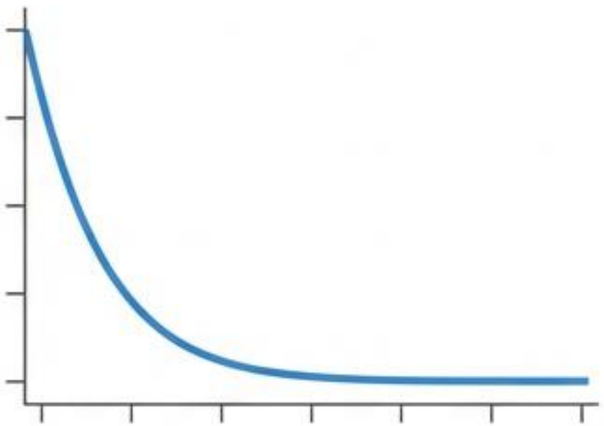
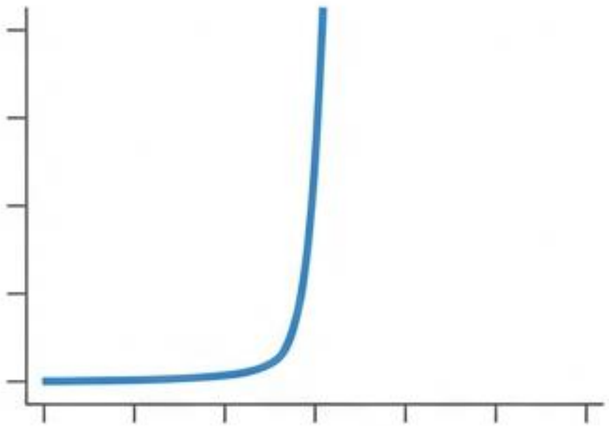
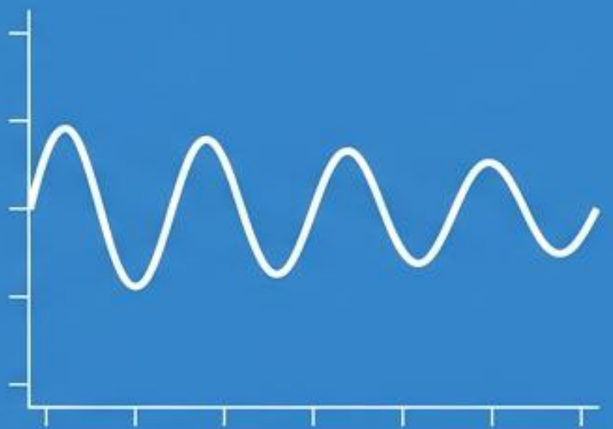
Discovery: A standard way to request trainable weights.

Concept: The Linear Layer



- **Role:** Feature Transformation (Pixels → Meaning)
- **Mechanism:** Dot product measures similarity between input and learned patterns.
- **Bias:** Shifts the activation function independent of input.

Systems Constraint: Initialization

Too Small	Too Large	Xavier Init
		
Vanishing Gradients	Exploding Gradients	Stable Variance

The Solution: Scale random weights by $\sqrt{(1 / fan\_in)}$ to preserve signal variance.

Code: Linear Initialization

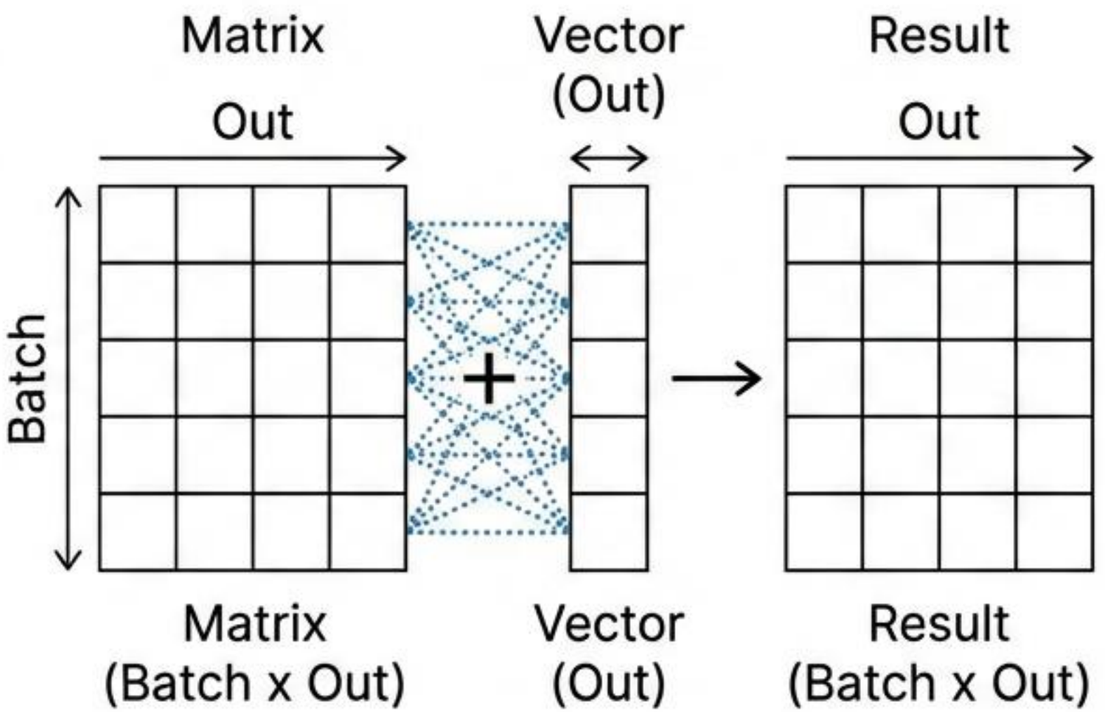
```
class Linear(Layer):
    def __init__(self, in_features, out_features, bias=True):
        self.in_features = in_features

        # Xavier Initialization
        scale = np.sqrt(1.0 / in_features)
        weight_data = np.random.randn(in_features, out_features) * scale

        self.weight = Tensor(weight_data)

        if bias:
            self.bias = Tensor(np.zeros(out_features))
        else:
            self.bias = None
```

Variance Preservation

Registering State

Code: Linear Forward

```python
def forward(self, x):
    """y = xW + b"""

    # 1. Matrix Multiplication
    output = x.matmul(self.weight)

    # 2. Bias Addition (Broadcasting)
    if self.bias is not None:
        output = output + self.bias

    return output
```



Code: Parameter Management

```
    def parameters(self):
        """Return list of trainable parameters"""
        params = [self.weight]

        if self.bias is not None:
            params.append(self.bias)

        return params
```

The Layer acts as a registry. It is responsible for reporting what it owns to the Optimizer. If the bias is disabled, it is excluded from the list.

Systems Reality: Memory Footprint

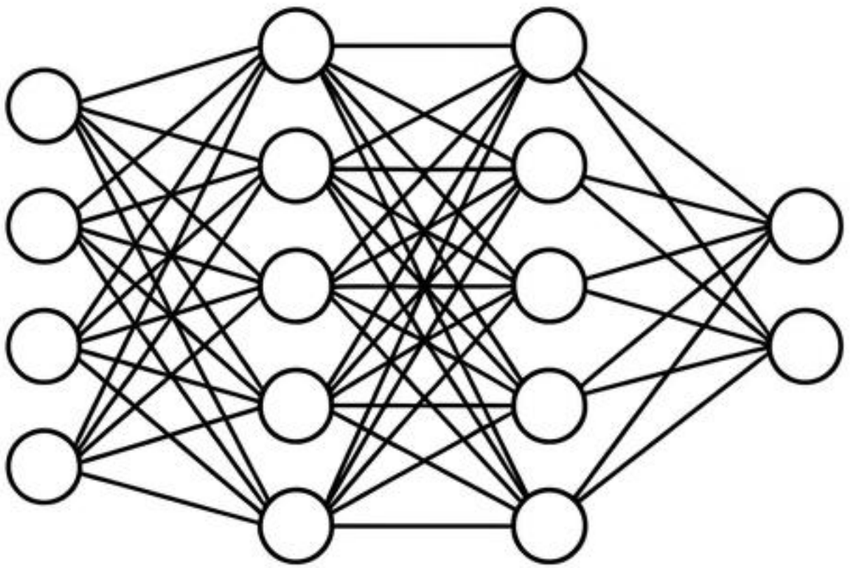
Case Study: Linear(784, 256) [Float32]

- Weight Matrix: $784 \times 256 \times 4$ bytes ≈ 800 KB
- Bias Vector: 256×4 bytes ≈ 1 KB

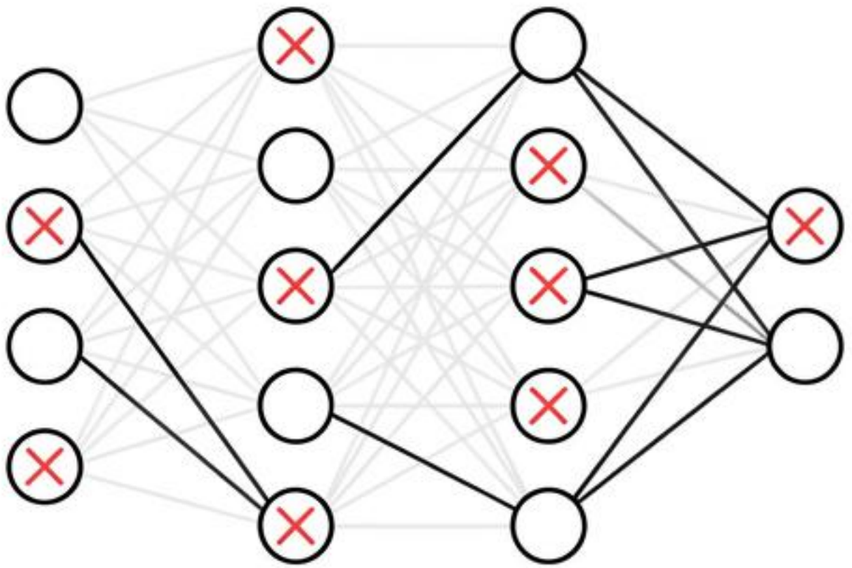
Linear layers are memory-heavy. Doubling the width quadruples the parameter count.

Concept: Dropout

Standard Training



Dropout Training (p=0.5)

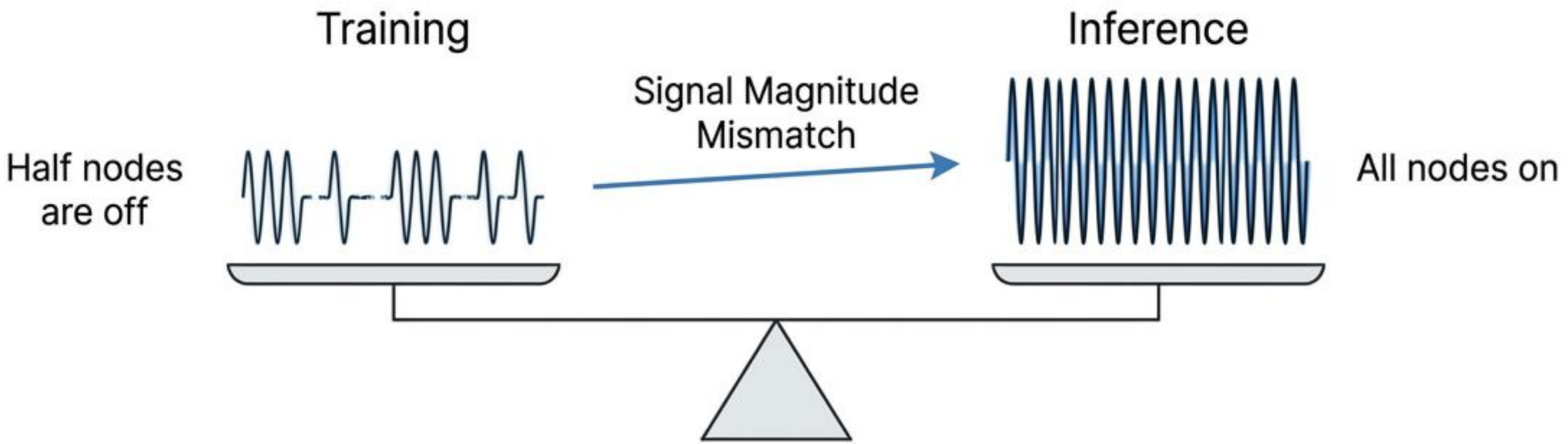


Standard Training

Dropout Training (p=0.5)

The Problem: Overfitting (Memorization).
The Solution: Randomly disable neurons during training to force redundancy.
Intuition: No single neuron can be a single point of failure.

Constraint: The Scaling Invariant



Inverted Dropout Solution: To fix the mismatch, we boost the signal during training.

$$\texttt{Scale Factor} = \frac{1}{1 - p}$$

If p=0.5 (50% drop), we multiply survivors by **2.0** to maintain expected magnitude.

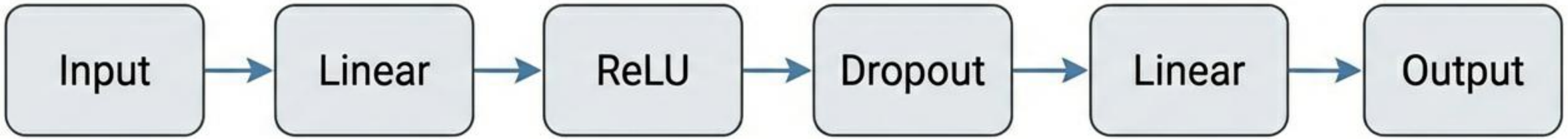
Code: Dropout Implementation

```python
class Dropout(Layer):
    def __init__(self, p=0.5):
        self.p = p

    def forward(self, x, training=True):
        if not training: return x

        # 1. Create Mask
        keep_prob = 1.0 - self.p
        mask = np.random.random(x.data.shape) < keep_prob

        # 2. Scale Factor (Inverted Dropout)
        scale = 1.0 / keep_prob

        # 3. Apply
        return x * Tensor(mask) * Tensor(scale)
```

Pass-through at Inference

Invariant Enforcement

Concept: Composition



$$y = f_2(\text{dropout}(\sigma(f_1(x))))$$

Deep Learning is simply function composition.
We need a container to manage the pipeline.

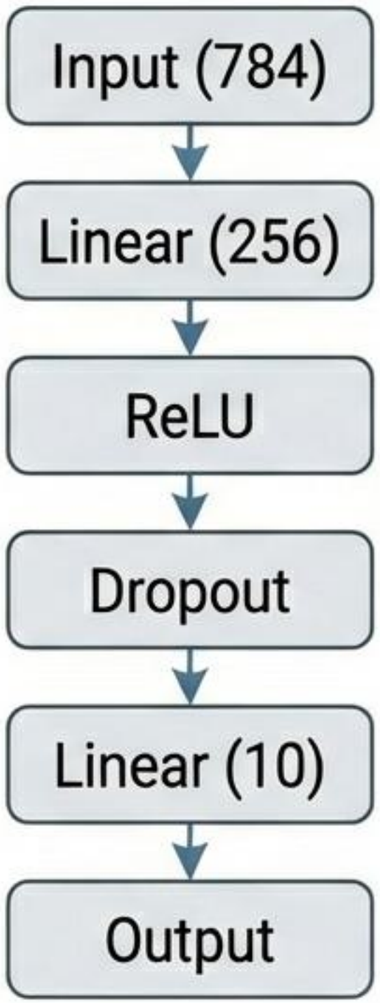
Code: The Sequential Container

```
class Sequential(Layer):
    def __init__(self, *layers):
        self.layers = list(layers)

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

    def parameters(self):
        params = []
        for layer in self.layers:
            params.extend(layer.parameters())
        return params
```

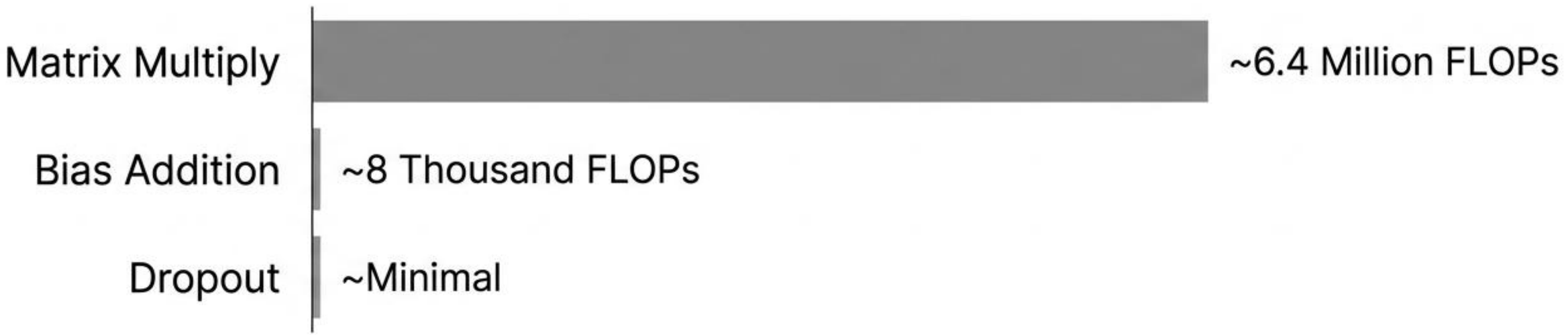
Simplicity is power. `forward` loops through computation. `parameters` recursively collects state.

Putting It Together

```
model = Sequential(
    Linear(784, 256),
    ReLU(),
    Dropout(0.5),
    Linear(256, 10)
)
```



Systems Analysis: Compute Intensity



For Linear(784, 256) with Batch=32.

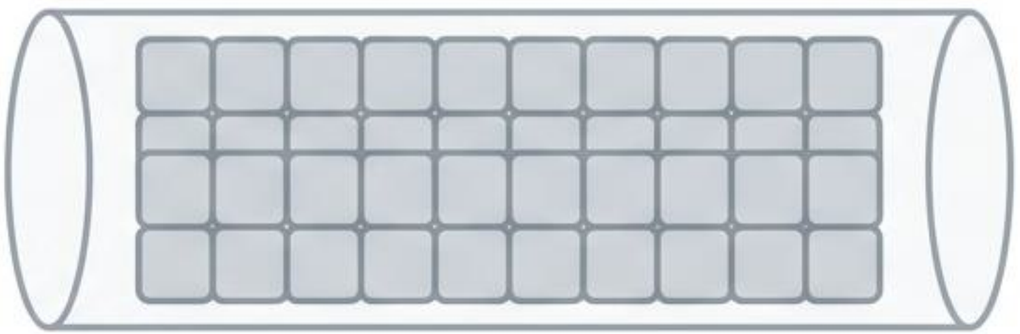
99.9% of compute time is spent in Matrix Multiplication.
Optimizing Deep Learning = Optimizing Matmul.

Systems Analysis: The Power of Batching

Batch Size = 1

~800 samples/sec

Batch Size = 32



~10,000 samples/sec

Why Batch?

- 1. Amortizes Python 1. Amortizes Python overhead.
- 2. Vectorization (SIM 2. Vectorization (SIMD) efficiency.
- 3. Cache locality. 3. Cache locality.

Common Failure Modes

Shape Mismatch
Symptom: `ValueError: (32, 128) @ (256, 10)`
Fix: Output of Layer N must match Input of Layer N+1.

The Inference Bug
Symptom: Training loss is good, but Test accuracy is terrible.
Fix: Forgot `training=False` during evaluation. Dropout is still killing signals.

Silent Failure
Symptom: Model doesn't learn.
Fix: `parameters()` method is incomplete. Optimizer cannot see the weights.

TinyTorch vs. PyTorch

	TinyTorch	PyTorch
API	`Linear(in, out)`	`nn.Linear(in, out)`
Container	`Sequential`	`nn.Sequential`
Parameters	returns `List`	returns `Generator`
Backend	NumPy	C++ / CUDA

You have implemented the core API of the industry standard.
The logic is identical; only the engine differs.

Module Summary

- [✓] **Abstraction**: Unified `Layer` interface
- [✓] **Linear**: The learned transformation ($y = xW + b$)
- [✓] **Dropout**: Robustness via randomness
- [✓] **Sequential**: Composition logic
- [✓] **State**: Parameter management

Coming Next: Module 04

LOSS FUNCTIONS

The network can predict, but it doesn't know if it's right or wrong. We need to measure error to drive learning.