FOUNDATION TIER

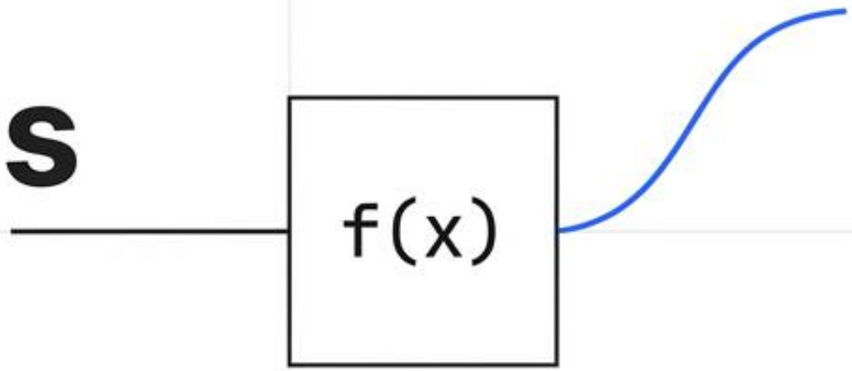
MODULE 02

Activations

Deep learning's nonlinear engine and the functions that enable learning

TINYTORCH FOUNDATION TIER

Module 02: Activations



Intelligence Through Nonlinearity

From Linear Algebra to Neural Computation

Prerequisites: Module 01 (Tensor)
Goal: Implement ReLU, Sigmoid, Tanh, GELU, Softmax

The Linear Trap

The Math of Collapse

W_1 = First Layer Weights

W_2 = Second Layer Weights

$f(x) = W_2(W_1x)$

$f(x) = (W_2W_1)x$

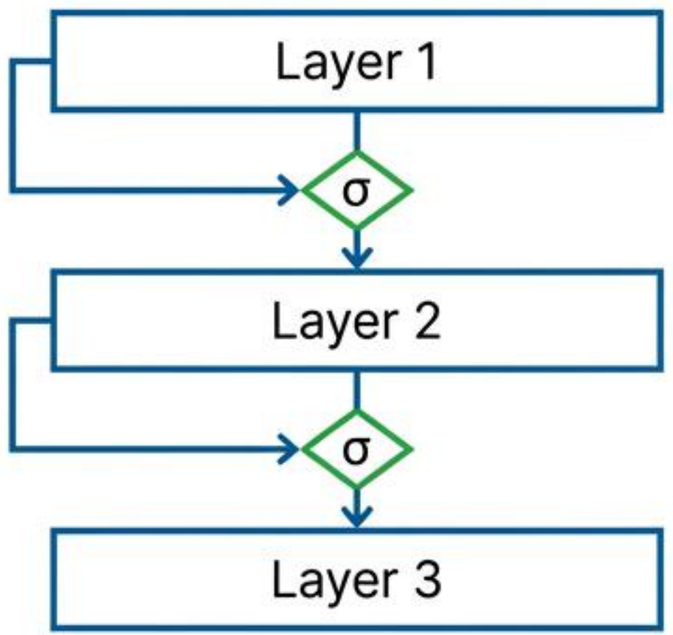
$f(x) = W_{new} \cdot x$

Result: Depth collapses.

A 100-layer linear network is mathematically identical to a 1-layer network.

The Activation Fix

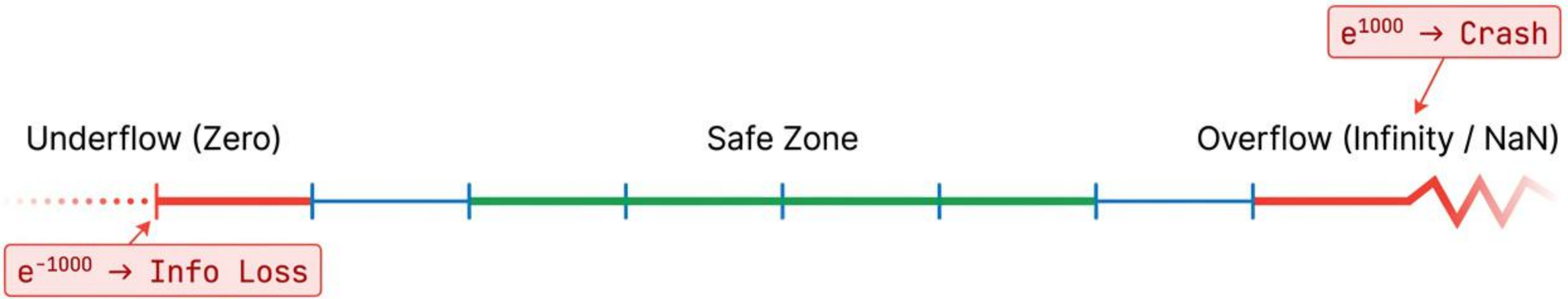
$f(x) = \sigma(W_2 \cdot \sigma(W_1x))$



Nonlinearity prevents matrix collapse, enabling hierarchical feature learning (Edges → Shapes → Objects)

Systems Reality: Math vs. Machine

Why implementation is harder than theory.



The Ideal Math

f(x) = e^x is continuous and infinite.

The Engineering Constraint

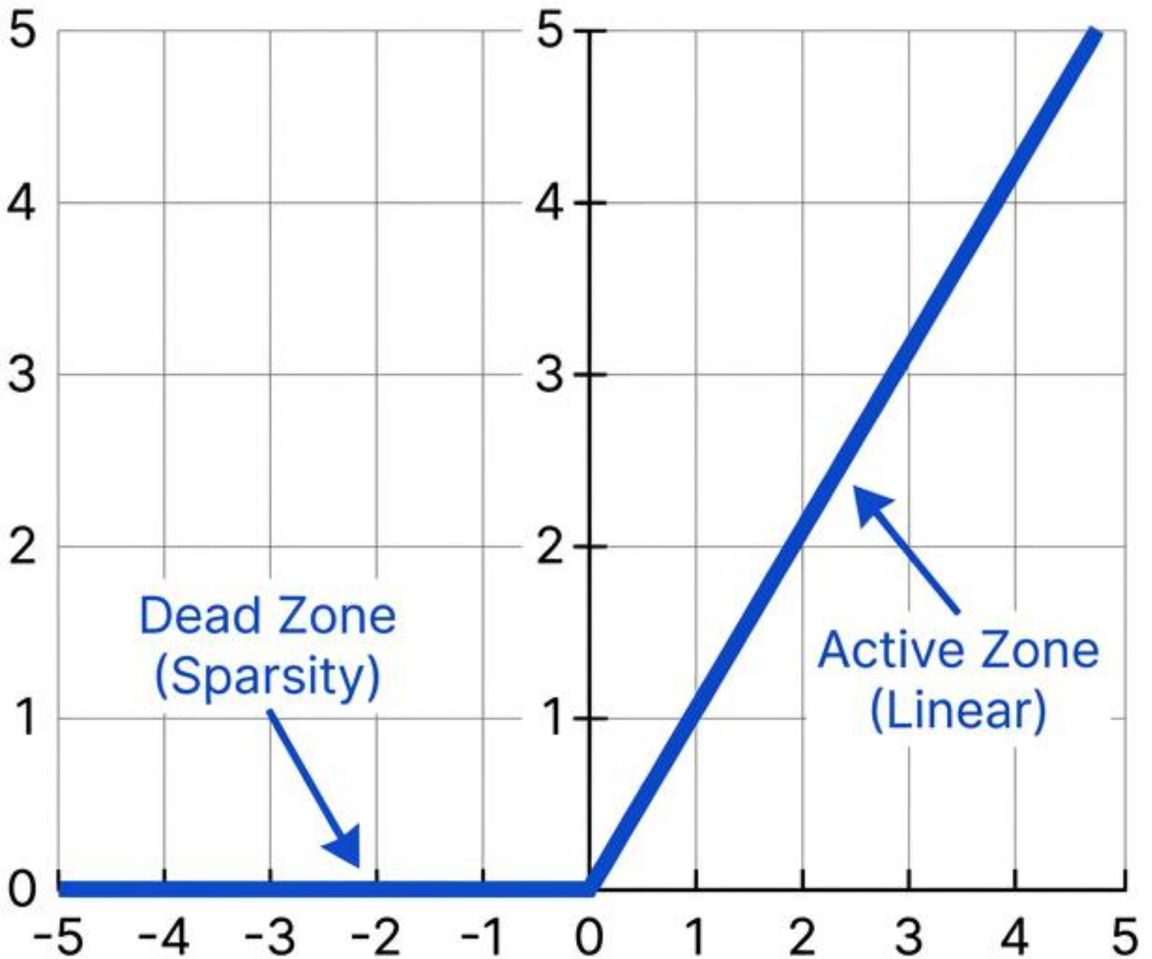
Exponential functions are **expensive** (~4x cost of multiplication). **Implementation must prioritize Numerical Stability.**

ReLU (Rectified Linear Unit)

Role: The default engine of modern Deep Learning.

Formula: `f(x) = max(0, x)`

- **Sparsity:** Negative inputs become exact zeros.
- **Efficiency:** Multiplying by zero is computationally free.
- **Gradient:** No vanishing gradient for positive values.



Implementation: ReLU

tinytorch/core/activations.py

```
class ReLU:
    def forward(self, x: Tensor) -> Tensor:
        """
        Apply ReLU: max(0, x)
        Cost: 1x (Baseline)
        """
        # Element-wise maximum. Fast and heavily optimized in C.
        result = np.maximum(0, x.data)
        return Tensor(result)
```

Systems Insight

Performance Note:
We rely on 'np.maximum'. It maps to optimized C code. This sets the baseline cost (1x) for all other activations.

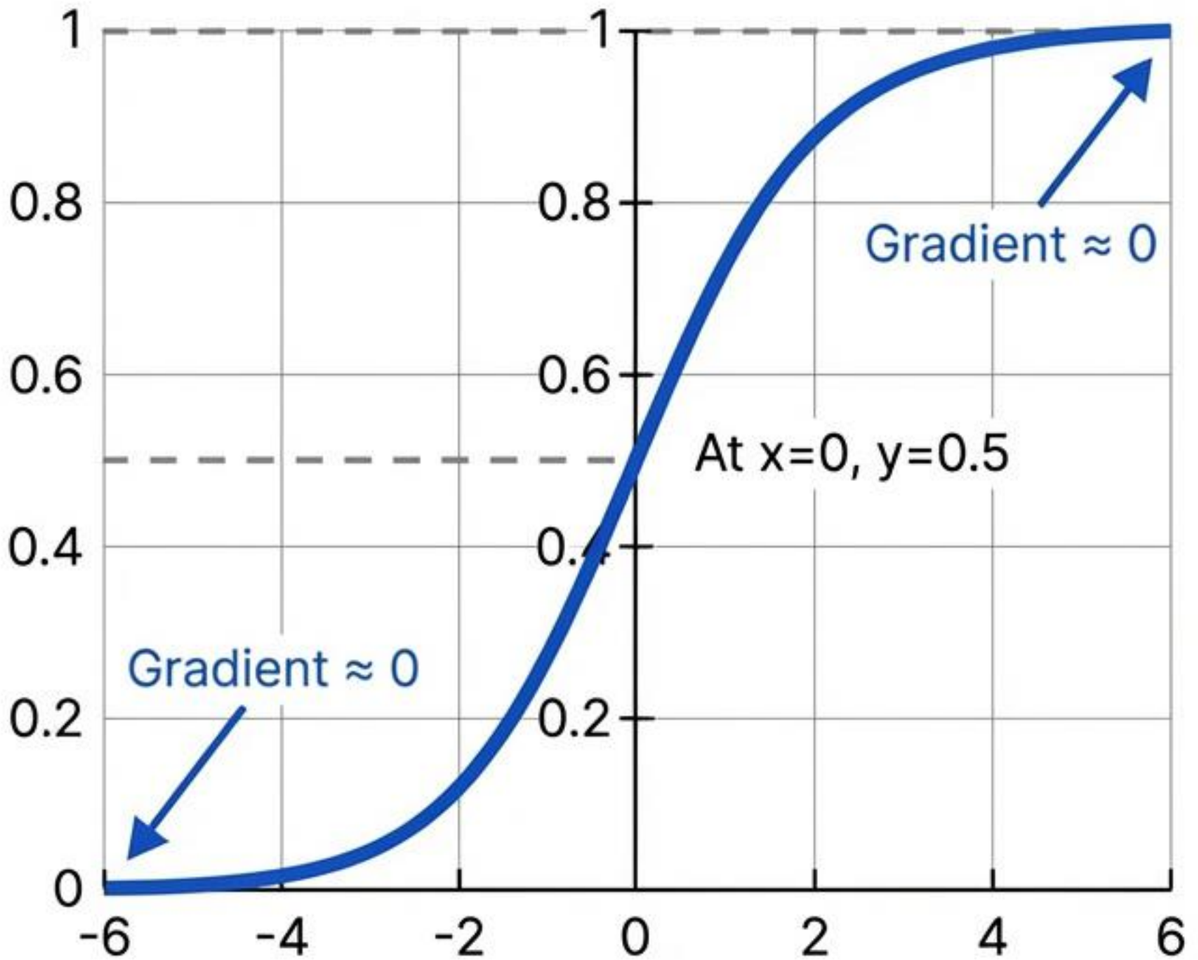
Sigmoid: The Probability Gate

Role: Binary classification outputs (e.g., 'Is this spam?').

Formula: $\sigma(x) = 1 / (1 + e^{-x})$

Range: (0, 1)
Cost: ~3-4x slower than ReLU (requires division and exponential).

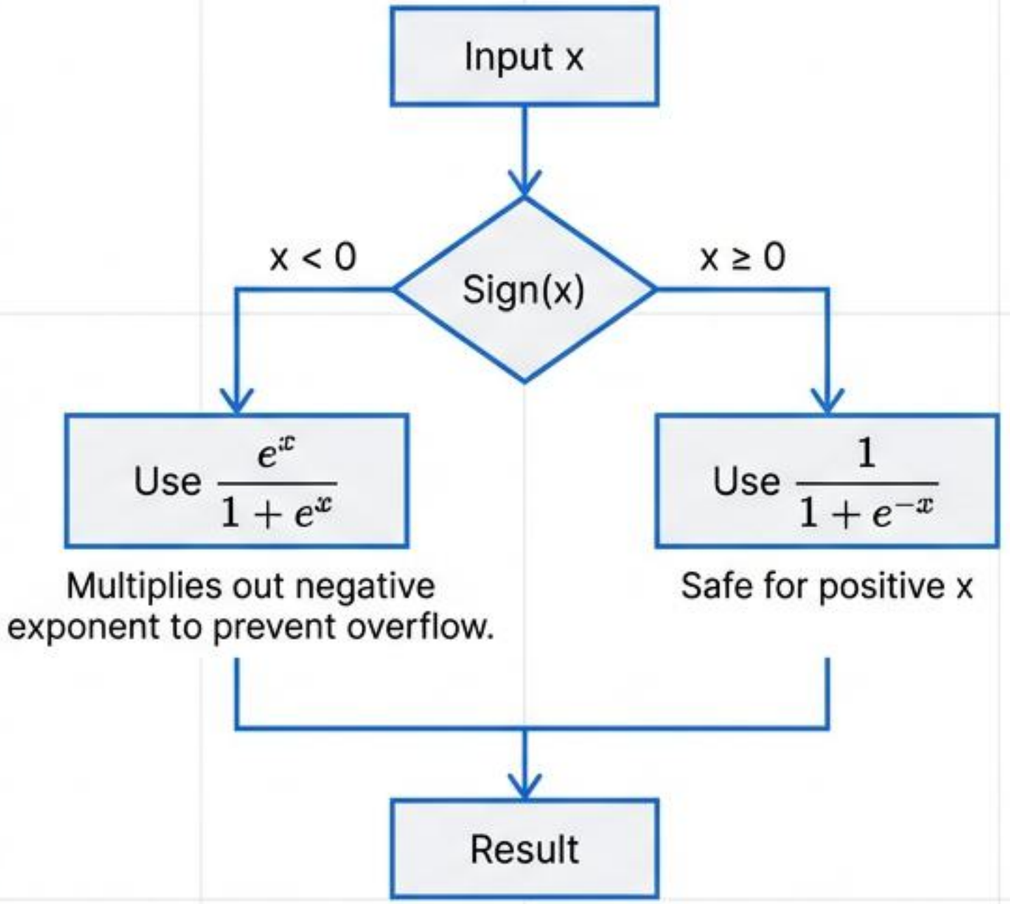
Risk: Saturation. Gradients vanish at extremes, killing training in deep networks.



Implementation: Stable Sigmoid

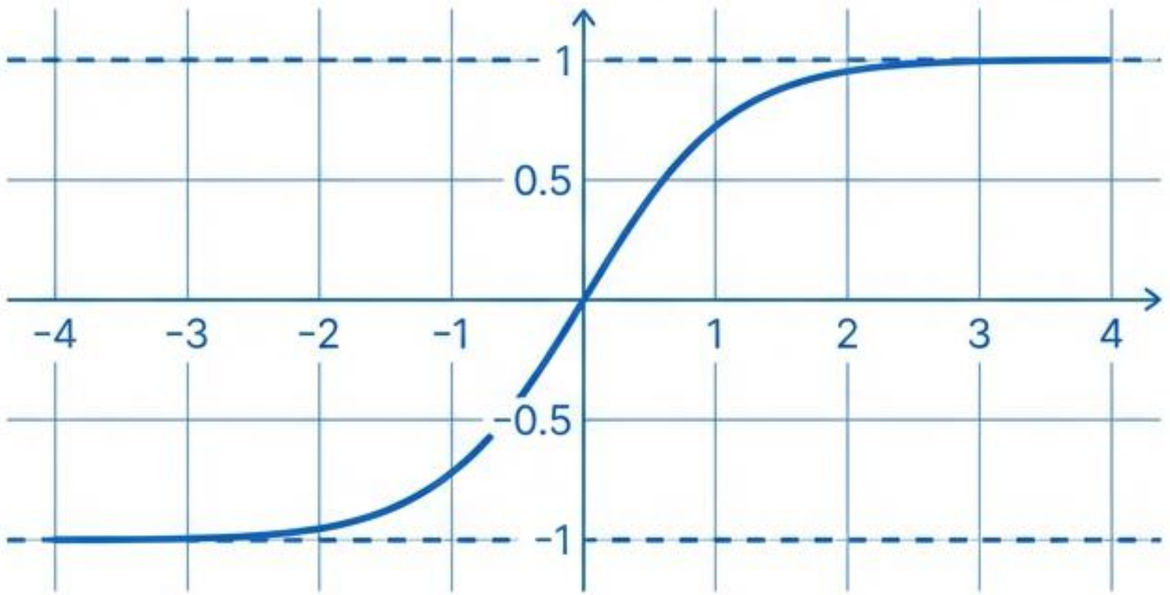
Handling Overflow and Underflow

```python
def forward(self, x: Tensor) -> Tensor:
    # 1. Clip to prevent raw overflow
    z = np.clip(x.data, -500, 500)

    # 2. Stable computation mask
    result = np.zeros_like(z)

    # Positive input: Standard formula
    pos_mask = z >= 0
    result[pos_mask] = 1.0 / (1.0 + np.exp(-z[pos_mask]))

    # Negative input: Alternative formula
    neg_mask = z < 0
    exp_z = np.exp(z[neg_mask])
    result[neg_mask] = exp_z / (1.0 + exp_z)

    return Tensor(result)
```



Tanh (Hyperbolic Tangent)

Role: Recurrent Neural Networks (RNNs).
Concept: Zero-centered output aids gradient flow.
Formula: $\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$

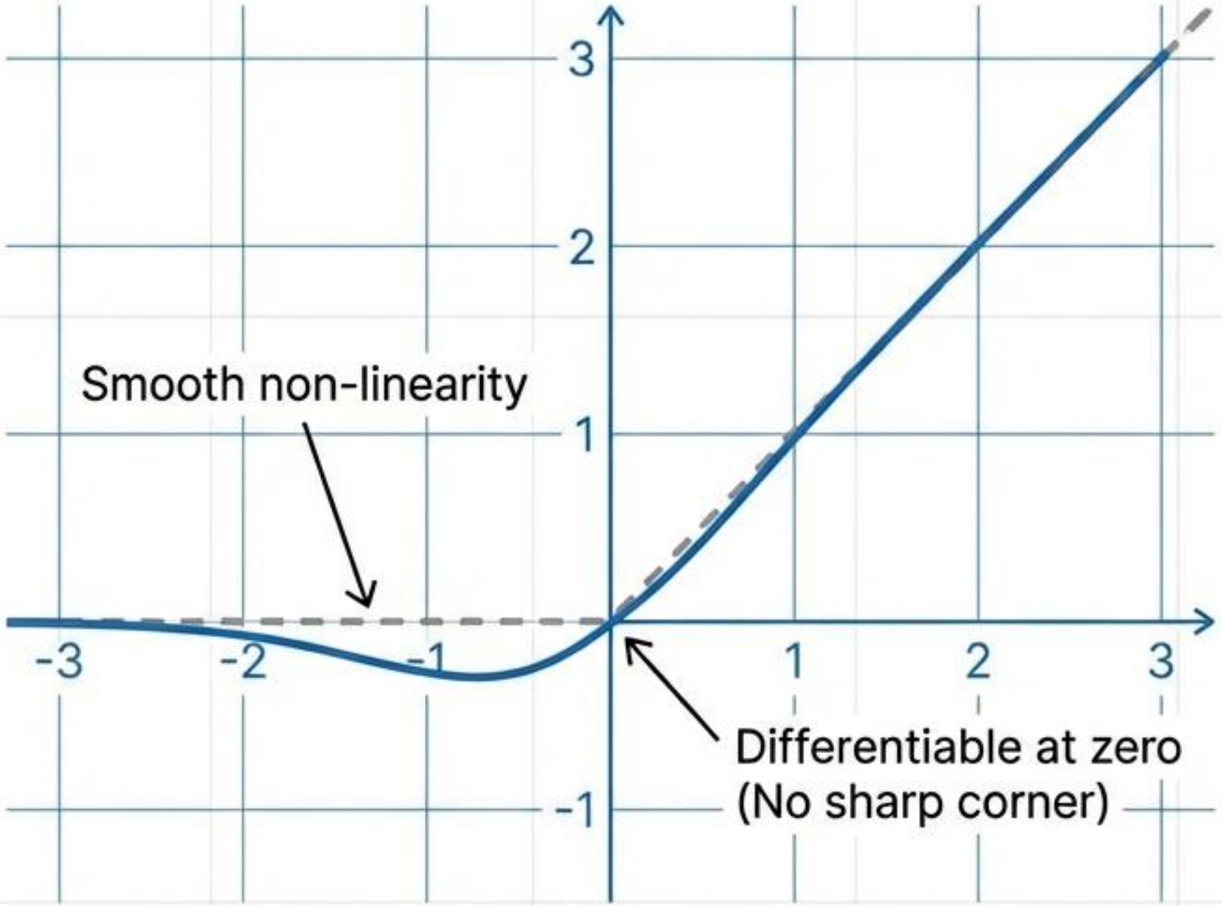


Range (-1, 1)

```
class Tanh:
    def forward(self, x: Tensor) -> Tensor:
        # Relies on NumPy's internal optimization
        return Tensor(np.tanh(x.data))
```

GELU (Gaussian Error Linear Unit)

The Transformer Standard (BERT, GPT, LLMs)



Formula: $x \cdot \Phi(x)$
(where $\Phi(x)$ is the cumulative distribution function of the standard Gaussian distribution)

Insight: Weights inputs by their percentile in a Gaussian distribution.

Benefit: Smoothness aids optimization in high-dimensional spaces.

Implementation: GELU Approximation

Trading Exactness for Speed

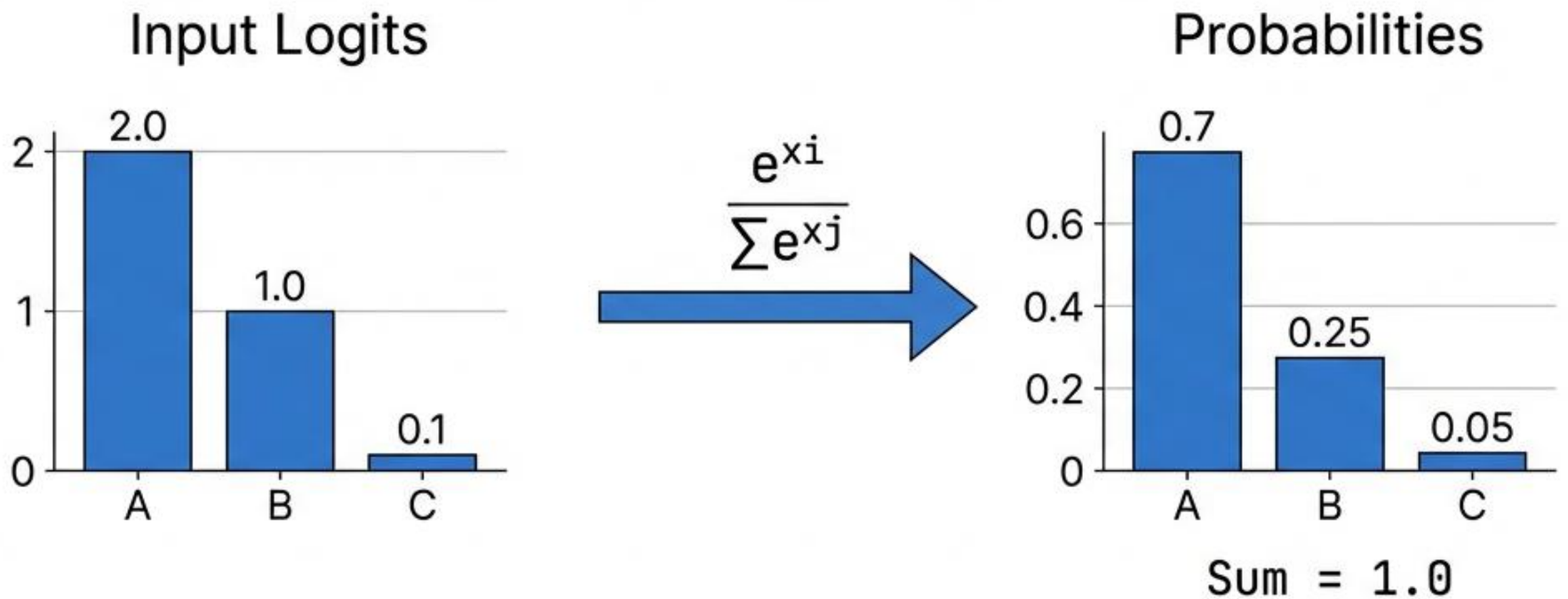
- Calculating the exact Gaussian CDF is computationally expensive.
- We use a high-precision approximation.
- Cost: ~4-5x slower than ReLU (due to exponential + div).
- Design Decision: In billion-parameter models, the accuracy gain justifies the compute cost.

```
def forward(self, x: Tensor) -> Tensor:
    # Approximation: x * sigmoid(1.702 * x)
    # 1.702 is derived from sqrt(2/pi)

    sigmoid_part = 1.0 / (1.0 + np.exp(-1.702 * x.data))
    return Tensor(x.data * sigmoid_part)
```

Magic Constant

Softmax: The Distribution Builder

Coupled Activation for Multi-Class Classification



Mechanism:

1. Amplifies large values super-linearly.
2. Normalizes output to sum to 1.
3. Dimensions are coupled (unlike element-wise ReLU/Sigmoid).

Implementation: Softmax Stability

The Max-Subtraction Trick

The Problem

- Direct computation of e^{1000} causes **Overflow (NaN)**.

The Mathematical Identity

$$\text{Softmax}(x) = \text{Softmax}(x - C)$$

We choose C = max(x). This ensures the largest exponent is $e^0 = 1$.

```
def forward(self, x: Tensor, dim: int = -1) -> Tensor:
  # 1. Shift values so max is 0 (Safe!)
  x_max = np.max(x.data, axis=dim, keepdims=True)
  x_shifted = x - Tensor(x_max)

  # 2. Compute safe exponentials
  exp_values = Tensor(np.exp(x_shifted.data))

  # 3. Normalize
  exp_sum = np.sum(exp_values.data, axis=dim, keepdims=True)
  return exp_values / Tensor(exp_sum)
```

The Activation Landscape

Decision Matrix

Activation	Relative Cost	Stability	Primary Use
ReLU	1x (Baseline)	High ✅	Hidden Layers (CNN/MLP)
GELU	4x	High ✅	Hidden Layers (Transformers)
Sigmoid	3x	Risky (Vanishing Grad) ❌	Binary Output Only
Softmax	5x	Requires Max-Trick ➡️	Multi-class Output Only

Heuristic: Start with ReLU. Optimize with GELU. Terminate with Softmax.

TinyTorch vs. PyTorch

API Alignment

TinyTorch (Your Build)

```
from tinytorch.core.activations
    import ReLU, Softmax

relu = ReLU()
y = relu(x)

# Multi-class
softmax = Softmax()
probs = softmax(logits)
```

PyTorch (Standard)

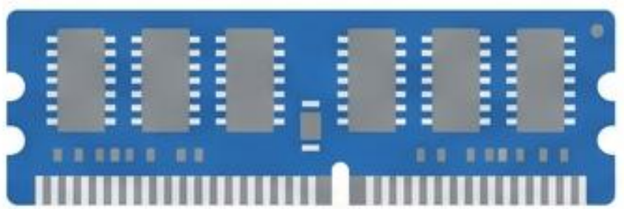
```
import torch.nn.functional as F

y = F.relu(x)

# Multi-class
probs = F.softmax(logits, dim=-1)
```

TinyTorch uses NumPy backends. PyTorch uses C++/CUDA backends. The logic is identical.

Systems Check: Scale & Cost

Reasoning about Resources

Memory Footprint



Scenario: Batch 32, Layer Size 4096, Float32

`32 × 4096 × 4 bytes ≈ 512 KB`

This memory must be preserved for Backpropagation (Module 06).

Compute Latency

ReLU: `1ms`

GELU: `5ms`

In a 100-layer model, this adds significant overhead. Only use GELU if accuracy justifies the cost.

Module Complete: What's Next?

Helvetica Now Display

Status Check

- ✅ Module 01: Data Representation (Tensor)
- ✅ Module 02: Nonlinearity (Activations)
- ☐ Module 03: Learnable Parameters (Layers)

Coming Up

Module 03: Layers

We will combine Tensors + Activations + Weights.

Goal: Building the fundamental atom of neural networks.

Repository Link: tinytorch/core/activations.py