Machine Learning

Imperial College London

Nuri Cingillioglu

Part 4: Connectionist Renaissance

The return of the ~~Jedi~~ Neural Networks

Why did linear models hit a ceiling?

Part 2, glorified curve-fitting. But the reality of perceptual data:

- For a modest 100×100 image ($d = 10^4$), we cannnot just hand-craft features. Polynomial expansion goes BOOM...

The Fundamental Question

Instead of manually engineering fixed non-linear functions or features $\phi(\mathbf{x})$, can we learn $\phi_{\mathbf{W}}(\mathbf{x})$ **end-to-end directly from raw data**?

This is the central thesis of **Deep Learning**.

Key Insight

Learn Intermediate Representations

$$\mathbf{x} \xrightarrow{\mathbf{W}^{(1)}} \mathbf{h}^{(1)} \xrightarrow{\mathbf{W}^{(2)}} \mathbf{h}^{(2)} \xrightarrow{\mathbf{W}^{(3)}} \cdots \xrightarrow{\mathbf{W}^{(L)}} \hat{\mathbf{y}}$$

Each layer transforms data into a new representation

Early Layers

Low-level sensory primitives (edges, gradients, frequencies)

Middle Layers

Combinatorial motifs (textures, contours, phonemes)

Deep Layers

High-level semantic parts (faces, wheels, words, concepts)

Multi-Layer Perceptrons (MLPs)

A Multi-Layer Perceptron (MLP) stacks affine transformations interleaved with element-wise non-linear activation functions $\sigma(\cdot)$:

For layer $l \in \{1, \ldots, L\}$:

$$\mathbf{h}^{(l)} = \sigma(\mathbf{z}^{(l)}) = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

- $\mathbf{h}^{(0)} = \mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$ (Input layer)
- $\mathbf{W}^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$ (Synaptic weight matrix)
- $\mathbf{b}^{(l)} \in \mathbb{R}^{d_l}$ (Bias vector)
- $\mathbf{z}^{(l)}$ (Pre-activation vector)
- $\mathbf{h}^{(l)}$ (Post-activation representation)

Also known as **feedforward neural networks (FFNNs)**.

Why the Non-Linearity

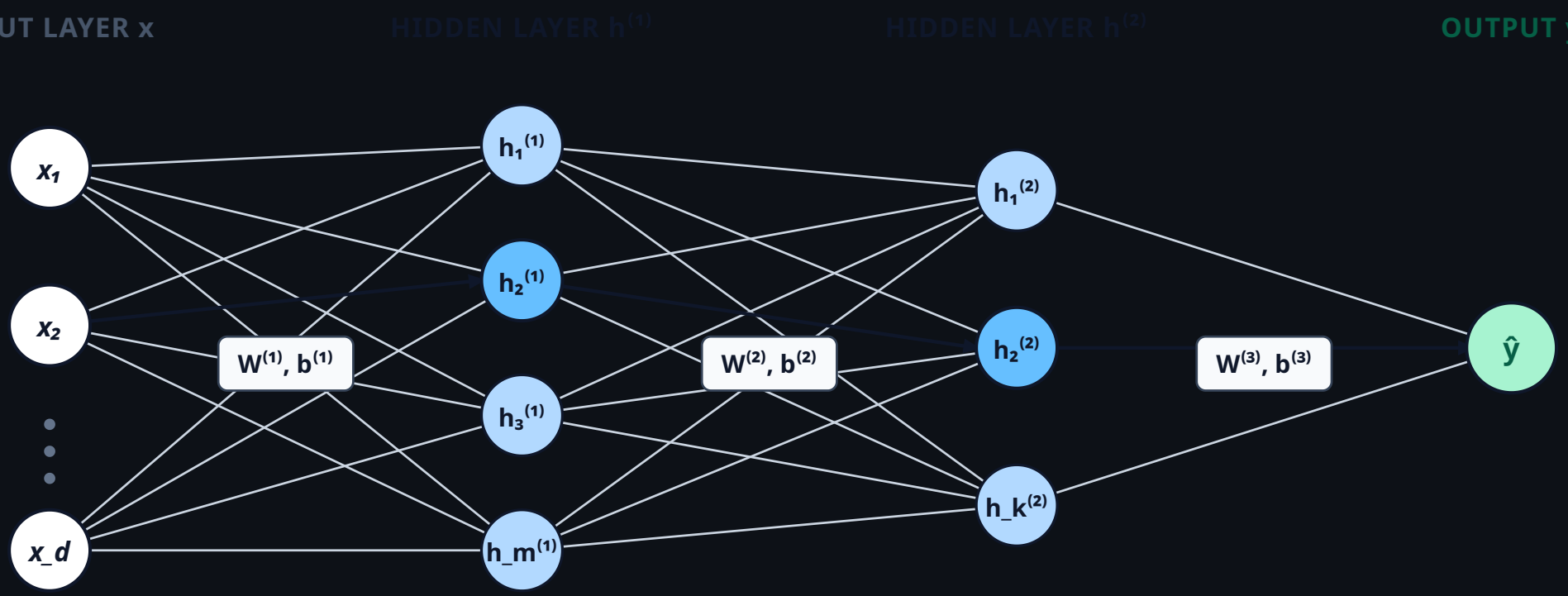
If $\sigma(z) = z$ (linear identity), the entire L-layer network collapses into a single matrix product:

$$\hat{\mathbf{y}} = \mathbf{W}^{(L)} \dots \mathbf{W}^{(2)} \mathbf{W}^{(1)} \mathbf{x} = \mathbf{W}_{\text{eff}} \mathbf{x}$$

Composition of linear operators is strictly linear.

> Please note the mathematical nature of this construction. While the biological inspiration is admirable, we are in not in Kansas anymore.

MLP - Feedforward Neural Network (FFNN)



Forward Pass: $h^{(l)} = \sigma(W^{(l)} h^{(l-1)} + b^{(l)})$ | $\hat{y} = g(W^{(L)} h^{(L-1)} + b^{(L)})$

Resolving the XOR Curse

$$\text{XOR}(x_1, x_2) = (x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$$

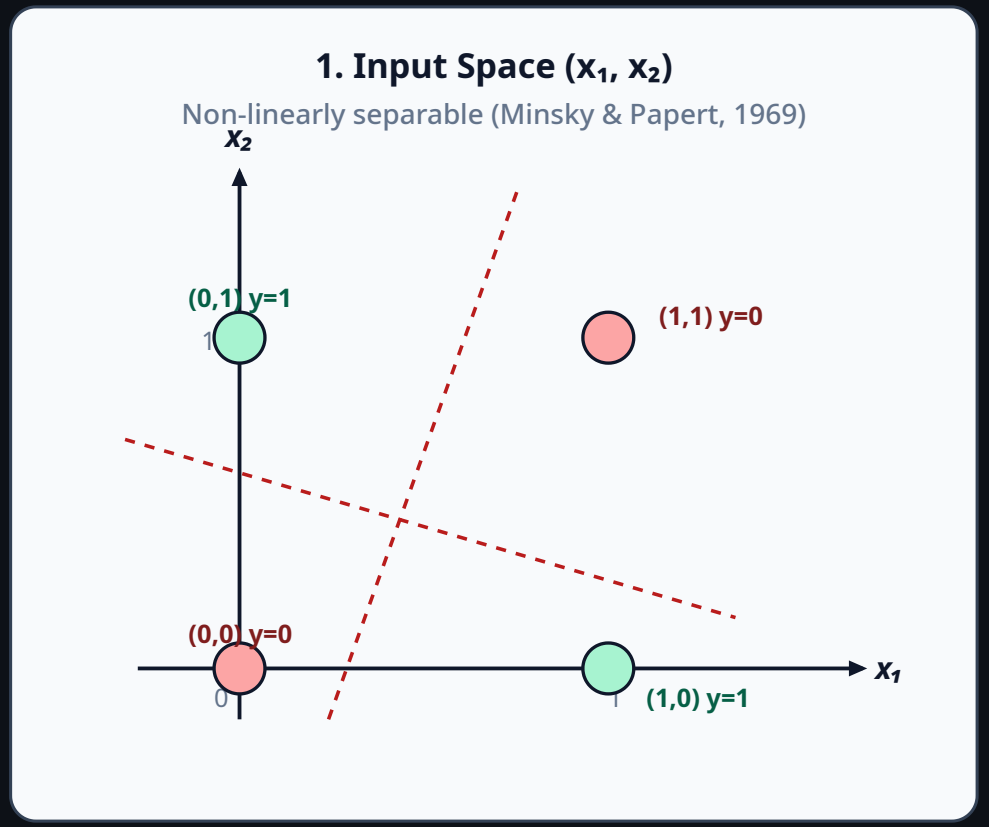
Consider a 2-layer network with step activation $H(z) = \mathbf{1}[z \geq 0]$:

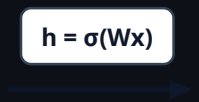
$$\mathbf{h} = H\left(\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} -0.5 \\ 1.5 \end{bmatrix}\right), \quad \hat{y} = H\left(\begin{bmatrix} 1 & 1 \end{bmatrix}\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} - 1.5\right)$$

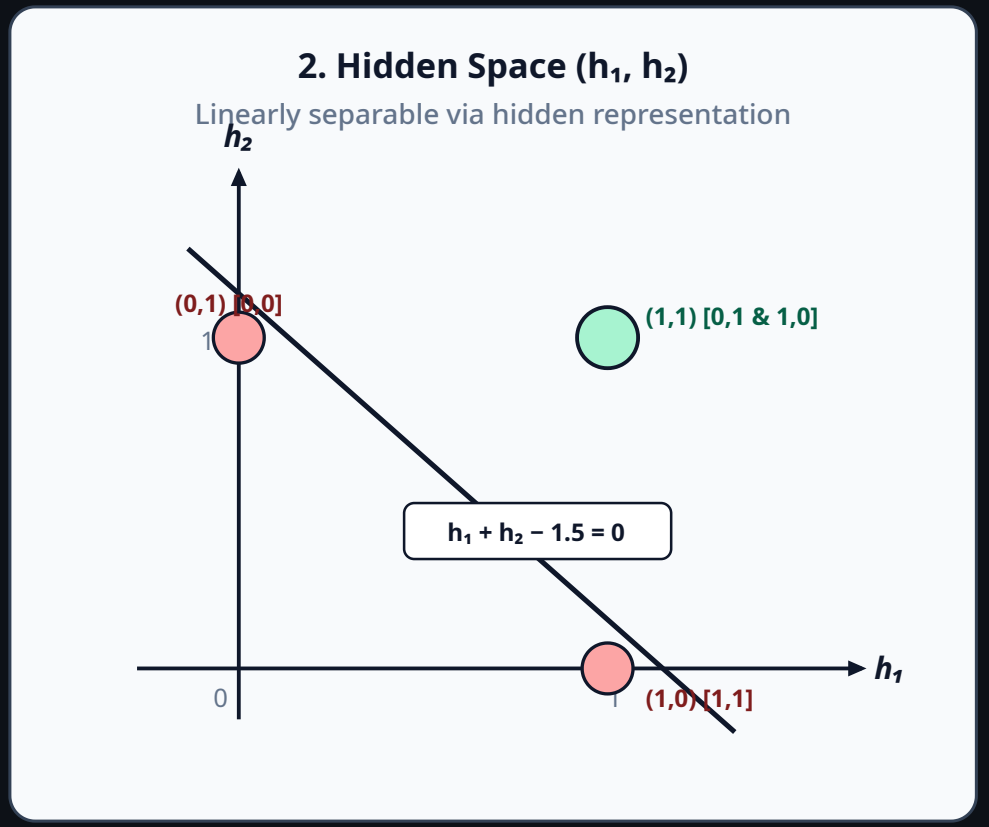
- $(0,0) \to \mathbf{z}^{(1)} = [-0.5, 1.5]^T \to \mathbf{h} = [0,1]^T \to \hat{y} = H(-0.5) = \mathbf{0}$
- $(0,1) \to \mathbf{z}^{(1)} = [0.5, 0.5]^T \to \mathbf{h} = [1,1]^T \to \hat{y} = H(0.5) = \mathbf{1}$
- $(1,0) \to \mathbf{z}^{(1)} = [0.5, 0.5]^T \to \mathbf{h} = [1,1]^T \to \hat{y} = H(0.5) = \mathbf{1}$
- $(1,1) \to \mathbf{z}^{(1)} = [1.5, -0.5]^T \to \mathbf{h} = [1,0]^T \to \hat{y} = H(-0.5) = \mathbf{0}$

> Finally, XOR is solved...

XOR: Hidden Layer







Batch Computation

In practice, we stack B input vectors into a data matrix $\mathbf{X} \in \mathbb{R}^{B \times d_{\text{in}}}$:

$$\mathbf{H}^{(0)} = \sigma\left(\mathbf{X}(\mathbf{W}^{(0)})^T + \mathbf{1}_B(\mathbf{b}^{(0)})^T\right) \in \mathbb{R}^{B \times d_0}$$

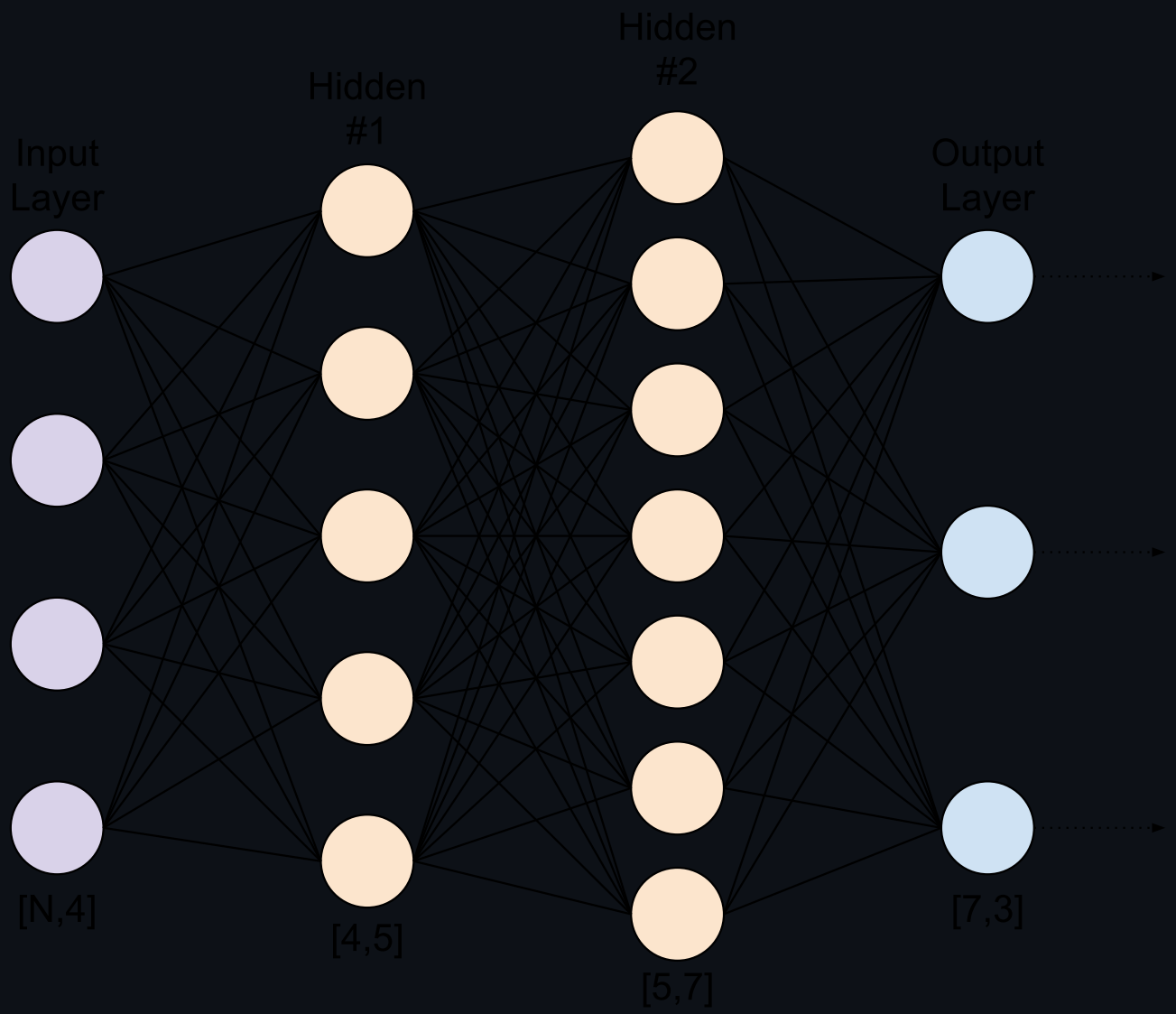
$$\mathbf{H}^{(l)} = \sigma\left(\mathbf{H}^{(l-1)}(\mathbf{W}^{(l)})^T + \mathbf{1}_B(\mathbf{b}^{(l)})^T\right) \in \mathbb{R}^{B \times d_l}$$

> Shape check: $(B, d_0) = (B, d_{in})(d_0, d_{in})^T + (B, 1)(d_0, 1)^T$

Computational Efficiency

- The forward pass is converted into a General Matrix Multiplication (**GEMM**).
- Modern GPUs (tensor cores) are hardware-optimised for dense matrix multiplication.
- Give me more GPUs! More matrix multiplication $\rightarrow$ **MOAR** AI!

> We'll use both notation going foward depending on whether $\mathbf{X}$ is batched or not.


Input Layer
Hidden #1
Hidden #2
Output Layer
[N,4]
[4,5]
[5,7]
[7,3]

PyTorch Parameter Level

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.W_1 = nn.Parameter(torch.randn(hidden_size, input_size))
        self.b_1 = nn.Parameter(torch.randn(hidden_size))
        self.W_2 = nn.Parameter(torch.randn(output_size, hidden_size))
        self.b_2 = nn.Parameter(torch.randn(output_size))

    def forward(self, x):
        h = torch.sigmoid(x @ self.W_1.T + self.b_1)
        y = h @ self.W_2.T + self.b_2
        return y
```

PyTorch Module Level

```
import torch.nn as nn
import torch.nn.functional as F

class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        # Notice how a "linear" layer is abstracted away
        self.l1 = nn.Linear(input_size, hidden_size)
        self.l2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h = F.sigmoid(self.l1(x))
        y = self.l2(h)
        return y
```

PyTorch Sequential

```
import torch.nn as nn
import torch.nn.functional as F

model = nn.Sequential(
    nn.Linear(input_size, hidden_size),
    nn.Sigmoid(),
    nn.Linear(hidden_size, output_size),
)
```

My advice is to not write this code by hand anymore. Connect the practice and theory yourself but let this plumbing be handled by LLMs. Back then, I used to have 50 tabs open just to get a new research model working, now I just ask a LLM. Why does this work so well in this case?

> There are other abstractions but these are the basics. The library is packed with stuff for almost any deep learning architecture. Take a look at PyTorch documentation for more.

Bio vs. ANN

Property	Biological Human Brain	Artificial Neural Network (GPU)
Scale	$\approx 8.6 \times 10^{10}$ neurons, $\sim 10^{14}$ synapses	10^{7} to 10^{12} parameters
Power	$\approx 20\ \mathrm{Watts}$ (a dim lightbulb!)	Megawatts (data centre clusters)
Signalling Mode	Asynchronous, discrete analogue action potentials (spikes)	Synchronous, dense floating-point tensors (FP16/FP8)
Synaptic Plasticity	Local chemical rules (STDP, Hebbian: *"fire together, wire together"*)	Global synchronous backpropagation (Reverse-mode autodiff)
Connectivity	Highly sparse, 3D anatomical goop	Dense matrix multiplication blocks

> **Spoiler:** ANNs are not biological simulations, they are differentiable mathematical function approximators optimised by numerical calculus.

Okay Steve, how do we train it?

Recall our emerging update rule:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

1. Without any hidden layers, everything is the same. Loss functions, softmax, etc.
2. The idea of descending in the direction of the gradient lowers the loss.
3. AI happens.

This time there are thousands of parameters all chained together and it spits out some numbers.

$$\hat{y} = \sigma(w^{(2)}\sigma(w^{(1)}x + b^{(1)}) + b^{(2)}) + \dots$$

Who is to blame to the high loss?

Backpropagation

Follow the ~~money~~ signal.

Neural networks are simply a **chain of elementary differentiable operations** ($+, \times, \sigma, \ldots$).

Forward Pass

- Feed inputs forward through elementary operations.
- Compute and cache intermediate values at every step:

$$\mathbf{x} \to \mathbf{z}^{(1)} \to \mathbf{h}^{(1)} \to \mathbf{z}^{(2)} \to \cdots \to \hat{\mathbf{y}} \to \mathcal{L}$$

Backward Pass

- Start at the scalar loss $\mathcal{L}$ with seed $\frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$.
- Propagate raw partial derivatives right-to-left using the chain rule.
- Evaluates gradients for **all parameters in a single backward pass**!

The pinnacle of calculus

$$\frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$$

And then we ask:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = ?$$

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{\partial \frac{1}{2}(\hat{y} - y)^2}{\partial \hat{y}} = \hat{y} - y$$

then:

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = (\hat{y} - y) \cdot \sigma'(z)$$

> Magic... 🪄

The Scalar Chain of Computation

To build intuition, consider a single neuron with scalar input x, weight w, bias b, and target y:

4 Elementary Operations

1. **Multiply Node:** $u = w \cdot x$
2. **Addition Node:** $z = u + b$
3. **Activation Node:** $\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$
4. **Loss Node:** $\mathcal{L} = \frac{1}{2}(\hat{y} - y)^2$

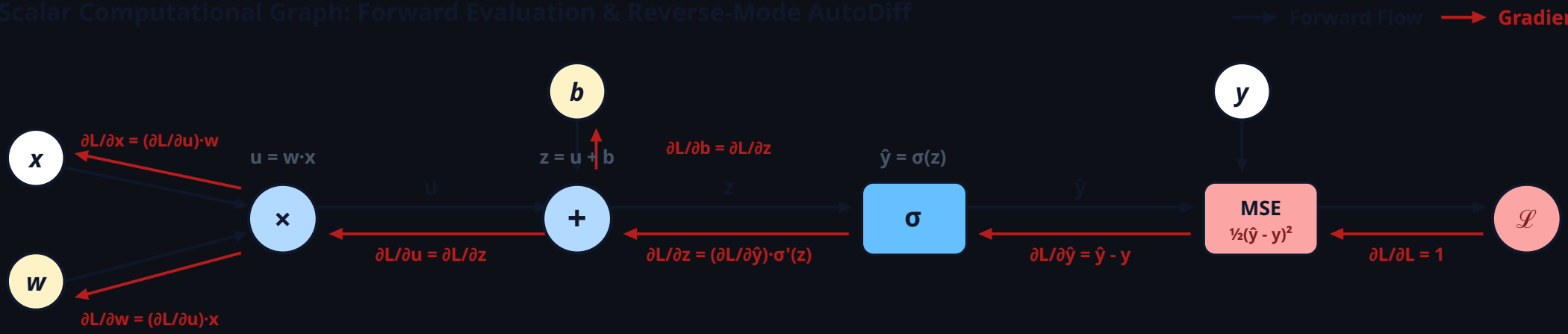
(All tensors here are scalars of shape 1×1)

Concrete Forward Example

$x = 2.0,\ w = 1.5,\ b = 0.5,\ y = 1.0$:

- $u = 1.5 \times 2.0 = 3.0$
- $z = 3.0 + 0.5 = 3.5$
- $\hat{y} = \sigma(3.5) = \frac{1}{1+e^{-3.5}} \approx 0.9707$
- $\mathcal{L} = \frac{1}{2}(0.9707 - 1.0)^2 \approx 0.00043$

Scalar Computational Graph: Forward Evaluation & Reverse-Mode AutoDiff

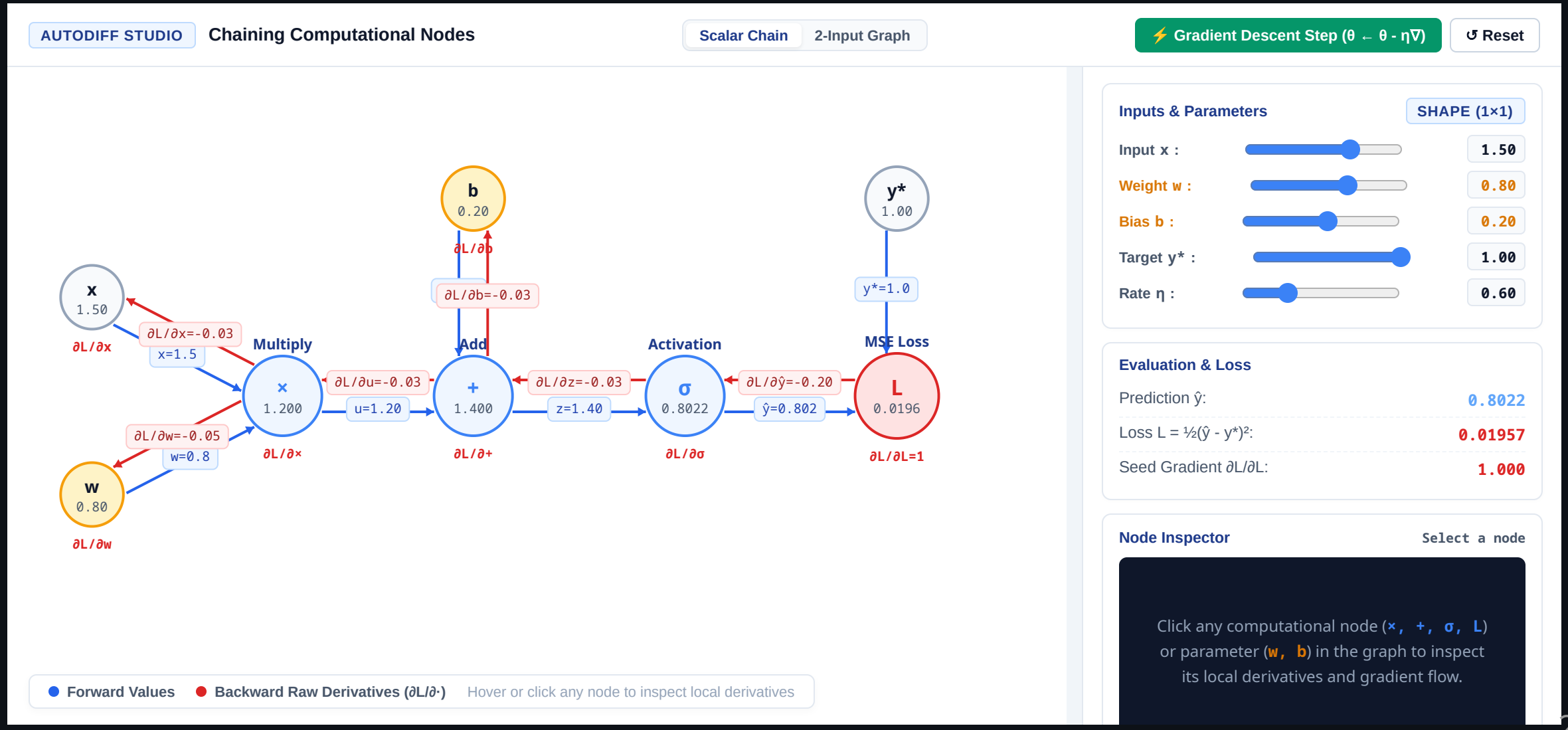


Reverse-Mode Chain Rule: Parameter gradients are computed in a single sweep: $\partial L/\partial w = (\hat{y} - y) \cdot \sigma'(z) \cdot x$ | $\partial L/\partial b = (\hat{y} - y) \cdot \sigma'(z)$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial b} = (\hat{y} - y) \cdot \sigma'(z) \cdot 1$$

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial u} \cdot \frac{\partial u}{\partial w} = (\hat{y} - y) \cdot \sigma'(z) \cdot 1 \cdot x$$

Backpropagation Demo



The full soup

I won't hold back here, we will do the full derivation for a single layer, no shortcuts. *Batch mode.*

$$H = \sigma(Z) = \sigma(XW + B)$$

where

- $X \in \mathbb{R}^{N \times D_{in}}$ (batch of N inputs, D_{in} features)
- $W \in \mathbb{R}^{D_{in} \times D_{out}}$ (weights)
- $B \in \mathbb{R}^{N \times D_{out}}$ (stacked bias $B = \mathbf{1}_N b$, where $b \in \mathbb{R}^{1 \times D_{out}}$)
- $Z \in \mathbb{R}^{N \times D_{out}}$ (pre-activations)
- $H \in \mathbb{R}^{N \times D_{out}}$ (post-activation hidden representations)

Why XW instead of Wx? In deep learning libraries (PyTorch, JAX), inputs are matrices of N samples. Multiplying on the right keeps batch size N as the leading dimension without transpose overhead. $(N, D_{in}) \times (D_{in}, D_{out}) = (N, D_{out})$.

Single Layer: Concrete 2×3 Setup

Let $N = D_{in} = 2, D_{out} = 3$:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix}, \quad W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix}, \quad B = \begin{bmatrix} \mathbf{b} \\ \mathbf{b} \end{bmatrix} = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 \end{bmatrix}$$

Multiplying $Z = XW + B$:

$$Z = \begin{bmatrix} x_{1,1}w_{1,1} + x_{1,2}w_{2,1} + b_1 & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} + b_2 & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} + b_3 \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} + b_1 & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} + b_2 & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} + b_3 \end{bmatrix}$$

The Upstream Gradient Matrix: $\frac{\partial \mathcal{L}}{\partial Z}$

The upper layer passes us the incoming upstream gradient matrix w.r.t our outputs $Z \in \mathbb{R}^{2\times 3}$:

$$\frac{\partial \mathcal{L}}{\partial Z} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z_{1,1}} & \frac{\partial \mathcal{L}}{\partial z_{1,2}} & \frac{\partial \mathcal{L}}{\partial z_{1,3}} \\ \frac{\partial \mathcal{L}}{\partial z_{2,1}} & \frac{\partial \mathcal{L}}{\partial z_{2,2}} & \frac{\partial \mathcal{L}}{\partial z_{2,3}} \end{bmatrix}$$

We need to compute three partial derivative matrices:

1. $\frac{\partial \mathcal{L}}{\partial X} \in \mathbb{R}^{2\times 2}$ — to **pass backward** to earlier layers.
2. $\frac{\partial \mathcal{L}}{\partial W} \in \mathbb{R}^{2\times 3}$ — to **update weights** via gradient descent.
3. $\frac{\partial \mathcal{L}}{\partial b} \in \mathbb{R}^{1\times 3}$ — to **update bias** via gradient descent.

> **Shape Rule:** Because scalar loss $\mathcal{L} \in \mathbb{R}$ is differentiated w.r.t each entry, every gradient matrix has the **exact same shape** as the variable itself.

Journey of a Single Input: $x_{1,1}$

How does $x_{1,1}$ affect the scalar loss $\mathcal{L}$? Look at where $x_{1,1}$ appears in Z:

$$Z = \begin{bmatrix} \mathbf{x_{1,1}}w_{1,1} + x_{1,2}w_{2,1} + b_1 & \mathbf{x_{1,1}}w_{1,2} + x_{1,2}w_{2,2} + b_2 & \mathbf{x_{1,1}}w_{1,3} + x_{1,2}w_{2,3} + b_3 \\ x_{2,1}w_{1,1} + x_{2,2}w_{2,1} + b_1 & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} + b_2 & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} + b_3 \end{bmatrix}$$

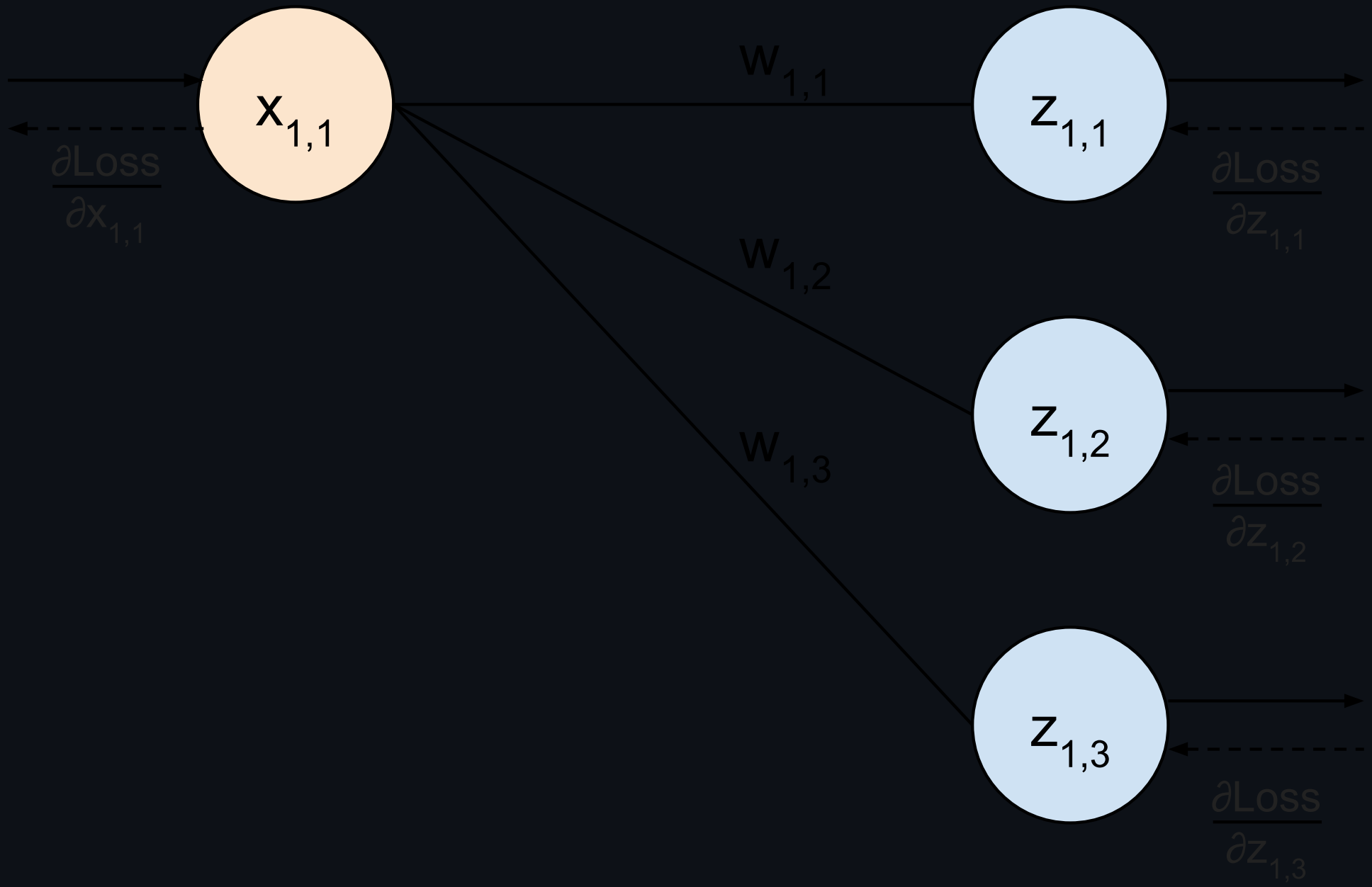
$x_{1,1}$ appears **only in the first row** of Z, affecting $z_{1,1}$, $z_{1,2}$, and $z_{1,3}$:

$$\frac{\partial z_{1,1}}{\partial x_{1,1}} = w_{1,1}, \quad \frac{\partial z_{1,2}}{\partial x_{1,1}} = w_{1,2}, \quad \frac{\partial z_{1,3}}{\partial x_{1,1}} = w_{1,3}, \quad \frac{\partial z_{2,j}}{\partial x_{1,1}} = 0$$

By the multivariable chain rule, sum along all outgoing paths:

$$\frac{\partial \mathcal{L}}{\partial x_{1,1}} = \frac{\partial \mathcal{L}}{\partial z_{1,1}} w_{1,1} + \frac{\partial \mathcal{L}}{\partial z_{1,2}} w_{1,2} + \frac{\partial \mathcal{L}}{\partial z_{1,3}} w_{1,3}$$

> **Intuition:** How $x_{1,1}$ affects loss depends on the weights it multiplies with and upstream error signals!


$x_{1,1}$
$w_{1,1}$
$w_{1,2}$
$w_{1,3}$
$z_{1,1}$
$z_{1,2}$
$z_{1,3}$
$\frac{\partial Loss}{\partial x_{1,1}}$
$\frac{\partial Loss}{\partial z_{1,1}}$
$\frac{\partial Loss}{\partial z_{1,2}}$
$\frac{\partial Loss}{\partial z_{1,3}}$

Input Gradient Matrix: $\frac{\partial \mathcal{L}}{\partial X} = \frac{\partial \mathcal{L}}{\partial Z} W^T$

Repeating this for all 4 entries in X gives the complete matrix:

$$\frac{\partial \mathcal{L}}{\partial X} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z_{1,1}} w_{1,1} + \frac{\partial \mathcal{L}}{\partial z_{1,2}} w_{1,2} + \frac{\partial \mathcal{L}}{\partial z_{1,3}} w_{1,3} & \frac{\partial \mathcal{L}}{\partial z_{1,1}} w_{2,1} + \frac{\partial \mathcal{L}}{\partial z_{1,2}} w_{2,2} + \frac{\partial \mathcal{L}}{\partial z_{1,3}} w_{2,3} \\ \frac{\partial \mathcal{L}}{\partial z_{2,1}} w_{1,1} + \frac{\partial \mathcal{L}}{\partial z_{2,2}} w_{1,2} + \frac{\partial \mathcal{L}}{\partial z_{2,3}} w_{1,3} & \frac{\partial \mathcal{L}}{\partial z_{2,1}} w_{2,1} + \frac{\partial \mathcal{L}}{\partial z_{2,2}} w_{2,2} + \frac{\partial \mathcal{L}}{\partial z_{2,3}} w_{2,3} \end{bmatrix}$$

Extracting the incoming upstream gradient matrix:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial X}}_{(2\times 2)} = \underbrace{\begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z_{1,1}} & \frac{\partial \mathcal{L}}{\partial z_{1,2}} & \frac{\partial \mathcal{L}}{\partial z_{1,3}} \\ \frac{\partial \mathcal{L}}{\partial z_{2,1}} & \frac{\partial \mathcal{L}}{\partial z_{2,2}} & \frac{\partial \mathcal{L}}{\partial z_{2,3}} \end{bmatrix}}_{(N\times D_{out})} \underbrace{\begin{bmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \\ w_{1,3} & w_{2,3} \end{bmatrix}}_{(D_{out}\times D_{in})} = \underbrace{\frac{\partial \mathcal{L}}{\partial Z}}_{(N\times D_{out})} \underbrace{W^T}_{(D_{out}\times D_{in})}$$

Journey of a Single Weight: $w_{1,1}$

Now consider weight $w_{1,1}$. Where does $w_{1,1}$ appear in Z?

$$Z = \begin{bmatrix} x_{1,1}\mathbf{w_{1,1}} + x_{1,2}w_{2,1} + b_1 & x_{1,1}w_{1,2} + x_{1,2}w_{2,2} + b_2 & x_{1,1}w_{1,3} + x_{1,2}w_{2,3} + b_3 \\ x_{2,1}\mathbf{w_{1,1}} + x_{2,2}w_{2,1} + b_1 & x_{2,1}w_{1,2} + x_{2,2}w_{2,2} + b_2 & x_{2,1}w_{1,3} + x_{2,2}w_{2,3} + b_3 \end{bmatrix}$$

$w_{1,1}$ appears in column 1 for **both** data points ($z_{1,1}$ and $z_{2,1}$):

$$\frac{\partial z_{1,1}}{\partial w_{1,1}} = x_{1,1}, \qquad \frac{\partial z_{2,1}}{\partial w_{1,1}} = x_{2,1}$$

By chain rule, sum contributions across all samples in the batch:

$$\frac{\partial \mathcal{L}}{\partial w_{1,1}} = \frac{\partial \mathcal{L}}{\partial z_{1,1}} x_{1,1} + \frac{\partial \mathcal{L}}{\partial z_{2,1}} x_{2,1}$$

> **Intuition:** A shared parameter affects every sample in the batch; its gradient **accumulates** across all batch samples!

Weight Gradient Matrix: $\frac{\partial \mathcal{L}}{\partial W} = X^T \frac{\partial \mathcal{L}}{\partial Z}$

Writing this out for all 6 weights:

$$\frac{\partial \mathcal{L}}{\partial W} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z_{1,1}} x_{1,1} + \frac{\partial \mathcal{L}}{\partial z_{2,1}} x_{2,1} & \frac{\partial \mathcal{L}}{\partial z_{1,2}} x_{1,1} + \frac{\partial \mathcal{L}}{\partial z_{2,2}} x_{2,1} & \frac{\partial \mathcal{L}}{\partial z_{1,3}} x_{1,1} + \frac{\partial \mathcal{L}}{\partial z_{2,3}} x_{2,1} \\ \frac{\partial \mathcal{L}}{\partial z_{1,1}} x_{1,2} + \frac{\partial \mathcal{L}}{\partial z_{2,1}} x_{2,2} & \frac{\partial \mathcal{L}}{\partial z_{1,2}} x_{1,2} + \frac{\partial \mathcal{L}}{\partial z_{2,2}} x_{2,2} & \frac{\partial \mathcal{L}}{\partial z_{1,3}} x_{1,2} + \frac{\partial \mathcal{L}}{\partial z_{2,3}} x_{2,2} \end{bmatrix}$$

Extracting matrices reveals the exact matrix product:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial W}}_{(2\times 3)} = \underbrace{\begin{bmatrix} x_{1,1} & x_{2,1} \\ x_{1,2} & x_{2,2} \end{bmatrix}}_{(D_{in}\times N)} \underbrace{\begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z_{1,1}} & \frac{\partial \mathcal{L}}{\partial z_{1,2}} & \frac{\partial \mathcal{L}}{\partial z_{1,3}} \\ \frac{\partial \mathcal{L}}{\partial z_{2,1}} & \frac{\partial \mathcal{L}}{\partial z_{2,2}} & \frac{\partial \mathcal{L}}{\partial z_{2,3}} \end{bmatrix}}_{(N\times D_{out})} = \underbrace{X^T}_{(D_{in}\times N)} \underbrace{\frac{\partial \mathcal{L}}{\partial Z}}_{(N\times D_{out})}$$

> Shape check: $(D_{in} \times N) \times (N \times D_{out}) = (D_{in} \times D_{out})$, exactly matching W!

Bias Gradient & Activation

Bias Gradient: $\frac{\partial \mathcal{L}}{\partial b}$

The bias $b \in \mathbb{R}^{1 \times D_{out}}$ is added to each row:

$$\frac{\partial \mathcal{L}}{\partial b_j} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial z_{i,j}} \cdot 1$$

In matrix form:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial b}}_{(1 \times D_{out})} = \underbrace{\mathbf{1}_N^T}_{(1 \times N)} \underbrace{\frac{\partial \mathcal{L}}{\partial Z}}_{(N \times D_{out})}$$

If this isn't clear, follow where b_1 etc appears.

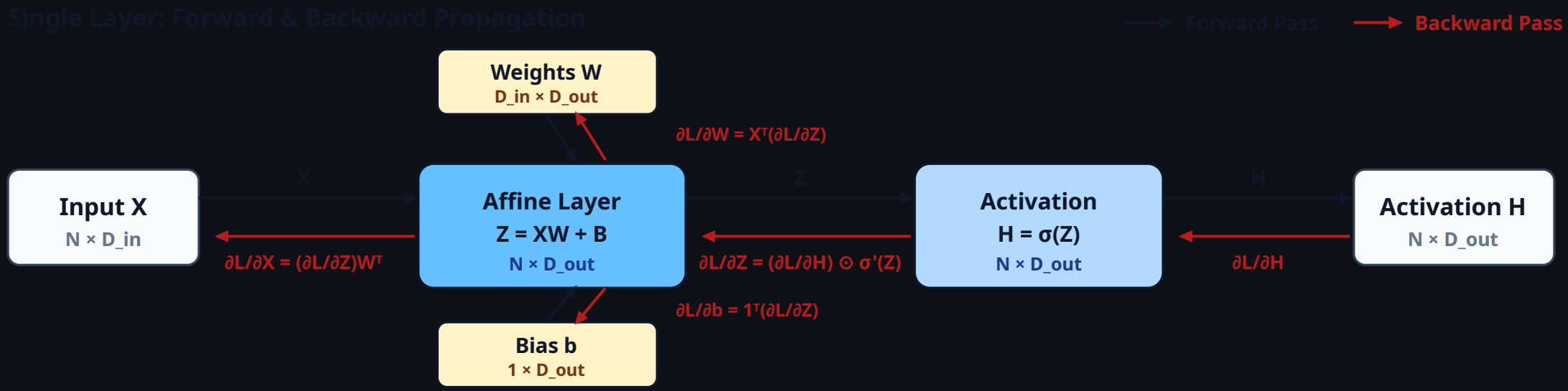
Activation: $\frac{\partial \mathcal{L}}{\partial Z}$

For elementwise activation $H = \sigma(Z)$:
Each $z_{i,j}$ affects only $h_{i,j}$ with local derivative $\sigma'(z_{i,j})$:

$$\frac{\partial \mathcal{L}}{\partial z_{i,j}} = \frac{\partial \mathcal{L}}{\partial h_{i,j}} \sigma'(z_{i,j})$$

In full matrix form:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial Z}}_{(N \times D_{out})} = \underbrace{\frac{\partial \mathcal{L}}{\partial H}}_{(N \times D_{out})} \odot \underbrace{\sigma'(Z)}_{(N \times D_{out})}$$



Shape Compatibility: $\partial L/\partial W = [X^T]_(D_{in} \times N) \cdot [\partial L/\partial Z]_(N \times D_{out}) = (D_{in} \times D_{out})$ | $\partial L/\partial X = [\partial L/\partial Z]_(N \times D_{out}) \cdot [W^T]_(D_{out} \times D_{in}) = (N \times D_{in})$

No shortcuts, no Greek letter soups: Every matrix equation follows directly from the scalar multivariable chain rule on concrete entries!

> I first wrote these notes back in 2019...

~~Burn~~ Learn baby Learn

Parameter update via Gradient Descent:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}}, \qquad b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial b^{(l)}}$$

All your hopes and dreams will fill the blank space below in this slide...

> Curious about the slide title, click here

David Rumelhart (1942–2011)

In 1986, a landmark paper appeared in *Nature* that ignited the modern connectionist era:

"Learning representations by back-propagating errors"

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams (Nature, 1986)

Historical context: Reverse-mode automatic differentiation was introduced earlier by Seppo Linnainmaa (1970) and adapted for neural networks by Paul Werbos (1974, 1982). Rumelhart came up with it independently and shared it with Hinton.



Geoffrey Hinton

"Dave Rumelhart came up with this idea of backpropagation. He wasn't the first to come up with it, but ...
I first of all explained to him why it wouldn't work, based on an argument in Rosenblatt's book, which showed that essentially it was an algorithm that couldn't break symmetry. If you had a deterministic learning rule, and you started off with two identical hidden units, they would stay identical, so the deterministic learning rule was no good.

Now, as you know, the standard way of doing learning is to have random initial weights, or you could just put a little bit of noise in the learning rule -either would be fine. So Dave Rumelhart pointed out that we didn't need to worry about symmetry.

The next argument I gave him was that it would get stuck in local minima. There was no guarantee you'd find a global optimum. Since you're bound to get stuck in local minima, it wasn't really worth investigating. Nevertheless, Dave programmed it up and did a few things with it. I think he showed it could learn **X-OR [exclusive OR]**."

> Talking Nets: An Oral History of Neural Networks, Chapter 16 page 376.

David Rumelhart

Science is human.

- In the late 1990s, Rumelhart began experiencing progressive memory and speech difficulties.
- Diagnosed with **Pick's disease** (frontotemporal lobar degeneration).
- The scientist who unlocked how to train artificial neural networks lost his own cognitive abilities, passing away in 2011.

> The prestigious **Rumelhart Prize** is awarded annually today to honour lifetime contributions to the theoretical foundations of human cognition.

Bio Backprop?

Neuroscientists widely agree that exact backpropagation is **not** how biological brains learn:

Backwards

Biological synapses are strictly unidirectional chemical junctions.

Global Signals

Biological neurons update synapses based on **local signals** (pre-synaptic firing, post-synaptic voltage, and neuromodulators like dopamine).

Synchrony

The brain operates as an asynchronous, continuous-time dynamical system with no master clock.

Modern computational neuroscience alternatives: Feedback Alignment (Lillicrap et al., 2016), Equilibrium Propagation (Scellier & Bengio, 2017), and Predictive Coding (Friston, 2005).

Vanishing & Exploding Gradients

Networks deeper than 3-4 layers may fail to train. Why?

$$\underbrace{\frac{\partial \mathcal{L}}{\partial X}}_{(N \times D_{in})} = \underbrace{\frac{\partial \mathcal{L}}{\partial Z}}_{(N \times D_{out})} \underbrace{W^T}_{(D_{out} \times D_{in})} = \underbrace{\frac{\partial \mathcal{L}}{\partial H}}_{(N \times D_{out})} \odot \underbrace{\sigma'(Z)}_{(N \times D_{out})} \underbrace{W^T}_{(D_{out} \times D_{in})}$$

1. **Weight matrices $\mathbf{W}^T$:** Depends on initialisation scale. If we initialise with $\mathcal{N}(0, 0.01)$, every step we multiply by small numbers.
2. **Activation slopes $\sigma'(\mathbf{z}^{(l)})$:**
   - Sigmoid: $\max \sigma' = 0.25$
   - Tanh: $\max \sigma' = 1.0$ (and drops to ≈ 0 in saturation)

Vanishing & Exploding Gradients

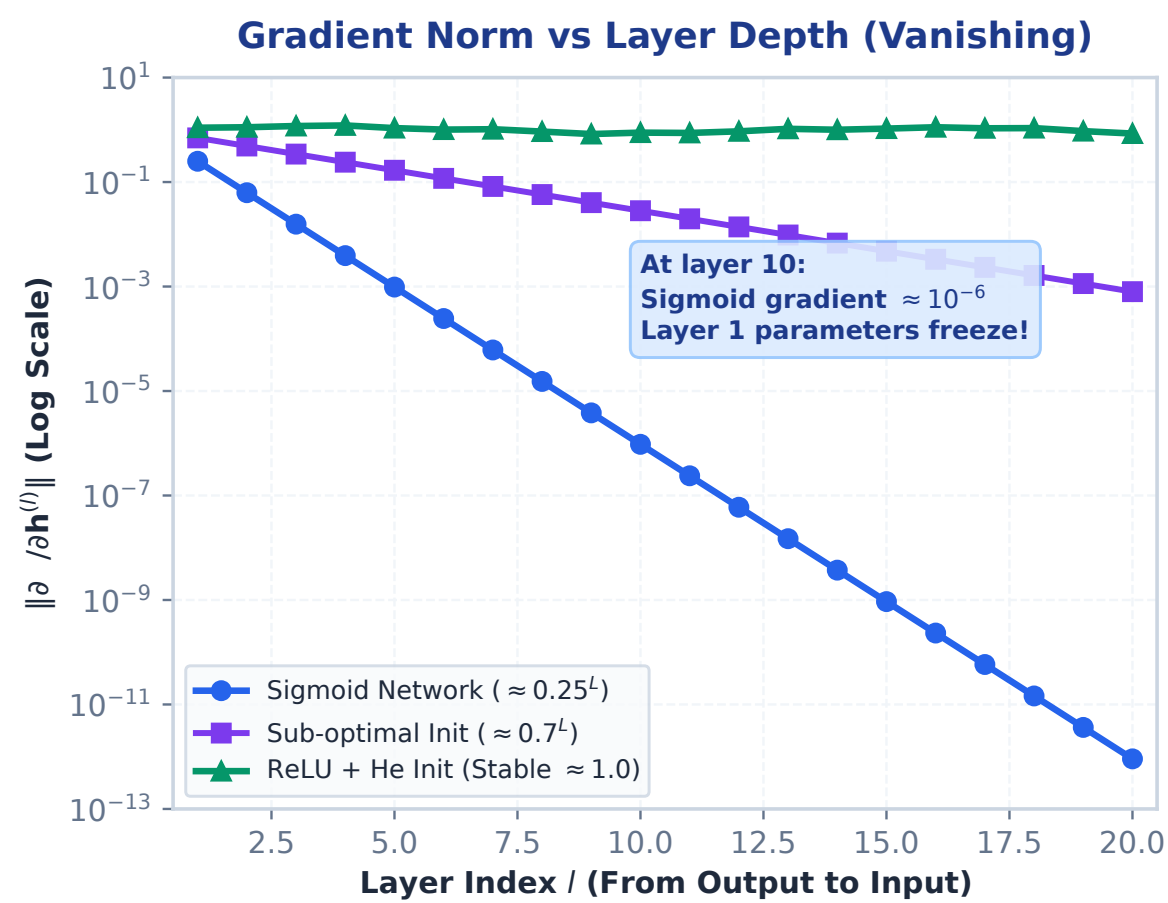
The gradient norm depends on the **singular values** of the product $\sigma'(Z)W^T$

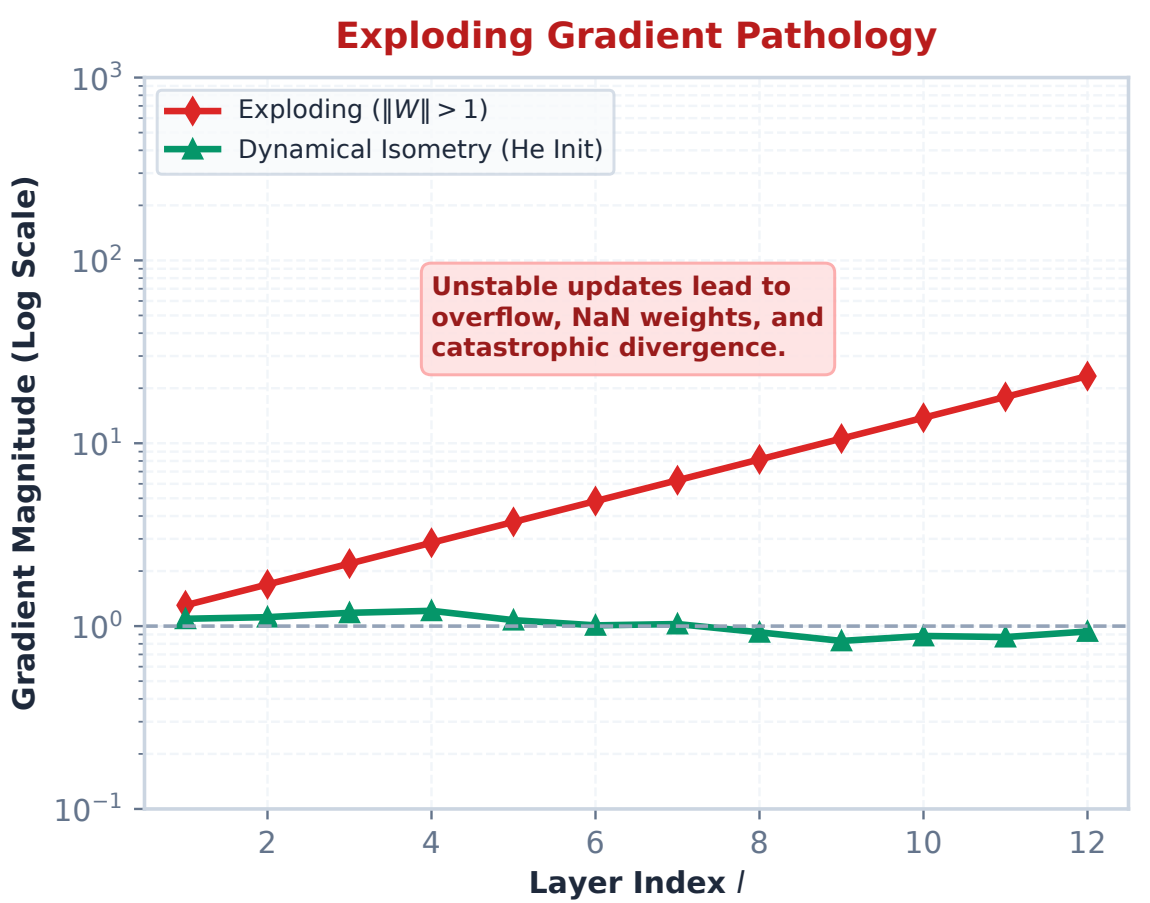
Symptom	Condition	Consequence
Vanishing Gradients	Singular values < 1	- $\|\nabla_{\mathbf{h}^{(0)}}\mathcal{L}\| \to 0$ - Early layers receive zero updates and freeze. - Deep net acts like a shallow net.
Exploding Gradients	Singular values > 1	- $\|\nabla_{\mathbf{h}^{(0)}}\mathcal{L}\| \to \infty$ - Parameter updates oscillate wildly. - Numerical overflow (`NaN` loss).

> What were singular values again?

Singular values measure how much the matrix stretches or shrinks vectors in different directions.

Visualising the Gradient Collapse





Fix this Steve

1. **Non-saturating Activations:**
   - Swap Sigmoid/Tanh for **ReLU** / **Leaky ReLU** ($\sigma'(z) = 1$ for $z > 0$).
2. **Variance-Preserving Initialisation:**
   - **He / Xavier Initialisation** scales weights by layer fan-in/fan-out to ensure variance stays ≈ 1.
3. **Architectural Highway (Residual Connections):**
   - **ResNets:** $\mathbf{h}^{(l+1)} = \mathbf{h}^{(l)} + \mathcal{F}(\mathbf{h}^{(l)})$.
   - Gradient flow includes an additive identity term $\mathbf{I}$, preventing exponential decay.
4. **Normalisation & Safeguards:**
   - **LayerNorm / BatchNorm** + **Gradient Clipping**.

Activation Functions

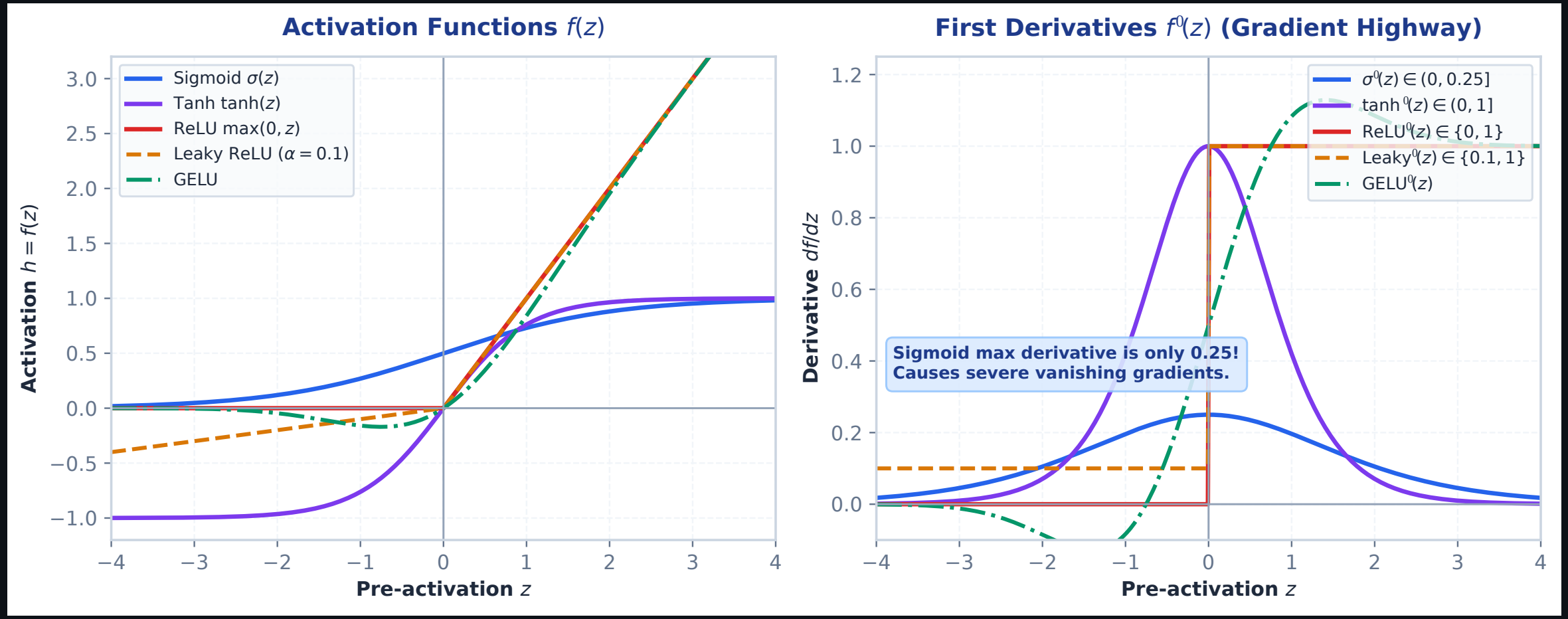
The traditional Sigmoid function was the primary culprit behind vanishing gradients:

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad \sigma'(z) = \sigma(z)(1-\sigma(z)) \leq 0.25$$

Every single sigmoid layer shrinks the gradient by at least $4\times$ (0.25)! Ouch...

Activation	Formula	Derivative Range	Properties
Sigmoid	$\frac{1}{1+e^{-z}}$	$(0, 0.25]$	Non-zero centred, severe vanishing gradient
Tanh	$\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$(0, 1.0]$	Zero-centred, but still saturates at $\pm\infty$
ReLU	$\max(0, z)$	$\{0, 1\}$	Constant gradient 1 for $z > 0$; dying ReLU risk
Leaky ReLU	$\max(\alpha z, z)$	$\{\alpha, 1\}$	Prevents dying neurons by keeping $\alpha = 0.01$ slope

Activation Functions



Weight Initialisation

If all weights are initialised to zero, every neuron in a layer computes the exact same gradient $\rightarrow$ **symmetry** Hinton was talking about.

To break symmetry, we sample weights randomly with zero mean:

$$W_{ij} \sim \mathcal{N}(0, \sigma_w^2) \quad \text{or} \quad W_{ij} \sim \mathcal{U}(-a, a), \quad b_i = 0$$

However, if the variance σ_w^2 is chosen poorly, our network values will either exponentially explode or vanish to zero as depth increases.

Variance Propagation Derivation (Linear Forward Pass)

For a single pre-activation unit at layer l receiving inputs from layer $l-1$:

$$z_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} h_j^{(l-1)}$$

Assumptions:

1. Weights w_{ij} and inputs $h_j^{(l-1)}$ are independent.
2. Zero-mean distributions: $\mathbb{E}[w_{ij}] = 0$ and $\mathbb{E}[h_j^{(l-1)}] = 0$.

$$\begin{aligned}\mathrm{Var}(w_{ij} h_j^{(l-1)}) &= \mathbb{E}[(w_{ij} h_j^{(l-1)})^2] - (\mathbb{E}[w_{ij} h_j^{(l-1)}])^2 \\ &= \mathbb{E}[(w_{ij})^2]\mathbb{E}[(h_j^{(l-1)})^2] - (\mathbb{E}[w_{ij}]\mathbb{E}[h_j^{(l-1)}])^2 = \mathrm{Var}(w)\mathrm{Var}(h^{(l-1)})\end{aligned}$$

Summing over all n_{in} independent inputs:

$$\text{Var}(z_i) = \sum_{j=1}^{n_{\text{in}}} \text{Var}(w_{ij} h_j^{(l-1)}) = n_{\text{in}} \text{Var}(w) \text{Var}(h^{(l-1)})$$

For $\tanh$ near zero, $h_j^{(l-1)} \approx z_j^{(l-1)}$, so $\text{Var}(h^{(l-1)}) \approx \text{Var}(z^{(l-1)})$:

$$\text{Var}(z_i) = n_{\text{in}} \text{Var}(w) \text{Var}(z_j^{(l-1)})$$

What we want (layer-to-layer variance preservation):

$$\text{Var}(z_i) = \text{Var}(z_j^{(l-1)}) \implies n_{\text{in}} \text{Var}(w) = 1 \implies \text{Var}(w) = \frac{1}{n_{\text{in}}}$$

- $\text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$ by definition of variance.
- For independent random variables X and Y, $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$ and $\mathbb{E}[(XY)^2] = \mathbb{E}[X^2]\mathbb{E}[Y^2]$.
- Since $\mathbb{E}[w] = 0$ and $\mathbb{E}[h^{(l-1)}] = 0$, $\text{Var}(w) = \mathbb{E}[w^2]$ and $\text{Var}(h^{(l-1)}) = \mathbb{E}[(h^{(l-1)})^2]$.

Backward Pass & The Glorot / Xavier Solution

During backpropagation, gradients flow backward from layer l to layer $l-1$:

$$\frac{\partial \mathcal{L}}{\partial h_j^{(l-1)}} = \sum_{i=1}^{n_{\text{out}}} w_{ij}^{(l)} \frac{\partial \mathcal{L}}{\partial z_i^{(l)}} \implies \text{Var}\left(\frac{\partial \mathcal{L}}{\partial h_j^{(l-1)}}\right) = n_{\text{out}} \text{Var}(w) \text{Var}\left(\frac{\partial \mathcal{L}}{\partial z_i^{(l)}}\right)$$

- Preserving forward activation variance requires: $\text{Var}(w) = \frac{1}{n_{\text{in}}}$
- Preserving backward gradient variance requires: $\text{Var}(w) = \frac{1}{n_{\text{out}}}$

Glorot & Bengio (2010) took the harmonic mean of both constraints:

$$\text{Var}(w) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

Practical Implementation:

- **Normal Distribution:**

$$W \sim \mathcal{N}\left(0,\, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right)$$

- **Uniform Distribution** (since $\operatorname{Var}(\mathcal{U}(-a, a)) = \frac{a^2}{3}$):

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}},\, +\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

- **Biases:** Initialised to zeros ($b = 0$).
- **Best for:** Symmetric activations with unit derivative at zero (e.g., $\tanh$).

The He / Kaiming Solution (ReLU)

Consider the signal propagation through a layer hop with ReLU:

$$z_j^{(l-1)} \xrightarrow{\text{ReLU}} h_j^{(l-1)} \xrightarrow{W^{(l)}} z_i^{(l)}$$

ReLU acts as a half-wave rectifier: $h_j^{(l-1)} = \max(0, z_j^{(l-1)})$. By definition of expectation:

$$\mathbb{E}[(h_j^{(l-1)})^2] = \mathbb{E}[(\max(0, z_j^{(l-1)}))^2] = \int_{-\infty}^{0} 0^2 \cdot p(z_j)\, dz_j + \int_{0}^{\infty} z_j^2 p(z_j)\, dz_j$$

Revisit:

$$z_j^{(l-1)} = \sum_{k=1}^{n_{in}} w_{jk}^{(l-1)} h_k^{(l-2)} \sim \mathcal{N}(0, \ldots)$$

By the Central Limit Theorem, previous-layer pre-activation $z_j^{(l-1)}$ is normally distributed and symmetric around zero ($p(-z) = p(z)$ and $\mathbb{E}[z_j^{(l-1)}] = 0$).

$$\mathbb{E}[(h_j^{(l-1)})^2] = \int_0^\infty z^2 p(z)\,dz = \frac{1}{2}\int_{-\infty}^\infty z^2 p(z)\,dz = \frac{1}{2}\mathbb{E}[(z_j^{(l-1)})^2] = \frac{1}{2}\mathrm{Var}(z_j^{(l-1)})$$

Substitute back into the layer:

$$\begin{aligned}
\mathrm{Var}(z_i^{(l)}) &= \sum_{j=1}^{n_{\text{in}}} \mathrm{Var}(w_{ij}^{(l)} h_j^{(l-1)}) \\
&= \sum_{j=1}^{n_{\text{in}}} \mathrm{Var}(w)\mathbb{E}[(h_j^{(l-1)})^2] \\
&= \sum_{j=1}^{n_{\text{in}}} \mathrm{Var}(w) \cdot \frac{1}{2}\mathrm{Var}(z_j^{(l-1)}) \\
&= \frac{1}{2} n_{\text{in}} \mathrm{Var}(w)\mathrm{Var}(z_j^{(l-1)}) \\
\implies \mathrm{Var}(w) &= \frac{2}{n_{\text{in}}}
\end{aligned}$$

Practical Implementation:

- **Normal Distribution (`he_normal`):**

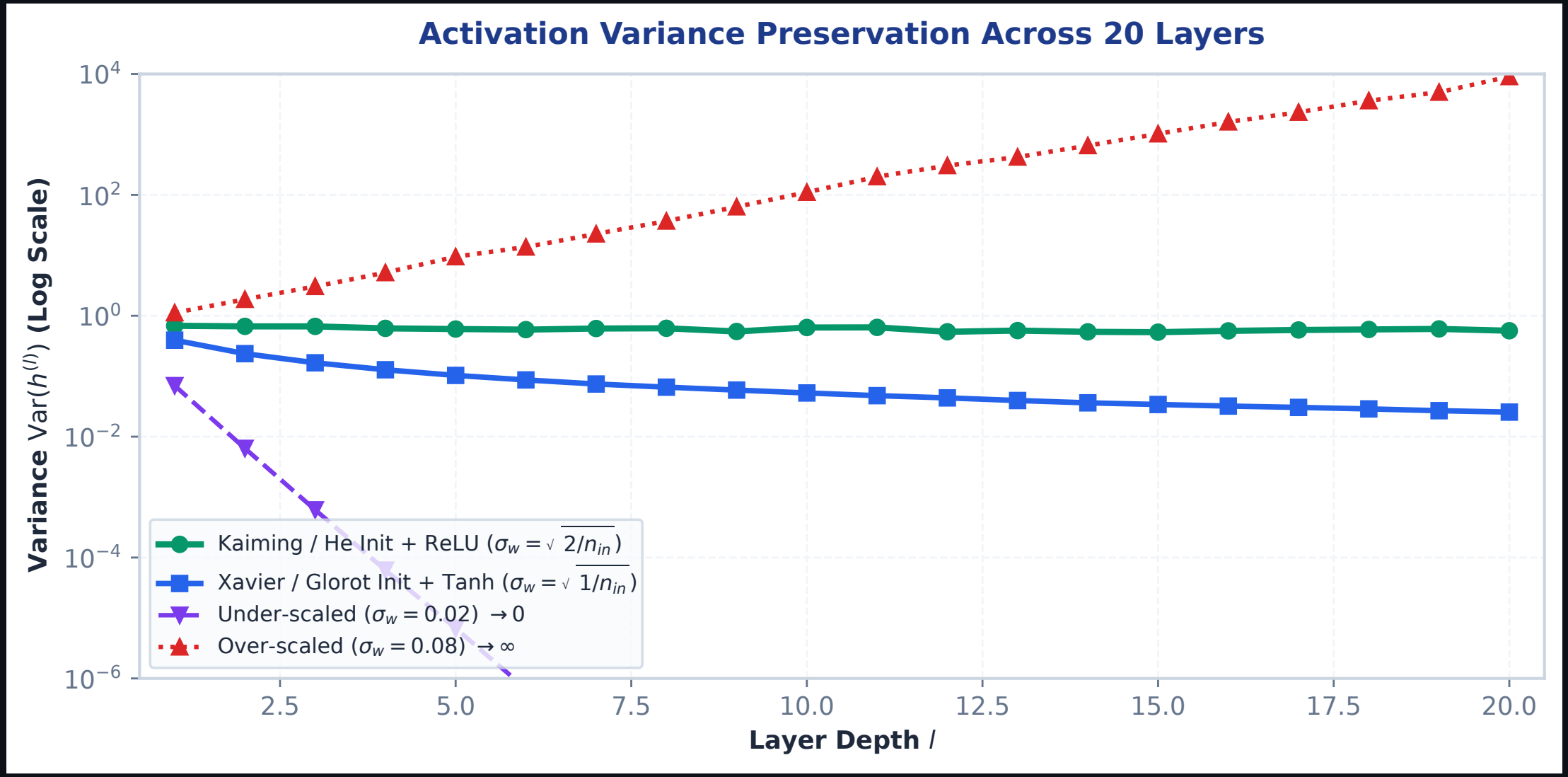
$$W \sim \mathcal{N}\left(0,\, \frac{2}{n_{\text{in}}}\right)$$

- **Uniform Distribution (`he_uniform`):**

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}}}},\, +\sqrt{\frac{6}{n_{\text{in}}}}\right)$$

- **Biases:** Initialised to zeros ($b = 0$).
- **Best for:** ReLU, Leaky ReLU (scaled with gain), and GELU variants.

It works


Activation Variance Preservation Across 20 Layers
Variance Var($h^{(l)}$) (Log Scale)
10^4
10^2
10^0
10^{-2}
10^{-4}
10^{-6}
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Layer Depth l
Kaiming / He Init + ReLU ($\sigma_w = \sqrt{2/n_{in}}$)
Xavier / Glorot Init + Tanh ($\sigma_w = \sqrt{1/n_{in}}$)
Under-scaled ($\sigma_w = 0.02$) $\to 0$
Over-scaled ($\sigma_w = 0.08$) $\to \infty$

Beyond Vanilla SGD

Vanilla Stochastic Gradient Descent updates parameters along the negative gradient:

$$\theta_{t+1} = \theta_t - \eta \mathbf{g}_t, \quad \mathbf{g}_t = \nabla \mathcal{L}(\theta_t)$$

In deep networks, vanilla SGD struggles with three pervasive geometric landscapes:

1. U-shaped Ravines

Curvature along one axis is vastly higher than along another. SGD oscillates wildly across the walls while crawling along the base.

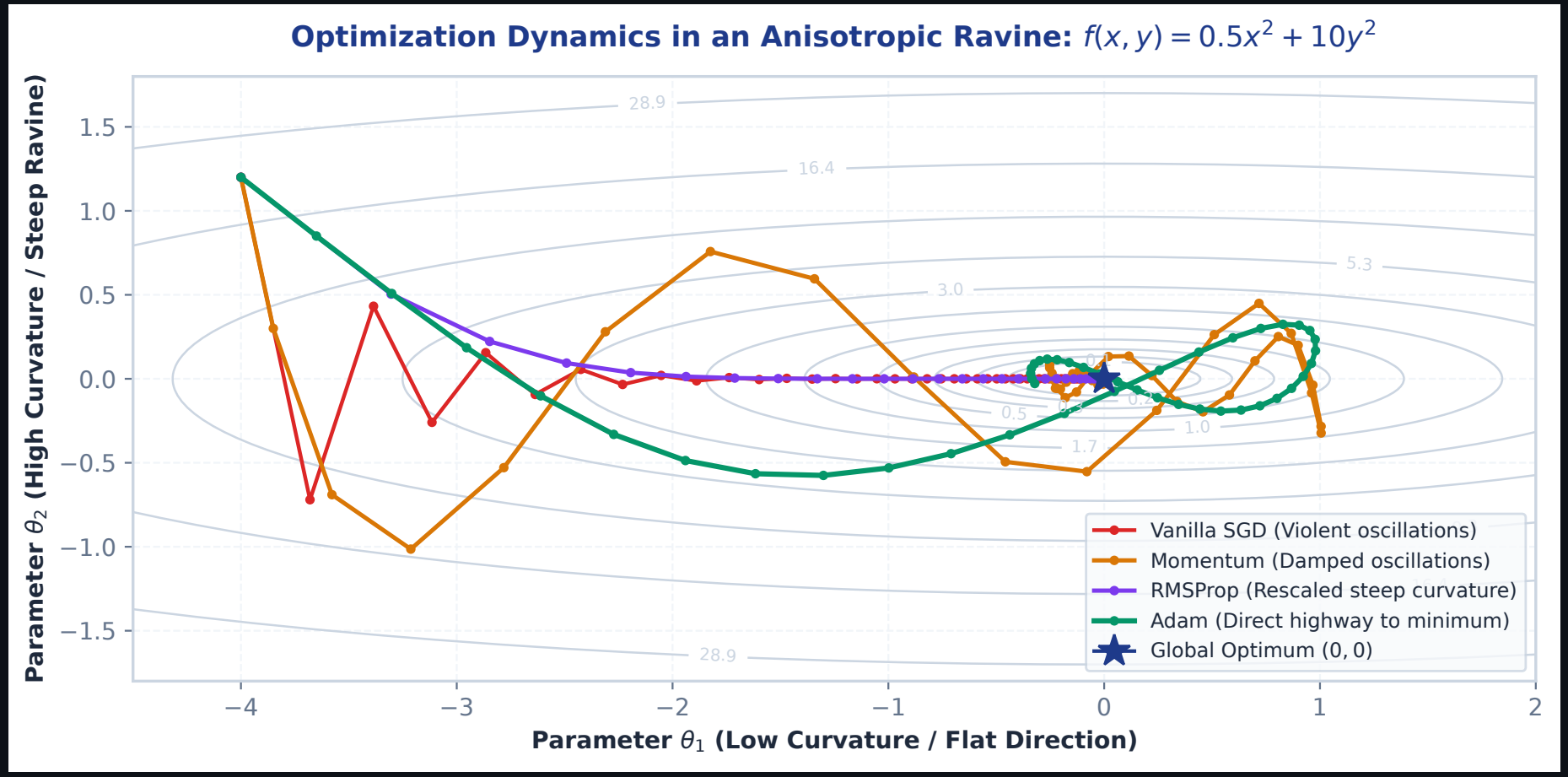
2. Saddle Points & Plateaus

Points where $\nabla \mathcal{L} \approx \mathbf{0}$. SGD stalls indefinitely in zero-gradient zones.

3. Noisy Mini-Batches

Variance across mini-batch gradients can lead to erratic, wandering optimisation trajectories.

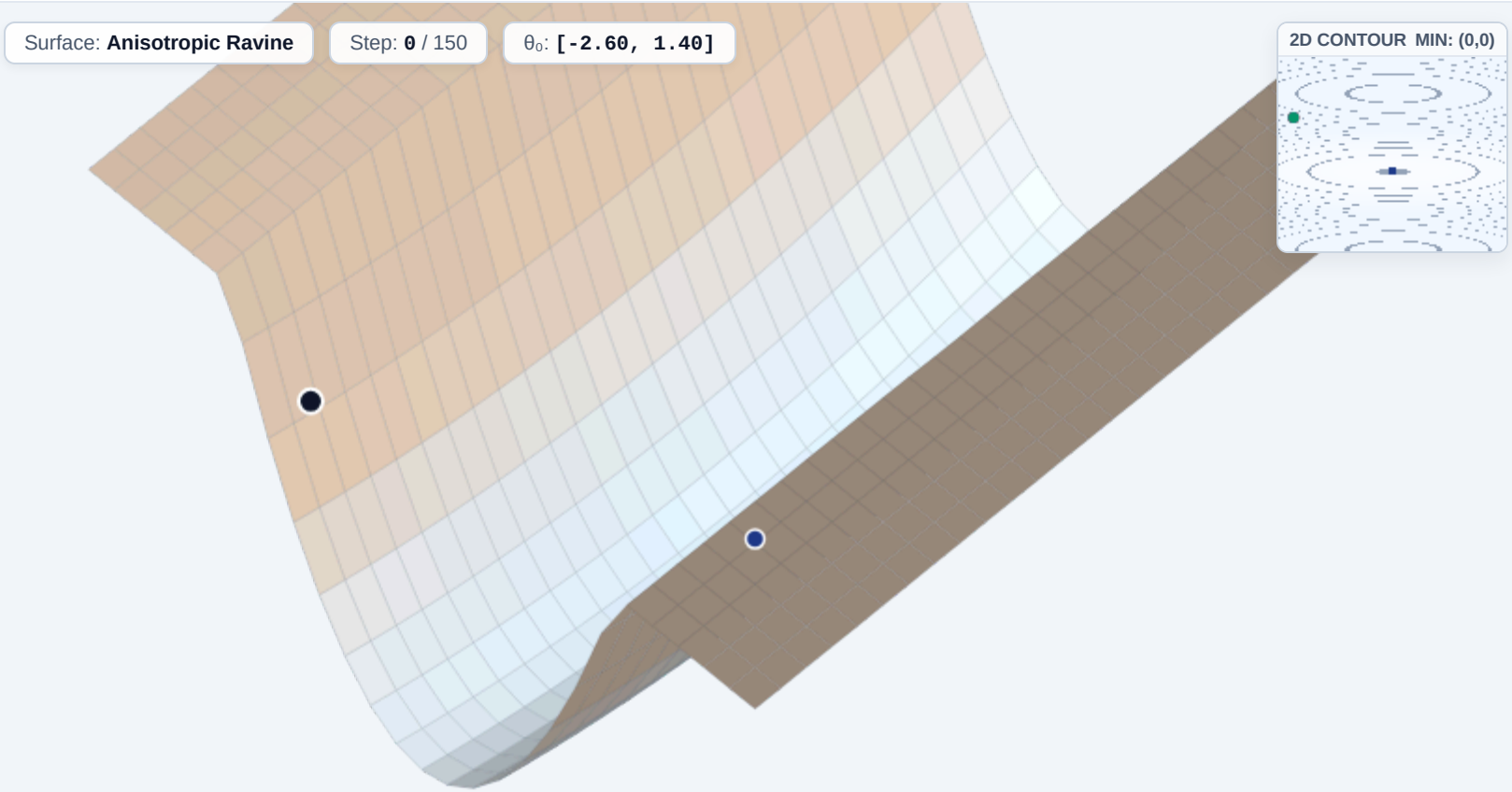
Optimisers


Optimization Dynamics in an Anisotropic Ravine: $f(x, y) = 0.5x^2 + 10y^2$
Parameter θ_2 (High Curvature / Steep Ravine)
Parameter θ_1 (Low Curvature / Flat Direction)
28.9
16.4
5.3
3.0
1.7
1.0
0.5
28.9
Vanilla SGD (Violent oscillations)
Momentum (Damped oscillations)
RMSProp (Rescaled steep curvature)
Adam (Direct highway to minimum)
Global Optimum (0, 0)

Optimisers

LECTURE LAB **Optimisation Dynamics Studio (SGD vs. Momentum vs. RMSProp vs. Adam)**

▶ Run Trajectories | Step (S) | ↺ Reset (R)



Landscape & Hyperparameters

Objective Surface: Anisotropic Ravine (Ill·

Learning Rate η: 0.035

Simulation Speed: 2× Lecture Pace

Convergence Telemetry

Loss $\mathscr{L}(\theta)$

Optimiser	Loss	Dist to Min	Show
SGD	10.0368	2.95	☑
Momentum (γ=0.9)	10.0368	2.95	☑
RMSProp (β=0.99)	10.0368	2.95	☑
Adam (β_1=0.9, β_2=0.999)	10.0368	2.95	☑

Mathematical Mechanics

Anisotropic Ravine: The curvature in the y-direction is 25× higher than along the x-floor ($\lambda_1 \gg \lambda_2$). Vanilla SGD oscillates violently between canyon walls. Momentum cancels perpendicular oscillations and accelerates along the floor, while Adam dynamically normalises step sizes via $\sqrt{\hat{v}_t}$.

Momentum

Intuition: Model optimisation as a heavy physical ball rolling down a potential well with inertia.

Polyak Momentum (1964)

Maintain an Exponential Moving Average of gradients (momentum):

$$\mathbf{m}_t = \beta \mathbf{m}_{t-1} + (1 - \beta)\mathbf{g}_t$$

$$\theta_{t+1} = \theta_t - \eta \mathbf{m}_t$$

- $\beta \approx 0.9$ (momentum coefficient).
- Oscillations across steep walls cancel out ($+g - g \approx 0$).
- Consistent gradients along the valley accumulate momentum $\rightarrow$ we start rolling down!

Adaptive Learning Rates: RMSProp & Adam

Instead of a single scalar learning rate η for all parameters, can we adapt η_i per coordinate?

RMSProp (Hinton, 2012)

Scale updates by the root mean square of recent gradients (usually $\beta = 0.9$):

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\mathbf{g}_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbf{v}_t} + \epsilon} \odot \mathbf{g}_t$$

- Large gradients $\rightarrow$ large v_i $\rightarrow$ smaller step (damps oscillations).
- Tiny gradients $\rightarrow$ small v_i $\rightarrow$ larger step (speeds up flat valleys).

Where is RMS? $\mathbf{g}_t^2$ is the square, $\mathbf{v}_t$ is the exponentially weighted mean square (variance estimate), and $\sqrt{\mathbf{v}_t}$ is the root.

Adam (Kingma & Ba, 2014)

Combines **momentum** ($\mathbf{m}_t$, 1st moment) and **RMSProp** ($\mathbf{v}_t$, 2nd moment):

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}, \quad \hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} \odot \hat{\mathbf{m}}_t$$

Standard defaults: $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$. The division by $1-\beta^t$ is a correction for unbiased estimate of the moments. Notice that as $t \to \infty$, $1-\beta^t \to 1$.

> What is the **moment**? The k-th moment of a random variable X is defined as $\mathbb{E}[X^k]$.

AdamW (Loshchilov & Hutter, 2017)

In Adam, adding L_2 regularisation $\frac{1}{2}\lambda\|\theta\|_2^2$ comes out as (ignoring the momentum term):

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}(\mathbf{g}_t + \lambda\theta_t)$$

Weights with large gradients receive **less** weight decay than weights with small gradients! This is of course a problem.

Instead, decouple weight decay completely from the adaptive gradient moment updates:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}\hat{\mathbf{m}}_t - \eta\lambda\theta_t$$

This decoupled formulation is the universal standard for training modern deep learning (Transformers, LLMs) and our cheap MLPs.

Regularisation

Explicit Penalty

The intuition works the same from our curve-fitting days.

- **Weight Decay (L_2):** Penalises large weights, favouring smooth decision surfaces:

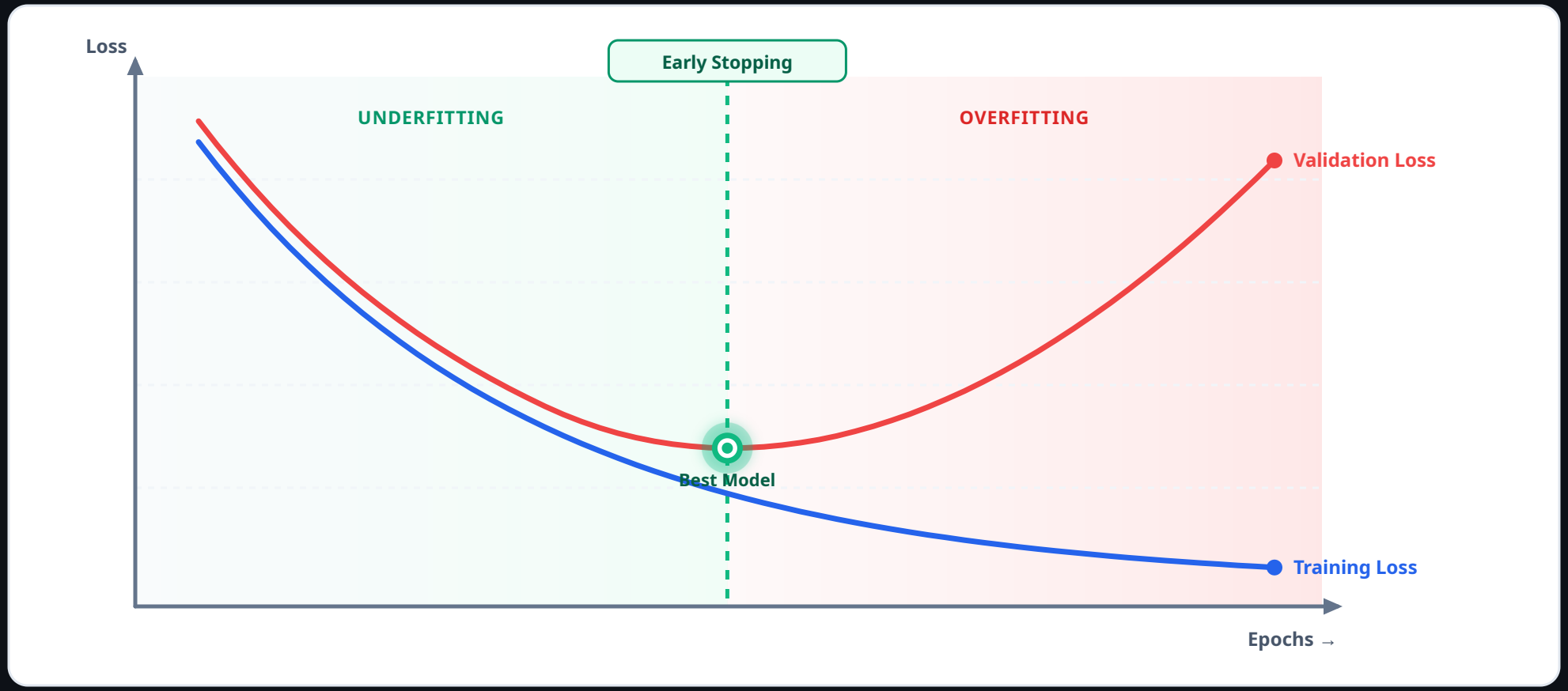
$$\mathcal{L}_{\text{total}} = \mathcal{L}_0 + \frac{\lambda}{2} \sum_l \|\mathbf{W}^{(l)}\|_2^2$$

- **Sparsity (L_1):** Drives non-essential weights to exactly zero:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_0 + \lambda \sum_l \|\mathbf{W}^{(l)}\|_1$$

Early Stopping

Halting training when validation loss begins to rise limits effective capacity.



Dropout

Introduced by Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014):

Training Phase

At each training step, randomly zero out each hidden unit with probability p (typically $p = 0.5$):

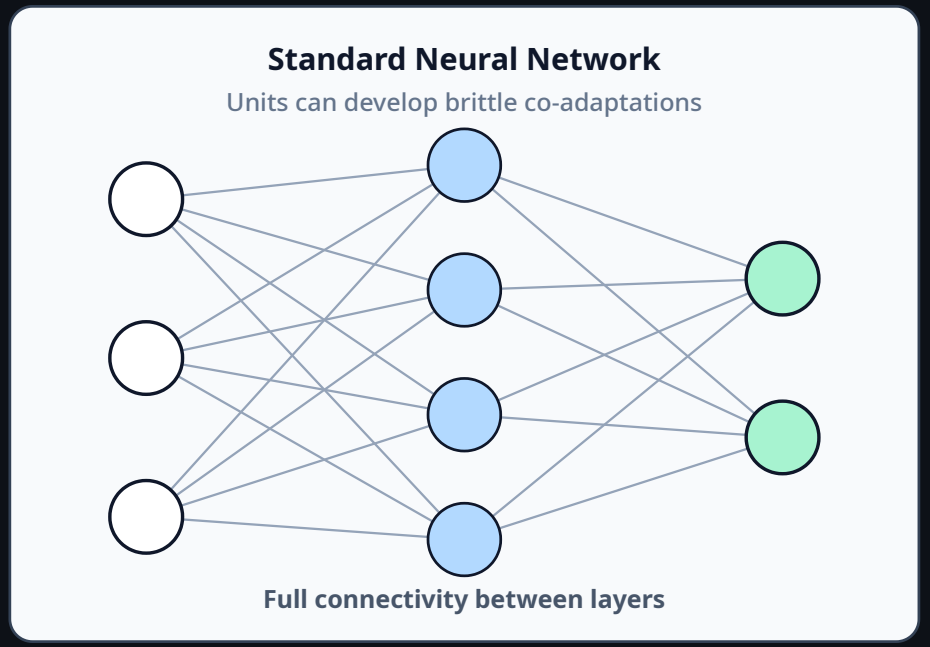
$$\mathbf{r} \sim \text{Bernoulli}(1 - p)$$

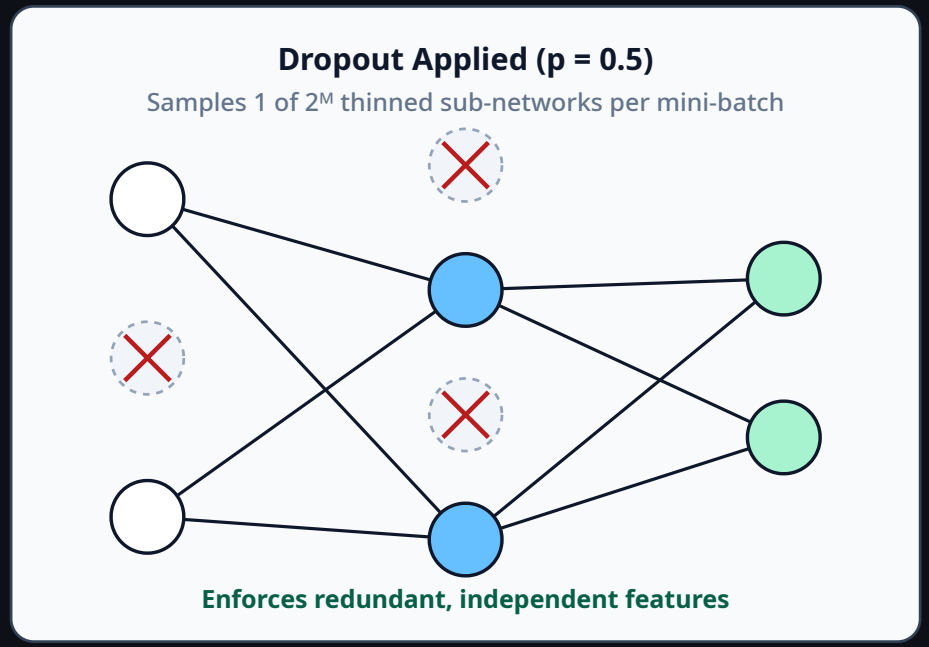
$$\tilde{\mathbf{h}} = \frac{1}{1 - p}(\mathbf{h} \odot \mathbf{r})$$

(Inverted dropout scales by $\frac{1}{1-p}$ during training so inference requires zero modifications as in `torch.nn.Dropout`*).*

Why Dropout? Intuition:

1. **Breaks Co-Adaptation:** Neurons cannot rely on specific neighbouring units to fix their mistakes; each unit must learn independently useful features.
2. **Implicit Ensemble:** Training with dropout samples 1 of $2^{|\theta|}$ possible subnetworks at every step. At test time, using all neurons acts as an ensemble over $2^{|\theta|}$ models.





Internal Shift

What the layer is learning changes as the previous layers are learning!

$$\mathbf{h}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)})$$

Deeper layers are forced to constantly adapt to changing input distributions, slowing convergence and requiring delicate learning rates.

Normalisation

Transform pre-activations to have zero mean and unit variance:

$$\hat{\mathbf{z}} = \frac{\mathbf{z} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

$$\mathbf{y} = \gamma\hat{\mathbf{z}} + \beta$$

$(\gamma, \beta) \in \mathbb{R}^{d_{out}}$ are learnable parameters that allow the network to recover the original identity mapping if needed.

Batch vs. Layer Normalisation

Where do we compute the statistics μ and σ^2?

Batch Normalisation

(Ioffe & Szegedy, 2015)

- Normalises across the **batch dimension** N for each feature independently.
- Computes mini-batch stats during training; uses running exponential averages at inference.
- **Drawback:** Breaks when batch size N is small; impossible for variable-length sequences with zero padding.

Layer Normalisation

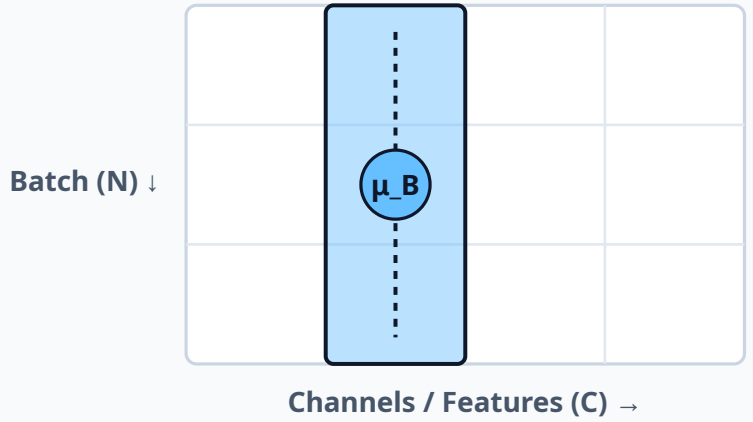
(Ba, Kiros, Hinton, 2016)

- Normalises across all **features** for each individual sample independently.
- Exactly identical computation during training and inference!
- Fully independent of batch size; the universal standard in **Transformers and LLMs**.

BatchNorm vs. LayerNorm

Batch Normalisation (BatchNorm)

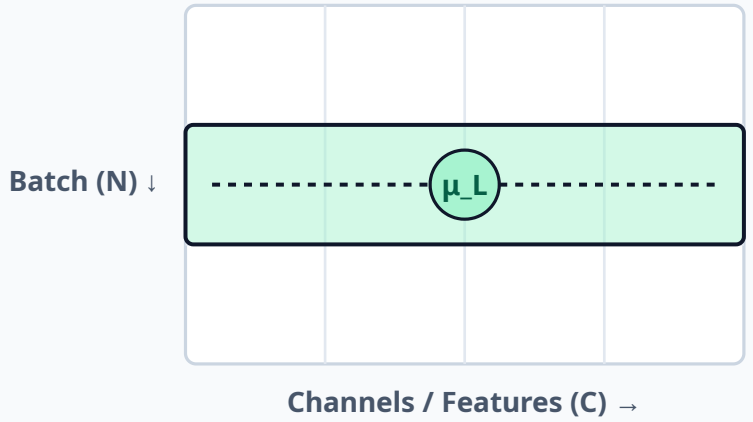
Normalises across mini-batch (N) for each feature (C)



Breaks when batch size N is small or variable

Layer Normalisation (LayerNorm)

Normalises across all features (C) for each sample (n)



✓ Batch-independent — standard in Transformers & RNNs

Okay Steve, why now?

Neural networks were theorised in the 1950s and backprop was popularised in 1986. Why did the deep learning revolution only go BOOM in 2012?

1. Compute

- **NVIDIA CUDA (2006)** repurposed GPUs from 3D gaming to general-purpose matrix engines.
- $100\times$ speedup over CPUs made multi-day experiments feasible.

2. Big Data

- High-capacity neural networks overfit on small datasets.
- The web enabled **ImageNet (Fei-Fei Li, 2009)**: 14 million annotated images across 20,000 categories.

3. Soup

- Non-saturating **ReLU** activations.
- Principled **initialisation**.
- **Adam / AdamW** optimisers.
- **Dropout & LayerNorm**.

PyTorch: Does all this for you

PyTorch bridges mathematical formulation and hardware execution for you:

1. **Autograd Engine (Reverse-Mode AD)**
    - Constructs a **dynamic computational graph** (define-by-run) on the fly during the forward pass.
    - Automatically computes gradients via `.backward()`.
2. **Hardware Acceleration via Tensor Abstraction**
    - High-dimensional arrays (`torch.Tensor`) backed by CUDA/ROCm C++ kernels.
    - Maps batched forward/backward passes directly to GPU **GEMM** operations.

`torch.Tensor`	`torch.autograd`	`torch.nn`
N-dimensional array with gradient tracking	Automatic differentiation $\rightarrow$ backpropagation	Neural networks (`Module`, `Parameter`, layers, losses)

MNIST Example

1. Model Definition

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28 * 28, 128),
            nn.ReLU(),
            nn.Linear(128, 10),
        )

    def forward(self, x):
        return self.net(x)
```

2. Training Loop

```
model = MLP().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
lossfn = nn.CrossEntropyLoss()

for x, y in train_loader:
    ...
    optimizer.zero_grad()
    out = model(x)
    loss = lossfn(out, y)
    loss.backward()
    optimizer.step()
    ...
```

Questions?

That wraps up the core of neural networks.

Inductive Biases & The Architecture Zoo

An **inductive bias** is the set of prior assumptions an algorithm encodes regarding the geometry and structure of the target function.

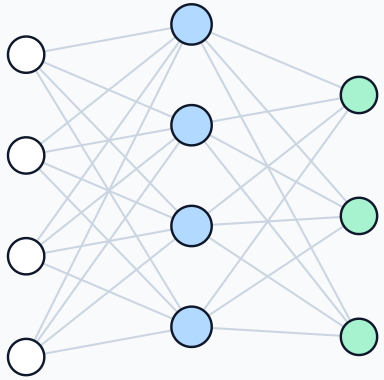
- Fully-Connected (MLP) $\rightarrow$ All to all, no assumptions
- Convolutional (CNN) $\rightarrow$ Spatial Locality & Translation Equivariance
- Recurrent $\rightarrow$ Sequential, Temporal
- Attention $\rightarrow$ Relational

> We can mix and match, e.g. CNN -> RNN -> Attention to process videos.

Inductive Biases: Architectures

Fully Connected (MLP)

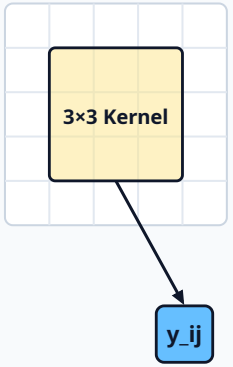
Dense / Minimal Prior Bias



All-to-all parametric connections

Convolutional (CNN)

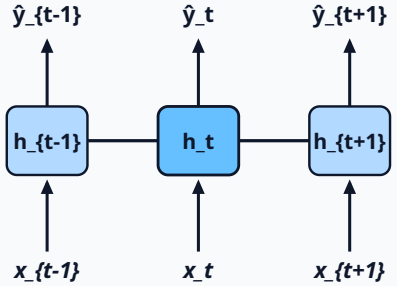
Local & Translation Equivariant



Shared weights across spatial grid

Recurrent (RNN)

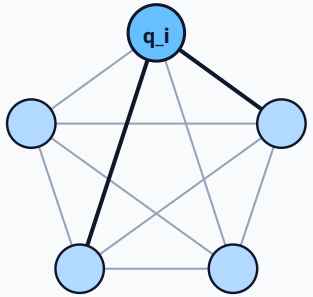
Sequential & Temporal Invariant



Recurrent memory across time

Transformer (Attention)

Relational & Direct Routing



Dynamic content-based routing

Convolutional Neural Networks

Two problems with fully-connected neural networks for computer vision:

1. Parameter Explosion

A high-resolution image ($1000 \times 1000 \times 3$) has 3×10^6 input dimensions. Connecting this to 1,000 hidden units requires:

$$3 \times 10^6 \times 10^3 = \mathbf{3 \times 10^9} \text{ parameters!}$$

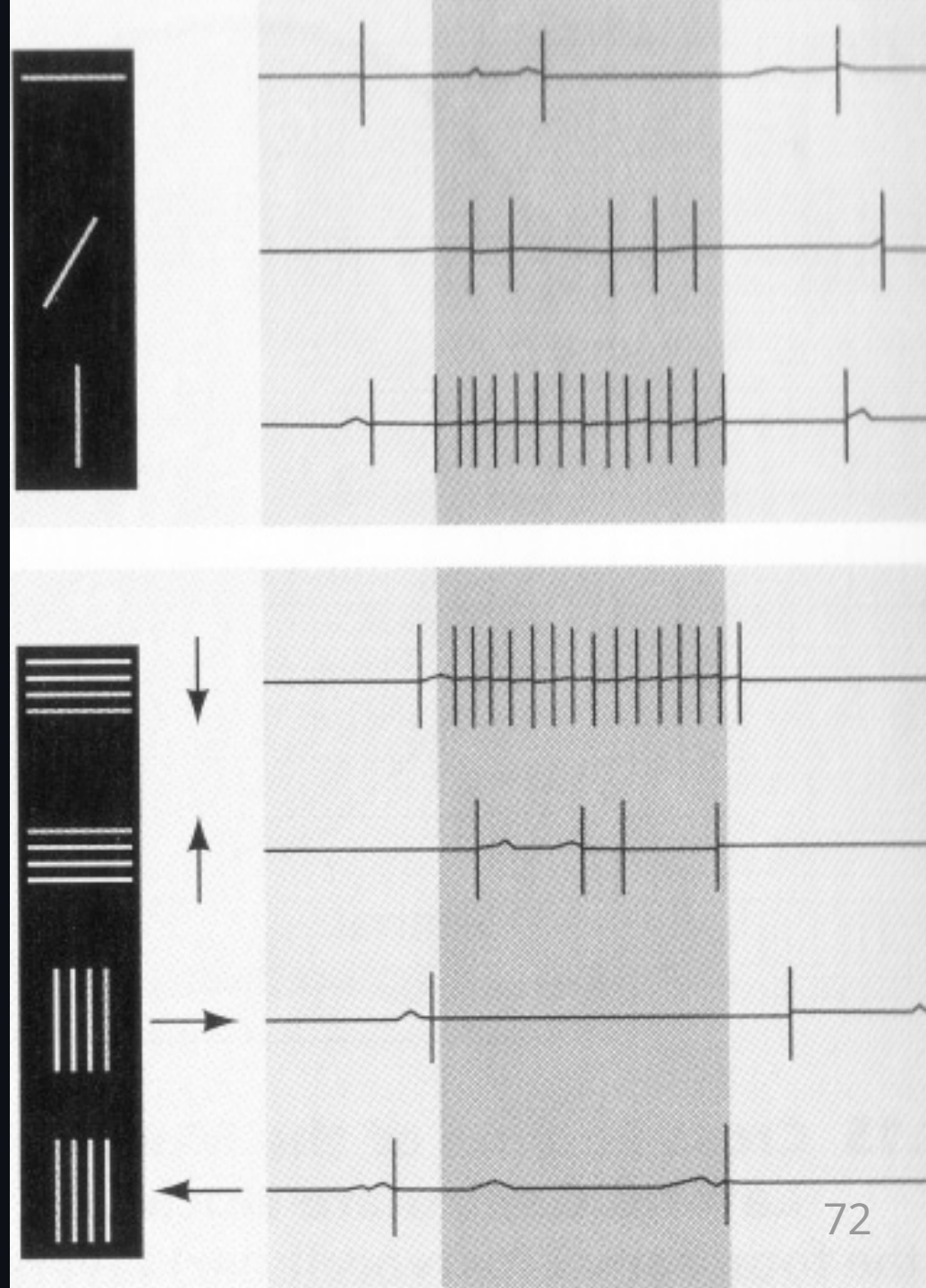
A single dense layer (MLP) = gigabytes of memory.

2. Destruction of 2D Topology

Flattening an image into a 1D vector $\mathbf{x} \in \mathbb{R}^{3 \cdot 10^6}$ destroys spatial neighbourhood relationships. Neighbouring pixels (x, y) and $(x+1, y)$ are treated independently.

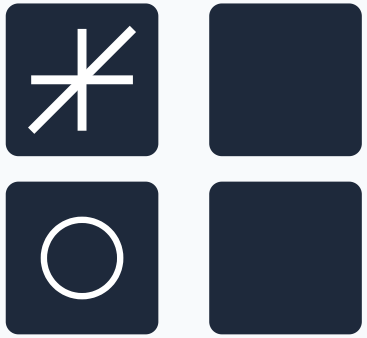
Biological Inspiration: Hubel & Wiesel (1962, 1981 Nobel Prize)

David Hubel and Torsten Wiesel discovered that neurons in the mammalian primary visual cortex (V1) have **local receptive fields** sensitive to oriented edges, organised into hierarchical simple and complex cell columns.



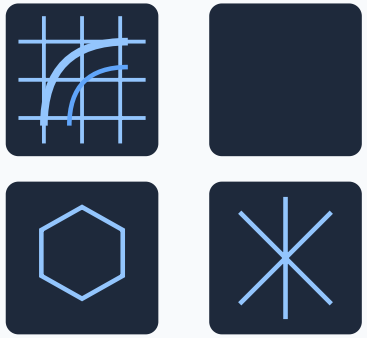
The Hierarchy of Learned Representations

LAYER 1: EDGES



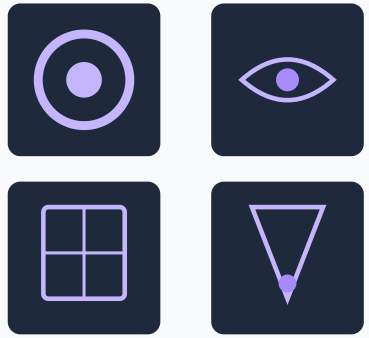
Gabor-like edge filters
Local gradients & lines

LAYER 2: TEXTURES



Corners, motifs, curves
Compound edge textures

LAYER 3: PARTS



Wheels, eyes, petals
Structural components

LAYER 4: OBJECTS

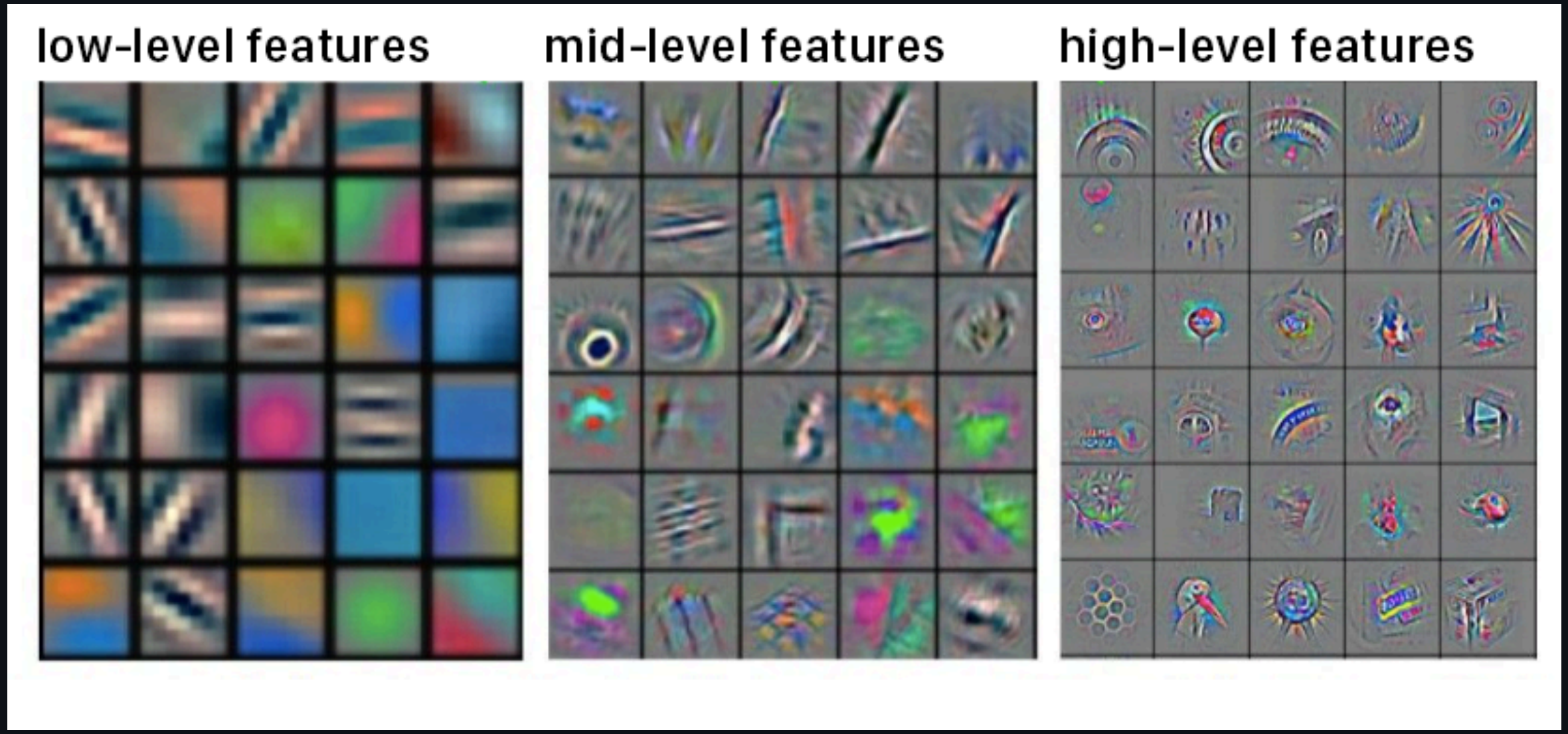
🚗 **Automobile**

🐕 **Golden Retriever**

✈️ **Airliner**

Semantic classes
Invariant object concepts

Hierarchy of Abstraction: Increasing Receptive Field & Semantic Invariance



Source: https://twopointseven.github.io/2017-10-29/cnn/

The 2D Convolution Operation

Instead of all-to-all dense matrix multiplication, a convolutional layer slides a small parameter kernel $\mathbf{K} \in \mathbb{R}^{K_h \times K_w}$ across the input image:

$$(I * K)(i, j) = \sum_m \sum_n I(i+m, j+n) K(m, n)$$

1. Local Connectivity

Each neuron connects only to a tiny spatial patch (e.g. 3×3 or 5×5), mimicking biological receptive fields.

2. Weight Sharing

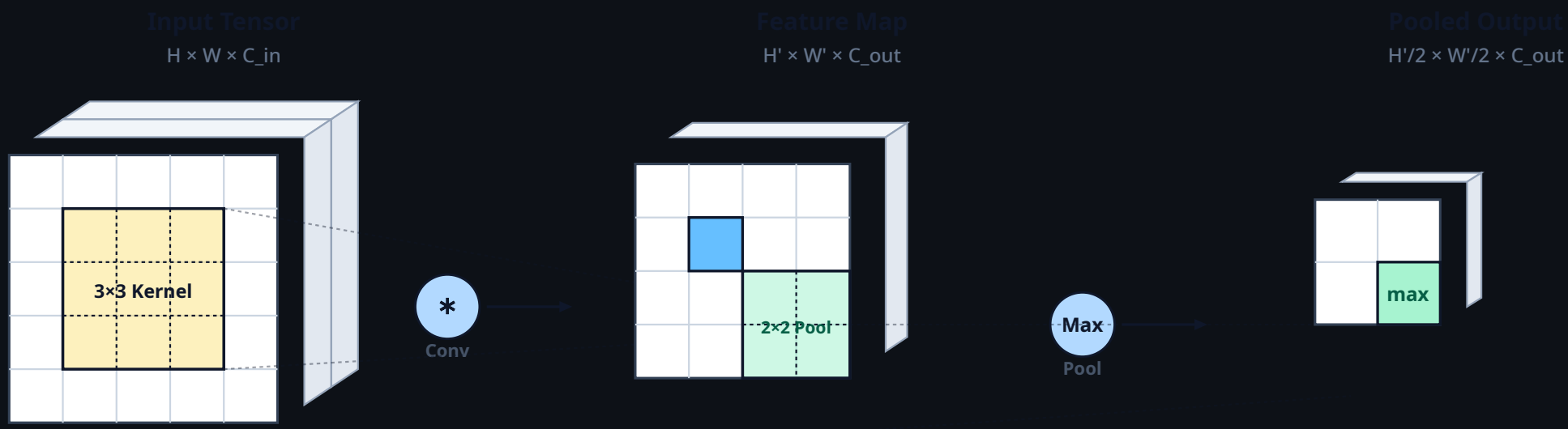
Kernel weights are reused across every single pixel location in the image.

3. Translation Equivariance

If an object shifts in the input by g, its activation representation shifts by the exact same amount:

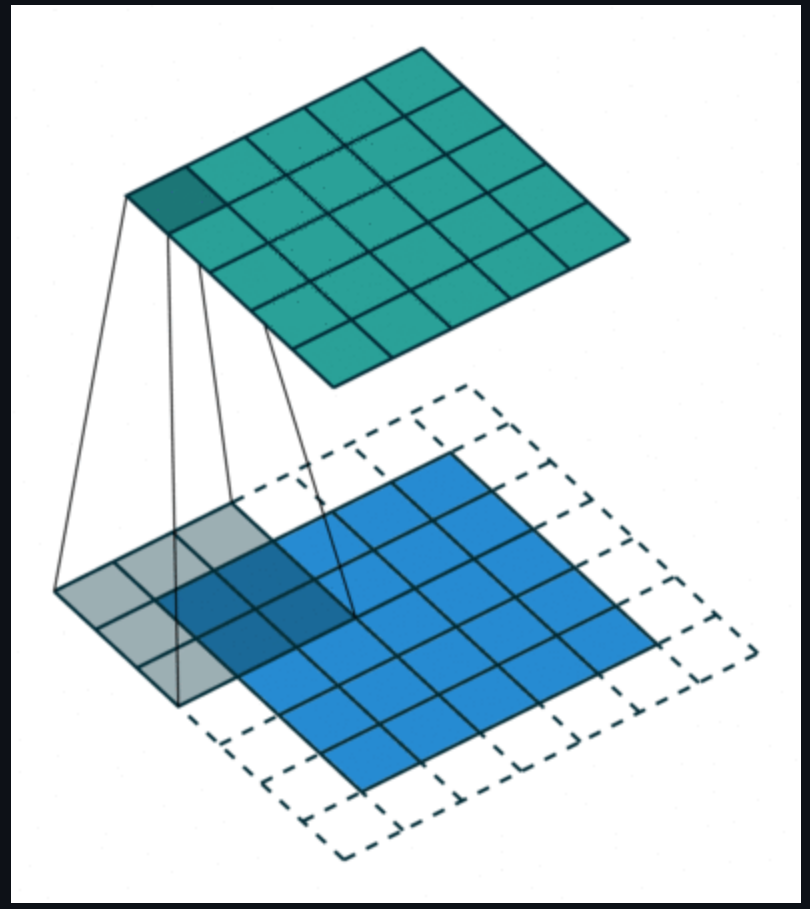
$$f(T_g(\mathbf{x})) = T_g(f(\mathbf{x}))$$

2D Convolution & Max Pooling

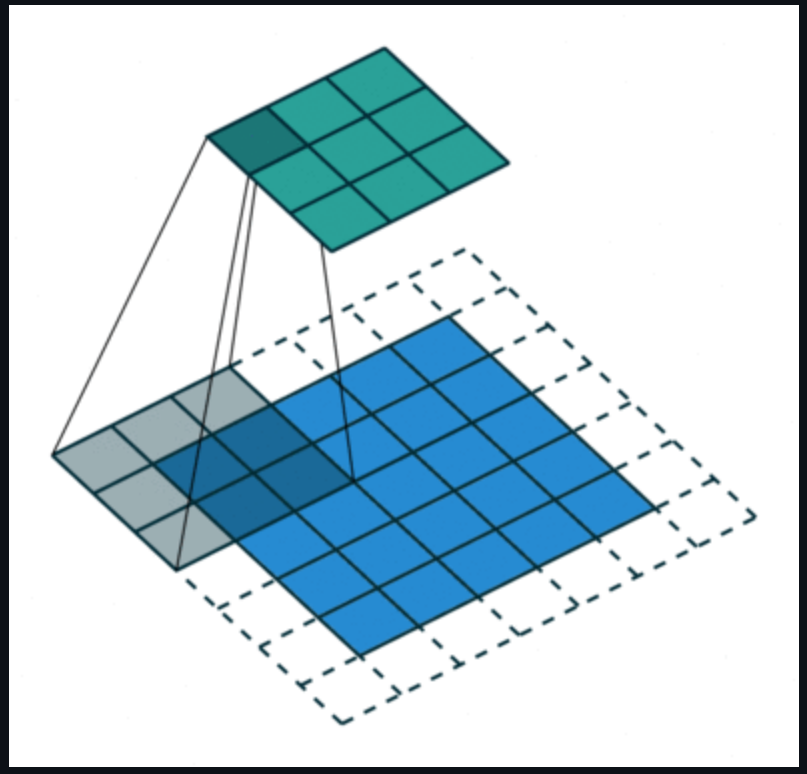


Inductive Biases: Local Receptive Fields • Shared Filter Weights • Downsampling Translation Invariance

No Stride, Padding

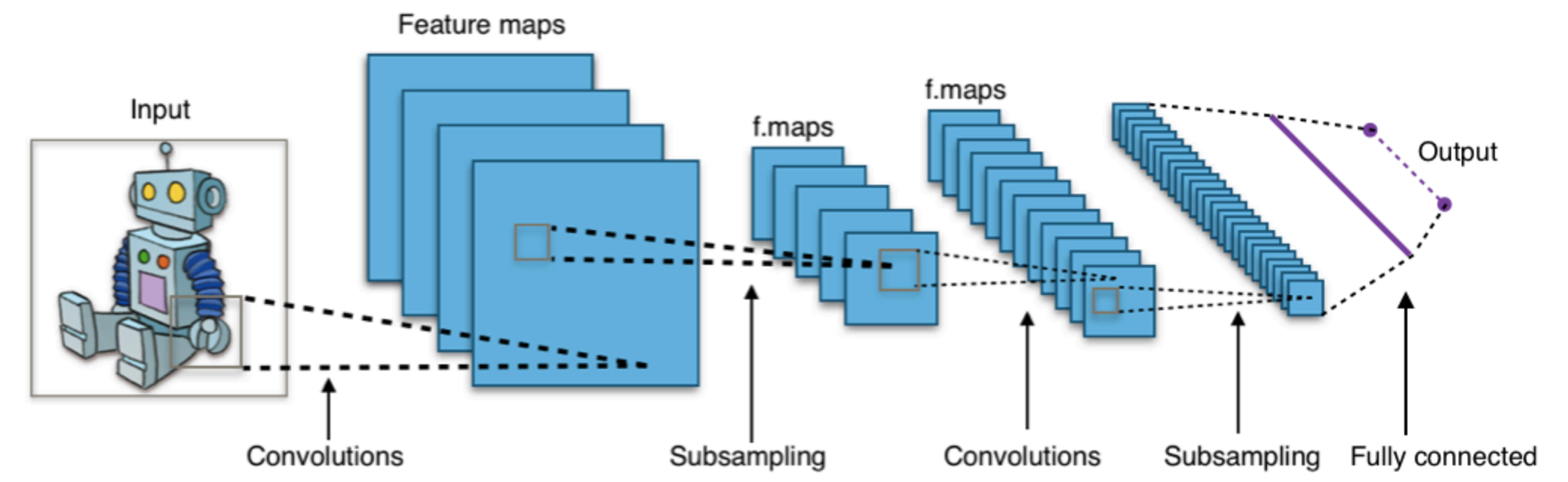


Stride and Padding



The same sliding window business happens for convolutional and pooling layers.

Typical CNN Architecture



Hyperparameters of the Convolutional Layer

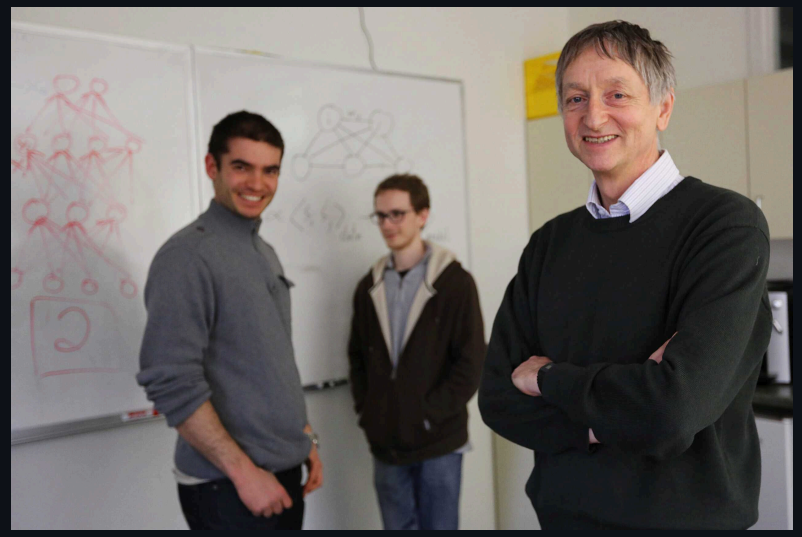
Given input spatial dimension W, kernel size K, padding P, and stride S:

$$W_{\text{out}} = \left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1$$

- **Stride S:** Number of pixels the kernel jumps per step (downsamples when $S > 1$).
- **Padding P:** Border pixels added to preserve spatial boundaries:
  - *Valid padding:* $P = 0$ (output shrinks).
  - *Same padding:* $P = \frac{K-1}{2}$ (for odd K and $S = 1$, $W_{\text{out}} = W$).

AlexNet & The 2012 ImageNet Breakthrough

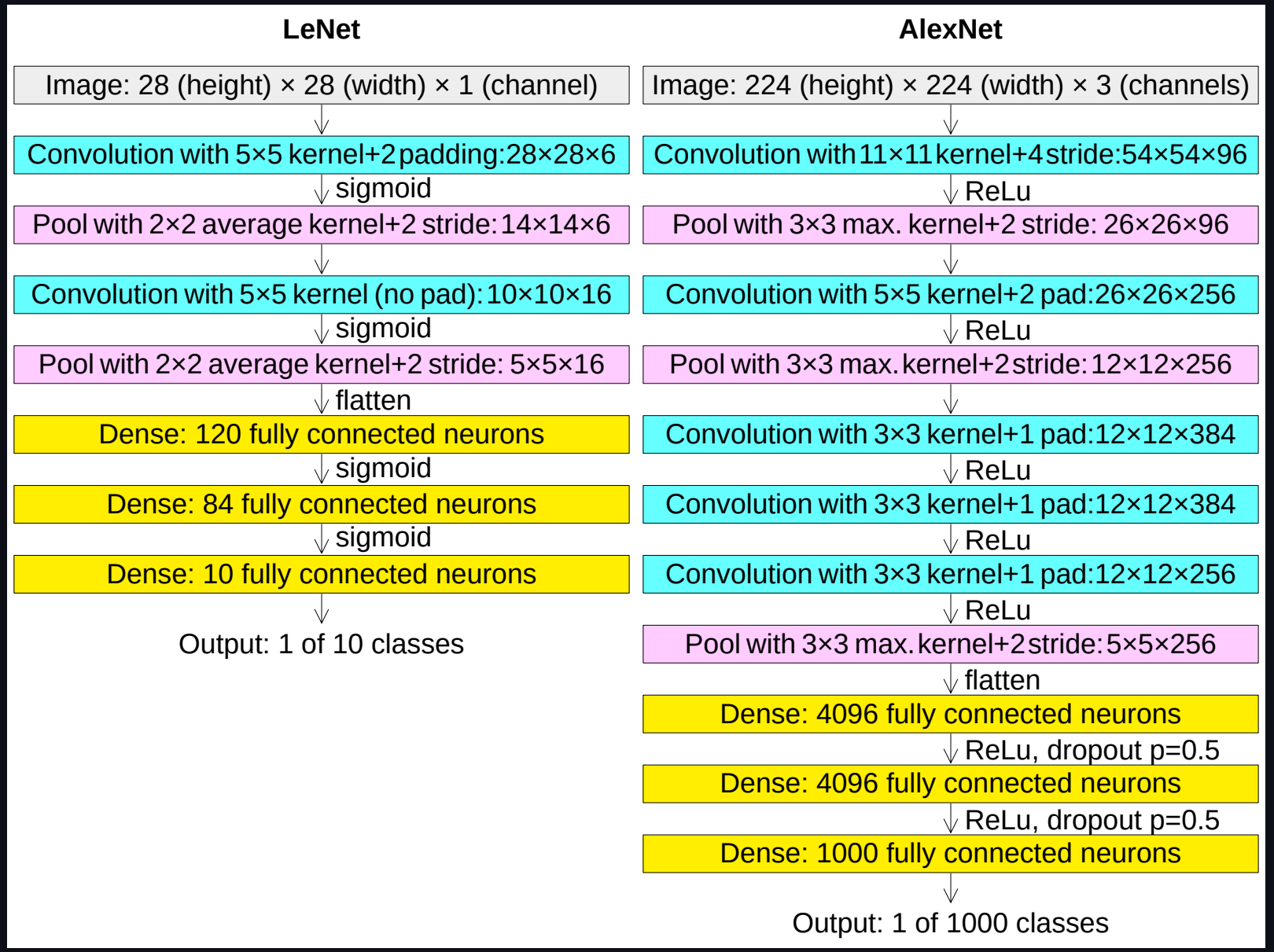
The modern era of deep learning officially began in October 2012 at the ImageNet Large Scale Visual Recognition Challenge (ILSVRC):



Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever
Credit: Computer History Museum

The 10.9% Error Drop

- **Traditional Computer Vision:** SIFT / HOG features + SVM achieved **26.2% error**.
- **AlexNet (CNN): 15.3% error**!
- Architecture:
  - 5 convolutional layers + 3 dense layers (60M parameters).
  - First large-scale use of **ReLU** activations.
  - **Dropout (0.5)** in dense layers.


LeNet
Image: 28 (height) × 28 (width) × 1 (channel)
Convolution with 5×5 kernel+2 padding:28×28×6
sigmoid
Pool with 2×2 average kernel+2 stride: 14×14×6
Convolution with 5×5 kernel (no pad): 10×10×16
sigmoid
Pool with 2×2 average kernel+2 stride: 5×5×16
flatten
Dense: 120 fully connected neurons
sigmoid
Dense: 84 fully connected neurons
sigmoid
Dense: 10 fully connected neurons
Output: 1 of 10 classes
AlexNet
Image: 224 (height) × 224 (width) × 3 (channels)
Convolution with 11×11 kernel+4 stride:54×54×96
ReLu
Pool with 3×3 max. kernel+2 stride: 26×26×96
Convolution with 5×5 kernel+2 pad:26×26×256
ReLu
Pool with 3×3 max. kernel+2 stride: 12×12×256
Convolution with 3×3 kernel+1 pad:12×12×384
ReLu
Convolution with 3×3 kernel+1 pad:12×12×384
ReLu
Convolution with 3×3 kernel+1 pad:12×12×256
ReLu
Pool with 3×3 max. kernel+2 stride: 5×5×256
flatten
Dense: 4096 fully connected neurons
ReLu, dropout p=0.5
Dense: 4096 fully connected neurons
ReLu, dropout p=0.5
Dense: 1000 fully connected neurons
Output: 1 of 1000 classes

Residual Networks (ResNet)

After AlexNet (8 layers) and VGG (19 layers), researchers attempted to train 30, 50, and 100 layer plain networks. They went BOOM (in a bad way):

The Degradation Barrier

(He et al., 2015)
A 56-layer plain network suffered **higher training error** than a 20-layer network.

- This was **not overfitting** (both training and test error were worse).
- Worst case idea: **identity mapping** $\mathcal{H}(\mathbf{x}) = \mathbf{x}$. But unable to learn identity functions via gradient descent.

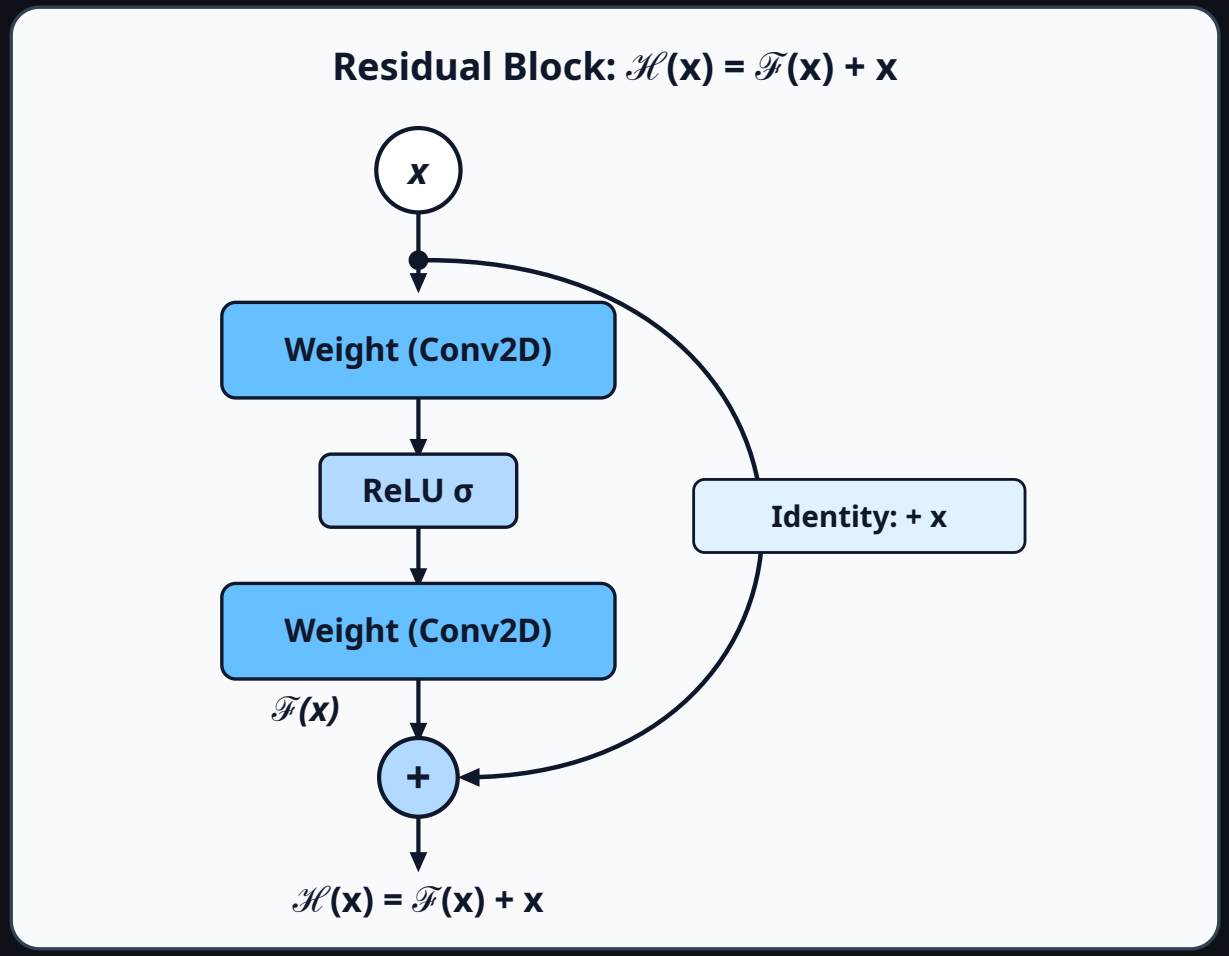
The Residual Solution

Instead of forcing weight layers to fit target mapping $\mathcal{H}(\mathbf{x})$, parameterise the **residual**:

$$\mathcal{H}(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x}$$

If identity is optimal, optimiser simply drives $\mathcal{F}(\mathbf{x}) \to \mathbf{0}$.

Residual Block



Why ResNets Can Train 100+ Layers

Consider the chain rule gradient of a residual block with loss $\mathcal{L}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \mathcal{H}} \frac{\partial \mathcal{H}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \mathcal{H}} \left(\frac{\partial \mathcal{F}}{\partial \mathbf{x}} + \mathbf{I} \right) = \frac{\partial \mathcal{L}}{\partial \mathcal{H}} \frac{\partial \mathcal{F}}{\partial \mathbf{x}} + \frac{\partial \mathcal{L}}{\partial \mathcal{H}}$$

Even if $\frac{\partial \mathcal{F}}{\partial \mathbf{x}}$ completely vanishes, the gradient flows directly backwards unimpeded to layer 0!

ResNet-152 (ILSVRC 2015)

- Trained a 152-layer network cleanly.
- Achieved **3.57% top-5 error**, surpassing human-level performance (5.1%) on ImageNet.
- Residual skip connections are now an indispensable architectural primitive across all of deep learning (including Transformers!).

Recurrent Neural Networks (RNNs)

Images are static spatial grids. But language, audio, DNA, and financial time-series are sequential and variable-length: $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$.

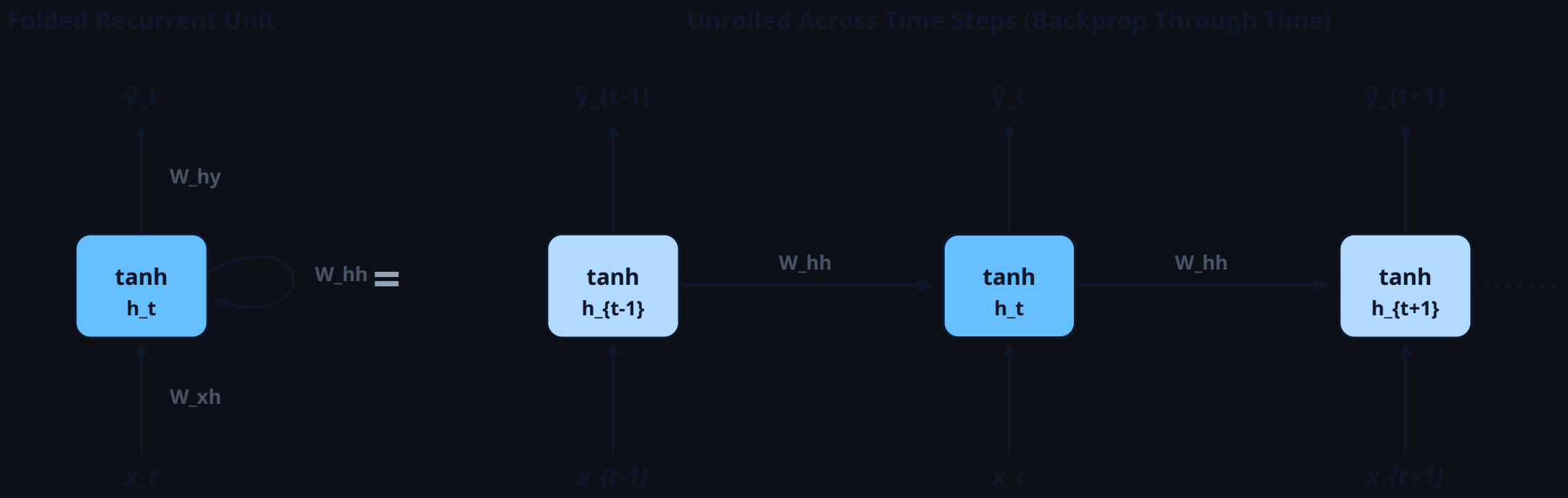
Maintain an internal hidden memory state $\mathbf{h}_t \in \mathbb{R}^d$:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h)$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y)$$

- $\mathbf{W}_{hh}$ (Hidden-to-hidden recurrent transitions).
- $\mathbf{W}_{xh}$ (Input-to-hidden projections).
- **Parameter Sharing:** The exact same weight matrices are reused at every time step t!

RNN Computational Graph



$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t + b_h)$ | Parameters (W_{hh}, W_{xh}, W_{hy}) are shared across all time steps

Backpropagation Through Time (BPTT)

To train an RNN, we unroll the computation graph across T steps and accumulate gradients:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{hh}}$$

Follow the ~~money~~ gradient

This time $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}$ involves repeated applications of $\mathbf{W}_{hh}^T$ and derivatives of $\tanh$.

$$h_{t-1} \xrightarrow{\mathbf{W}_{hh}} \xrightarrow{\tanh} h_t \xrightarrow{\mathbf{W}_{hh}} \xrightarrow{\tanh} h_{t+1}$$

Vanishing & Exploding Gradients

- Gradients can vanish like before $\implies$ difficult to learn long-range dependencies.
- Gradients can explode, like before $\implies$ must use something like **gradient clipping**.
- Gated units (LSTM / GRU) solved vanishing gradients via additive linear cell states:

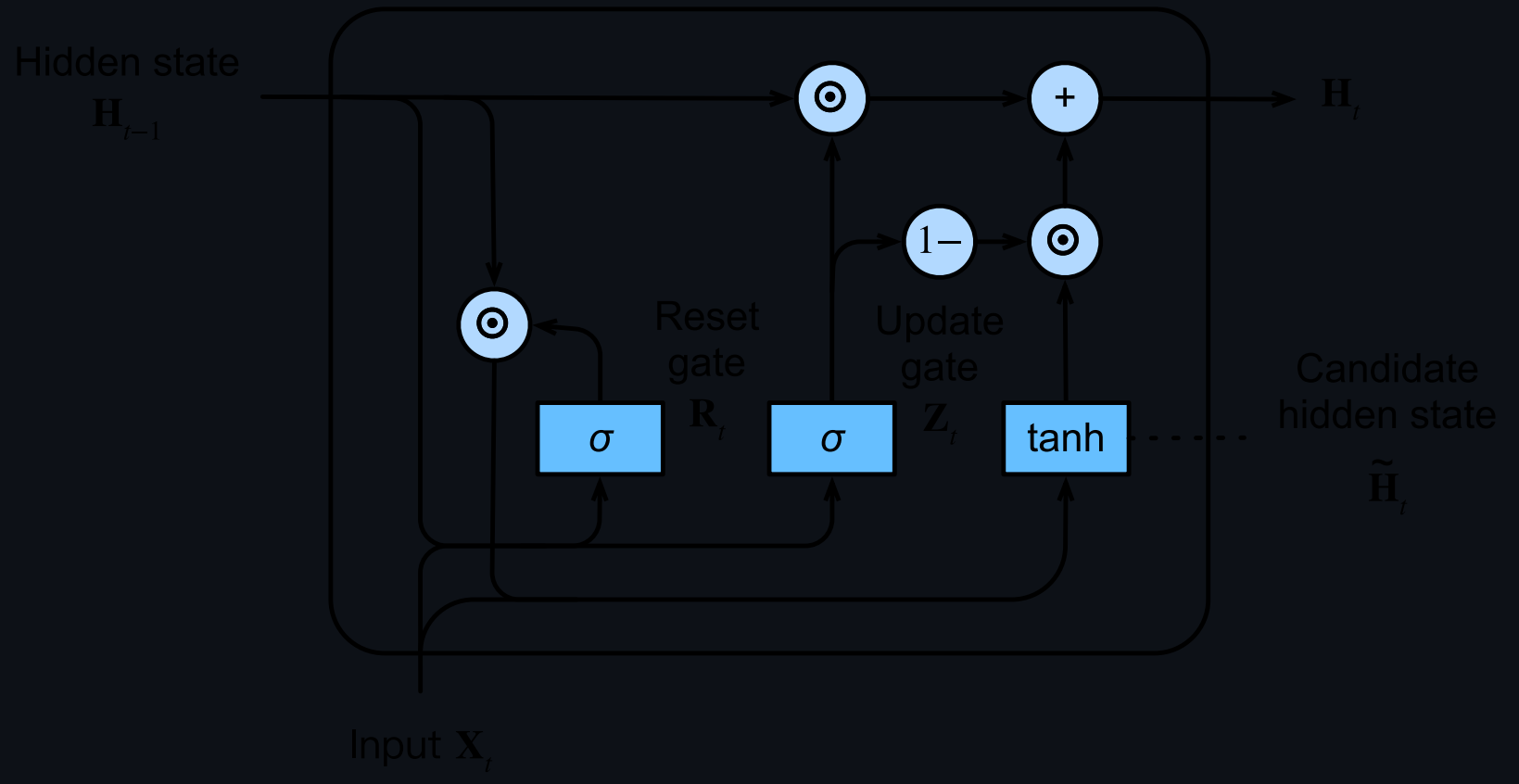
GRU (slightly simpler of two):

$$
\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{W}_z \mathbf{x}_t + \mathbf{U}_z \mathbf{h}_{t-1}) \\
\mathbf{r}_t &= \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}_r \mathbf{h}_{t-1}) \\
\tilde{\mathbf{h}}_t &= \tanh(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h (\mathbf{r}_t \odot \mathbf{h}_{t-1})) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t
\end{aligned}
$$

where:

- $\mathbf{z}_t$ is the **update gate** (how much of the past information to keep)
- $\mathbf{r}_t$ is the **reset gate** (how much of previous state to retain for candidate)

Gated Recurrent Unit (GRU)





Two problems with RNNs

- **The Information Bottleneck**
  Compressing arbitrary long-range semantics into a fixed vector causes catastrophic forgetting of early sentence context.
- **Sequential Computation**
  Since each hidden state $\mathbf{h}_t$ depends on the previous one, the computation cannot be parallelised across the time dimension. This prevents RNNs from taking full advantage of modern parallel hardware like GPUs.

The Attention Mechanism

Let the model learn to access and combine only the relevant information.

In sequence-to-sequence translation, an RNN encoder compresses an entire 50-word sentence into a single fixed-size vector $\mathbf{h}_T$:

$$\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T \xrightarrow{\text{RNN Encoder}} \mathbf{h}_T \xrightarrow{\text{RNN Decoder}} \mathbf{y}_1, \ldots, \mathbf{y}_{T'}$$

Bahdanau et al. (2014) Soft Alignment

Instead of a static bottleneck $\mathbf{h}_T$, allow the decoder to look back and attend dynamically to **all** encoder hidden states via content-based addressing!

Seq2Seq Example: English $\rightarrow$ French

Task: translate **"The black cat slept quietly"** $\rightarrow$ **"Le chat noir dormait paisiblement"**.

1. Encoder (Source Sentence)

Input sequence of tokens ($T = 5$):

$$\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5)$$

$$[\text{"The"}, \text{"black"}, \text{"cat"}, \text{"slept"}, \text{"quietly"}]$$

At each step j, the encoder RNN computes:

$$\mathbf{h}_j = \text{RNN}_{\text{enc}}(\mathbf{x}_j, \mathbf{h}_{j-1})$$

- Produces hidden states $\{\mathbf{h}_1, \ldots, \mathbf{h}_5\}$.
- Each $\mathbf{h}_j$ encodes word $\mathbf{x}_j$ in context.
- **Bottleneck:** Vanilla Seq2Seq keeps only $\mathbf{h}_5$!

2. Decoder (Target Translation)

Target sequence of tokens ($T' = 5$):

$$\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \mathbf{y}_5)$$

$$[\text"Le", \text"chat", \text"noir", \text"dormait", \text"paisiblement"]$$

At step t, given state $\mathbf{s}_{t-1}$ and context $\mathbf{c}_t$:

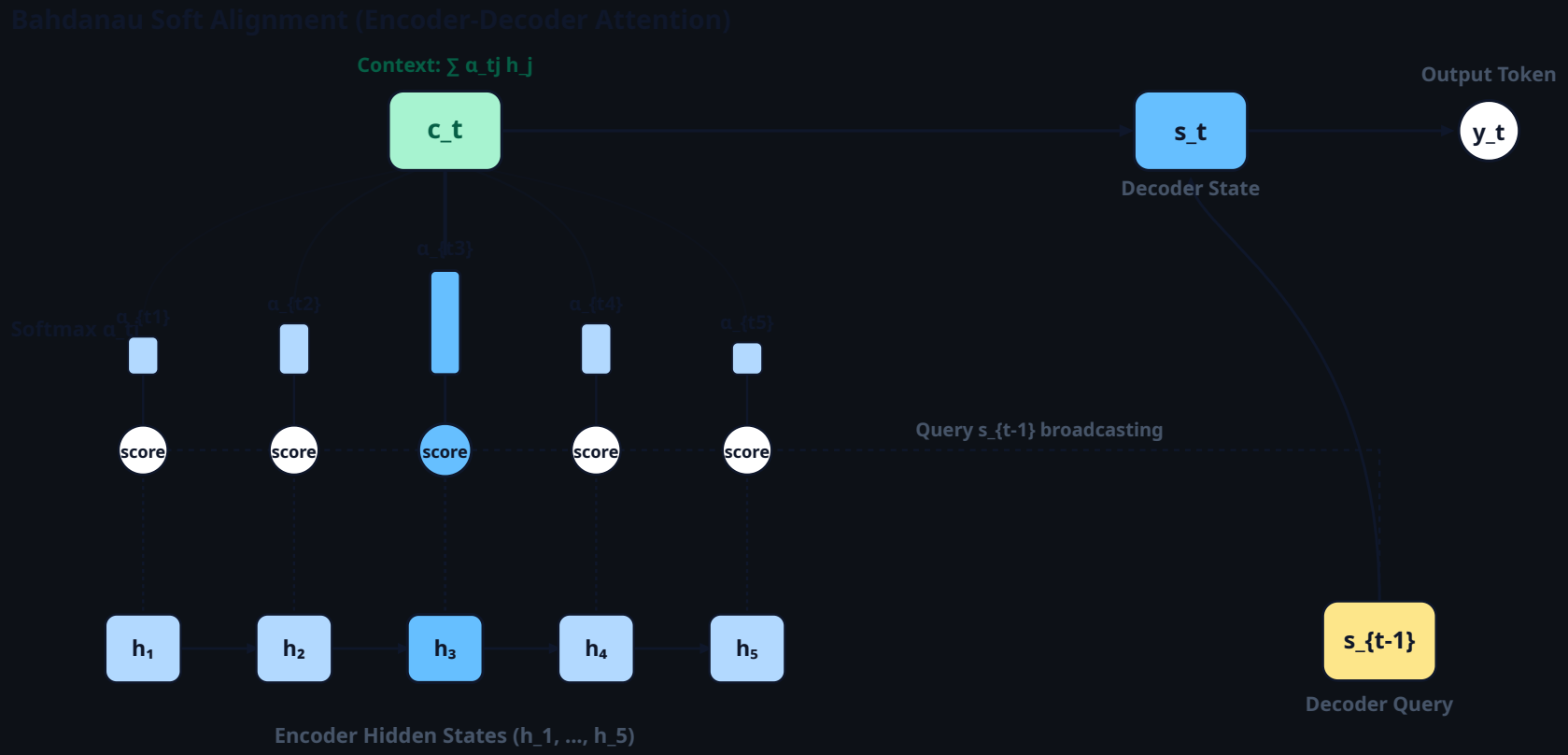
$$\mathbf{s}_t = \text{RNN}_{\text{dec}}(\mathbf{y}_{t-1}, \mathbf{s}_{t-1}, \mathbf{c}_t)$$

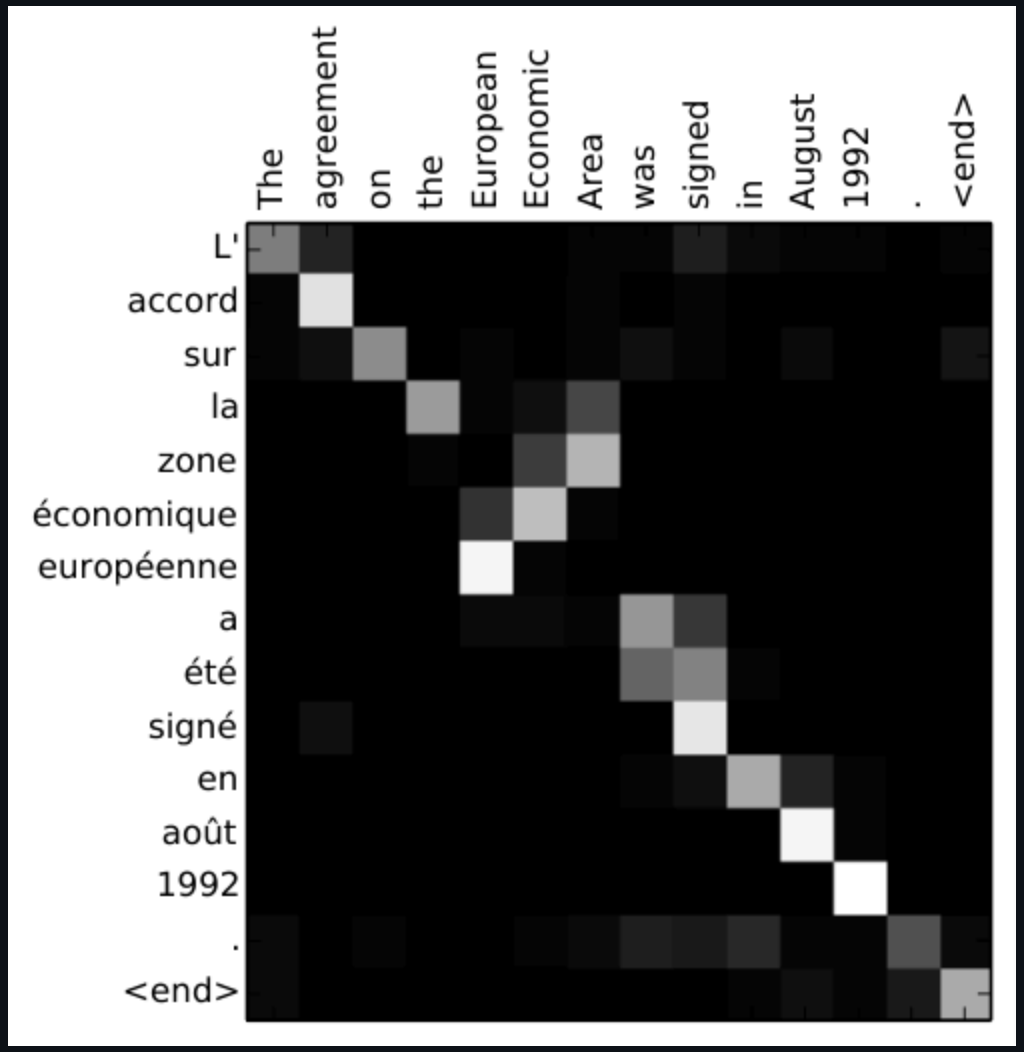
- **Word reordering:** "black cat" $\rightarrow$ "chat noir".
- Generating $\mathbf{y}_3 = \text"noir"$ needs $\mathbf{h}_2$ ("black")!
- How does decoder get $\mathbf{c}_t$? $\implies$ **Attention**

$$\mathbf{c}_t = \sum_{j=1}^{T} \alpha_{tj} \mathbf{h}_j$$

$$\alpha_{tj} = \text{softmax}(\text{score}(\mathbf{h}_j, \mathbf{s}_{t-1}))$$

$$\text{score}(\mathbf{h}_j, \mathbf{s}_{t-1}) = \mathbf{v}^T \tanh(\mathbf{W}_{attn}(\mathbf{h}_j \| \mathbf{s}_{t-1}))$$





Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." arXiv preprint arXiv:1409.0473 (2014).

Transformers: "Attention Is All You Need"

In 2017, Vaswani et al. (Google Brain / Google Research) posed a radical question:

> If attention provides dynamic information routing, **why keep the recurrent sequential layers at all?**

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

The model is like a database:

- **Query** — What am I looking for?
- **Key** — What information do I have?
- **Value** — What information do I want to retrieve?

Where does Q, K, V come from?

We learn them of course... Given token representations $\mathbf{X} \in \mathbb{R}^{T \times d}$:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V$$

where $\mathbf{W}_Q, \mathbf{W}_K \in \mathbb{R}^{d \times d_k}$ and $\mathbf{W}_V \in \mathbb{R}^{d \times d_v}$ learnable weights.

Where does $\mathbf{X}$ come from?

We use **learned token embeddings**: $\mathbf{x}_i \to \mathbf{E}_i \in \mathbb{R}^d$.

And **positional encodings**: $\mathbf{PE} \in \mathbb{R}^{T \times d}$:

$$\mathbf{X} = \mathbf{E} + \mathbf{PE}$$

> The deep dive is left for the NLP course!

Example

Input sequence: $\mathbf{x}_1 = \text{”river”}, \quad \mathbf{x}_2 = \text{”bank”}.$

- **Step 1 (Projections):** Compute $(\mathbf{q}_1, \mathbf{k}_1, \mathbf{v}_1)$ and $(\mathbf{q}_2, \mathbf{k}_2, \mathbf{v}_2)$.
- **Step 2 (Querying):** Word $\mathbf{x}_2$ ("bank") asks *"What kind of bank am I?"* via query $\mathbf{q}_2$:

$$e_{21} = \mathbf{q}_2 \cdot \mathbf{k}_1 = 6.0 \quad (\text{”river”})$$
$$e_{22} = \mathbf{q}_2 \cdot \mathbf{k}_2 = 1.0 \quad (\text{”bank”})$$

- **Step 3 (Softmax):** Scaled attention weights:

$$\alpha_{21} = 0.90 \quad (\text{”river”}), \quad \alpha_{22} = 0.10 \quad (\text{”bank”})$$

- **Step 4 (Value Mixing):**

$$\mathbf{z}_2 = 0.90\,\mathbf{v}_1 + 0.10\,\mathbf{v}_2$$

$\mathbf{z}_2$ now represents **river bank** rather than a financial bank!

Why Scale by $\frac{1}{\sqrt{d_k}}$?

Assume $\mathbb{E}[q_i] = \mathbb{E}[k_i] = 0$ and $\mathrm{Var}(q_i) = \mathrm{Var}(k_i) = 1$ for vectors $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$:

$$\mathbb{E}[\mathbf{q} \cdot \mathbf{k}] = 0, \quad \mathrm{Var}(\mathbf{q} \cdot \mathbf{k}) = \sum_{i=1}^{d_k} \mathrm{Var}(q_i)\mathrm{Var}(k_i) = d_k$$

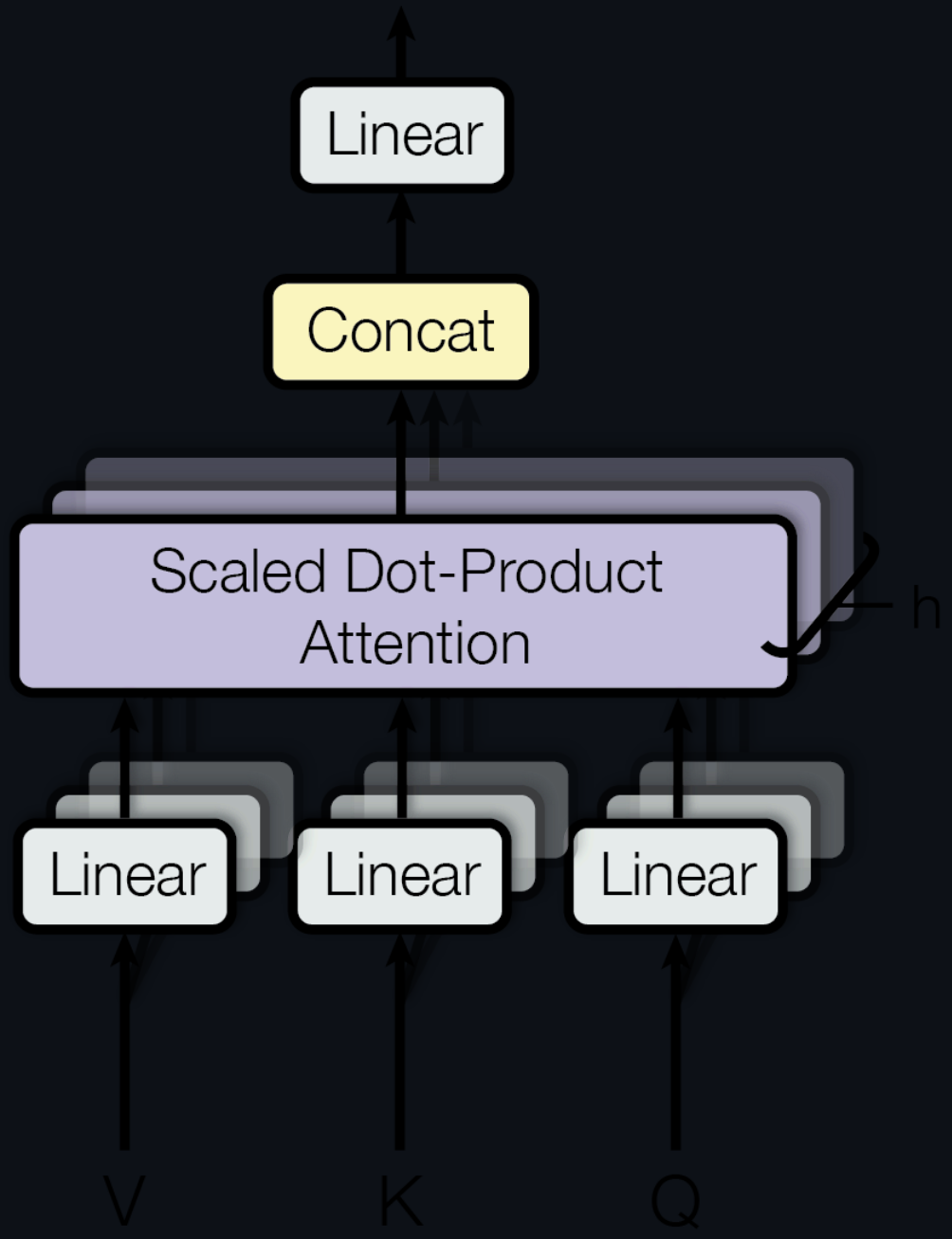
Dividing by $\sqrt{d_k}$ preserves unit variance and healthy gradient flow. $Var(aX) = a^2 Var(X)$

Multi-Head Self-Attention

Rather than a single attention map, project into h distinct representation subspaces in parallel:

$$\mathrm{MHA}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h)\mathbf{W}^O$$

Allows the model to attend simultaneously to syntactic relations, coreferences, and semantic attributes!


Linear
Concat
Scaled Dot-Product Attention
h
Linear
Linear
Linear
V
K
Q

Scaled Dot-Product Attention

1. Linear Projections

Input Tokens X (T × d)

Q — Query — X · W_Q

K — Key — X · W_K

V — Value — X · W_V

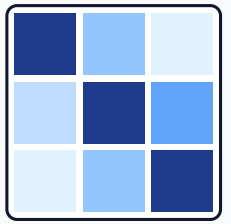
Content-addressable dynamic routing

2. Scaled Dot-Product & Softmax

MatMul: Q · K^T → T × T

Scale: ÷ √d_k (Stabilises variance)

Softmax(·) → Attention Matrix A



Weights A (T×T)

$$A_{i,j} = \exp(q_i k_j / \sqrt{d_k}) / \sum_l \exp(q_i k_l / \sqrt{d_k})$$

3. Weighted Values

Output = A · V

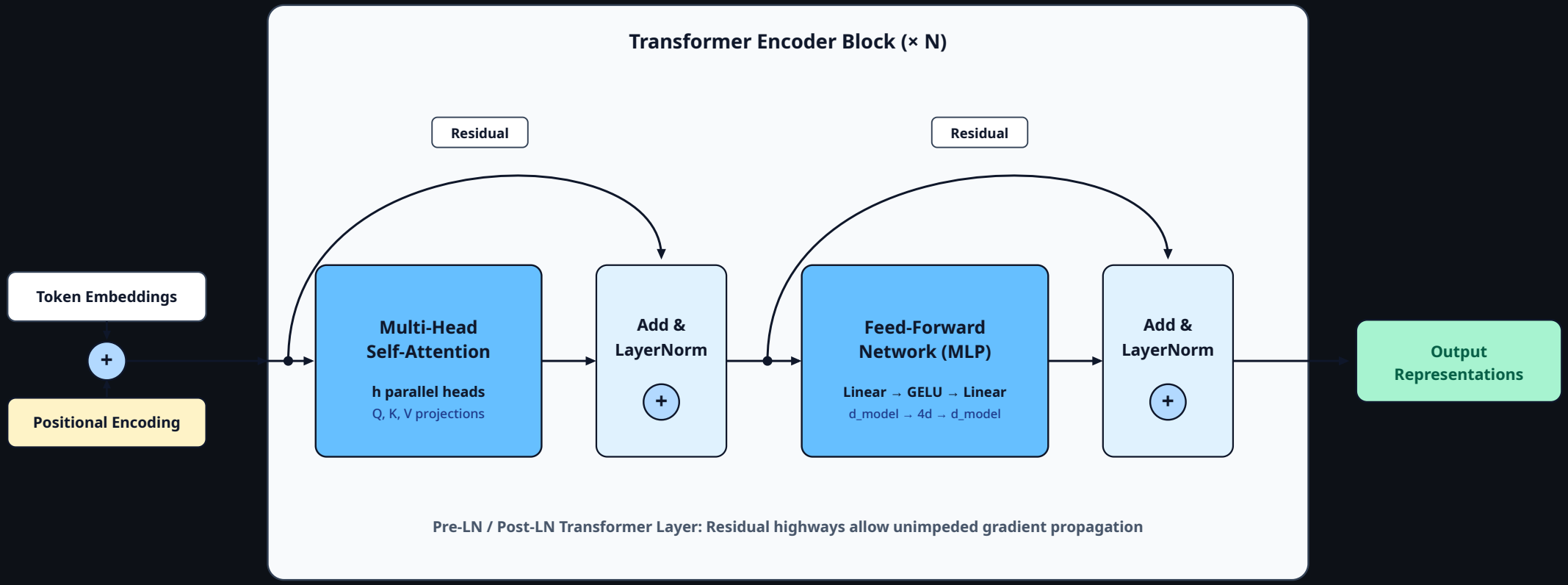
Contextual Embeddings

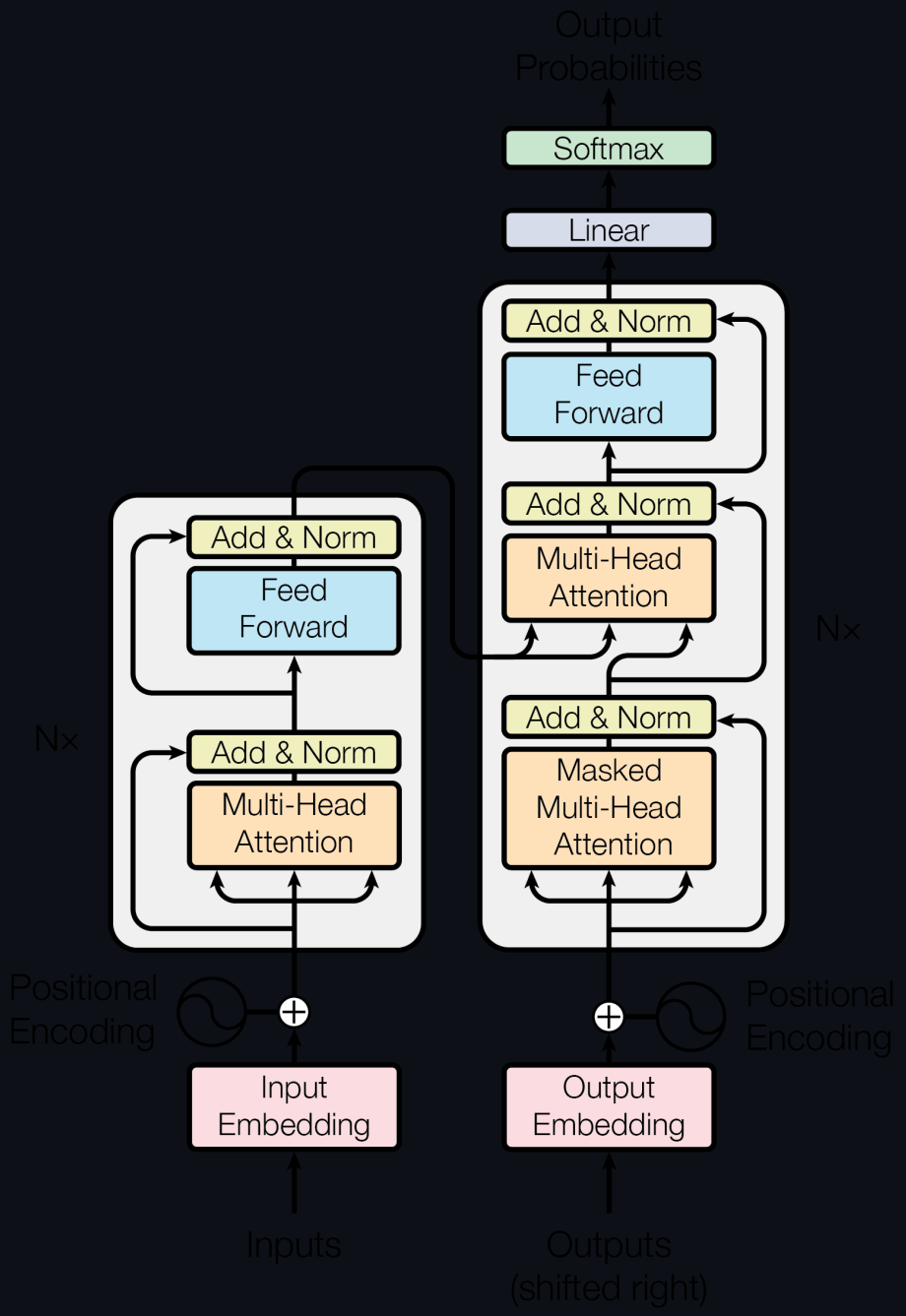
Each token updates its meaning based on all other tokens in parallel!

$$\text{Attention}(Q, K, V) = \text{softmax}(Q \cdot K^T / \sqrt{d_k}) \cdot V$$

Input-Input Layer5

The Complete Transformer Block




Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Chapter: Overview

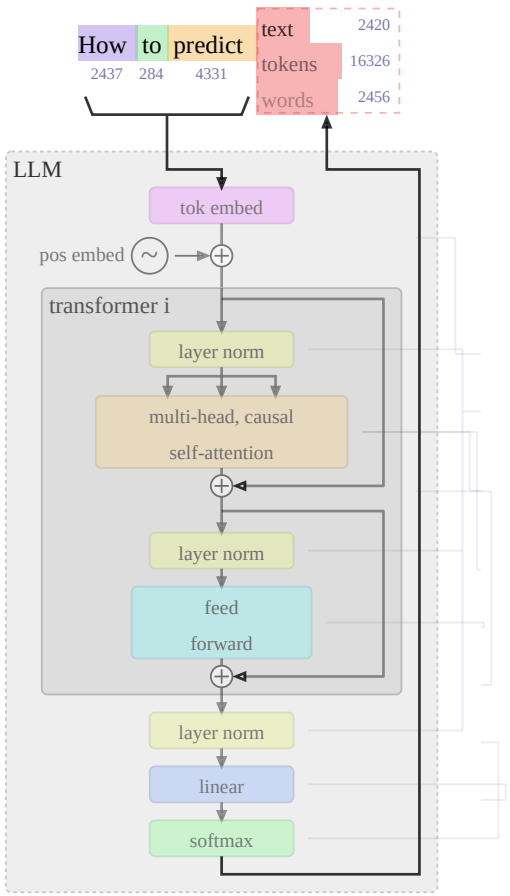


Table of Contents

Intro

Introduction

Preliminaries

Components

Embedding

Layer Norm

Self Attention

Projection

MLP

Transformer

Softmax

Output

GPT-2 (small) | nano-gpt | GPT-2 (XL) | GPT-3

Continue | Skip

Recurrent vs. Transformer Inductive Biases

Dimension	Recurrent Neural Network	Transformer (Self-Attention)
Computational Dependency	Strictly sequential: must compute $\mathbf{h}_{t-1}$ before $\mathbf{h}_t$	Fully parallel: all tokens processed simultaneously
Training Complexity	$O(T)$ sequential steps (cannot saturate GPU)	$O(1)$ **sequential operations** (pure GEMM matrix products!)
Maximum Path Length	$O(T)$ forward and backward steps across time	$O(1)$ **direct connection** between any two arbitrary tokens
Memory / Compute Scaling	$O(T \cdot d^2)$ computation, $O(T)$ memory	$O(T^2 \cdot d)$ pairwise attention matrix
Positional Awareness	Implicit in sequential transition order	Permutation equivariant! Needs **Positional Encodings** (RoPE / Sinusoidal)

Show me the code

```
class ResidualMLPBlock(nn.Module):
    def __init__(self, dim: int):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.linear1 = nn.Linear(dim, dim * 4)
        self.act = nn.GELU()
        self.linear2 = nn.Linear(dim * 4, dim)
        self.dropout = nn.Dropout(0.1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Pre-LN Residual Highway: x + F(Norm(x))
        residual = x
        out = self.linear2(self.dropout(self.act(self.linear1(self.norm(x)))))
        return residual + out
```

Training loop

```
import torch.optim as optim

model = ResidualMLPBlock(dim=256)
# Decoupled AdamW optimiser with cosine learning rate schedule
optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-2)
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=100)
criterion = nn.MSELoss()

for epoch in range(100):
    optimizer.zero_grad(set_to_none=True)  # Faster than setting to 0

    inputs = torch.randn(64, 256)
    targets = torch.randn(64, 256)

    predictions = model(inputs)
    loss = criterion(predictions, targets)

    loss.backward()                          # Reverse-mode autodiff (Backpropagation)
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0) # Gradient clipping

    optimizer.step()                         # AdamW parameter update
    scheduler.step()                         # Learning rate decay
```

Where is my artificial brain?

I think we started really well this time, more neurons, layers etc. but went all mechanical, fixing things until they work...

AI $\rightarrow$ ML $\rightarrow$ Numbers game

Next Part: What happens when we abandon continuous differentiable hyperplanes entirely for orthogonal, rule-based space partitioning? Welcome to *Decision Trees, Ensembles, and Collective Intelligence*!