

Neural Networks

From one neuron to deep networks

Mikhail Mikhasenko · Lecture 3A

Where we are

The recipe from Lecture 1B

1. A **model** $f(x; \theta)$ with parameters θ
2. A **loss** $L(\theta)$ that measures mistakes on training data
3. An **optimizer** that lowers $L(\theta)$

Trees (2A, 2B): piecewise-constant models grown by greedy cuts.

Today

Neural networks are smooth functions with many parameters, fitted by following the gradient of the loss.

- What does a network compute?
- Why do we need hidden layers and nonlinearities?
- How do we compute gradients efficiently, and which way do we step?

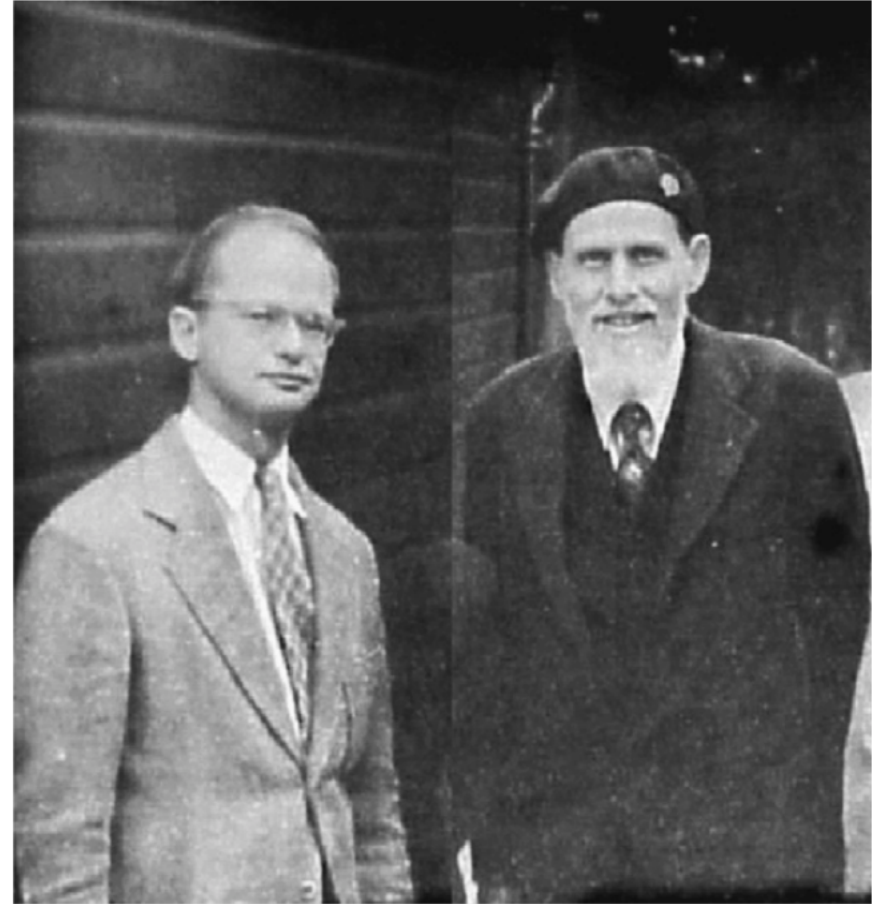
Origins

1943 McCulloch & Pitts: a binary-threshold **neuron model**

1949 Hebb: connections that fire together strengthen

1958 Rosenblatt: the **perceptron** learns weights from labelled examples (Mark I Perceptron: 20×20 photocell images)

1969 Minsky & Papert, *Perceptrons*: the limits of one layer



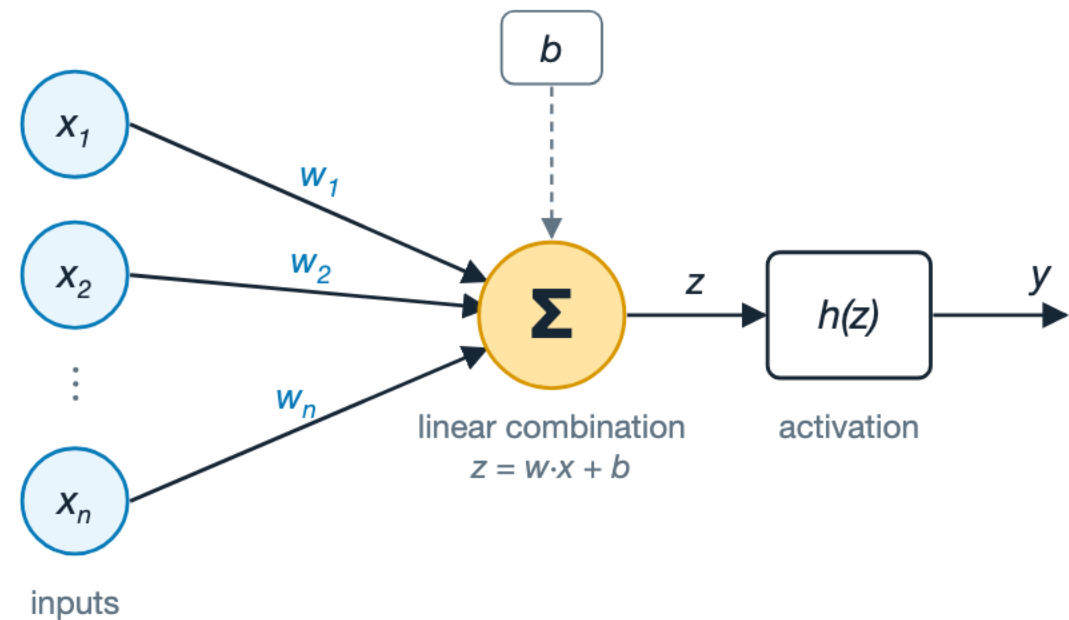
Warren McCulloch and Walter Pitts, Chicago 1949

The artificial neuron

$$y = h\left(\sum_{i=1}^n w_i x_i + b\right) = h(\mathbf{w} \cdot \mathbf{x} + b)$$

- **weights** w_i : how strongly each input counts
- **bias** b : shifts the threshold
- **activation** h : a fixed nonlinear function

An affine map followed by a nonlinearity.
The perceptron uses a step: $h(z) = 1$ if $z \geq 0$, else 0.



One neuron, one line

Geometry

The boundary $\mathbf{w} \cdot \mathbf{x} + b = 0$ is a straight line (a hyperplane in n dimensions).

- $\mathbf{w}$ is **normal** to the line and points towards $z > 0$.
- b moves the line: its distance from the origin is $|b|/\|\mathbf{w}\|$.
- For a sigmoid, larger $\|\mathbf{w}\|$ makes the transition sharper.

Perceptron learning rule

For each example, with prediction $\hat{y} \in \{0, 1\}$:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta (y - \hat{y}) \mathbf{x}, \quad b \leftarrow b + \eta (y - \hat{y})$$

Correct predictions change nothing; a mistake tilts the line towards the misclassified point.

If a separating line **exists**, the rule finds one in a finite number of steps.

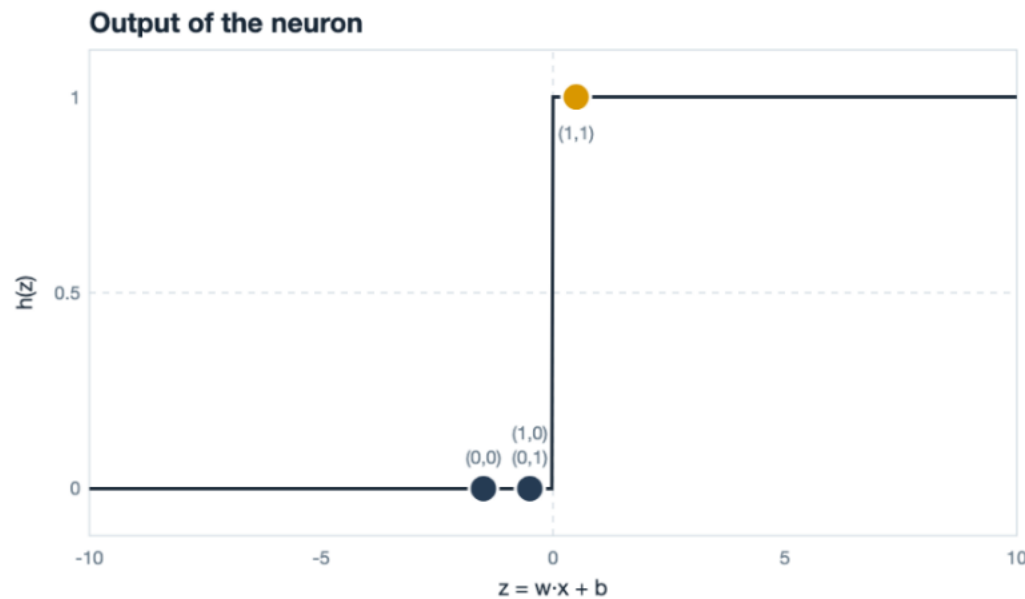
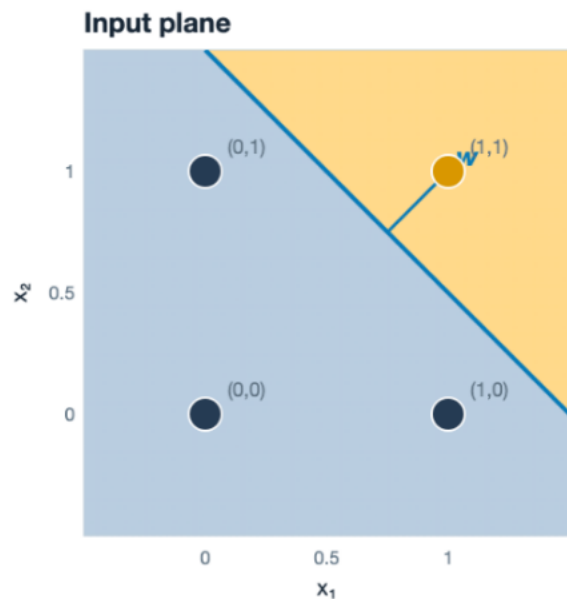
One neuron draws one line

One neuron draws one line

● target 1 ● target 0 ○ misclassified

The neuron computes $z = w_1x_1 + w_2x_2 + b$ and outputs $h(z)$. The boundary $z = 0$ is a straight line with normal vector $\mathbf{w}$.

Task $h(z)$ w_1 w_2 b



Correct inputs z for (0,0) (0,1) (1,0) (1,1) Training epochs
4 / 4 **-1.5 -0.5 -0.5 0.5** **0**

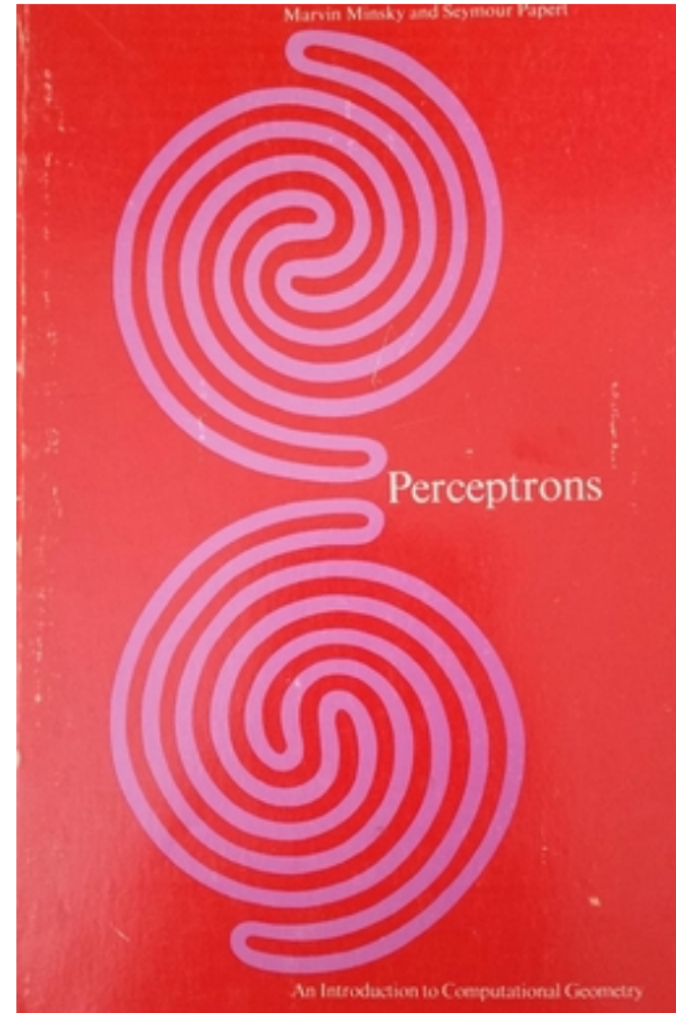
Find weights for AND and OR by hand. Then try XOR, and let the training rule try too.

The XOR problem and the first AI winter

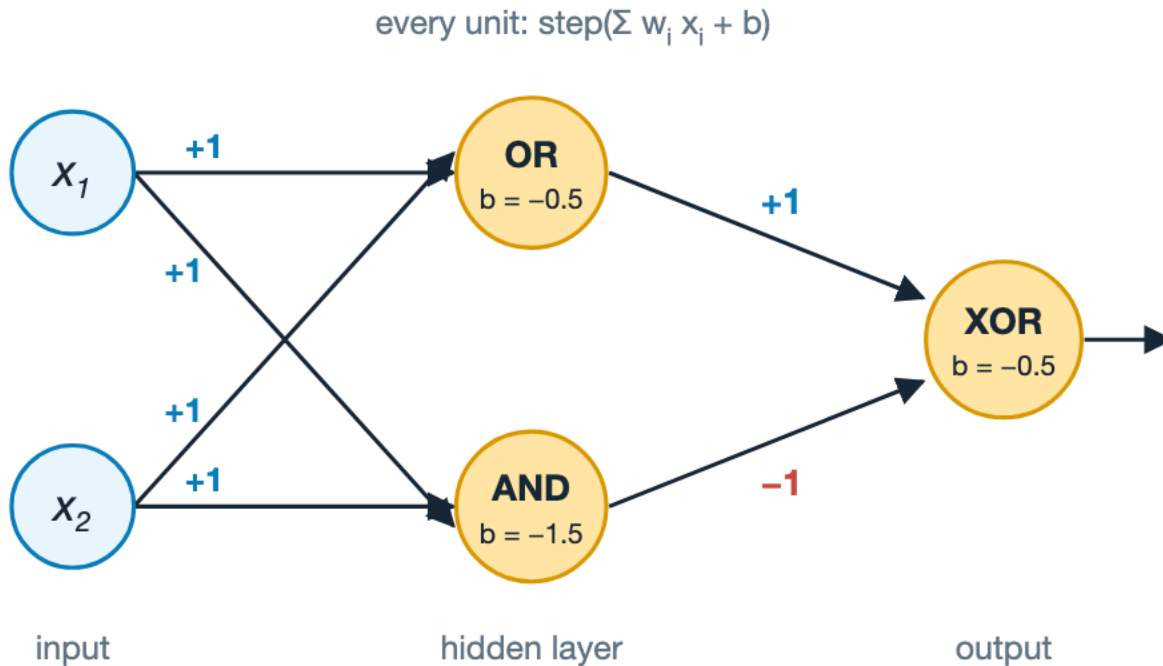
1969: Minsky and Papert prove that a single-layer perceptron cannot compute some simple functions, for example XOR.

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

No single line separates the ones from the zeros. Funding and interest in neural networks declined through the 1970s.



One hidden layer solves XOR



x_1	x_2	OR	AND	OR - AND
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

The hidden units compute **new features**. In the (OR, AND) plane, XOR is separable by one line.

Multilayer perceptron

Stack layers of neurons. With $\mathbf{a}^{(0)} = \mathbf{x}$:

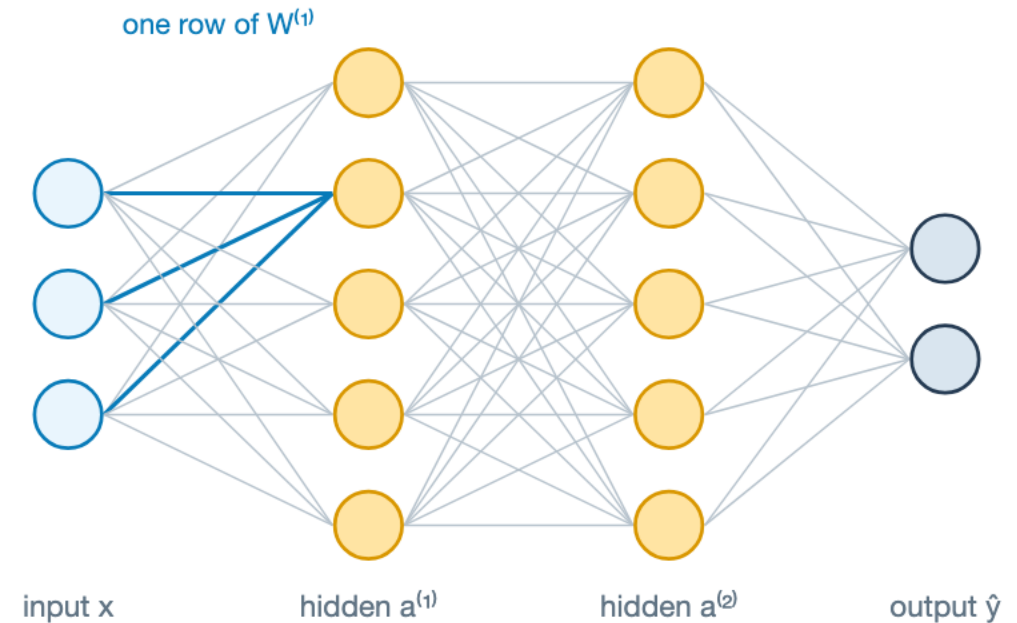
$$\mathbf{a}^{(l)} = h\left(W^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right), \quad l = 1, \dots, L$$

$W^{(l)}$ has shape $n_l \times n_{l-1}$: each **row** holds the weights of one unit.

Dense (fully connected): every unit sees every unit of the previous layer.

Example, MNIST 784–800–10:

$784 \cdot 800 + 800 + 800 \cdot 10 + 10 = 636\,010$
parameters.



Why the nonlinearity?

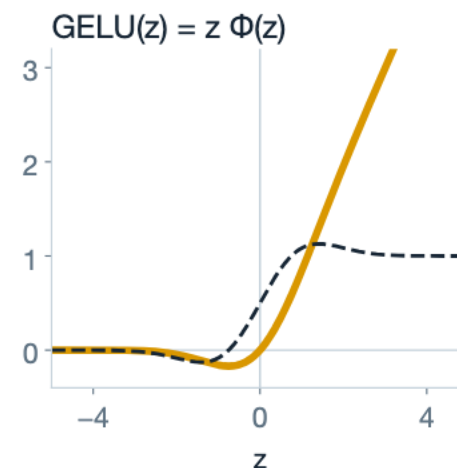
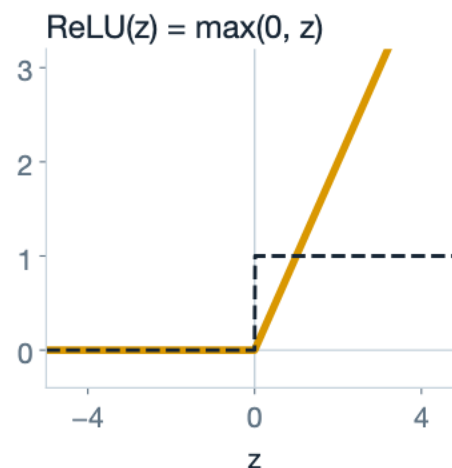
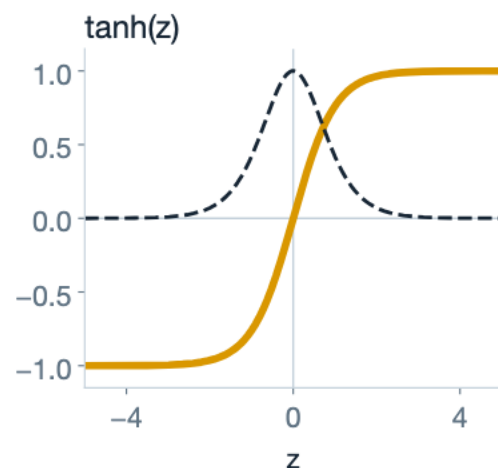
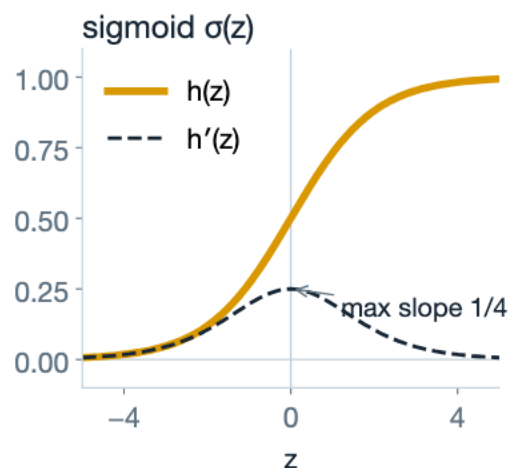
Without h , two layers collapse into one:

$$W^{(2)} \left(W^{(1)} \mathbf{x} + \mathbf{b}^{(1)} \right) + \mathbf{b}^{(2)} = W^{(2)} \underbrace{W^{(1)}}_{\mathbf{W}} \mathbf{x} + \underbrace{W^{(2)} \mathbf{b}^{(1)} + \mathbf{b}^{(2)}}_{\mathbf{b}}$$

Any depth of linear layers is still **one linear map**: one line, no XOR.

The activation bends the space between layers. Composing many simple bends builds complicated functions.

Activation functions



Sigmoid, tanh saturate: $\sigma'(z) \leq \frac{1}{4}$.
Ten layers can shrink a gradient by $4^{-10} \approx 10^{-6}$ (**vanishing gradient**). Still used for outputs and gates.

ReLU $\max(0, z)$: cheap, slope 1 for $z > 0$. Units stuck at $z < 0$ stop learning (“dying ReLU”); leaky ReLU keeps a small slope.

GELU $z \Phi(z)$, Φ = Gaussian CDF: a smooth ReLU, standard in transformers (BERT, GPT-2).

Universal approximation

1989, Cybenko: one hidden layer is enough. For a sigmoidal σ , any continuous f on a bounded domain and any $\varepsilon > 0$ there exist N and parameters with

$$g(\mathbf{x}) = \sum_{j=1}^N v_j \sigma(\mathbf{w}_j^\top \mathbf{x} + b_j), \quad |g(\mathbf{x}) - f(\mathbf{x})| < \varepsilon.$$

Hornik (1991), Leshno et al. (1993): any non-polynomial activation works, including ReLU.

What it does not say

- how large N must be (it can grow exponentially with dimension)
- how to **find** the weights
- how well g generalizes beyond the training data

In practice, **deep** networks reach the same accuracy with far fewer units than one wide layer.

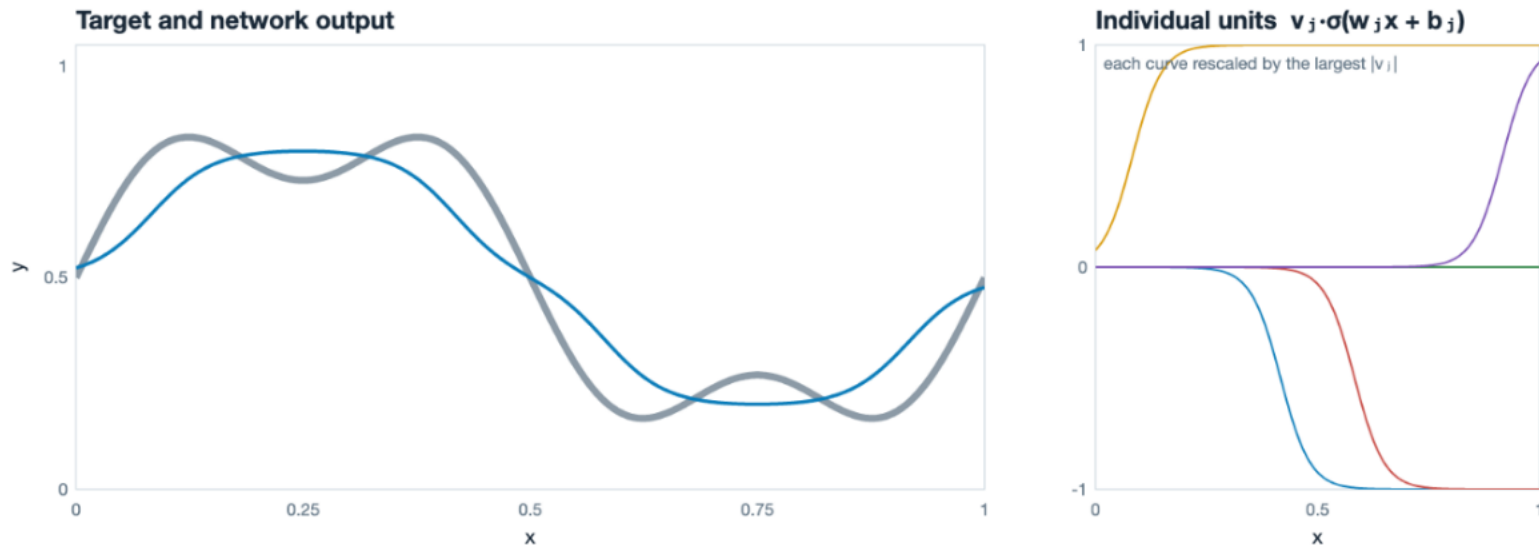
Universal approximation: the visual proof

Universal approximation, one unit at a time

— target — network $g(x)$

One hidden layer: $g(x) = c_0 + \sum_j v_j h(w_j x + b_j)$. Here the weights are **constructed**, not trained: each unit adds one step (sigmoid) or one kink (ReLU).

Target smooth wave h sigmoid Hidden units 6 Steepness 20 ☒ show units



Hidden units N	Parameters $3N + 1$	Max $ g - f $	RMS error
6	19	0.147	0.0887

Increase the number of units and watch the error fall. The theorem says such weights **exist**; it says nothing about how many units you need or whether training finds them.

Output layers and losses

Task	Output unit	Loss
Regression	identity: $\hat{y} = z$	squared error $\frac{1}{2}(\hat{y} - y)^2$
Binary classification	sigmoid $\hat{p} = \sigma(z)$	cross-entropy $-y \log \hat{p} - (1 - y) \log (1 - \hat{p})$
K classes	softmax $p_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$	cross-entropy $-\log p_y$
Positive quantity	softplus $\log (1 + e^z)$	e.g. Gaussian negative log-likelihood

For sigmoid or softmax with cross-entropy, the gradient at the output is simply $\partial L / \partial z_k = p_k - \mathbb{1}[k = y]$.

Gradient descent

The loss averages over N training examples:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^N \ell(f(x_i; \theta), y_i)$$

Step against the gradient:

$$\theta_{n+1} = \theta_n - \eta \nabla_{\theta} L(\theta_n)$$

η is the **learning rate**.

Like walking downhill in the dark with a flashlight:
you only see the slope at your feet.

Near a minimum with curvature λ the error is
multiplied by $1 - \eta\lambda$ at every step:

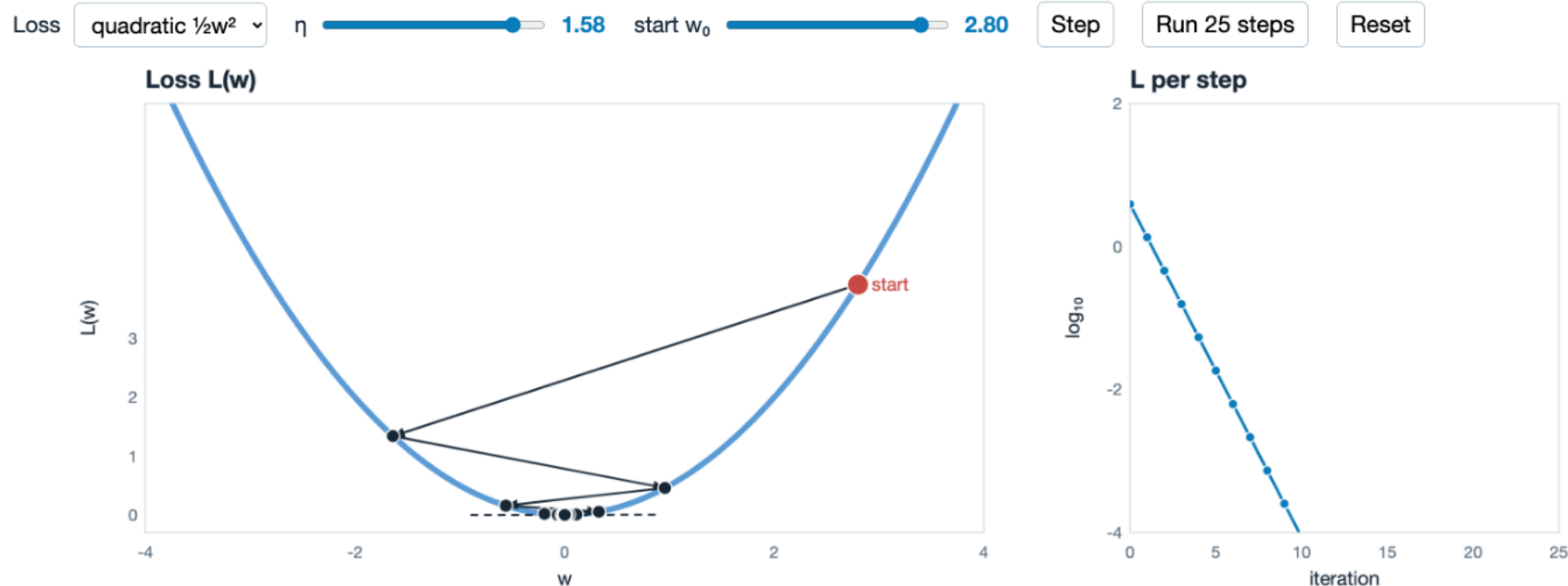
- $\eta\lambda < 1$: smooth approach
- $1 < \eta\lambda < 2$: zig-zag, still converges
- $\eta\lambda > 2$: diverges

Choosing the learning rate

Choosing the learning rate

update: $w \leftarrow w - \eta L'(w)$

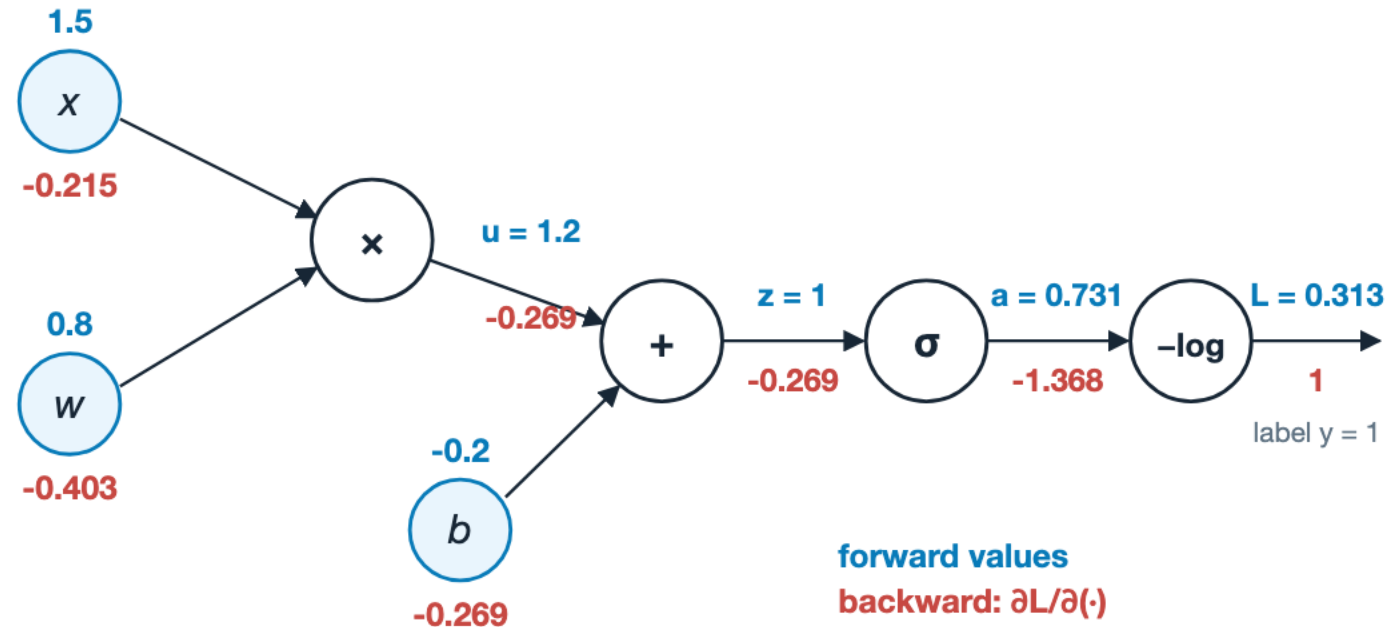
Gradient descent only sees the local slope. The learning rate η decides how far to trust it.



Steps	Current w	$L(w)$	$\eta \cdot L''(w^*)$ near this minimum
14	0.002	0.0000	1.58 (zig-zag, converges)

For $\frac{1}{2}w^2$ the curvature is 1: try $\eta = 0.3, 1.5$ and 2.2 . With two minima, change the start point.

Backpropagation: the chain rule on a graph



$$\frac{\partial L}{\partial w} = \underbrace{-\frac{1}{a}}_{\partial L / \partial a} \cdot \underbrace{a(1-a)}_{\partial a / \partial z} \cdot \underbrace{1}_{\partial z / \partial u} \cdot \underbrace{x}_{\partial u / \partial w} = (a - y) x = -0.403$$

Forward: compute and store every intermediate value. **Backward:** multiply local derivatives from the loss towards the inputs.

Backpropagation through layers

Define the error signal $\delta^{(l)} = \partial L / \partial \mathbf{z}^{(l)}$, where $\mathbf{z}^{(l)} = W^{(l)} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$:

$$\delta^{(L)} = \hat{\mathbf{p}} - \mathbf{y}, \quad \delta^{(l)} = \left(W^{(l+1)\top} \delta^{(l+1)} \right) \odot h'(\mathbf{z}^{(l)})$$

$$\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)} \mathbf{a}^{(l-1)\top}, \quad \frac{\partial L}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}$$

One backward pass costs about as much as one forward pass, whether the network has ten parameters or ten billion.

1970 Linnainmaa: reverse-mode automatic differentiation

1974 Werbos: applied to neural networks

1986 Rumelhart, Hinton & Williams: rediscovered and popularized



2024 Nobel Prize in Physics,
Hopfield & Hinton: “for foundational discoveries and inventions that enable machine learning with artificial neural networks”

Mini-batches: stochastic gradient descent

With $N = 10^6$ examples, the exact gradient is expensive. Estimate it on a random **batch** B of m examples ($m \approx 32\text{--}512$):

$$\mathbf{g} = \frac{1}{m} \sum_{i \in B} \nabla_{\theta} \ell_i, \quad \mathbb{E}[\mathbf{g}] = \nabla_{\theta} L$$

The loss being minimized changes a little every step. One pass over all data is an **epoch**.

With a batch dimension the input is a **matrix** X of shape $m \times d$:

$$Z^{(1)} = XW^{(1)\top} + \mathbf{1}\mathbf{b}^{(1)\top}$$

Every layer is one matrix multiplication, which GPUs execute very fast.

Gradient noise shrinks like $1/\sqrt{m}$, and a little noise can help escape saddle points and sharp minima.

Momentum

Keep the inertia of the motion:

$$\mathbf{v}_n = \beta \mathbf{v}_{n-1} + \mathbf{g}_n, \quad \theta_{n+1} = \theta_n - \eta \mathbf{v}_n$$

Unrolled, $\mathbf{v}_n = \sum_{k \geq 0} \beta^k \mathbf{g}_{n-k}$: an **exponentially decaying** memory of past gradients, about $1/(1 - \beta) = 10$ steps for $\beta = 0.9$.

- Consistent directions add up: up to $1/(1 - \beta)$ times faster.
- Oscillating directions cancel: less zig-zag.
- Speed can carry the parameters through small bumps and plateaus.

Polyak's “heavy ball” (1964).

Adaptive step sizes: AdaGrad and RMSProp

Give every parameter its own step size. Per coordinate:

$$s_n = \begin{cases} s_{n-1} + g_n^2 & \text{AdaGrad} \\ \beta_2 s_{n-1} + (1 - \beta_2) g_n^2 & \text{RMSProp} \end{cases}$$

$$\theta_{n+1} = \theta_n - \eta \frac{g_n}{\sqrt{s_n} + \epsilon}$$

- Steep directions get small steps; flat ones get relatively large steps.
- Rarely updated parameters (sparse features) catch up instead of being left behind.
- AdaGrad's s_n only grows, so its steps shrink towards zero. RMSProp forgets old gradients.

Adam: adaptive moment estimation

$$\begin{aligned}\mathbf{m}_n &= \beta_1 \mathbf{m}_{n-1} + (1 - \beta_1) \mathbf{g}_n, & \mathbf{s}_n &= \beta_2 \mathbf{s}_{n-1} + (1 - \beta_2) \mathbf{g}_n^2 \\ \widehat{\mathbf{m}}_n &= \frac{\mathbf{m}_n}{1 - \beta_1^n}, & \widehat{\mathbf{s}}_n &= \frac{\mathbf{s}_n}{1 - \beta_2^n}, & \theta_{n+1} &= \theta_n - \eta \frac{\widehat{\mathbf{m}}_n}{\sqrt{\widehat{\mathbf{s}}_n} + \varepsilon}\end{aligned}$$

Defaults: $\eta = 10^{-3}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$.

Momentum (first moment $\mathbf{m}$):
speed along consistent directions

RMSProp (second moment $\mathbf{s}$):
per-parameter step sizes

Bias correction: $\mathbf{m}$ and $\mathbf{s}$ start at zero

The default choice for most networks today, often as AdamW with decoupled weight decay.

Optimizers on a loss surface

Optimizers on a loss surface

☒ GD ☒ Momentum ☒ RMSProp ☒ Adam

Click on the map to choose a starting point. All optimizers start together and see the same gradients.

Surface narrow valley

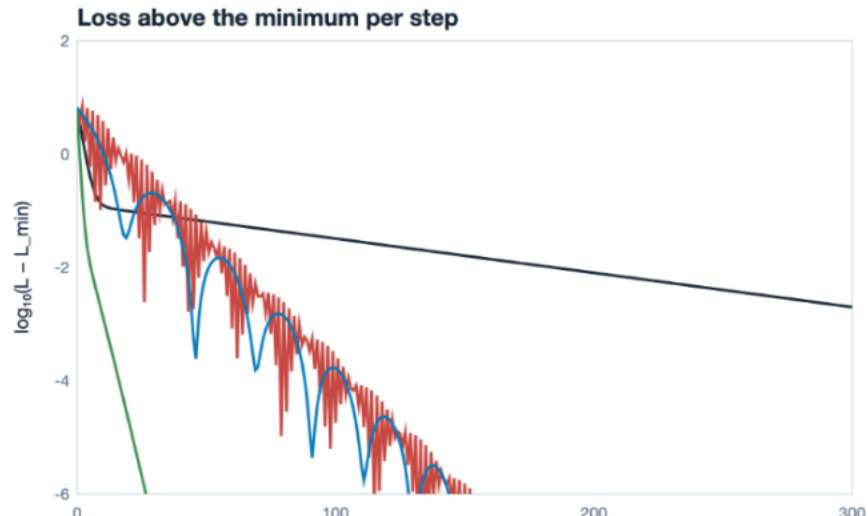
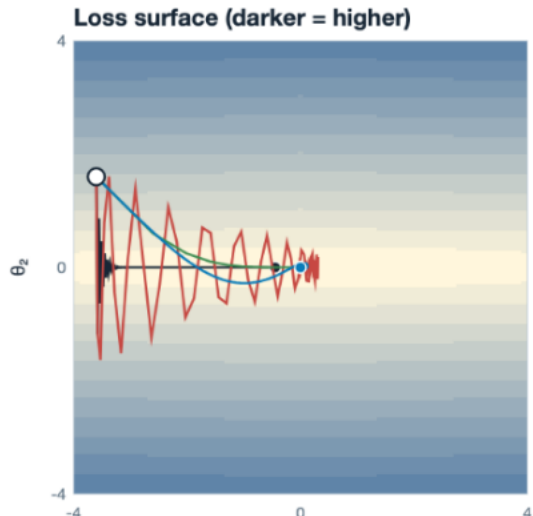
η (GD, momentum) 0.347

α (RMSProp, Adam) 0.100

gradient noise 0.00

Run

Reset



Step	GD	Momentum	RMSProp	Adam
300	2.0e-3	1.7e-13	8.8e-156	1.5e-13

Narrow valley: plain GD zig-zags across and crawls along. Momentum builds speed along the valley; RMSProp and Adam rescale each coordinate.

Initialization

Break the symmetry

If all weights start equal, all units in a layer compute the same function and receive the same gradient, forever. Start with **b** = 0 and **random** *W*.

Choose the scale

For $z = \sum_{i=1}^{n_{\text{in}}} w_i a_i$ with independent terms,

$\text{Var } z = n_{\text{in}} \text{Var } w \text{Var } a$. Keep the variance of signals and gradients constant from layer to layer.

Tanh, sigmoid: Glorot (Xavier), 2010

$$\text{Var } w = \frac{2}{n_{\text{in}} + n_{\text{out}}}, \quad w \sim U\left(\pm \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

ReLU: He (Kaiming), 2015

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

ReLU zeroes half the signal, hence the factor 2.

Overfitting and regularization

A large network can memorize the training data, noise included (bias and variance, Lecture 2A).

- **Validation set and early stopping:** stop when the validation loss rises.
- **L2 / weight decay:** minimize $L + \frac{\lambda}{2}\|\theta\|^2$; penalizes large weights (1990s).
- **More data, data augmentation:** shifted, flipped, cropped copies (Lecture 3B).
- **Dropout** (2012): during training, set each hidden unit to zero with probability p . Like averaging many thinned networks.
- **Batch normalization** (2015): normalize each unit over the batch, then learn a scale and a shift. Mainly stabilizes and speeds up training.
- **Ensembles:** average independently trained networks.

Train a small network

Train a small network

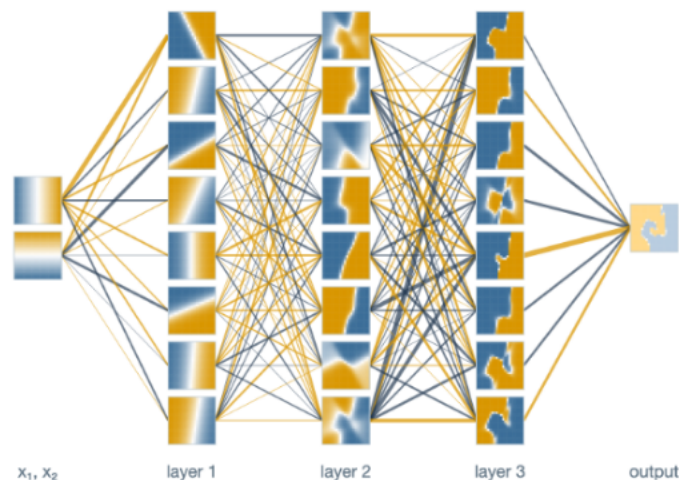
● class 1 ● class 0 weights: positive / negative

Every square shows what one unit computes over the whole input plane. Later units combine the patterns of earlier ones.

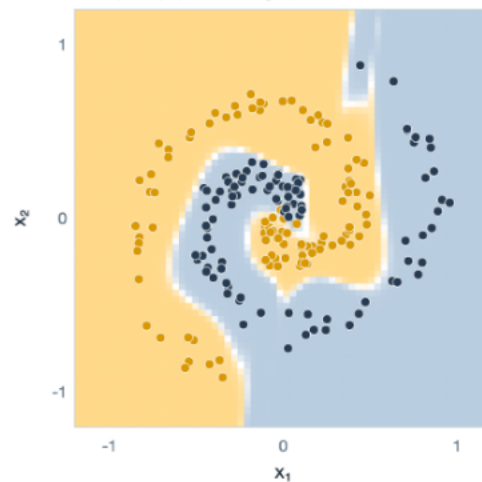
Data spiral Hidden layers 3 Units 8 h tanh lr 0.01 L2 0 Noise 0.10

☐ test points Train Reset

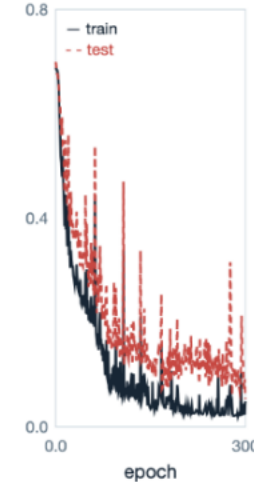
Network



Output $p(\text{class 1} | \mathbf{x})$



Loss



Epoch	Parameters	Train loss	Test loss	Train accuracy	Test accuracy
300	177	0.049	0.053	99%	98%

Try the spiral with 1 layer of 2 units, then 3 layers of 8. Switch to the linear activation: the depth no longer helps.

The deep-learning renaissance

2009 Training on GPUs

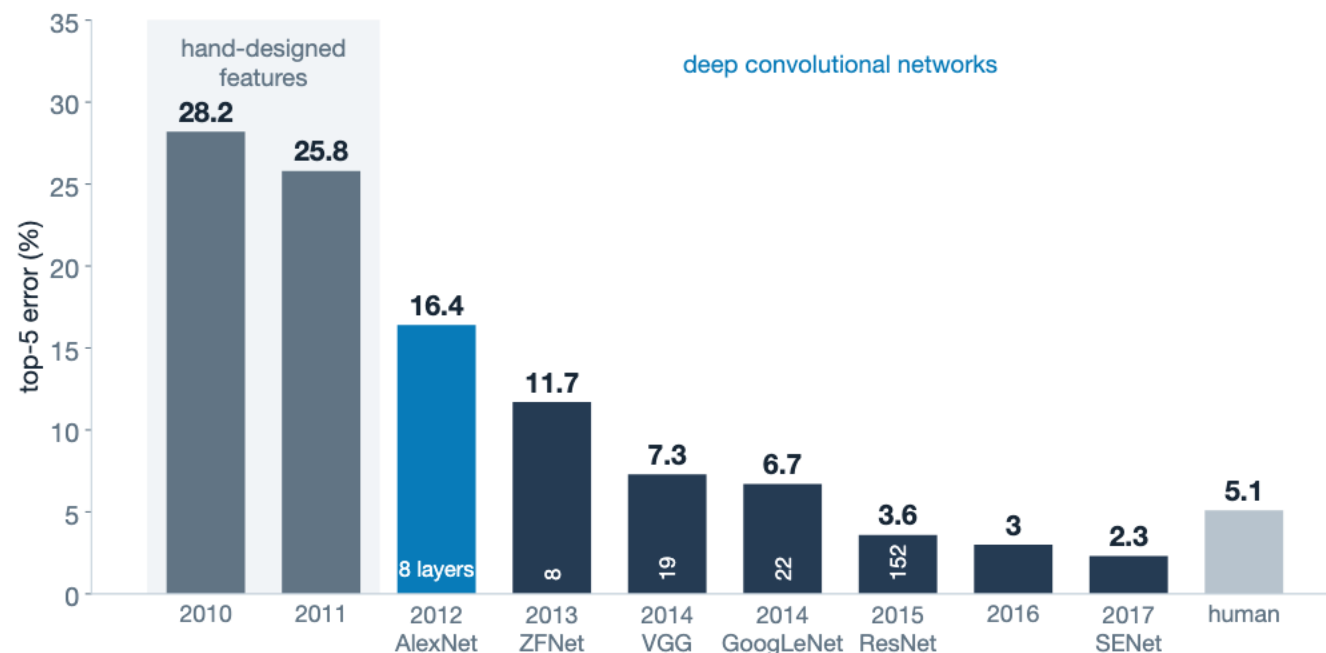
2010 Glorot & Bengio: why deep networks were hard to train, better initialization

2011–12 ReLU and dropout

2012 AlexNet wins ImageNet by a wide margin

2015 Deep networks dominate vision, speech and language benchmarks

Next lecture: [how convolutional networks see](#).



End

Thank you for the attention