Machine Learning and Optimization

Lecture 12: Neural Network Architecture and Expressivity

CSCI 4140/6960 — Spring 2026

Course website: `cs.rpi.edu/~gittea/.../mlandopt.html`

Learning Objectives

By the end of this lecture, you should be able to:

1. **Explain** how the RERM template carries over unchanged to neural networks, with only the predictor $f(x; \omega)$ changing
2. **Define** FCFFNs/MLPs, state the vector-form forward pass, and count parameters per layer
3. **Prove** that depth is useless without nonlinearity and state the Universal Approximation Theorem (with its limitations)
4. **Write** the RERM objective for an MLP and identify why it is nonconvex

Lecture Roadmap

Bridge from the First Half

What Is a Neural Network?

Expressivity: Why We Need Neural Networks

Vector Notation for MLPs

The RERM Problem for MLPs

Summary

Framing the Second Half

Everything we have done so far fits a common template:

1. Identify a model class motivated by problem structure (linear separability, noise model, etc.)
2. Frame learning as solving an optimization problem (RERM)
3. Analyze and apply optimization algorithms to solve it

Framing the Second Half

Everything we have done so far fits a common template:

1. Identify a model class motivated by problem structure (linear separability, noise model, etc.)
2. Frame learning as solving an optimization problem (RERM)
3. Analyze and apply optimization algorithms to solve it

The second half on neural networks follows the *same* template:

- The RERM framework carries over unchanged.
- Only the hypothesis class $f(x; \omega)$ changes.
- Optimization is harder; theory weaker; practice surprisingly strong.

Same Losses, Same Principles — New Predictor

The loss functions, noise models, and estimation principles from the first half carry over *unchanged*. Only the predictor $f(x; \omega)$ changes.

Setting	**First half (Convex)**	**Second half (NNs)**
Predictor	$\langle \omega, x \rangle + b$	$o^L(x)$
Gaussian $\to$ Regression	$(f - y)^2$	**Same**
Bernoulli $\to$ Binary	Cross-entropy	**Same**
Categorical $\to$ Multi	Softmax CE	**Same**
MAP $\to$ Penalty	$\|\omega\|_2^2$	**Same**

Same Losses, Same Principles — New Predictor

The loss functions, noise models, and estimation principles from the first half carry over *unchanged*. Only the predictor $f(x; \omega)$ changes.

Setting	**First half (Convex)**	**Second half (NNs)**
Predictor	$\langle \omega, x \rangle + b$	$o^L(x)$
Gaussian $\rightarrow$ Regression	$(f - y)^2$	**Same**
Bernoulli $\rightarrow$ Binary	Cross-entropy	**Same**
Categorical $\rightarrow$ Multi	Softmax CE	**Same**
MAP $\rightarrow$ Penalty	$\|\omega\|_2^2$	**Same**

The *only* difference: $f(x; \omega)$ is now a nonconvex function of ω, so the optimization problem is no longer convex. The entire *statistical* story is unchanged.

The Power of the RERM Template

The RERM template separates three independent design decisions:

Modeling: What is f?

Statistical: What is ℓ?

Computational: How to optimize?

Changing one component does not require rethinking the others.

The Power of the RERM Template

The RERM template separates three independent design decisions:

Modeling: What is f?

Statistical: What is ℓ?

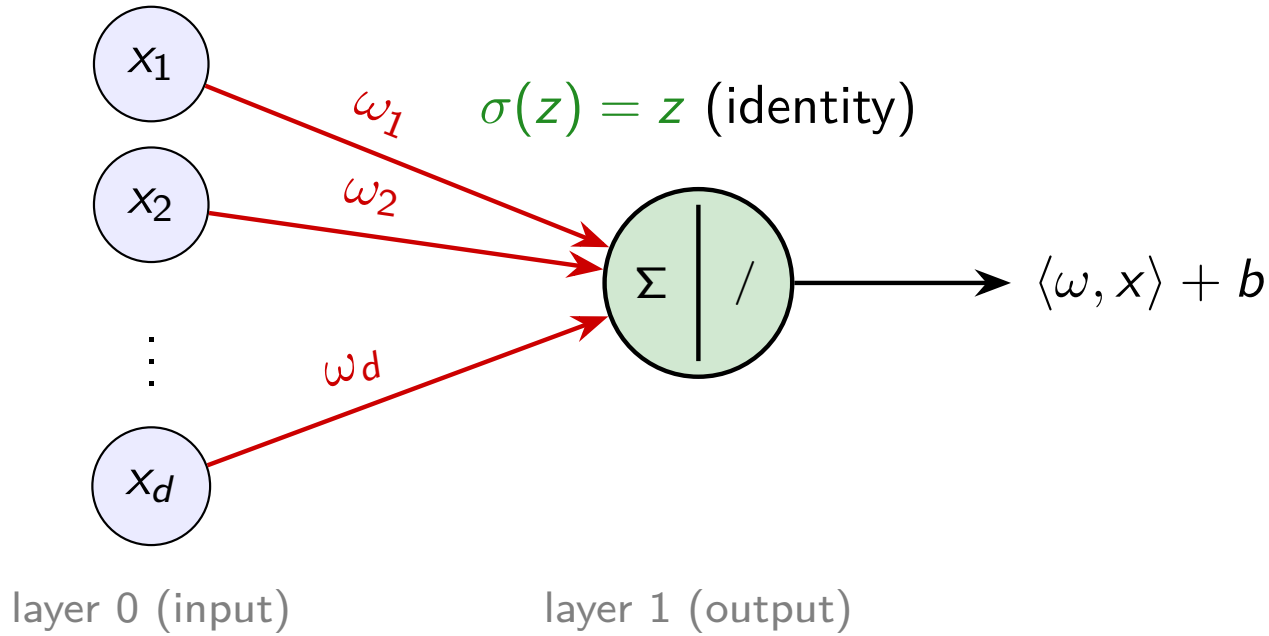
Computational: How to optimize?

Changing one component does not require rethinking the others.

Today we change only the **modeling** component:
$f(x;\omega) = \langle \omega, x \rangle + b$ becomes a neural network.

All Models Seen So Far Are Neural Networks

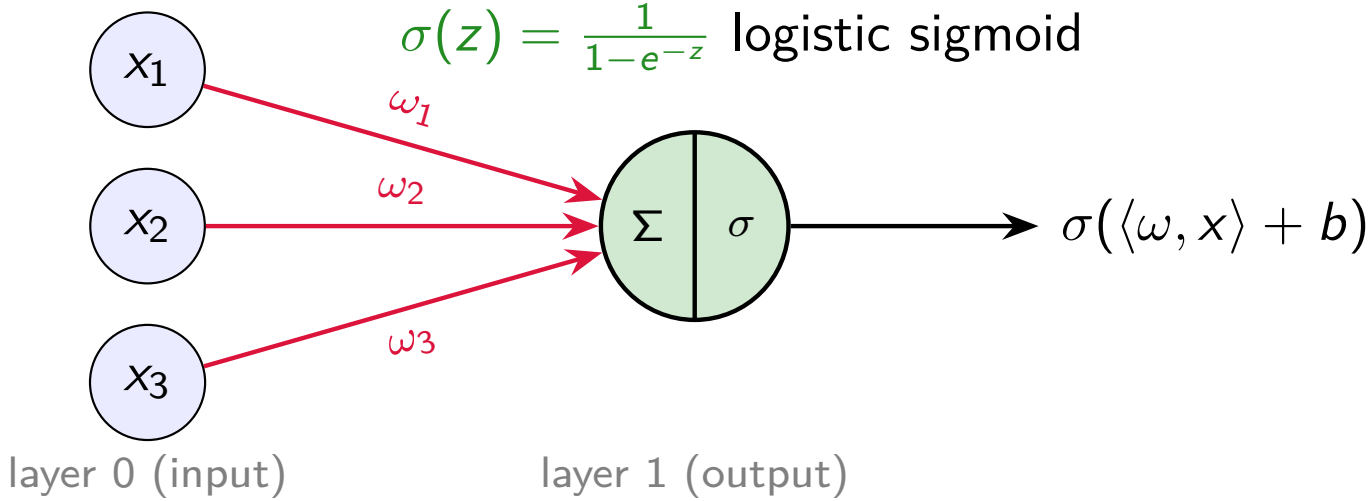
Linear regression is a 1-layer, 1-neuron NN with identity activation.



Parameters: $(\omega, b) \in \mathbb{R}^{d+1}$

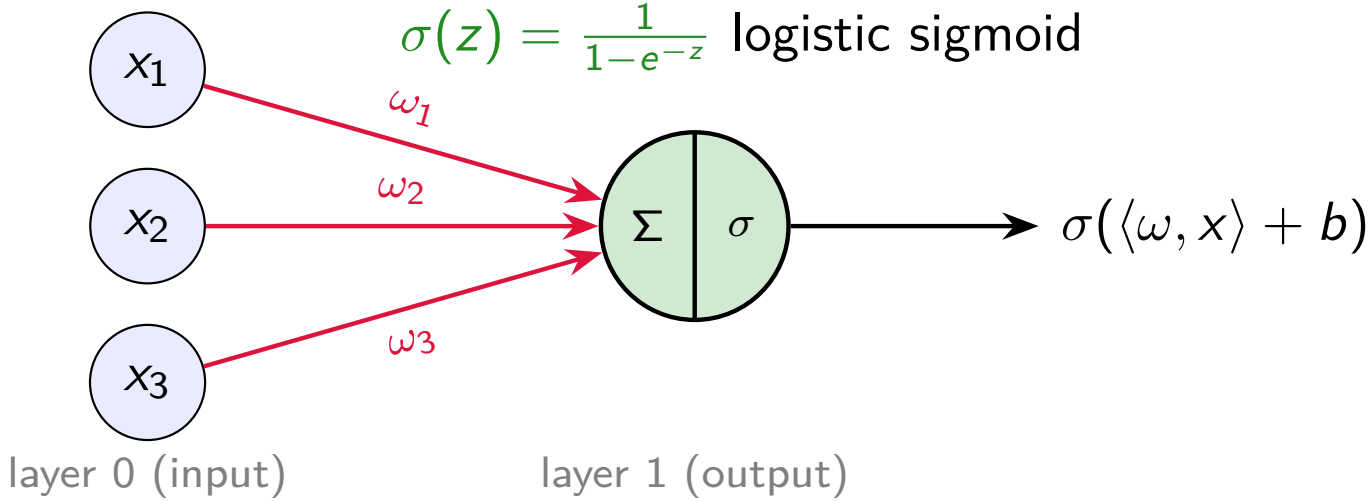
Train with $\ell(y, \hat{y}) = (y - \hat{y})^2 \longrightarrow$ recovers ordinary linear regression.

Logistic Regression as a 1-Layer, 1-Neuron FCFFN



Parameters: $(\omega, b) \in \mathbb{R}^{d+1}$

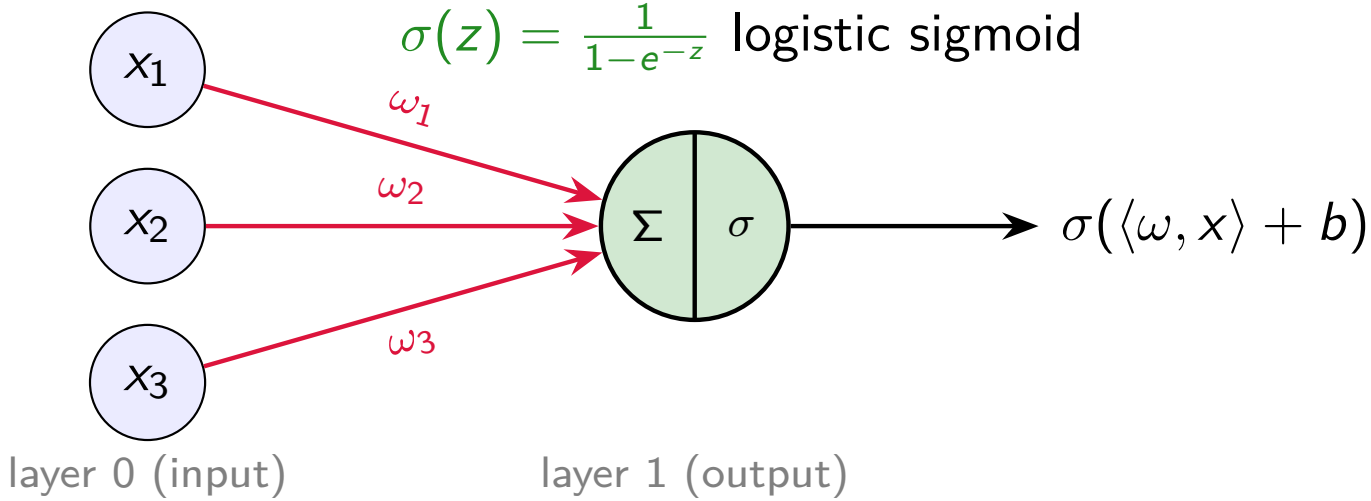
Logistic Regression as a 1-Layer, 1-Neuron FCFFN



Parameters: $(\omega, b) \in \mathbb{R}^{d+1}$

Similarly, multiclass logistic regression is a 1-layer FCFFN with K output neurons and softmax activation.

Logistic Regression as a 1-Layer, 1-Neuron FCFFN



Parameters: $(\omega, b) \in \mathbb{R}^{d+1}$

Similarly, multiclass logistic regression is a 1-layer FCFFN with K output neurons and softmax activation.

Layer counting convention: depth = number of hidden layers + 1 (for the output layer).
Linear and logistic regression are **depth-1** networks (0 hidden layers).

What Can Neural Networks Do That Fixed Feature Models Cannot?

All models from the first half take the form:

$$f(x;\omega) = g(\phi(x);\omega)$$

where the feature map ϕ is **fixed** and only g is learned.

What Can Neural Networks Do That Fixed Feature Models Cannot?

All models from the first half take the form:

$$f(x;\omega) = g(\phi(x);\omega)$$

where the feature map ϕ is **fixed** and only g is learned.

- **Hand-tuned features:** Domain expert designs ϕ (polynomial, Fourier, wavelet). Quality is limited by ϕ, but learning is convex in ω.
- **Kernel methods:** Implicitly define ϕ via $k(x, x')$. Expressive, but ϕ is fixed at design time and independent of data.

What Can Neural Networks Do That Fixed Feature Models Cannot?

All models from the first half take the form:

$$f(x;\omega) = g(\phi(x);\omega)$$

where the feature map ϕ is **fixed** and only g is learned.

- **Hand-tuned features:** Domain expert designs ϕ (polynomial, Fourier, wavelet). Quality is limited by ϕ, but learning is convex in ω.
- **Kernel methods:** Implicitly define ϕ via $k(x, x')$. Expressive, but ϕ is fixed at design time and independent of data.

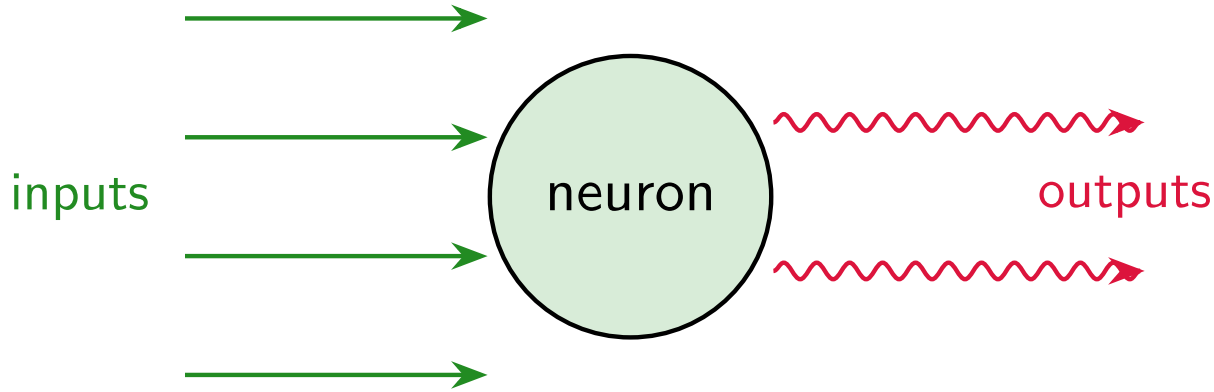
NNs allow **end-to-end learning of representations**. The hidden layers compute a data-dependent $\phi(x;\omega_{\text{hidden}})$, and the output layer is linear on top of these learned features. Both are learned simultaneously.

Biological Motivation and the Artificial Neuron

Motivated by biological neural networks (collections of neurons):

Biological Motivation and the Artificial Neuron

Motivated by biological neural networks (collections of neurons):

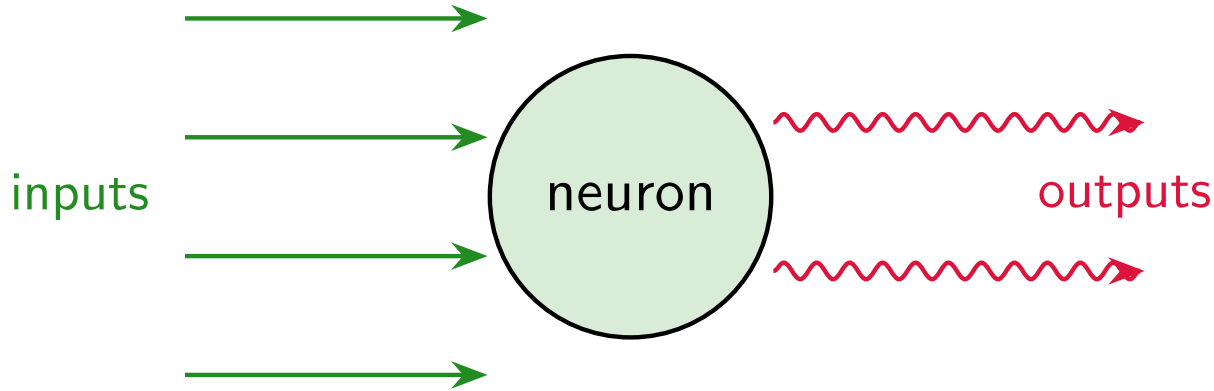


$$\text{output} = t(\text{input})$$

where t is nonlinear

Biological Motivation and the Artificial Neuron

Motivated by biological neural networks (collections of neurons):



$$\text{output} = t(\text{input})$$

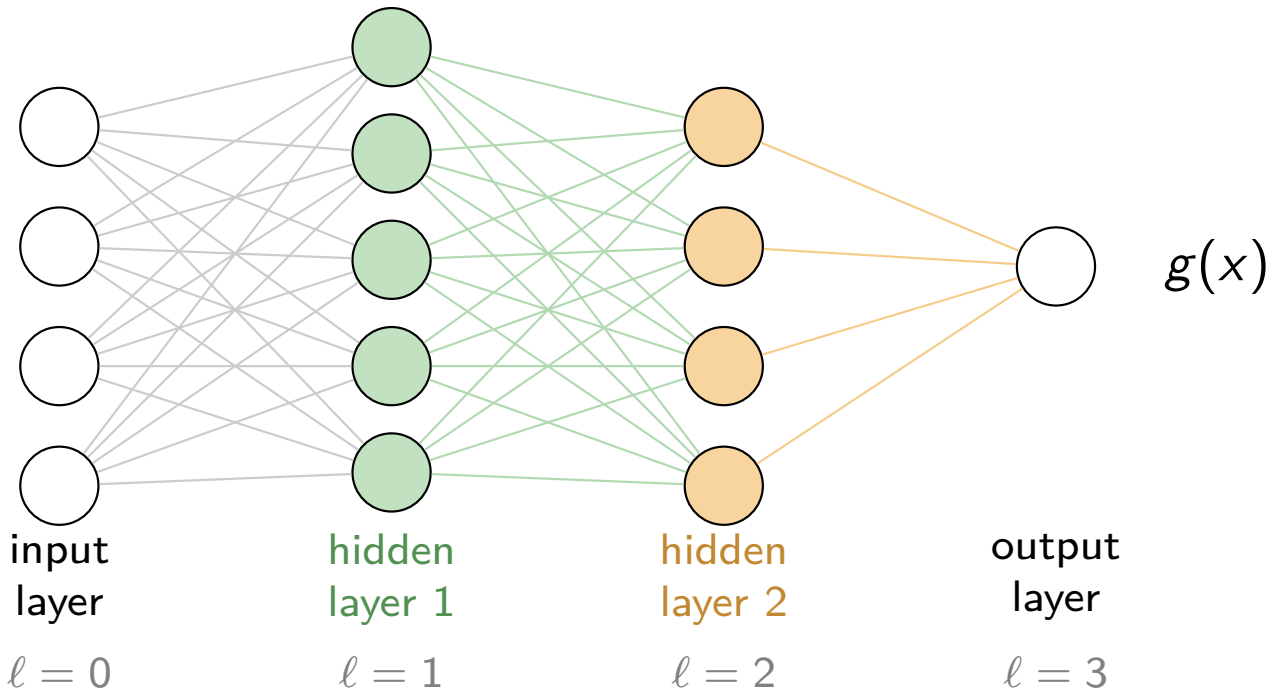
where t is nonlinear

How t is parameterized and how the neurons are connected together determines the type of neural network.

Fully-Connected Feedforward Neural Networks (FCFFNs)

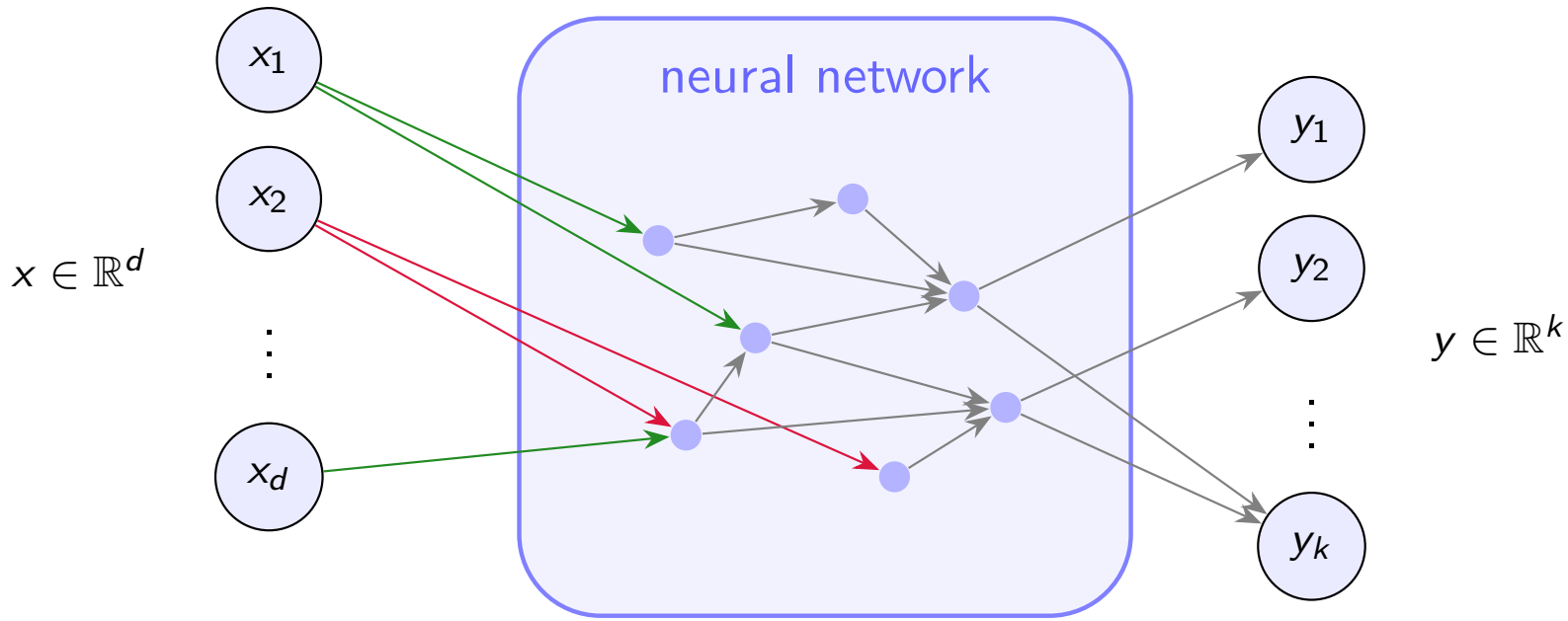
Also called FFNs or (when neurons are perceptron-type) MLPs.

Defining property: All neurons on layer ℓ are connected to <u>all</u> neurons on layer $\ell - 1$, and to <u>no other</u> neurons.



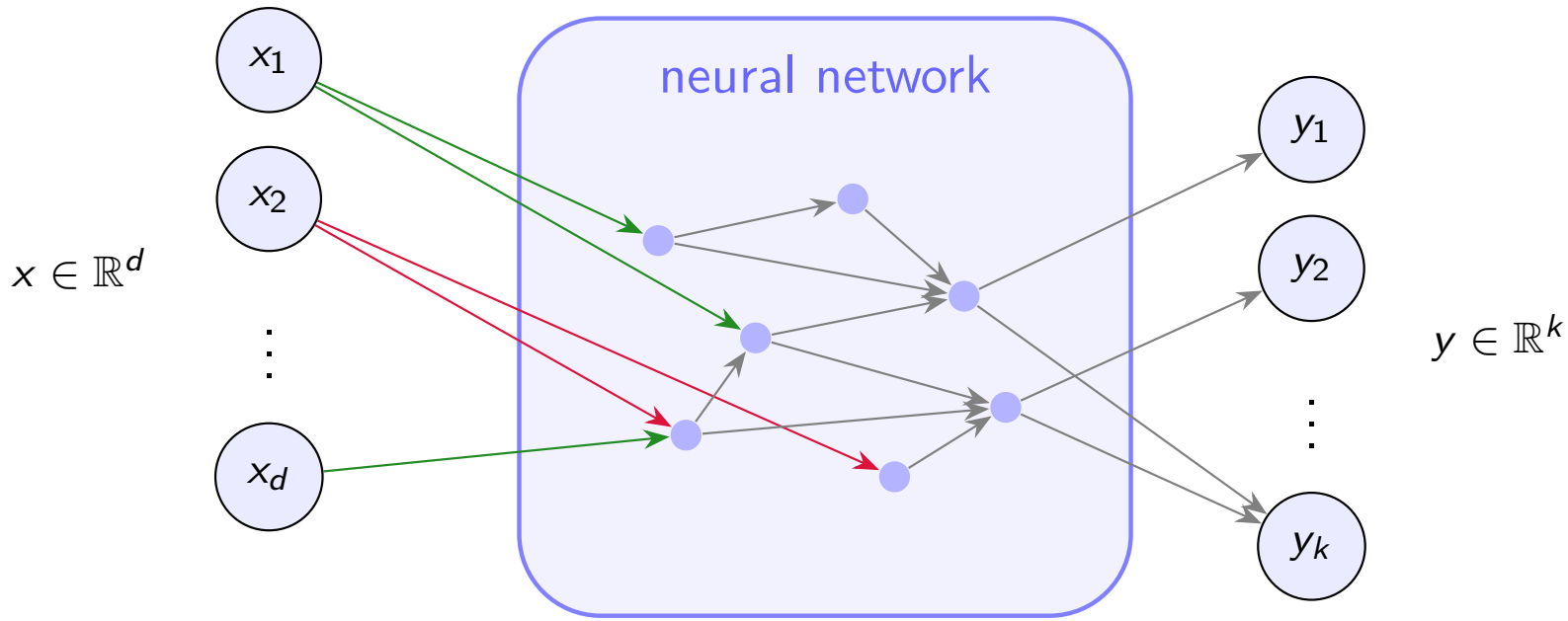
Neural Networks as Computational DAGs

FCFFNs are a special case of a more general structure: a **computational DAG** (directed acyclic graph).



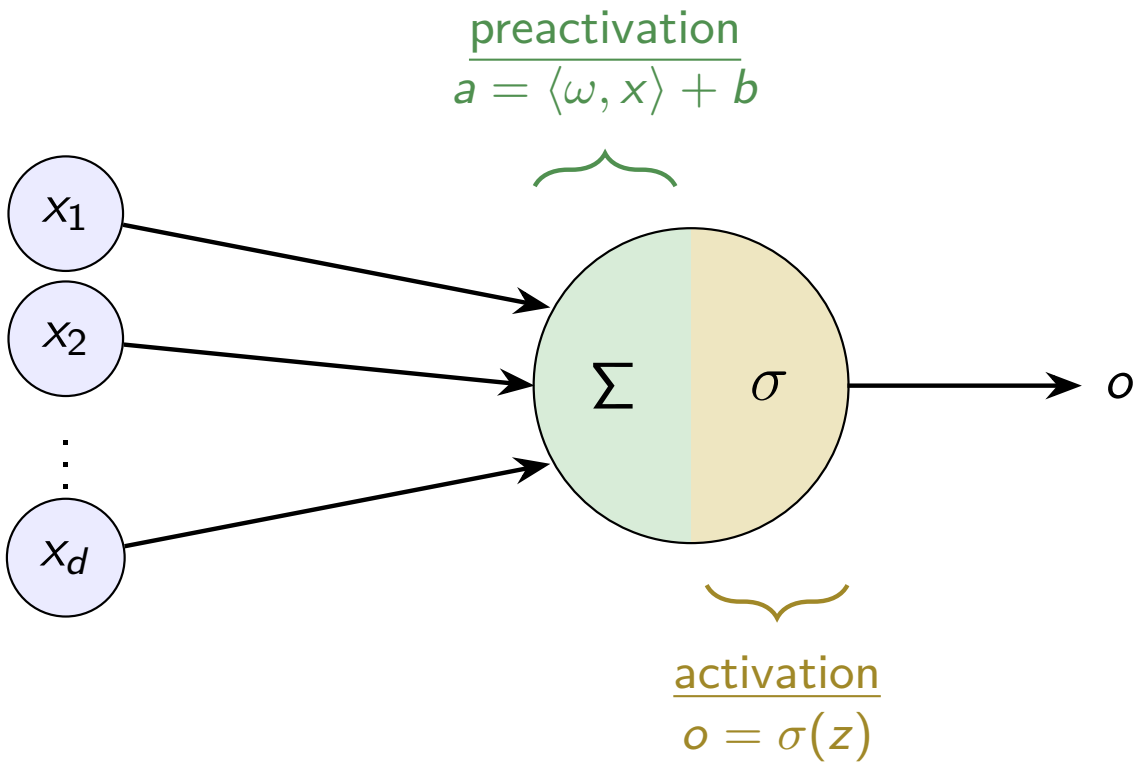
Neural Networks as Computational DAGs

FCFFNs are a special case of a more general structure: a **computational DAG** (directed acyclic graph).



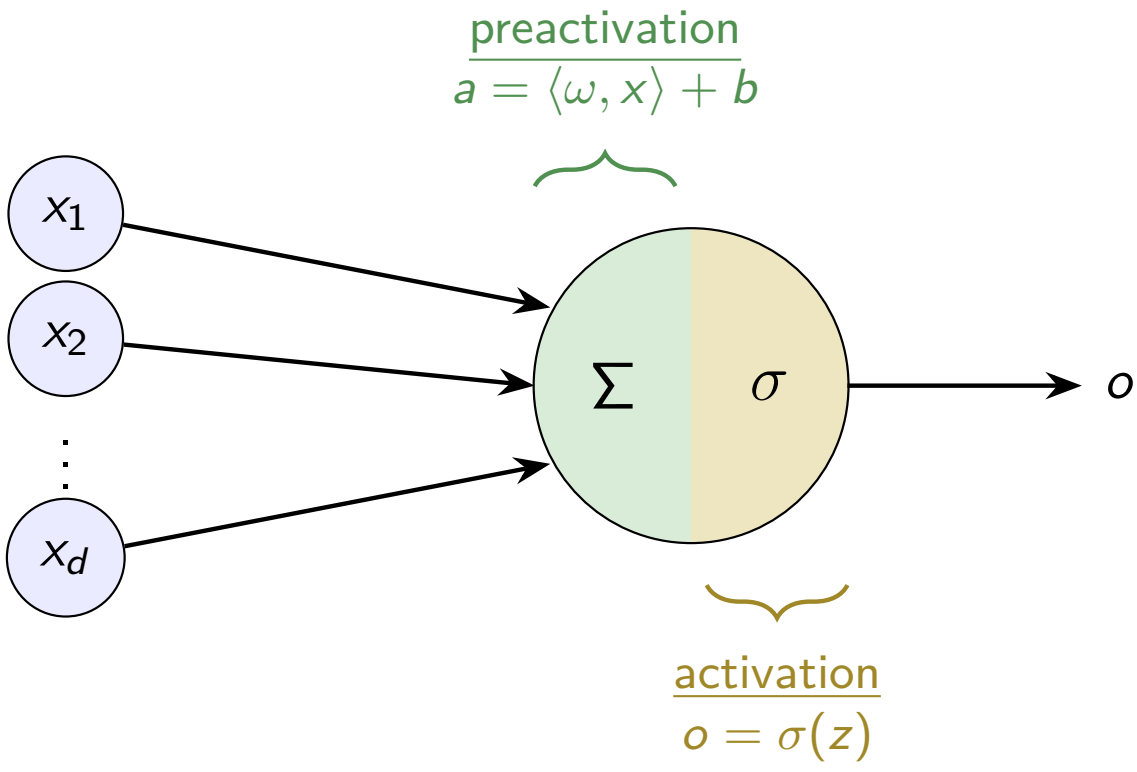
The DAG can have arbitrary acyclic structure; layers are a *choice*, not a requirement. The structure of this DAG is called the **architecture**.

Typical Setup of a Neuron (Perceptron-Type)



Parameters: $(\omega, b) \in \mathbb{R}^{d+1}$

Typical Setup of a Neuron (Perceptron-Type)

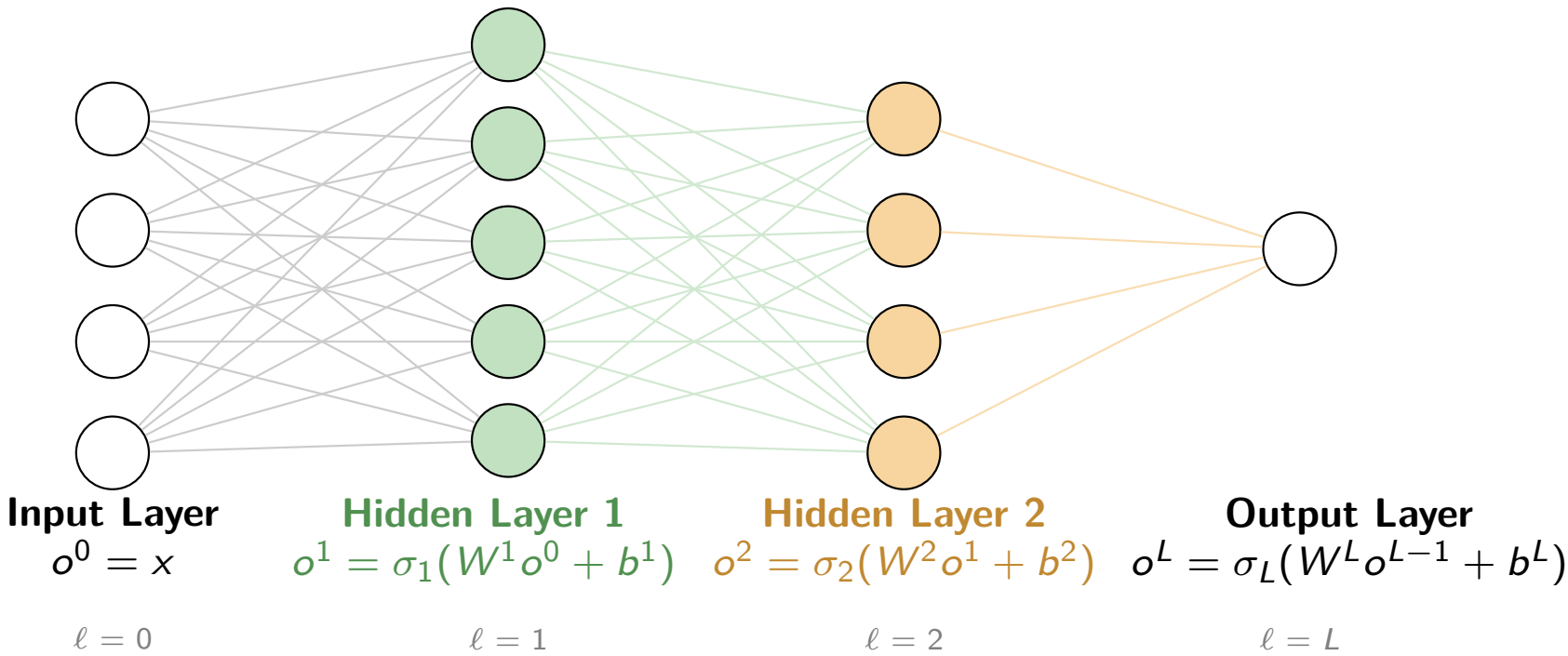


Parameters: $(\omega, b) \in \mathbb{R}^{d+1}$

These are called **perceptron-type neurons**.

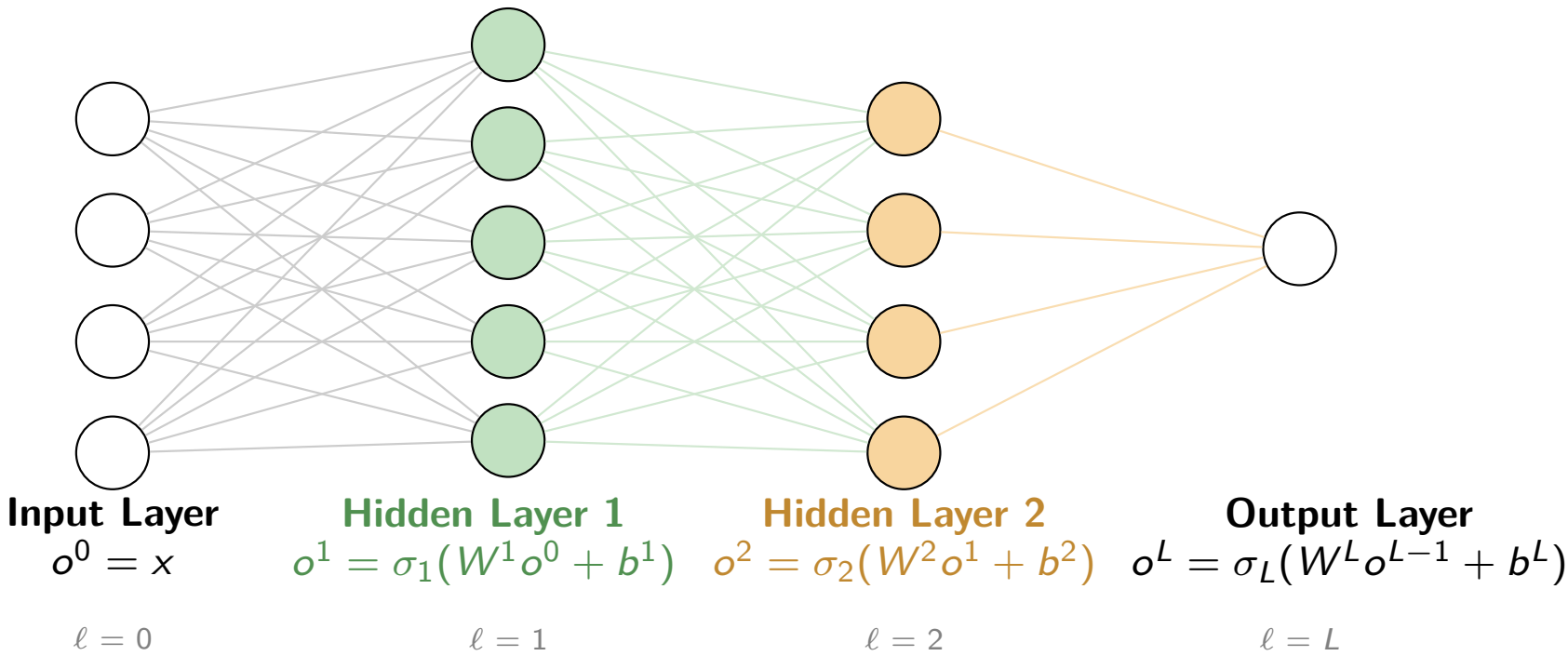
Multilayer Perceptron Networks

MLPs are FCFFNs where the neurons are all perceptron type.
Each layer ℓ takes in the output vector from the previous layer, $o^{\ell-1}$, and produces its output o^{ℓ}.



Multilayer Perceptron Networks

MLPs are FCFFNs where the neurons are all perceptron type.
Each layer ℓ takes in the output vector from the previous layer, $o^{\ell-1}$, and produces its output o^{ℓ}.



Each o^{ℓ} is a vector. The matrix W^{ℓ} contains the weights connecting layer $\ell - 1$ to ℓ, and b^{ℓ} is the bias vector for layer ℓ.

Why Must σ Be Nonlinear?

Claim: **Without nonlinearity, depth is mathematically useless**.

Why Must σ Be Nonlinear?

Claim: **Without nonlinearity, depth is mathematically useless**.

If we set $\sigma(z) = z$ (the identity function), our MLP computes:

$$o^1 = W^1 x + b^1$$

$$o^2 = W^2 o^1 + b^2 = W^2(W^1 x + b^1) + b^2$$

$$\vdots$$

$$o^L = W^L(W^{L-1}(\cdots(W^1 x + b^1)\cdots) + b^{L-1}) + b^L$$

Why Must σ Be Nonlinear?

Claim: Without nonlinearity, depth is mathematically useless.

If we set $\sigma(z) = z$ (the identity function), our MLP computes:

$$o^1 = W^1 x + b^1$$
$$o^2 = W^2 o^1 + b^2 = W^2(W^1 x + b^1) + b^2$$
$$\vdots$$
$$o^L = W^L(W^{L-1}(\cdots(W^1 x + b^1)\cdots) + b^{L-1}) + b^L$$

Expanding this composition, we find:

$$o^L = \underbrace{(W^L W^{L-1} \cdots W^1)}_{\tilde{W}} x + \underbrace{(\text{affine combination of } b^\ell)}_{\tilde{b}}$$

Why Must σ Be Nonlinear?

Claim: Without nonlinearity, depth is mathematically useless.

If we set $\sigma(z) = z$ (the identity function), our MLP computes:

$$o^1 = W^1 x + b^1$$
$$o^2 = W^2 o^1 + b^2 = W^2(W^1 x + b^1) + b^2$$
$$\vdots$$
$$o^L = W^L(W^{L-1}(\cdots(W^1 x + b^1)\cdots) + b^{L-1}) + b^L$$

Expanding this composition, we find:

$$o^L = \underbrace{(W^L W^{L-1} \cdots W^1)}_{\tilde{W}} x + \underbrace{(\text{affine combination of } b^\ell)}_{\tilde{b}}$$

An L-layer linear network computes exactly the same function class as a single-layer linear model: $o^L = \tilde{W}x + \tilde{b}$!

Why Must σ Be Nonlinear?

Claim: Without nonlinearity, depth is mathematically useless.

If we set $\sigma(z) = z$ (the identity function), our MLP computes:

$$o^1 = W^1 x + b^1$$
$$o^2 = W^2 o^1 + b^2 = W^2(W^1 x + b^1) + b^2$$
$$\vdots$$
$$o^L = W^L(W^{L-1}(\cdots(W^1 x + b^1)\cdots) + b^{L-1}) + b^L$$

Expanding this composition, we find:

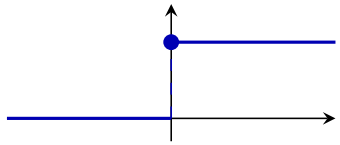
$$o^L = \underbrace{(W^L W^{L-1} \cdots W^1)}_{\tilde{W}} x + \underbrace{(\text{affine combination of } b^\ell)}_{\tilde{b}}$$

An L-layer linear network computes exactly the same function class as a single-layer linear model: $o^L = \tilde{W}x + \tilde{b}$!

Nonlinearity is what prevents the layers from "collapsing."

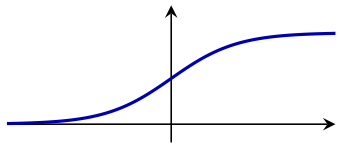
Common Activation Functions

$\sigma(x) = \text{step}(x)$



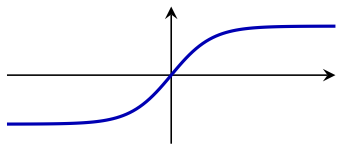
non-differentiable

$\sigma(x) = \frac{1}{1+e^{-x}}$



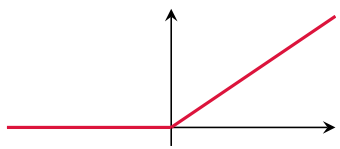
logistic sigmoid

$\sigma(x) = \tanh(x)$



tanh

$\sigma(x) = \max(0, x)$



ReLU

Why ReLU Dominates in Practice

Sigmoid and tanh **saturate** for large $|x|$: their derivatives approach 0.

Why ReLU Dominates in Practice

Sigmoid and tanh **saturate** for large $|x|$: their derivatives approach 0.

In deep networks, gradients are products of per-layer derivatives (chain rule), so saturation causes **vanishing gradients** — deep layers receive negligible gradient signal.

Why ReLU Dominates in Practice

Sigmoid and tanh **saturate** for large $|x|$: their derivatives approach 0.

In deep networks, gradients are products of per-layer derivatives (chain rule), so saturation causes **vanishing gradients** — deep layers receive negligible gradient signal.

ReLU avoids saturation on active units ($x > 0$), yielding better-conditioned Jacobians in deep compositions.

Why ReLU Dominates in Practice

Sigmoid and tanh **saturate** for large $|x|$: their derivatives approach 0.

In deep networks, gradients are products of per-layer derivatives (chain rule), so saturation causes **vanishing gradients** — deep layers receive negligible gradient signal.

ReLU avoids saturation on active units ($x > 0$), yielding better-conditioned Jacobians in deep compositions.

This links *architecture choice* to *optimization tractability* — a recurring theme in NN design.

Why ReLU Dominates in Practice

Sigmoid and tanh **saturate** for large $|x|$: their derivatives approach 0.

In deep networks, gradients are products of per-layer derivatives (chain rule), so saturation causes **vanishing gradients** — deep layers receive negligible gradient signal.

ReLU avoids saturation on active units ($x > 0$), yielding better-conditioned Jacobians in deep compositions.

This links *architecture choice* to *optimization tractability* — a recurring theme in NN design.

The activation function is a **hyperparameter**, chosen by validation just like the architecture, optimizer, and regularizer.

Why Do We Need NNs?

Models seen so far take the form $f(\langle \omega, x \rangle + b)$.

Why Do We Need NNs?

Models seen so far take the form $f(\langle \omega, x \rangle + b)$.

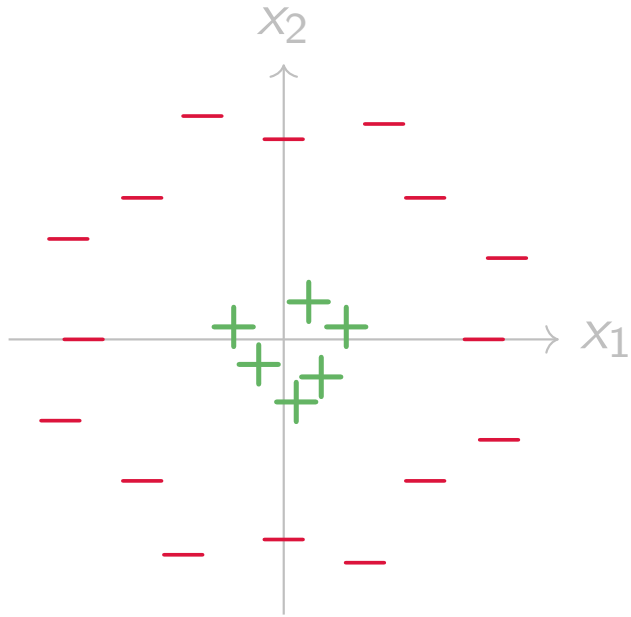
These are useful for problems with **linear separability**.

Why Do We Need NNs?

Models seen so far take the form $f(\langle \omega, x \rangle + b)$.

These are useful for problems with **linear separability**.

What about problems that are not linearly separable?

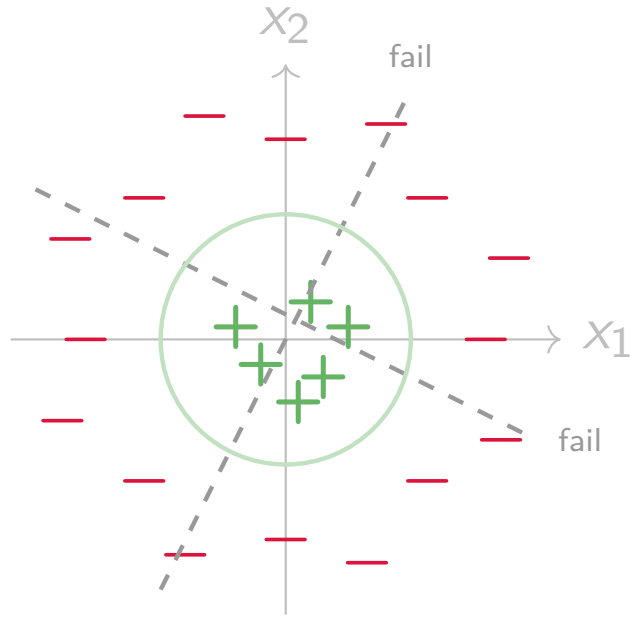


Why Do We Need NNs?

Models seen so far take the form $f(\langle \omega, x \rangle + b)$.

These are useful for problems with **linear separability**.

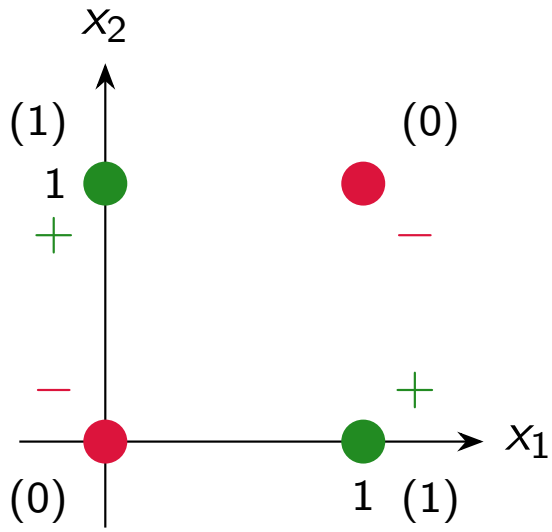
What about problems that are not linearly separable?



XOR Is Not Linearly Separable

Classic example! The XOR function $f(x_1, x_2) = x_1 \oplus x_2$:

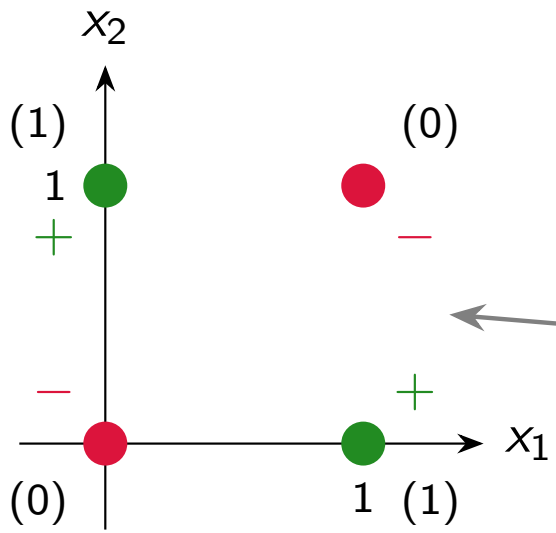
$$f(x_1, x_2) = \begin{cases} 1 & x_1 \neq x_2 \\ 0 & x_1 = x_2 \end{cases}$$



XOR Is Not Linearly Separable

Classic example! The XOR function $f(x_1, x_2) = x_1 \oplus x_2$:

$$f(x_1, x_2) = \begin{cases} 1 & x_1 \neq x_2 \\ 0 & x_1 = x_2 \end{cases}$$



These classes are **not** linearly separable, so SVM, logistic regression, etc. will not work.

NNs Can Learn XOR

A FCFFN with 2 hidden neurons can express XOR using threshold activation.

NNs Can Learn XOR

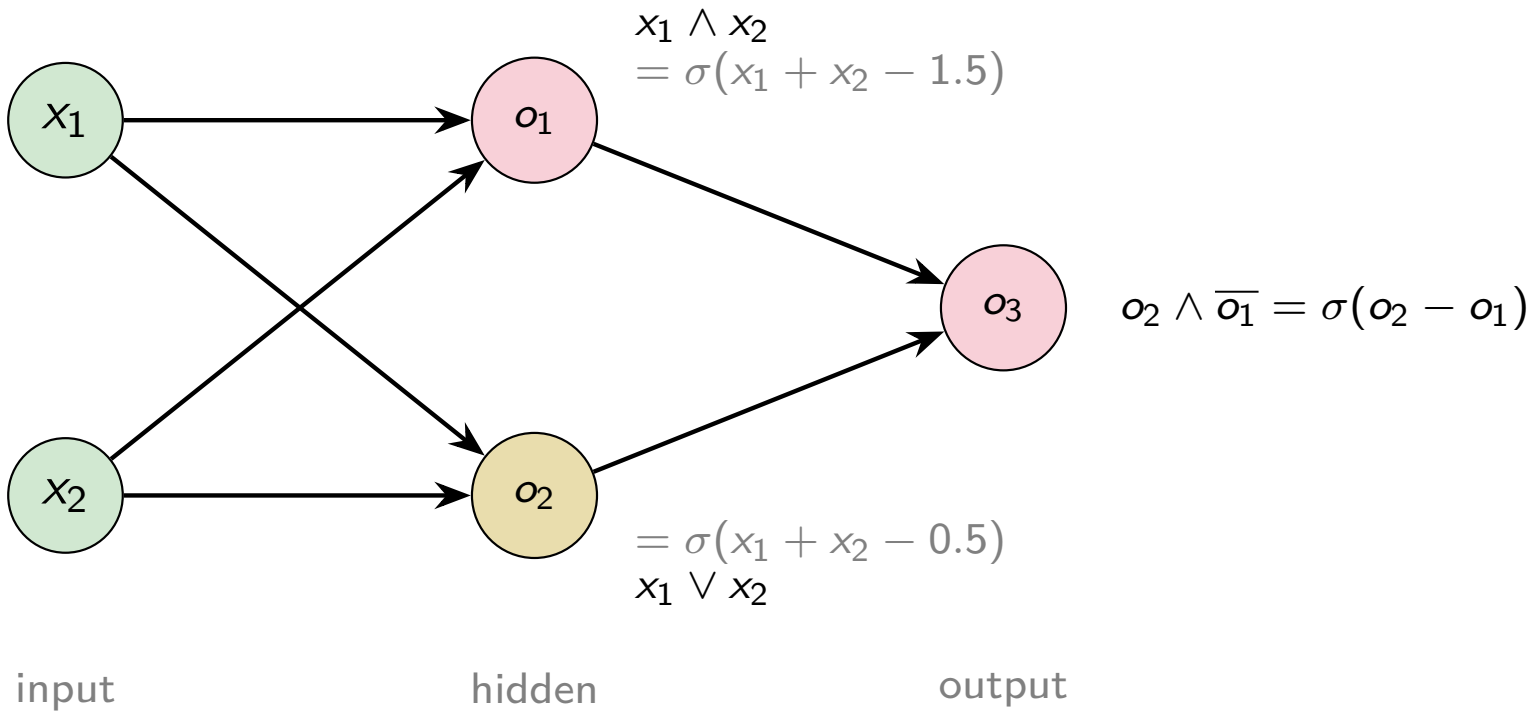
A FCFFN with 2 hidden neurons can express XOR using threshold activation.

Key identity: $\mathrm{XOR}(x_1, x_2) = (x_1 \vee x_2) \ \wedge \ \overline{(x_1 \wedge x_2)}$

NNs Can Learn XOR

A FCFFN with 2 hidden neurons can express XOR using threshold activation.

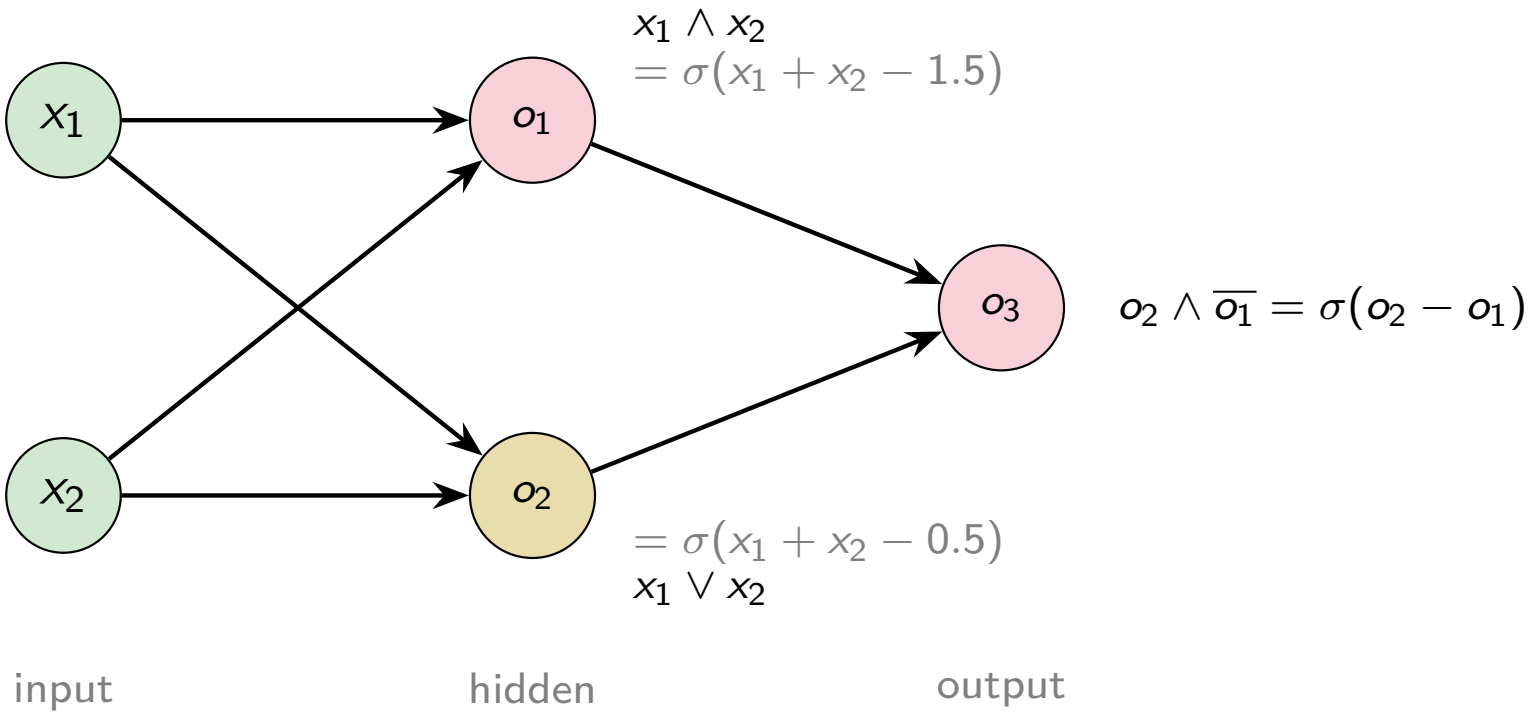
Key identity: $\mathrm{XOR}(x_1, x_2) = (x_1 \vee x_2) \wedge \overline{(x_1 \wedge x_2)}$



NNs Can Learn XOR

A FCFFN with 2 hidden neurons can express XOR using threshold activation.

Key identity: $\mathrm{XOR}(x_1, x_2) = (x_1 \vee x_2) \wedge \overline{(x_1 \wedge x_2)}$



Verification: Check all 4 input combinations.

Expressivity of NNs — Intuition

Question: What functions can NNs approximate?

Expressivity of NNs — Intuition

Question: What functions can NNs approximate?

Consider the set of 1-hidden-layer NNs with "bump-like" activation σ:

$$\Sigma_1 = \left\{ f \;\middle|\; f(x) = \sum_{i=1}^{k} a_i \, \sigma(\langle \omega_i, x \rangle + b_i) \right\}$$

Expressivity of NNs — Intuition

Question: What functions can NNs approximate?

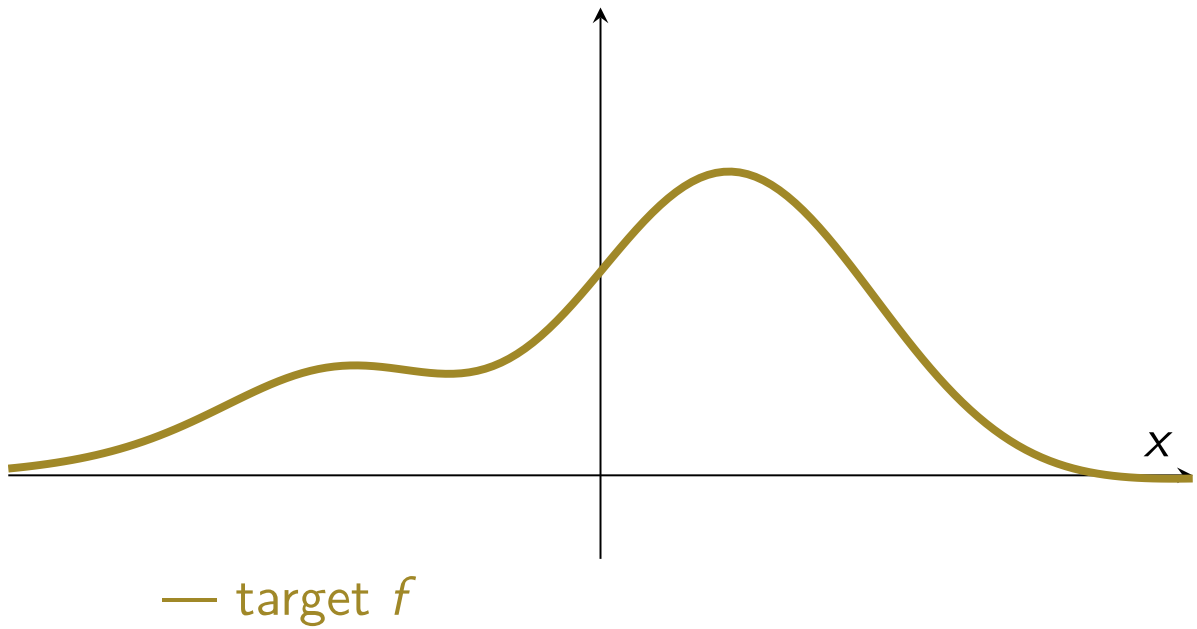
Consider the set of 1-hidden-layer NNs with "bump-like" activation σ:

$$\Sigma_1 = \left\{ f \;\middle|\; f(x) = \sum_{i=1}^{k} a_i \, \sigma(\langle \omega_i, x \rangle + b_i) \right\}$$

This is the set of superpositions of **scaled** (a_i), **skewed** (ω_i), **translated** (b_i) bumps.

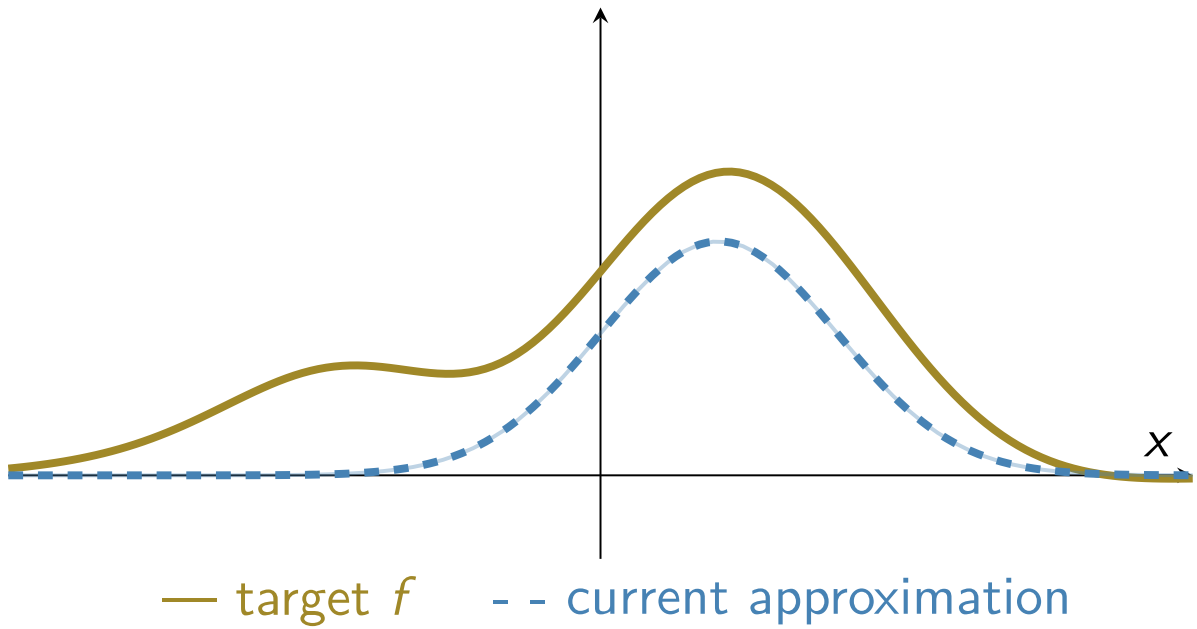
Superposition of Bumps

Intuitively, any function f on a compact set can be arbitrarily well approximation as a superposition of such scaled, skewed, and translated bumps.



Superposition of Bumps

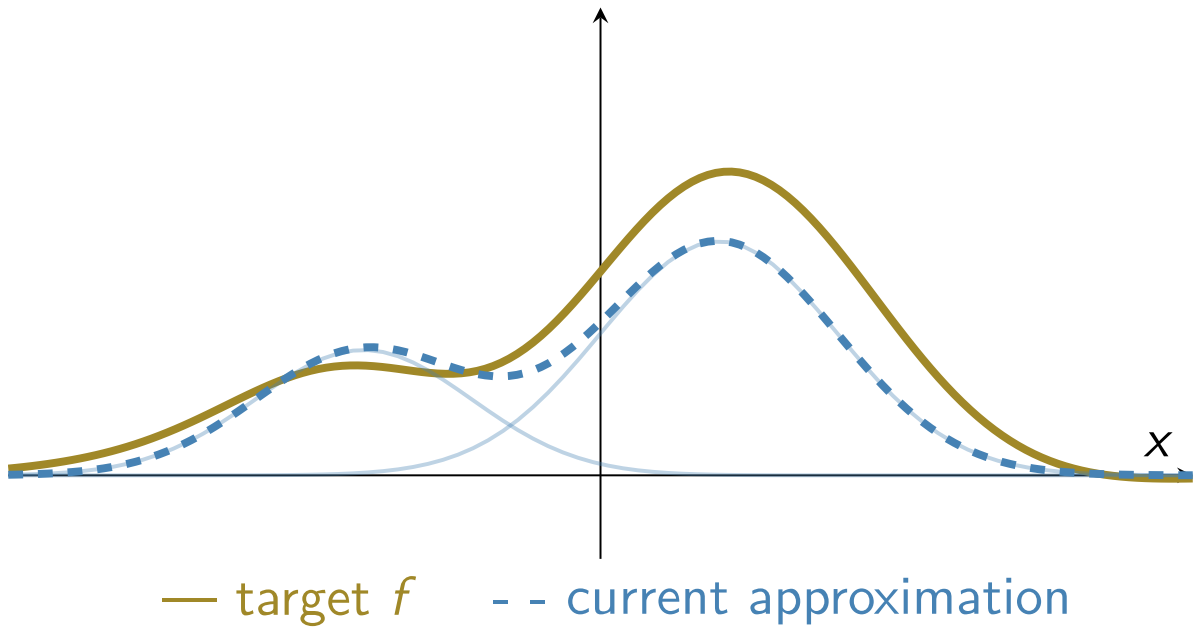
Intuitively, any function f on a compact set can be arbitrarily well approximation as a superposition of such scaled, skewed, and translated bumps.



More bumps give a better approximation. This connects to the idea that **wider networks approximate better**.

Superposition of Bumps

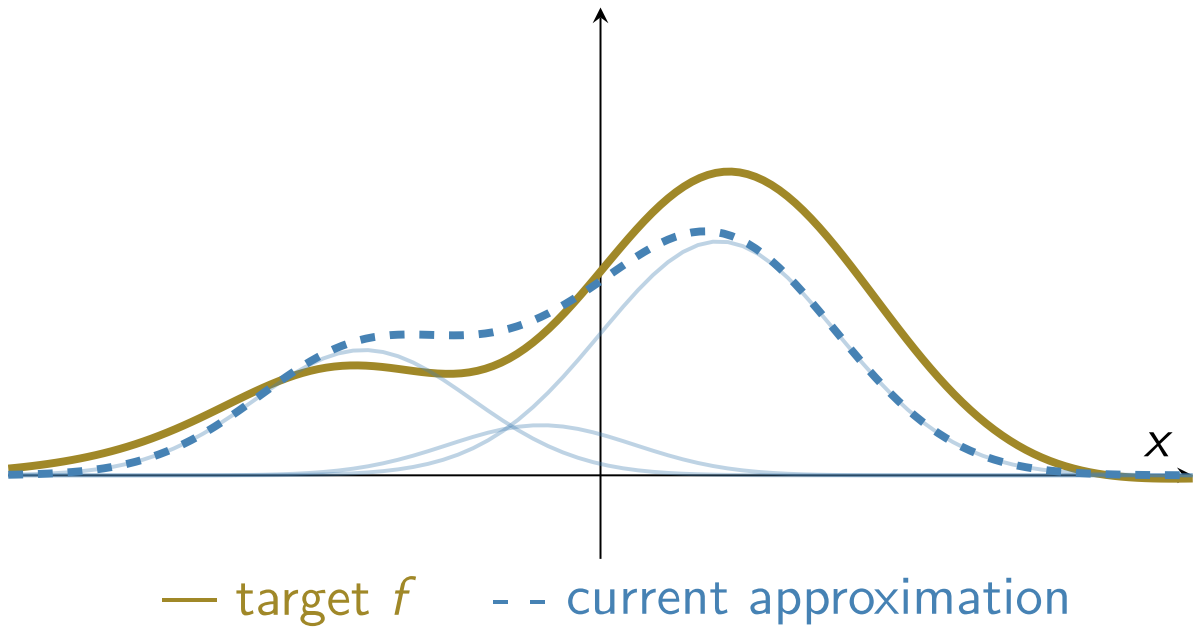
Intuitively, any function f on a compact set can be arbitrarily well approximation as a superposition of such scaled, skewed, and translated bumps.



More bumps give a better approximation. This connects to the idea that **wider networks approximate better**.

Superposition of Bumps

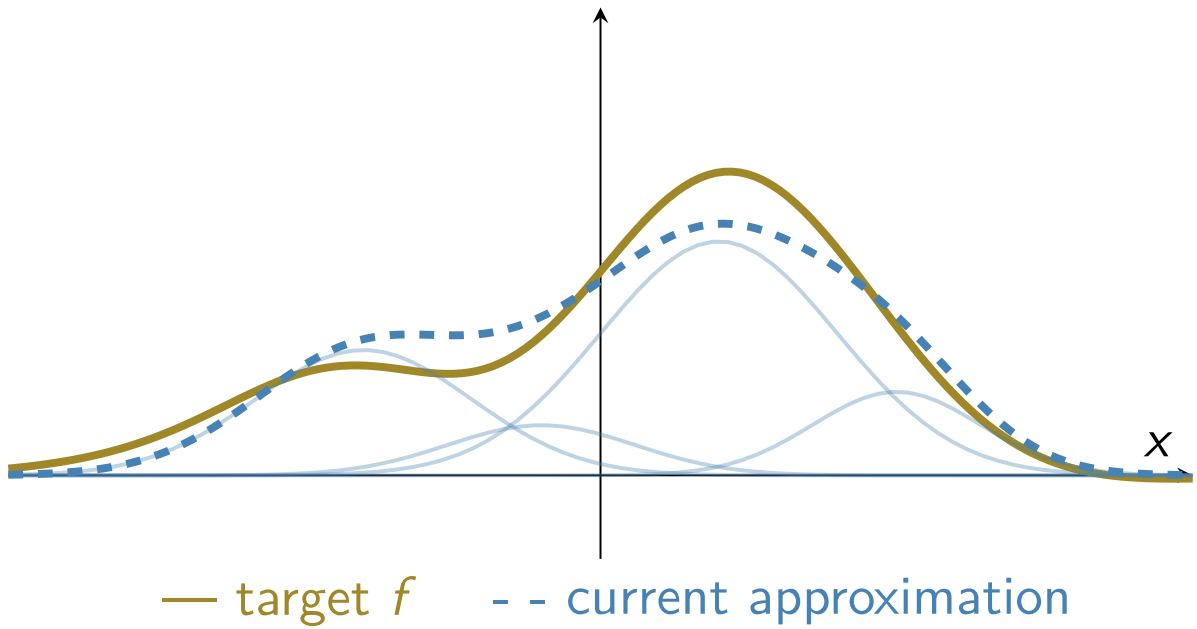
Intuitively, any function f on a compact set can be arbitrarily well approximation as a superposition of such scaled, skewed, and translated bumps.



More bumps give a better approximation. This connects to the idea that **wider networks approximate better**.

Superposition of Bumps

Intuitively, any function f on a compact set can be arbitrarily well approximation as a superposition of such scaled, skewed, and translated bumps.



More bumps give a better approximation. This connects to the idea that **wider networks approximate better**.

Universal Approximation Theorem

Theorem (Leshno, Lin, Pinkus & Schocken, 1993)

Let $\sigma : \mathbb{R} \to \mathbb{R}$ be locally bounded and piecewise continuous. Set

$$\Sigma_1 = \mathrm{span}\{\sigma(\langle \omega, x \rangle + b) : \omega \in \mathbb{R}^d, b \in \mathbb{R}\}.$$

Then Σ_1 is dense in $C(K)$ for every compact $K \subset \mathbb{R}^d$ **if and only if** σ is not a polynomial.

Universal Approximation Theorem

Theorem (Leshno, Lin, Pinkus & Schocken, 1993)

Let $\sigma : \mathbb{R} \to \mathbb{R}$ be locally bounded and piecewise continuous. Set

$$\Sigma_1 = \text{span}\{\sigma(\langle \omega, x \rangle + b) : \omega \in \mathbb{R}^d, b \in \mathbb{R}\}.$$

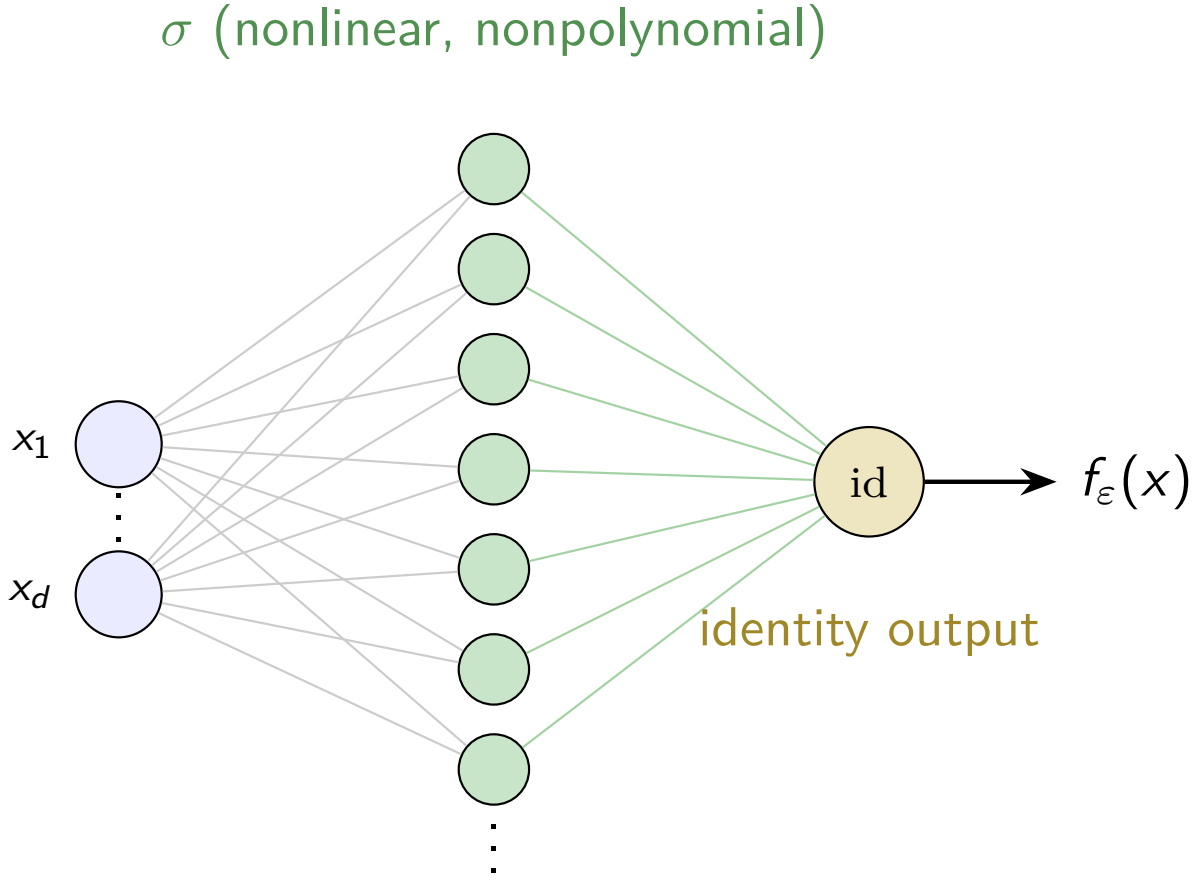
Then Σ_1 is dense in $C(K)$ for every compact $K \subset \mathbb{R}^d$ **if and only if** σ is not a polynomial.

That is, for any continuous $f : K \to \mathbb{R}$ and any $\varepsilon > 0$, there exists $f_\varepsilon \in \Sigma_1$ such that

$$\|f - f_\varepsilon\|_\infty = \max_{x \in K} |f(x) - f_\varepsilon(x)| < \varepsilon.$$

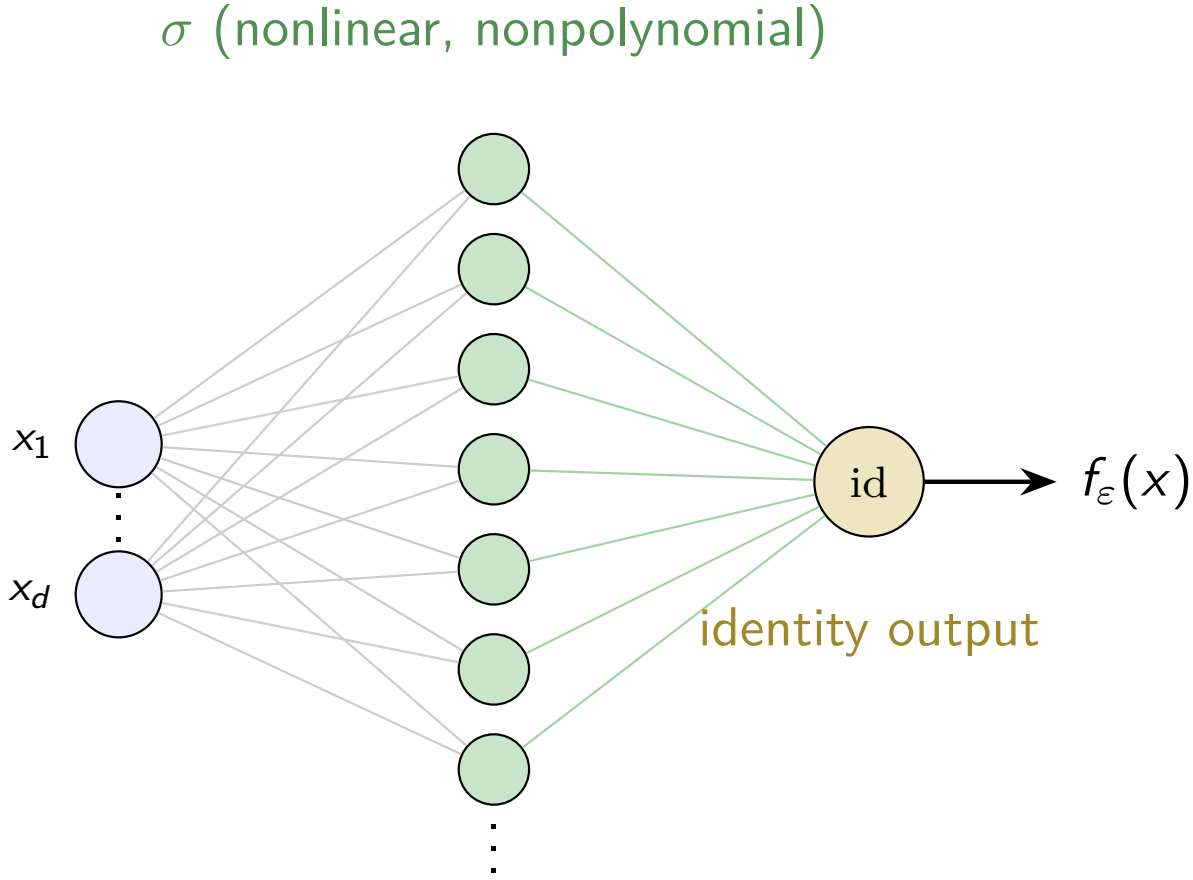
UAT: The Architecture

The theorem applies to 1-hidden-layer networks with nonpolynomial activation and identity at the output:



UAT: The Architecture

The theorem applies to 1-hidden-layer networks with nonpolynomial activation and identity at the output:



All common activations (sigmoid, tanh, ReLU) satisfy the hypotheses.

UAT: Important Caveats

This is an *existence* result. It says a sufficiently wide 1-hidden-layer network *exists* but says nothing about:

- how to *find* the weights, or
- how many neurons are needed.

UAT: Important Caveats

This is an *existence* result. It says a sufficiently wide 1-hidden-layer network *exists* but says nothing about:

- how to *find* the weights, or
- how many neurons are needed.

Curse of dimensionality: For general L-Lipschitz functions on $[0,1]^d$, one may need $O(1/\varepsilon^d)$ hidden neurons — exponential in d.

UAT: Important Caveats

This is an *existence* result. It says a sufficiently wide 1-hidden-layer network *exists* but says nothing about:

- how to *find* the weights, or
- how many neurons are needed.

Curse of dimensionality: For general L-Lipschitz functions on $[0,1]^d$, one may need $O(1/\varepsilon^d)$ hidden neurons — exponential in d.

Barron (1993) showed that for functions with bounded first moment of the Fourier transform, $O(1/\varepsilon^2)$ neurons suffice *independent of dimension* — NNs are particularly efficient for functions with spectral decay.

Misinterpretation Warning

Universal approximation does **not** mean:

Misinterpretation Warning

Universal approximation does **not** mean:

- Shallow networks are *parameter-efficient* (width may need to be exponential in d)

Misinterpretation Warning

Universal approximation does **not** mean:

- Shallow networks are *parameter-efficient* (width may need to be exponential in d)
- Shallow networks are *data-efficient* (sample complexity may be enormous)

Misinterpretation Warning

Universal approximation does **not** mean:

- Shallow networks are *parameter-efficient* (width may need to be exponential in d)
- Shallow networks are *data-efficient* (sample complexity may be enormous)
- Shallow networks are *easy to train* (finding the weights is a nonconvex optimization problem)

Misinterpretation Warning

Universal approximation does **not** mean:

- Shallow networks are *parameter-efficient* (width may need to be exponential in d)
- Shallow networks are *data-efficient* (sample complexity may be enormous)
- Shallow networks are *easy to train* (finding the weights is a nonconvex optimization problem)

Depth *is used to add parameter efficiency and inductive bias.*

Misinterpretation Warning

Universal approximation does **not** mean:

- Shallow networks are *parameter-efficient* (width may need to be exponential in d)
- Shallow networks are *data-efficient* (sample complexity may be enormous)
- Shallow networks are *easy to train* (finding the weights is a nonconvex optimization problem)

***Depth** is used to add parameter efficiency and inductive bias.*

Depth separation: There exist functions representable by depth-L networks with polynomial width that require exponential width at depth $L - 1$ (Telgarsky 2016; Eldan & Shamir 2016).

Misinterpretation Warning

Universal approximation does **not** mean:

- Shallow networks are *parameter-efficient* (width may need to be exponential in d)
- Shallow networks are *data-efficient* (sample complexity may be enormous)
- Shallow networks are *easy to train* (finding the weights is a nonconvex optimization problem)

Depth *is used to add parameter efficiency and inductive bias.*

Depth separation: There exist functions representable by depth-L networks with polynomial width that require exponential width at depth $L - 1$ (Telgarsky 2016; Eldan & Shamir 2016).

Depth gives fundamentally more efficient representations for certain function classes.

Vector Notation for MLPs

As we will use MLPs often, let's fix the notation.

Vector Notation for MLPs

As we will use MLPs often, let's fix the notation.

Let our MLP have L layers. Each layer ℓ has n_ℓ neurons:

- Layer 0 is the input layer: $n_0 = d$
- Layer L is the output layer: $n_L = k$

Vector Notation for MLPs

As we will use MLPs often, let's fix the notation.

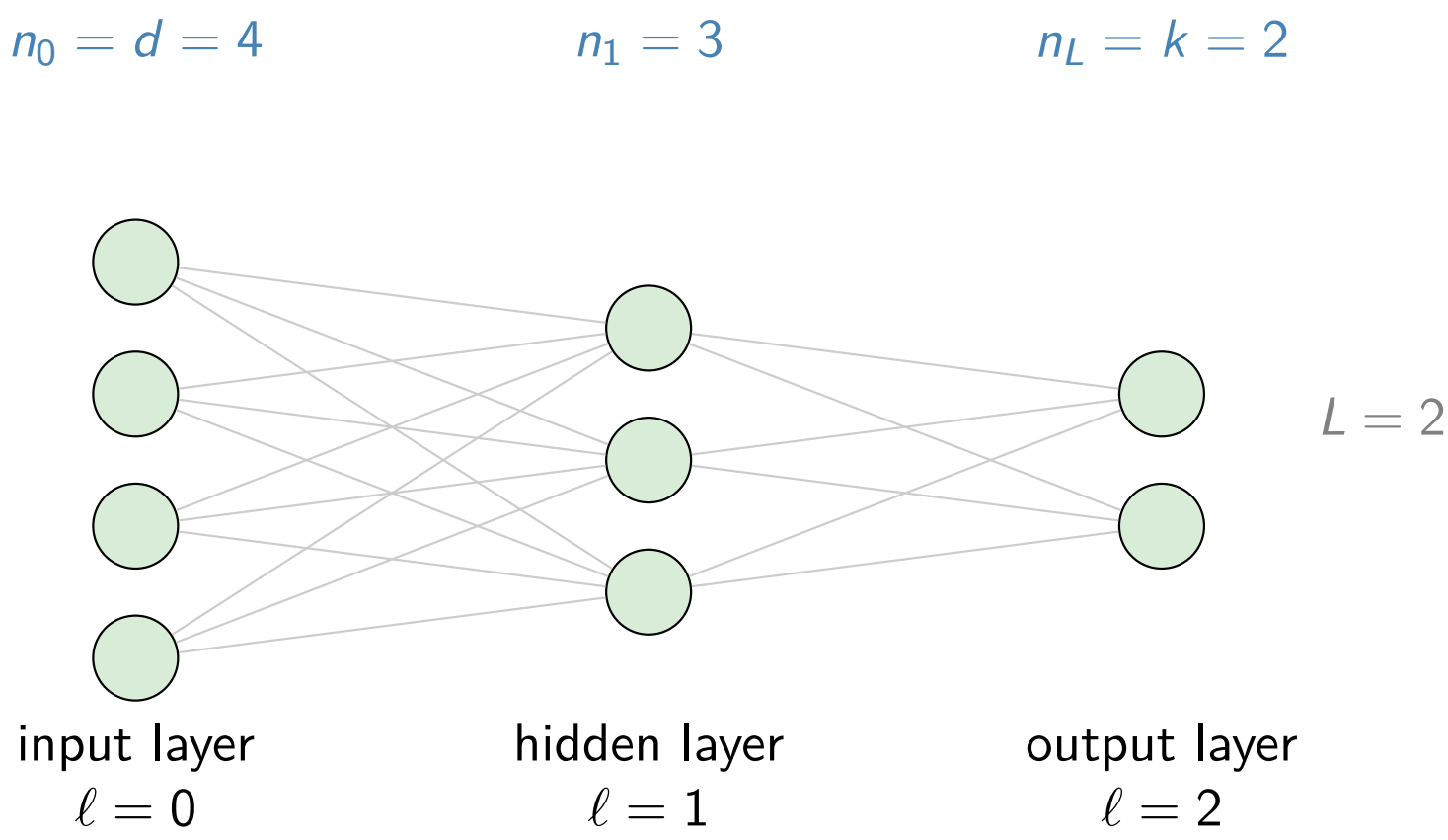
Let our MLP have L layers. Each layer ℓ has n_ℓ neurons:

- Layer 0 is the input layer: $n_0 = d$
- Layer L is the output layer: $n_L = k$

We write the preactivations and activations of layer ℓ as vectors in $\mathbb{R}^{n_\ell}$:

- $a^\ell \in \mathbb{R}^{n_\ell}$ — **preactivation vector**
- $o^\ell \in \mathbb{R}^{n_\ell}$ — **activation vector**

Example: MLP with $L = 2$



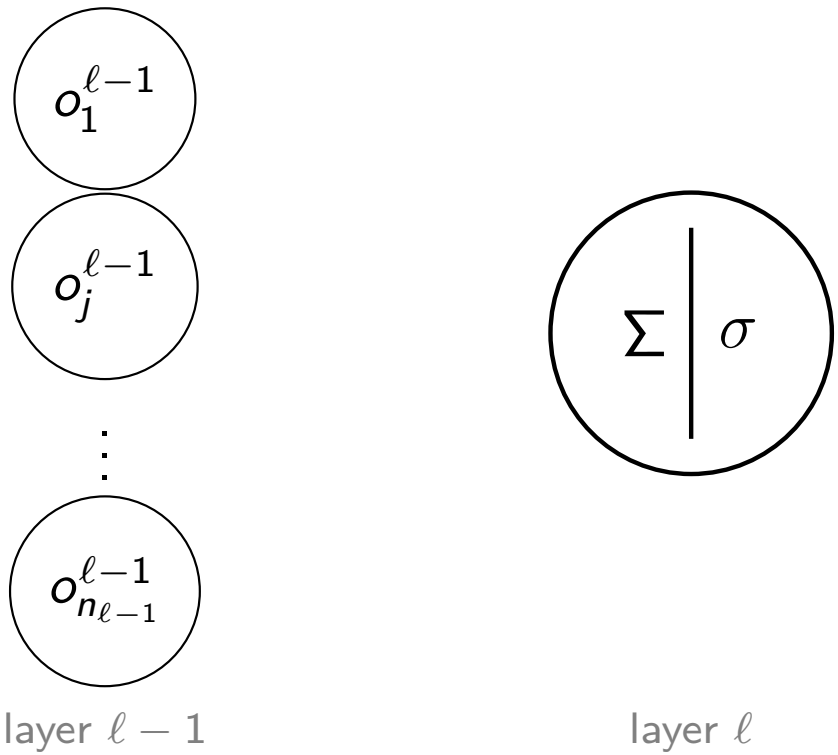
Compact notation for MLPs

We will use MLPs often, so let's fix a standard and concise notation.

Compact notation for MLPs

We will use MLPs often, so let's fix a standard and concise notation.

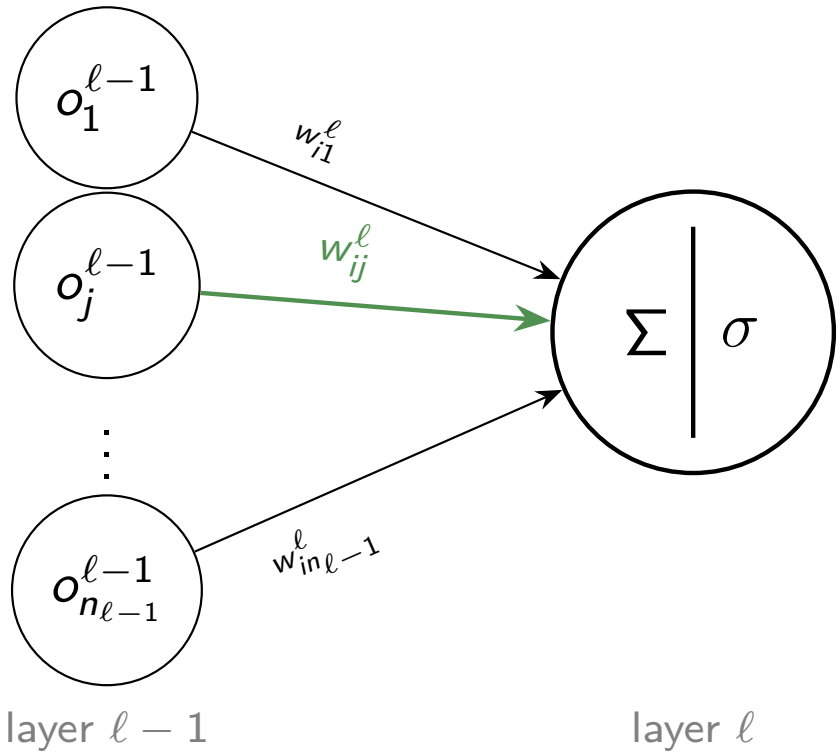
Each neuron i in layer ℓ computes a weighted sum of all outputs from layer $\ell - 1$.



Compact notation for MLPs

We will use MLPs often, so let's fix a standard and concise notation.

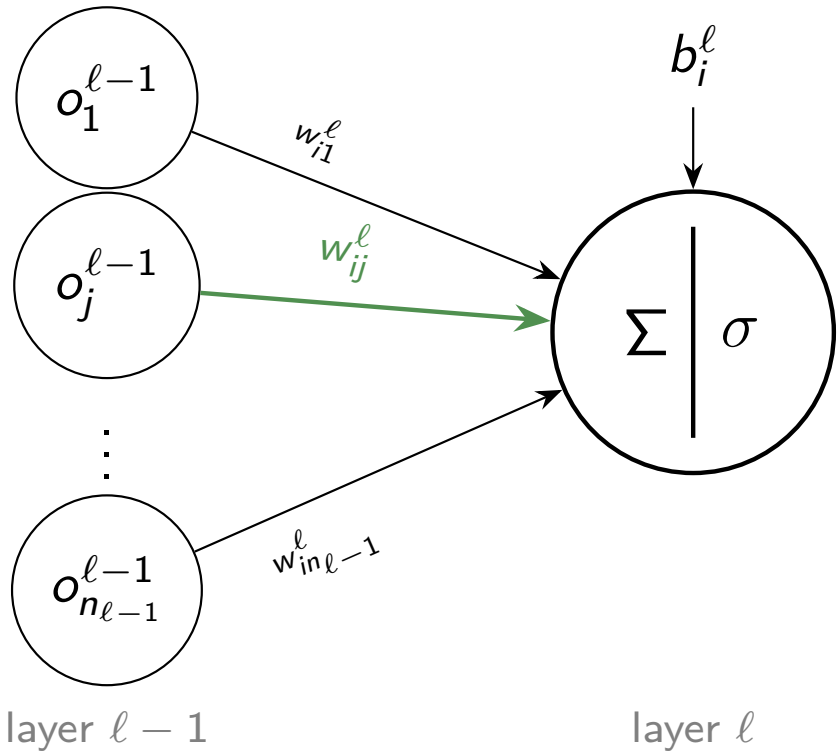
Each neuron i in layer ℓ computes a weighted sum of all outputs from layer $\ell - 1$.



Compact notation for MLPs

We will use MLPs often, so let's fix a standard and concise notation.

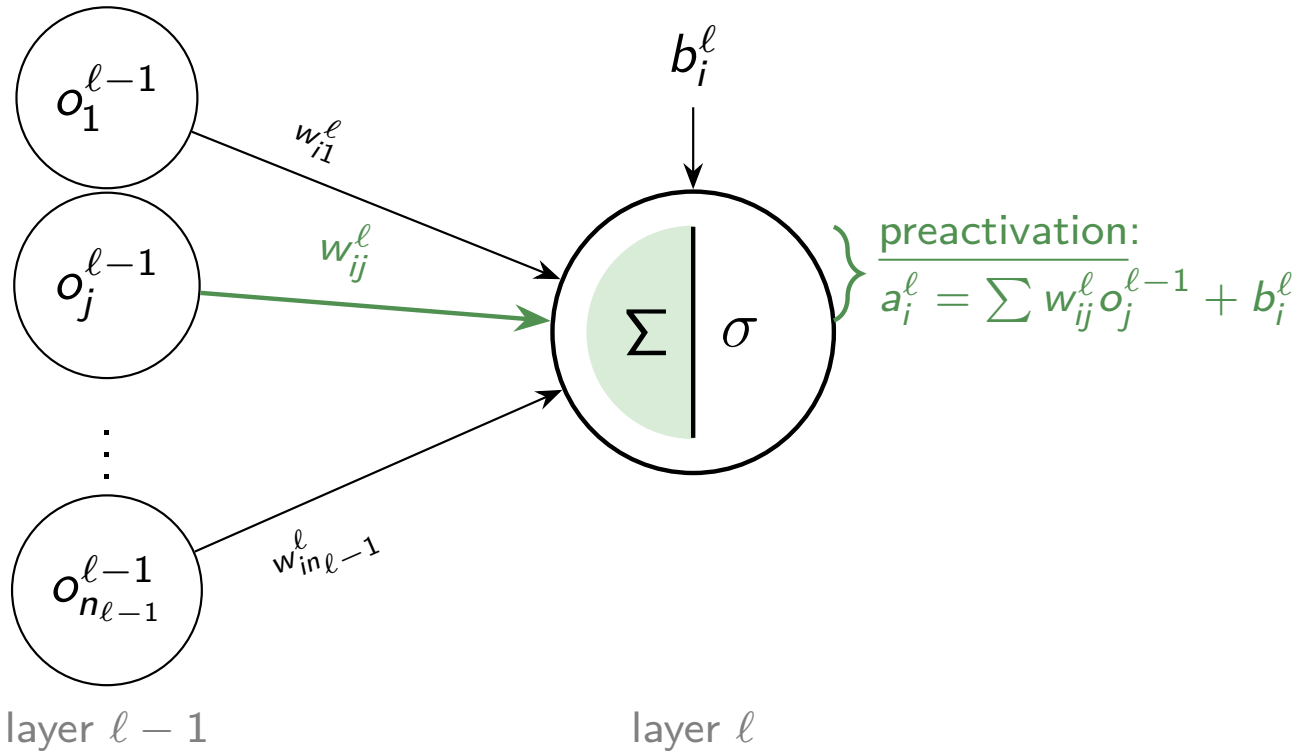
Each neuron i in layer ℓ computes a weighted sum of all outputs from layer $\ell - 1$.



Compact notation for MLPs

We will use MLPs often, so let's fix a standard and concise notation.

Each neuron i in layer ℓ computes a weighted sum of all outputs from layer $\ell - 1$.

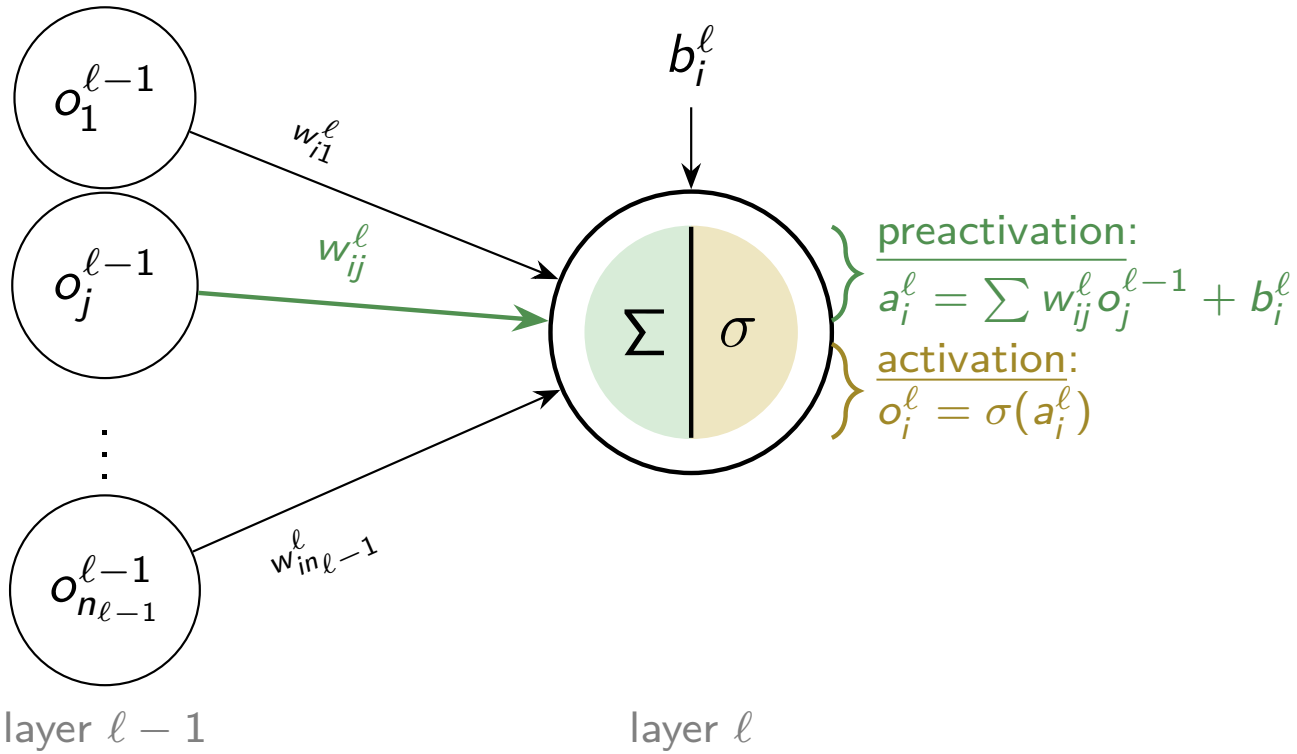


$$a_i^{\ell} = \langle \omega_i^{\ell}, o^{\ell-1} \rangle + b_i^{\ell}$$

Compact notation for MLPs

We will use MLPs often, so let's fix a standard and concise notation.

Each neuron i in layer ℓ computes a weighted sum of all outputs from layer $\ell - 1$.



$$a_i^{\ell} = \langle \omega_i^{\ell}, o^{\ell-1} \rangle + b_i^{\ell}$$

$$o_i^{\ell} = \sigma(a_i^{\ell})$$

The Weight Matrix and Bias Vector

We collect the parameters of layer ℓ into a **weight matrix** W^ℓ and **bias vector** b^ℓ.

Weight matrix $W^\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$:

$$W^\ell = \begin{pmatrix} — \ (\omega_1^\ell)^\top \ — \\ \vdots \\ — \ (\omega_{n_\ell}^\ell)^\top \ — \end{pmatrix} = \begin{pmatrix} w_{11}^\ell & \cdots & w_{1n_{\ell-1}}^\ell \\ \vdots & \ddots & \vdots \\ w_{n_\ell 1}^\ell & \cdots & w_{n_\ell n_{\ell-1}}^\ell \end{pmatrix}$$

The Weight Matrix and Bias Vector

We collect the parameters of layer ℓ into a **weight matrix** W^ℓ and **bias vector** b^ℓ.

Weight matrix $W^\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$:

$$W^\ell = \begin{pmatrix} — & (\omega_1^\ell)^\top & — \\ & \vdots & \\ — & (\omega_{n_\ell}^\ell)^\top & — \end{pmatrix} = \begin{pmatrix} w_{11}^\ell & \cdots & w_{1n_{\ell-1}}^\ell \\ \vdots & \ddots & \vdots \\ w_{n_\ell 1}^\ell & \cdots & w_{n_\ell n_{\ell-1}}^\ell \end{pmatrix}$$

Vectorized computation:

- **Preactivation**
  $a^\ell = W^\ell o^{\ell-1} + b^\ell$
- **Activation**
  $o^\ell = \sigma(a^\ell)$

The Weight Matrix and Bias Vector

We collect the parameters of layer ℓ into a **weight matrix** W^ℓ and **bias vector** b^ℓ.

Weight matrix $W^\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$:

$$W^\ell = \begin{pmatrix} — & (\omega_1^\ell)^\top & — \\ & \vdots & \\ — & (\omega_{n_\ell}^\ell)^\top & — \end{pmatrix} = \begin{pmatrix} w_{11}^\ell & \cdots & w_{1n_{\ell-1}}^\ell \\ \vdots & \ddots & \vdots \\ w_{n_\ell 1}^\ell & \cdots & w_{n_\ell n_{\ell-1}}^\ell \end{pmatrix}$$

Vectorized computation:

- **Preactivation**
  $a^\ell = W^\ell o^{\ell-1} + b^\ell$
- **Activation**
  $o^\ell = \sigma(a^\ell)$

Parameter Count

Total parameters for layer ℓ with n_ℓ neurons and $n_{\ell-1}$ inputs:

- **Weights:** $n_\ell \times n_{\ell-1}$, **Biases:** n_ℓ
- **Total:** $n_\ell(n_{\ell-1}+1)$

Full Expression for an L-Layer MLP

Recursive Definition: Each layer computes an activation o^ℓ from the previous layer's output:

$$o^\ell = \sigma(W^\ell \, o^{l-1} + b^\ell)$$

Full Expression for an L-Layer MLP

Recursive Definition: Each layer computes an activation o^ℓ from the previous layer's output:

$$o^\ell = \sigma(W^\ell \, o^{l-1} + b^\ell)$$

An L-layer MLP computes a nested function:

$$\begin{aligned} f(x;\omega) = o^L &= \sigma(W^L \, o^{L-1} + b^L) \\ &= \sigma\left(W^L \, \sigma(W^{L-1} \, o^{L-2} + b^{L-1}) + b^L\right) \end{aligned}$$

Full Expression for an L-Layer MLP

Recursive Definition: Each layer computes an activation o^ℓ from the previous layer's output:

$$o^\ell = \sigma(W^\ell \, o^{l-1} + b^\ell)$$

An L-layer MLP computes a nested function:

$$\begin{aligned} f(x;\omega) &= o^L = \sigma(W^L \, o^{L-1} + b^L) \\ &= \sigma\left(W^L \, \sigma(W^{L-1} \, o^{L-2} + b^{L-1}) + b^L\right) \\ &= \ldots \text{ (repeating the composition)} \ldots \end{aligned}$$

Full Expression for an L-Layer MLP

Recursive Definition: Each layer computes an activation o^ℓ from the previous layer's output:

$$o^\ell = \sigma(W^\ell \, o^{l-1} + b^\ell)$$

An L-layer MLP computes a nested function:

$$\begin{aligned} f(x;\omega) = o^L &= \sigma(W^L \, o^{L-1} + b^L) \\ &= \sigma\left(W^L \, \sigma(W^{L-1} \, o^{L-2} + b^{L-1}) + b^L\right) \\ &= \ldots \text{ (repeating the composition)} \ldots \\ &= \sigma\left(W^L \, \sigma\left(\ldots \sigma(W^1 x + b^1) \cdots + b^{L-1}\right) + b^L\right) \end{aligned}$$

Full Expression for an L-Layer MLP

Recursive Definition: Each layer computes an activation o^ℓ from the previous layer's output:

$$o^\ell = \sigma(W^\ell \, o^{l-1} + b^\ell)$$

An L-layer MLP computes a nested function:

$$\begin{aligned} f(x;\omega) = o^L &= \sigma(W^L \, o^{L-1} + b^L) \\ &= \sigma\left(W^L \, \sigma(W^{L-1} \, o^{L-2} + b^{L-1}) + b^L\right) \\ &= \ldots \text{ (repeating the composition)} \ldots \\ &= \sigma\left(W^L \, \sigma\left(\ldots \sigma(W^1 x + b^1) \cdots + b^{L-1}\right) + b^L\right) \end{aligned}$$

Parameters of the NN: All weights and biases are collected into the set ω:

$$\omega = \{W^1, \ldots, W^L, \; b^1, \ldots, b^L\}$$

Learning = RERM (Concrete Form for MLPs)

Given a particular architecture, learn the parameters to minimize:

$$\omega^* = \underset{\omega}{\operatorname{argmin}} \; \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i; \omega), \, y_i) + \lambda \, R(\omega)$$

where $f(x_i; \omega)$ is the NN output evaluated at input x_i with parameters ω.

Learning = RERM (Concrete Form for MLPs)

Given a particular architecture, learn the parameters to minimize:

$$\omega^* = \underset{\omega}{\operatorname{argmin}} \; \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i; \omega), \, y_i) + \lambda \, R(\omega)$$

where $f(x_i; \omega)$ is the NN output evaluated at input x_i with parameters ω.

The architecture determines:

- How many parameters there are (via $n_0, n_1, \ldots, n_L$)
- How f depends on ω (via the compositional structure)

Learning = RERM (Concrete Form for MLPs)

Given a particular architecture, learn the parameters to minimize:

$$\omega^* = \operatorname*{argmin}_{\omega} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i; \omega),\, y_i) + \lambda\, R(\omega)$$

where $f(x_i; \omega)$ is the NN output evaluated at input x_i with parameters ω.

The architecture determines:

- How many parameters there are (via $n_0, n_1, \ldots, n_L$)
- How f depends on ω (via the compositional structure)

Training a given architecture = solving this RERM (e.g., with SGD).

Learning = RERM (Concrete Form for MLPs)

Given a particular architecture, learn the parameters to minimize:

$$\omega^* = \operatorname*{argmin}_{\omega} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i; \omega),\, y_i) + \lambda\, R(\omega)$$

where $f(x_i; \omega)$ is the NN output evaluated at input x_i with parameters ω.

The architecture determines:

- How many parameters there are (via $n_0, n_1, \ldots, n_L$)
- How f depends on ω (via the compositional structure)

Training a given architecture = solving this RERM (e.g., with SGD).

Because $f(x; \omega)$ is a composition of nonlinear functions, the objective is **nonconvex** in ω — even when ℓ is convex.

Why Deep Learning Needs More Than Good Models

We have shown NNs are expressive. But expressivity alone does not explain the success of modern deep learning. Making this work requires the simultaneous availability of several factors.

Why Deep Learning Needs More Than Good Models

We have shown NNs are expressive. But expressivity alone does not explain the success of modern deep learning. Making this work requires the simultaneous availability of several factors.

Factor 1 — Expressive Architecture Design:
NN architectures encode **domain-specific inductive biases**:

- **CNNs** (LeCun et al., 1989): translation equivariance for images
- **RNNs/LSTMs** (Hochreiter & Schmidhuber, 1997): sequential structure for time series
- **Transformers** (Vaswani et al., 2017): long-range attention for sequences
- **GNNs**: message-passing on graph-structured data

Why Deep Learning Needs More Than Good Models

We have shown NNs are expressive. But expressivity alone does not explain the success of modern deep learning. Making this work requires the simultaneous availability of several factors.

Factor 1 — Expressive Architecture Design:
NN architectures encode **domain-specific inductive biases**:

- **CNNs** (LeCun et al., 1989): translation equivariance for images
- **RNNs/LSTMs** (Hochreiter & Schmidhuber, 1997): sequential structure for time series
- **Transformers** (Vaswani et al., 2017): long-range attention for sequences
- **GNNs**: message-passing on graph-structured data

Without the right architecture, a generic MLP would need astronomically many parameters to learn the same function.

Factor 2 — Large-Scale Data

Modern NNs are data-hungry: performance scales with dataset size in ways classical methods cannot match.

Factor 2 — Large-Scale Data

Modern NNs are data-hungry: performance scales with dataset size in ways classical methods cannot match.

Key enabling datasets:

- **ImageNet** (Deng et al., 2009): 14M+ labeled images $\rightarrow$ enabled the AlexNet breakthrough
- **Common Crawl / web-scale text**: enabled GPT, BERT, and modern LLMs
- **Domain-specific:** medical imaging, protein structure (AlphaFold), autonomous driving

Factor 2 — Large-Scale Data

Modern NNs are data-hungry: performance scales with dataset size in ways classical methods cannot match.

Key enabling datasets:

- **ImageNet** (Deng et al., 2009): 14M+ labeled images $\rightarrow$ enabled the AlexNet breakthrough
- **Common Crawl / web-scale text**: enabled GPT, BERT, and modern LLMs
- **Domain-specific:** medical imaging, protein structure (AlphaFold), autonomous driving

The empirical observation that performance improves as a power law in dataset size (**scaling laws**; Kaplan et al., 2020) is one of the most striking phenomena in modern ML.

Factor 3 — Compute

Training modern NNs requires enormous computational resources:

- **GPUs:** massively parallel matrix multiplications
- **TPUs and custom accelerators:** designed for NN workloads
- **Distributed training:** model and data parallelism across thousands of devices

Factor 3 — Compute

Training modern NNs requires enormous computational resources:

- **GPUs:** massively parallel matrix multiplications
- **TPUs and custom accelerators:** designed for NN workloads
- **Distributed training:** model and data parallelism across thousands of devices

The key enabling technology was **CUDA** (Nvidia, 2006), making GPUs programmable for general-purpose computation. Without GPU acceleration, training AlexNet would have taken months instead of days.

Factor 3 — Compute

Training modern NNs requires enormous computational resources:

- **GPUs:** massively parallel matrix multiplications
- **TPUs and custom accelerators:** designed for NN workloads
- **Distributed training:** model and data parallelism across thousands of devices

The key enabling technology was **CUDA** (Nvidia, 2006), making GPUs programmable for general-purpose computation. Without GPU acceleration, training AlexNet would have taken months instead of days.

With the parameter counts from Part 3 in hand, the need for large data (to constrain all those parameters) and massive compute (to process the resulting optimization problem) should feel concrete.

What We Learned

1. The **RERM template** carries over unchanged: same losses, same estimation principles. Only $f(x; \omega)$ changes.

What We Learned

1. The **RERM template** carries over unchanged: same losses, same estimation principles. Only $f(x; \omega)$ changes.
2. A neural network is a **computational DAG** of nonlinear neurons. FCFFNs/MLPs are the layered special case.

What We Learned

1. The **RERM template** carries over unchanged: same losses, same estimation principles. Only $f(x;\omega)$ changes.
2. A neural network is a **computational DAG** of nonlinear neurons. FCFFNs/MLPs are the layered special case.
3. **Nonlinearity + depth** gives expressivity (UAT), but expressivity $\neq$ efficiency. Depth separation shows depth is fundamentally more parameter-efficient.

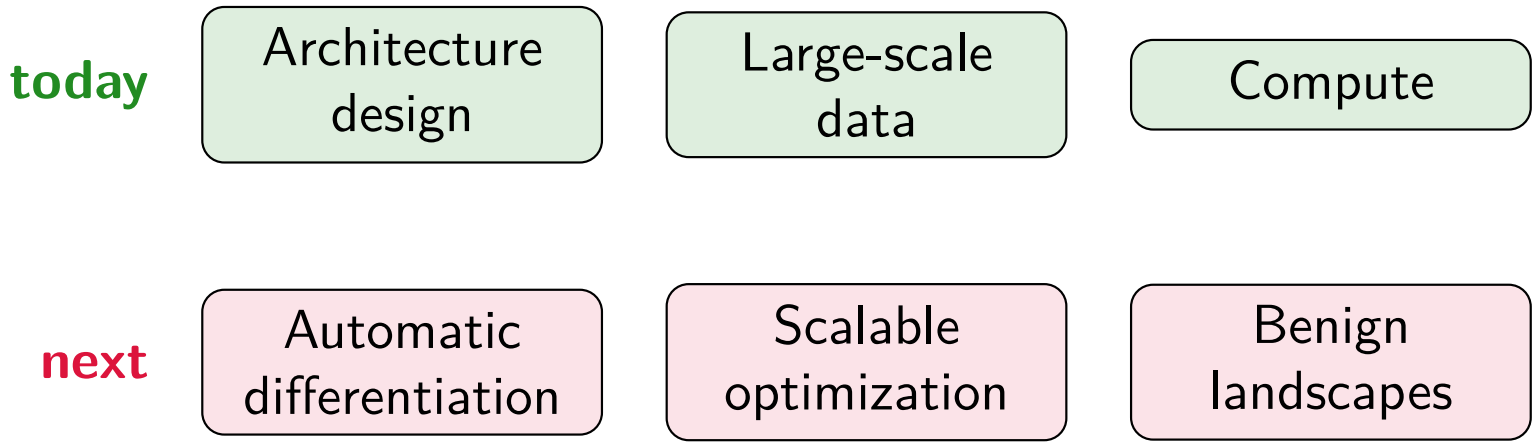
What We Learned

1. The **RERM template** carries over unchanged: same losses, same estimation principles. Only $f(x;\omega)$ changes.
2. A neural network is a **computational DAG** of nonlinear neurons. FCFFNs/MLPs are the layered special case.
3. **Nonlinearity + depth** gives expressivity (UAT), but expressivity $\neq$ efficiency. Depth separation shows depth is fundamentally more parameter-efficient.
4. The RERM for MLPs is a **nonconvex** optimization problem. Parameter counts grow as $\sum_{\ell}(n_{\ell} \cdot n_{\ell-1} + n_{\ell})$.

What We Learned

1. The **RERM template** carries over unchanged: same losses, same estimation principles. Only $f(x;\omega)$ changes.
2. A neural network is a **computational DAG** of nonlinear neurons. FCFFNs/MLPs are the layered special case.
3. **Nonlinearity + depth** gives expressivity (UAT), but expressivity $\neq$ efficiency. Depth separation shows depth is fundamentally more parameter-efficient.
4. The RERM for MLPs is a **nonconvex** optimization problem. Parameter counts grow as $\sum_\ell (n_\ell \cdot n_{\ell-1} + n_\ell)$.
5. Three of six factors enabling deep learning: **architecture**, **data**, **compute**. Three more next time.

What's Next



Next time: How to *train* these models. The RERM is nonconvex — what can we guarantee? We will develop convergence theory for GD/SGD on nonconvex objectives and confront overparameterization and double descent.

Questions?

Next time: training neural networks