Lecture 2: Neural Networks

Block 4.2 Advanced AI

Eoin O’Brien

Shallow neural networks

Shallow neural networks

- Linear regression gave us a useful starting point
- But a linear model can only describe one global linear relationship
- Neural networks combine:
  - linear transformations
  - nonlinear activation functions
  - further linear combinations
- This gives us a much richer family of functions

$$\mathbf{y} = f(\mathbf{x}; \boldsymbol{\phi})$$

The question for today

How can a network built mostly from linear operations represent a nonlinear function?

We will answer this in four views:

- algebraically
- geometrically
- architecturally
- in terms of expressive capacity

From a line to a flexible function

A 1D linear regression model has the form

$$y = \phi_0 + \phi_1 x$$

- Two parameters determine slope and intercept
- The slope is constant everywhere
- More data can improve our estimate of the parameters
  - But more data does **not** make the model family more expressive
  - A line is a line, no matter how you rotate or translate it

What changes in a neural network?

- Instead of fitting one global line...
  - We construct a function from multiple linear pieces.
- Each hidden unit contributes one simple nonlinear component.
  - Via the **activation function**
- The output layer combines those components into the final prediction.

Our first neural network

Start with:

- one scalar input x
- one scalar output y
- one hidden layer
- three hidden units
- ReLU as the activation function

$$y = \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11}x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21}x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31}x]$$

Parameters

The parameter set is

$$\boldsymbol{\phi} = \phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}$$

But the forward computation is simple:

1. compute three affine functions
2. pass them through ReLU
3. weight and sum the results

A family of functions

The equation does not describe just one function.

$$y = \mathrm{f}[x, \boldsymbol{\phi}]$$

- $\boldsymbol{\phi}$ selects one member of a **family of functions**
- changing $\boldsymbol{\phi}$ changes:
  - the positions of the joints
  - the slopes between joints
  - the overall vertical position

The architecture defines the family; the parameters choose a particular function from that family.

Inference

If the parameters are known, prediction is straightforward.

Given an input x and fixed parameters $\boldsymbol{\phi}$:

$$\hat{y} = \mathrm{f}[x, \boldsymbol{\phi}]$$

We simply evaluate the network from left to right.

This forward evaluation is **inference**.

Training data and loss

Suppose we have a supervised training dataset

$$\mathcal{D} = (x_i, y_i)_{i=1}^{I}$$

As with linear regression, we need a loss function that measures how well a particular choice of parameters fits the data.

For least squares:

$$\mathcal{L}[\boldsymbol{\phi}] = \sum_{i=1}^{I} (y_i - \mathrm{f}[x_i, \boldsymbol{\phi}])^2$$

Training the network

Training means searching for parameter values that minimise the loss.

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\arg\min}\, \mathcal{L}[\boldsymbol{\phi}]$$

So the supervised-learning framework has not changed:

- define a parameterised family of functions
- define a loss
- search for parameters that minimise that loss

What has changed is the **expressive power of the function family**.

Three stages of computation

$$x \longrightarrow \text{affine transformation} \longrightarrow \text{ReLU} \longrightarrow \text{linear combination} \longrightarrow y$$

The nonlinearity in the middle is the key ingredient.

The ReLU activation

The rectified linear unit is

$$\mathrm{ReLU}(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

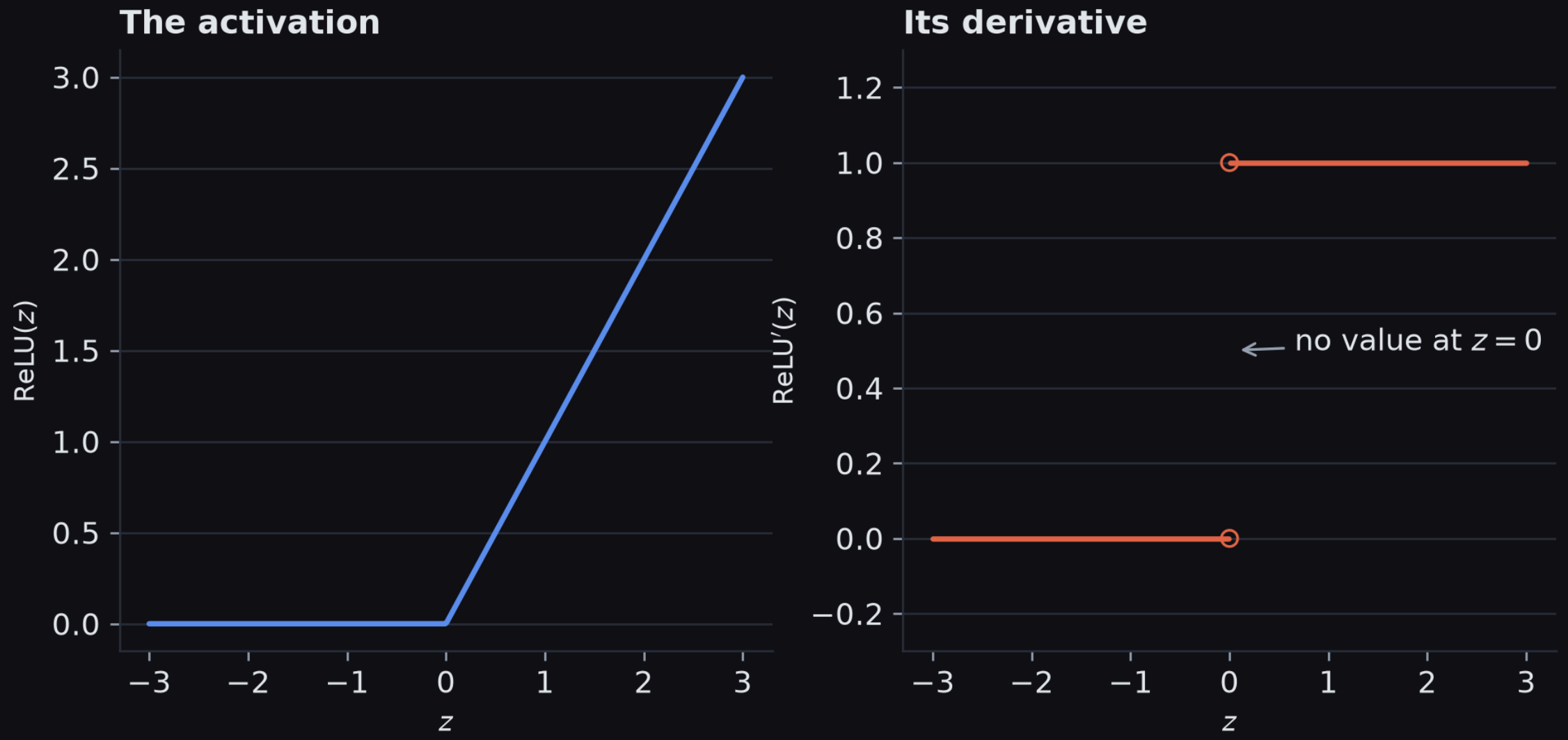
Equivalently,

$$\mathrm{ReLU}(z) = \max(0, z)$$

Note that ReLU is continuous but not differentiable everywhere!

- $\mathrm{ReLU}'(0)$ is 1 when approached from the right
- ...but 0 when approached from the left
- In practice, we assign it either 0 or 1

ReLU and its derivative



ReLU and its derivative over $z \in [-3, 3]$.

ReLU as a clipping operation

ReLU:

- clips negative inputs to zero
- leaves positive inputs unchanged
- is continuous
- is piecewise linear
- introduces a change in slope at $z = 0$

That change in slope is what makes the hidden unit useful.

One hidden unit

Consider

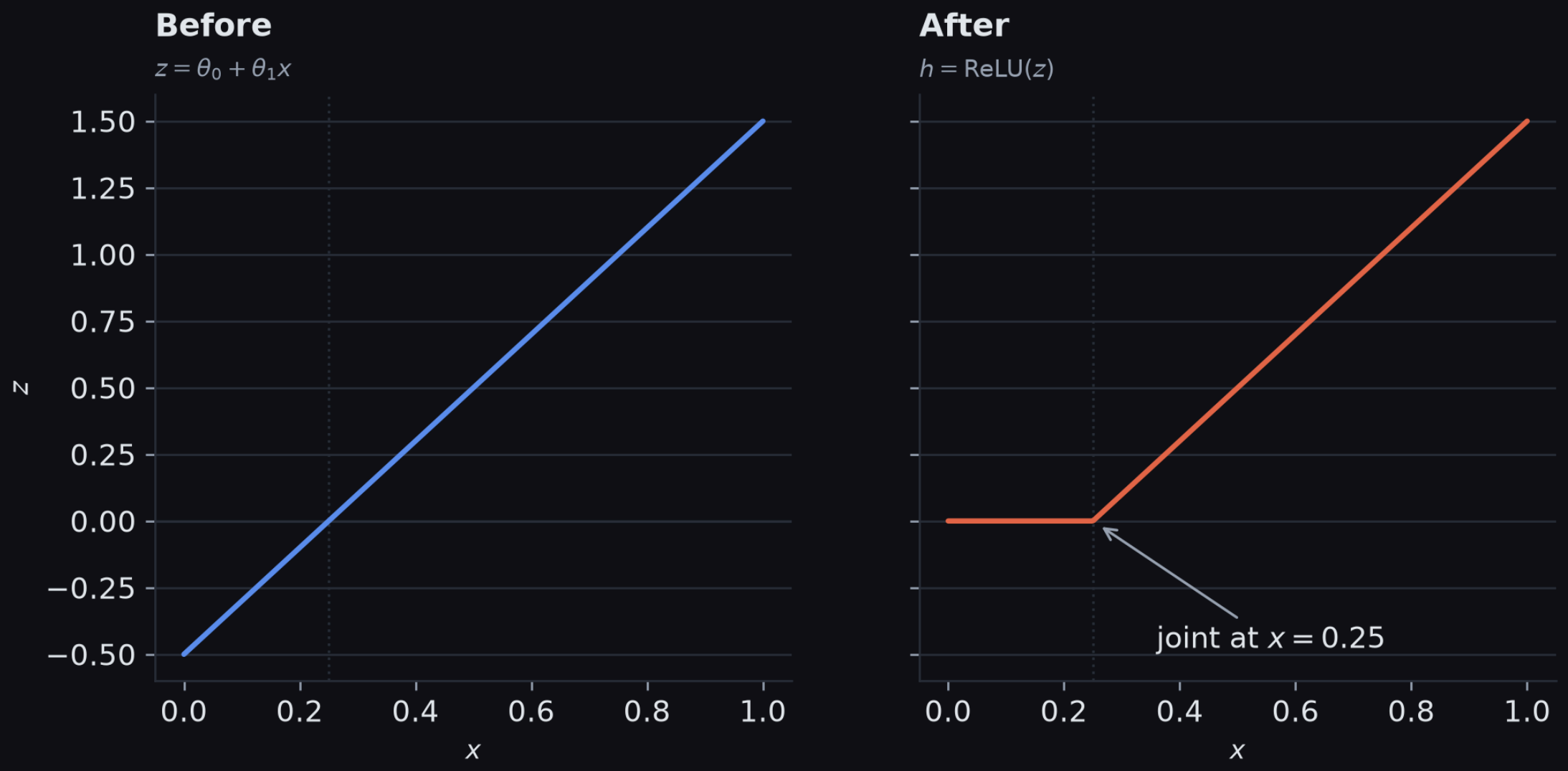
$$h = \mathrm{ReLU}(\theta_0 + \theta_1 x)$$

Before ReLU, we have an affine function:

$$z = \theta_0 + \theta_1 x$$

After ReLU, the negative part is clipped to zero.

Before and after the activation



One unit before and after the activation. The dotted line marks the joint on both panels.

ReLU introduces a joint

The hidden unit has two regimes:

$$h = \begin{cases} 0, & \theta_0 + \theta_1 x < 0 \\ \theta_0 + \theta_1 x, & \theta_0 + \theta_1 x \geq 0 \end{cases}$$

The point where the regime changes becomes a **joint** in the function.

Where is the joint?

The joint occurs when the pre-activation is zero:

$$\theta_0 + \theta_1 x = 0$$

So

$$x = -\frac{\theta_0}{\theta_1}$$

provided $\theta_1 \neq 0$.

What do the parameters do?

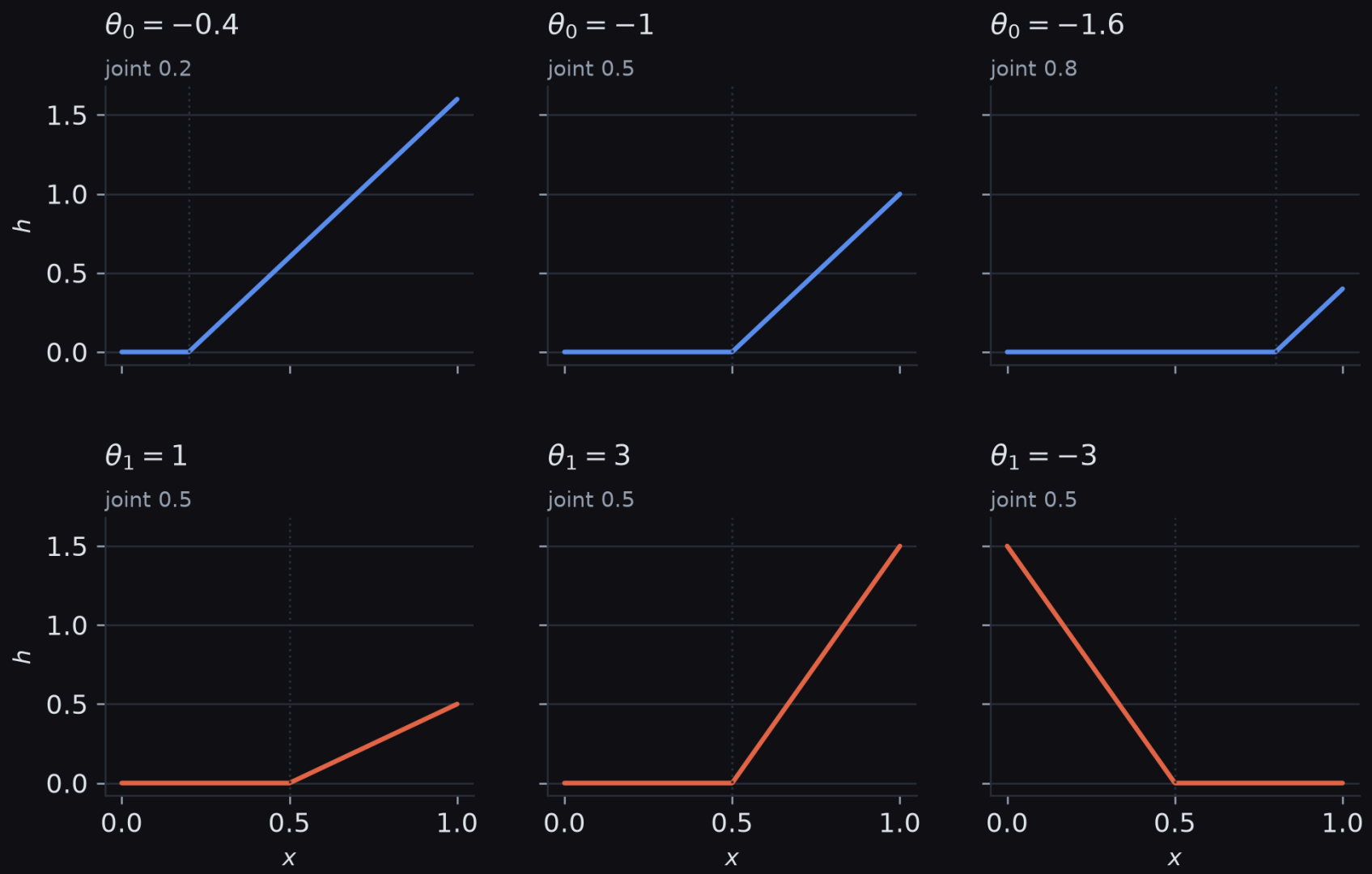
For

$$h = \mathrm{ReLU}(\theta_0 + \theta_1 x)$$

- θ_0 helps control **where** the unit switches on
- θ_1 controls the slope and direction of the underlying line

The hidden unit therefore has both a location and a shape.

Each parameter in isolation



Top row: θ_0 varied at fixed θ_1. Bottom row: θ_1 varied with the joint held at 0.5.

Hidden units

For the three-unit network, define

$$h_1 = \mathrm{ReLU}(\theta_{10} + \theta_{11}x)$$

$$h_2 = \mathrm{ReLU}(\theta_{20} + \theta_{21}x)$$

$$h_3 = \mathrm{ReLU}(\theta_{30} + \theta_{31}x)$$

These intermediate quantities are the **hidden units**.

The output layer

The hidden units are combined as

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

- ϕ_d scales the contribution from hidden unit d
- ϕ_0 shifts the overall output vertically

A useful mental model

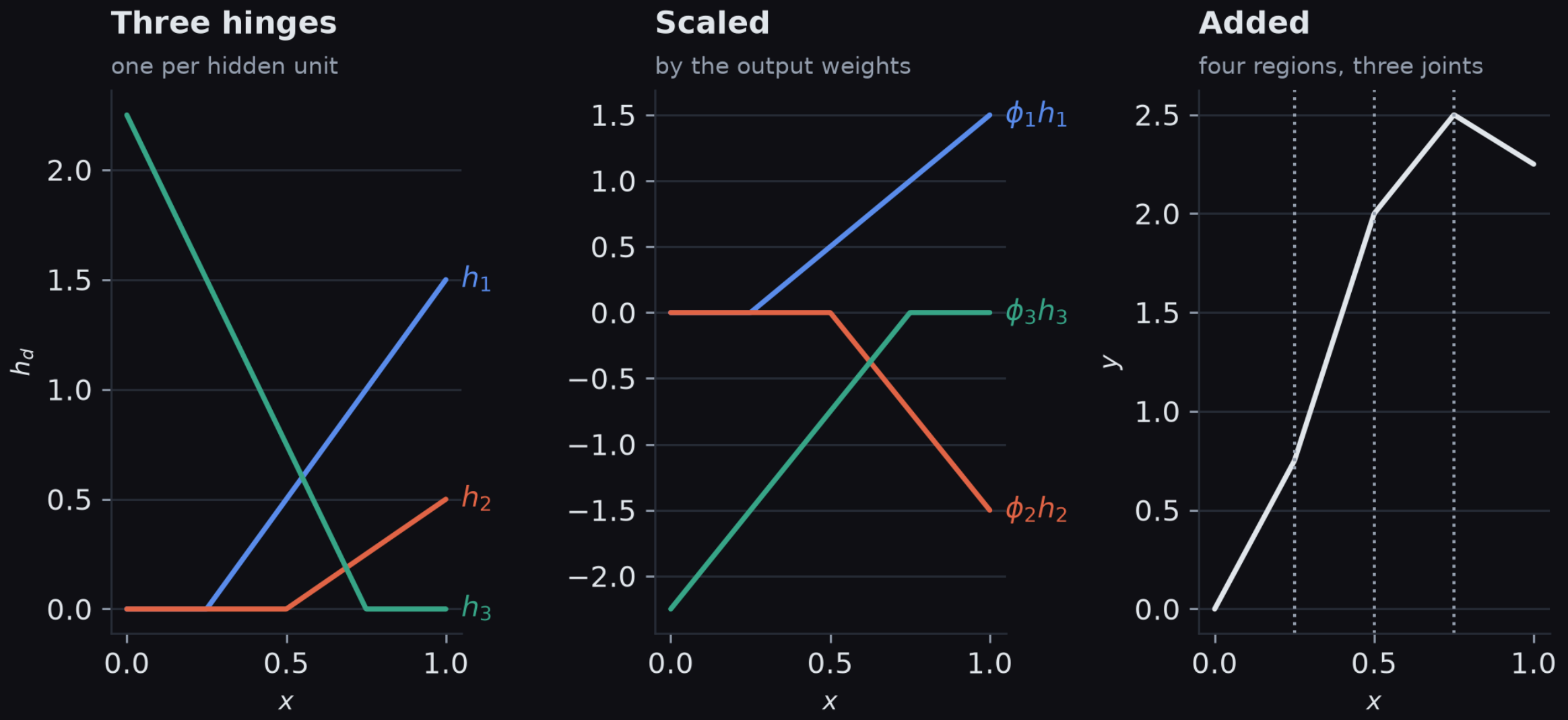
Each hidden unit contributes a **hinged line**.

$$h_d(x) = \mathrm{ReLU}(\theta_{d0} + \theta_{d1} x)$$

The network:

- creates several hinges
- scales them
- adds them together
- adds a final offset

Three hinges and their sum



Three hidden units, the same three scaled by their output weights, and their sum.

Why the result is piecewise linear

Each ReLU hidden unit is piecewise linear.

Within any region where the same hidden units remain active:

- each active hidden unit is affine
- each inactive hidden unit contributes zero
- the weighted sum is therefore affine

The network is affine within a region, but different affine functions apply in different regions.

Active and inactive units

For a hidden unit

$$h_d = \mathrm{ReLU}(z_d)$$

we call it:

- **inactive** when $z_d < 0$
- **active** when $z_d > 0$

At $z_d = 0$, the unit lies exactly on its activation boundary.

Activation patterns

Suppose in one interval:

- h_1 is active
- h_2 is inactive
- h_3 is active

Then

$$h_2 = 0$$

and only h_1 and h_3 influence the local slope.

The local slope

If h_1 and h_3 are active,

$$y = \phi_0 + \phi_1(\theta_{10} + \theta_{11}x) + \phi_3(\theta_{30} + \theta_{31}x)$$

Collecting the terms in x gives

$$\frac{dy}{dx} = \theta_{11}\phi_1 + \theta_{31}\phi_3$$

Not every regional slope is independent

Three hidden units can create four linear regions, but that does **not** give four freely chosen slopes.

- Each hidden unit contributes one slope term when active
- The slope in a region is the sum of contributions from its active units
- If all hidden units are inactive, the slope is 0
- For three hidden units, only three slope contributions are independently parameterised

This is an early example of an important idea:

> the geometry produced by a network is constrained by the way its parameters are shared across regions.

Activation patterns define linear regions

Each distinct active/inactive pattern defines one region.

Within that region:

$$y = ax + b$$

for some region-specific a and b.

Within each region, the network is still affine.

- The nonlinearity comes from switching between different affine functions as the input moves between regions.

Three hidden units, four regions

Each scalar ReLU hidden unit can introduce one joint.

With three hidden units:

- at most three distinct joints
- therefore at most four linear regions

$$D = 3 \quad \Longrightarrow \quad \text{at most } D + 1 = 4 \text{ regions}$$

Why “at most”?

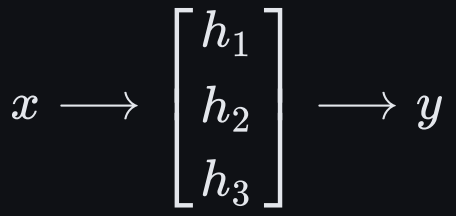
Two hidden units might:

- switch at the same point
- never switch over the domain of interest
- contribute slopes that cancel

So adding a hidden unit creates **capacity** for another region, but doesn’t guarantee that every region is used.

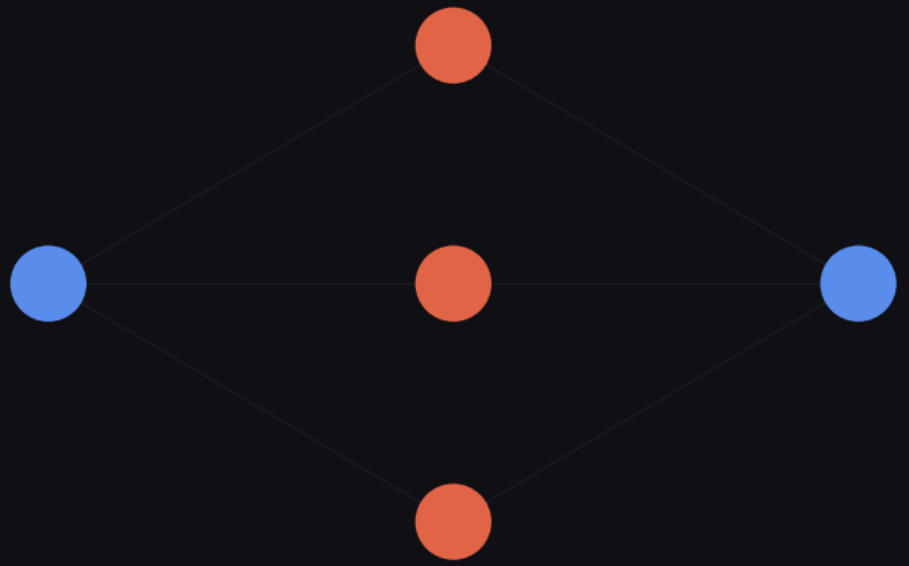
The network as a computation graph

The same equations can be depicted as a network.



- input on the left
- hidden units in the middle
- output on the right
- arrows represent learned connections

The computation graph



The three-unit network of the preceding equations.

What is actually on an arrow?

A connection corresponds to multiplication by a parameter.

For example,

$$\phi_1 h_1$$

means:

- source value: h_1
- connection weight: ϕ_1
- contribution to the target: $\phi_1 h_1$

What happens to biases in diagrams?

A bias can be represented by adding a constant input of 1.

$$\phi_0 \cdot 1 = \phi_0$$

But neural-network diagrams usually omit these constant nodes and draw only the main computational structure.

Why draw the network?

The graphical representation becomes useful when the algebra grows large.

It lets us see:

- which variables feed which units
- how layers are connected
- where activations occur
- how computation flows

The diagram does not define a different model; it is another representation of the same equations.

Increasing capacity

Generalise to D hidden units:

$$h_d = \text{ReLU}(\theta_{d0} + \theta_{d1}x), \qquad d = 1, \ldots, D$$

$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d$$

Width as capacity

For one scalar input:

- each hidden unit can add one joint
- D hidden units give at most D joints
- therefore at most $D + 1$ linear regions

$$\text{maximum number of regions} = D + 1$$

This gives a simple geometric interpretation of **width**.

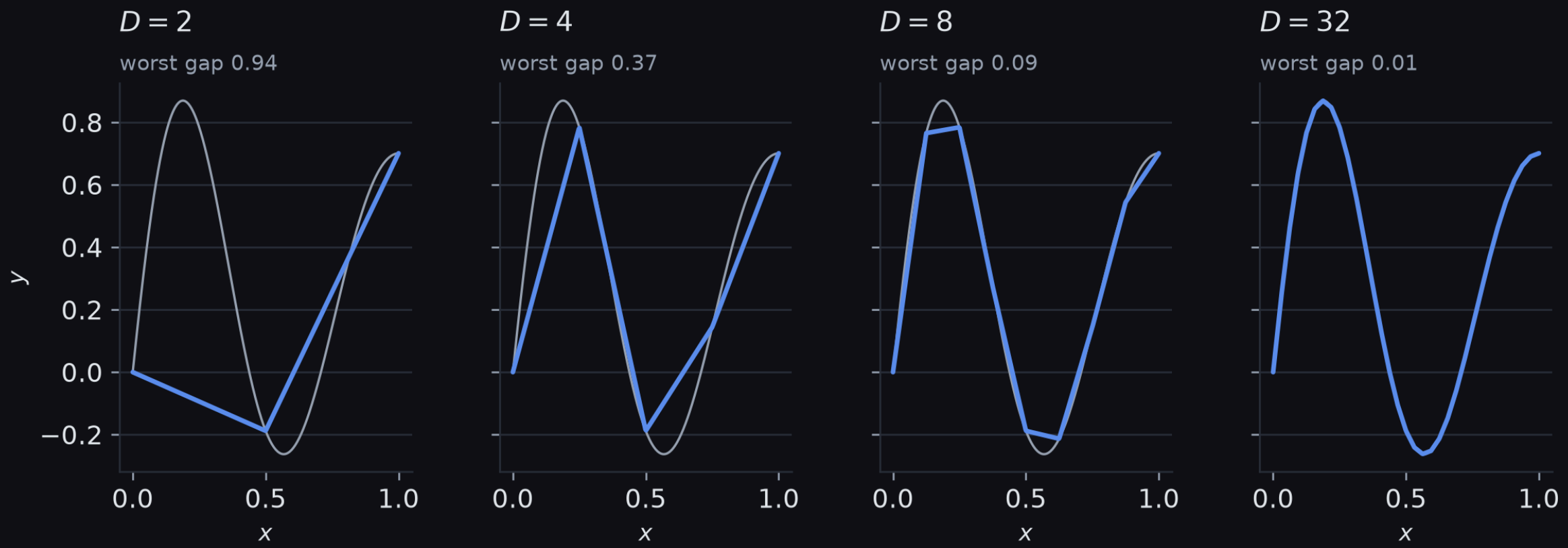
Approximating a curved function

Imagine approximating a smooth curve with straight segments.

- One line: crude
- A few segments: better
- Many short segments: increasingly accurate

A ReLU network constructs exactly this kind of piecewise-linear approximation.

Approximation at four widths



One target approximated by networks of width 2, 4, 8 and 32. The joints are placed evenly and the network matches the target at them; no optimiser is involved.

Reminder: what is a continuous function?

Informally, a function is **continuous** if its output does not suddenly jump when we make a sufficiently small change to its input.

A useful mental picture: you can draw a continuous 1D function without lifting your pen from the page.

More formally, continuity at x means that inputs sufficiently close to x produce outputs arbitrarily close to $f(x)$.

Continuity: the mathematical idea

For a scalar function f, continuity at x can be written

$$\forall \varepsilon > 0,\ \exists \delta > 0 \text{ such that } |x' - x| < \delta \Longrightarrow |f(x') - f(x)| < \varepsilon$$

You do **not** need to manipulate or memorise this definition.

- You **do** need to understand what a continuous function is!

The important points are:

- small input changes $\rightarrow$ small output changes
- no jumps or breaks
- ReLU itself is continuous, even though its slope changes at 0

Universal approximation

The universal approximation theorem is fundamentally an **existence result**.

For a suitable continuous target function on a compact domain, and for any desired approximation tolerance, there exists a sufficiently wide shallow network that can approximate that function while achieving that tolerance.

$$\forall \varepsilon > 0, \qquad \exists\, f_\phi \text{ such that } \|f - f_\phi\| < \varepsilon$$

Weasel words?

Universal approximation says:

- sufficiently wide shallow networks are extremely expressive
- they can approximate suitable continuous functions arbitrarily closely

It does **not** say:

- what width is required to approximate a specific function
- the representation is efficient
- training will find the required parameters
- the learned network will generalise

Universal approximation: study definition

A useful version to remember is:

> For any continuous function on a compact subset of $\mathbb{R}^n$, and for any specified accuracy, there exists a neural network with one hidden layer and a finite number of hidden units that approximates that function to the specified accuracy.

Two phrases matter:

- **continuous function**
- **compact domain**

For our purposes, think of a compact domain as a closed and bounded region of input space, such as a finite interval $[a, b]$.

Universal approximation: historical note

The theorem was established in several forms.

- **Cybenko (1989)** (Cybenko 1989): proved a universal approximation result for one-hidden-layer networks using a class of sigmoid activations
- **Hornik (1991)** (Hornik 1991): established results for a broader class of nonlinear activation functions

The theorem is not specifically a theorem about ReLU, but ReLU networks also possess universal approximation results.

Representation versus learning

There are (unfortunately) two distinct questions:

1. **Can this architecture represent the function?**
2. **Can our learning algorithm find good parameters?**

Universal approximation answers the first question, not the second.

Moving beyond scalar outputs

So far, we've discussed mapping scalar input to scalar output

$$x \in \mathbb{R} \qquad \longrightarrow \qquad y \in \mathbb{R}$$

More generally, we want to consider an arbitrary number of inputs mapped to an arbitrary number of outputs.

$$\mathbf{x} \in \mathbb{R}^{D_i} \qquad \longrightarrow \qquad \mathbf{y} \in \mathbb{R}^{D_o}$$

We can generalise the input and output sides separately.

Multiple outputs

Keep scalar x, but predict two outputs.

Let four hidden units be shared:

$$h_d = \mathrm{ReLU}(\theta_{d0} + \theta_{d1}x), \qquad d = 1, \ldots, 4$$

Each output is a different linear combination of the same hidden representation.

Two outputs from shared hidden units

$$y_1 = \phi_{10} + \sum_{d=1}^{4} \phi_{1d} h_d$$

$$y_2 = \phi_{20} + \sum_{d=1}^{4} \phi_{2d} h_d$$

The outputs share the hidden units but have different output weights.

Shared hidden units imply shared joints

The joint locations are determined by

$$h_d = \mathrm{ReLU}(\theta_{d0} + \theta_{d1} x)$$

Therefore:

- y_1 and y_2 have joints at the same input positions
- their slopes can differ
- their vertical offsets can differ

The hidden layer forms a shared representation used by all outputs.

Multivariate inputs

Now let

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2$$

A hidden unit receives a linear combination of both inputs:

$$h_d = \mathrm{ReLU}(\theta_{d0} + \theta_{d1} x_1 + \theta_{d2} x_2)$$

Pre-activation with two inputs

Define

$$z_d = \theta_{d0} + \theta_{d1}x_1 + \theta_{d2}x_2$$

Before ReLU, this is a plane over the (x_1, x_2) input space.

- θ_{d1} controls change along x_1
- θ_{d2} controls change along x_2
- θ_{d0} controls the offset from (0, 0)
- Remember, a plane is the 2D analogue of a line

The activation boundary

The hidden unit changes regime where

$$\theta_{d0} + \theta_{d1} x_1 + \theta_{d2} x_2 = 0$$

In a two-dimensional input space, this equation defines a **line**.

ReLU clips the plane

Apply

$$h_d = \mathrm{ReLU}(z_d)$$

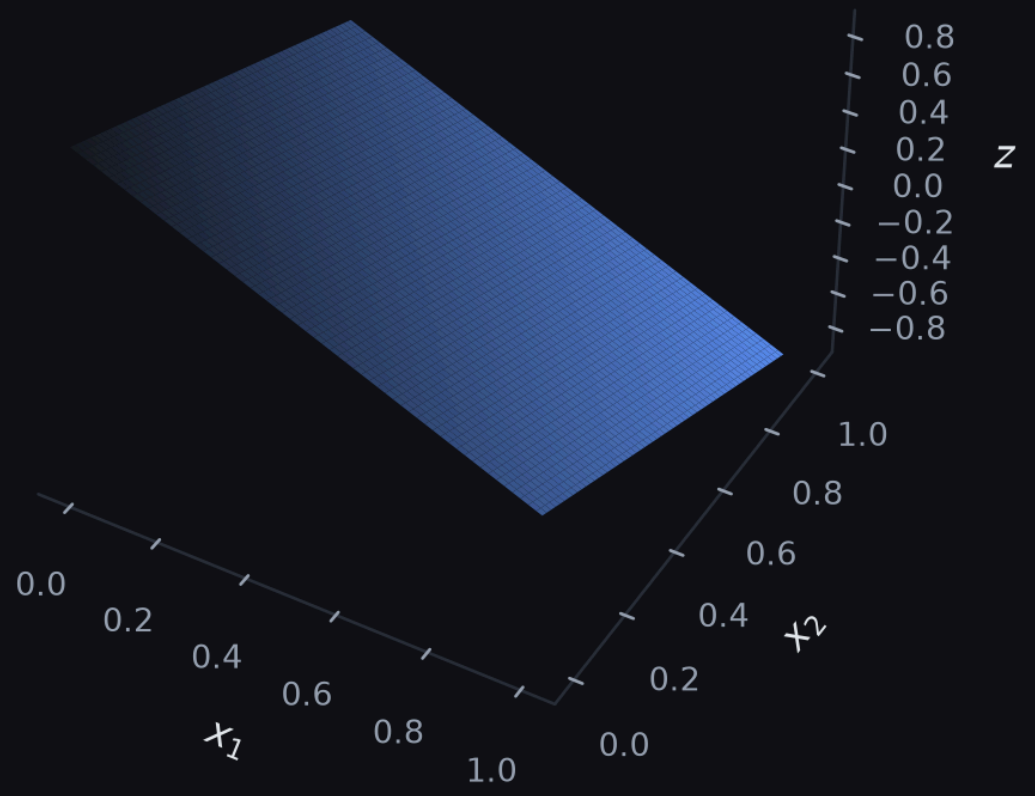
One side of the boundary is clipped to zero.

The other side retains the affine plane.

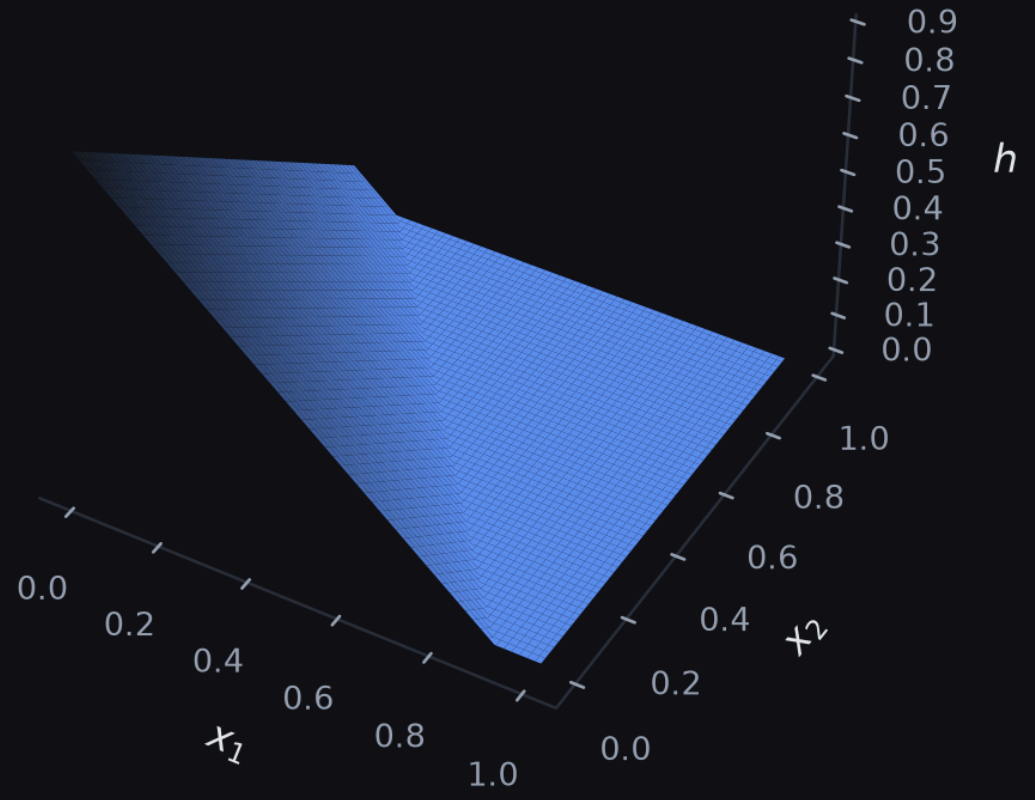
So the 1D “joint” generalises to a higher-dimensional activation boundary.

The clipped plane

Before



After



One unit over two inputs, before and after the activation.

Combining clipped planes

With three hidden units,

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

The output is a continuous piecewise-linear surface.

The regions are convex

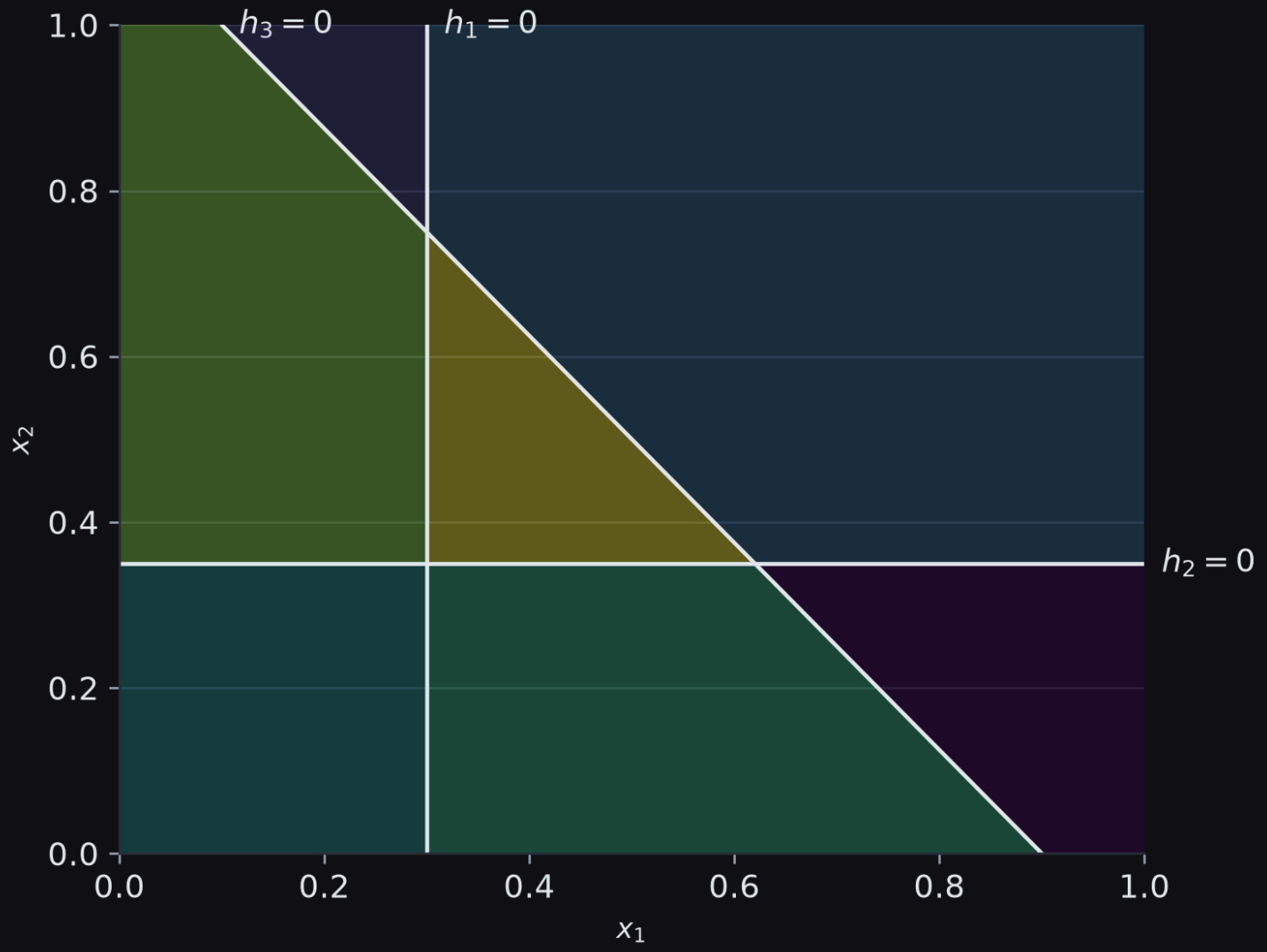
For a shallow ReLU network with two inputs:

- each activation boundary is a line
- intersections of these boundaries divide the input plane
- the resulting activation regions are **convex polygons**

In higher dimensions, these generalise to **convex polytopes**.

A set is convex if the straight line segment between any two points in the set remains entirely inside the set.

The regions in the input plane



Three activation boundaries over the unit square, and the seven regions they cut it into.

Multiple outputs share the same partition

The hidden layer determines the activation boundaries.

Therefore, when a network has several outputs:

- every output sees the **same partition** of the input space
- each region has the same active/inactive hidden-unit pattern for every output
- different outputs may use different affine functions inside those shared regions

The hidden representation determines **where** the pieces are; the output weights determine **what each output does inside them**.

From intervals to polygons

Inputs	Boundaries are	Regions are
one	points	intervals
two	lines	polygons
three	planes	polyhedral cells
more	hyperplanes	polytopes

Activation regions in higher dimensions

For D_i inputs, hidden unit d has boundary

$$\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i = 0$$

Different sides of the hyperplane correspond to different activation states.

Each region has one affine map

Once the active/inactive pattern is fixed, every ReLU is either:

- **inactive**, contributing 0
- **active**, contributing $\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i$

Therefore the full network reduces to an affine map inside that region.

Why dimensionality matters

Suppose we have D_i input dimensions and D_i suitably positioned activation boundaries.

Align one boundary with each coordinate axis.

Then the space is divided into 2^{D_i} orthants.

Examples

$$D_i = 1 \implies 2 \text{ regions}$$

$$D_i = 2 \implies 4 \text{ regions}$$

$$D_i = 3 \implies 8 \text{ regions}$$

Region counts can grow rapidly with input dimension.

General shallow network

Let

$$\mathbf{x} \in \mathbb{R}^{D_i}, \qquad \mathbf{y} \in \mathbb{R}^{D_o}$$

and let the network have D hidden units.

The dth hidden unit is

$$h_d = \mathrm{a}\left[\theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i\right]$$

General output equation

For output j,

$$y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} h_d$$

with

$$j = 1, \ldots, D_o$$

Compact vector notation

We can compress the indexed equations.

First compute the hidden pre-activations:

$$\mathbf{z} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$

Then apply ReLU elementwise:

$$\mathbf{h} = \mathrm{ReLU}(\mathbf{z})$$

Output layer in matrix form

The output layer is

$$\mathbf{y} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2$$

Substituting the hidden layer gives

$$\mathbf{y} = \mathbf{W}_2\,\mathrm{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

Check the dimensions

If

$$\mathbf{x} \in \mathbb{R}^{D_i}, \qquad \mathbf{h} \in \mathbb{R}^{D}, \qquad \mathbf{y} \in \mathbb{R}^{D_o}$$

then

$$\mathbf{W}_1 \in \mathbb{R}^{D \times D_i}, \qquad \mathbf{b}_1 \in \mathbb{R}^{D}$$

$$\mathbf{W}_2 \in \mathbb{R}^{D_o \times D}, \qquad \mathbf{b}_2 \in \mathbb{R}^{D_o}$$

How many parameters?

- **First layer**: $DD_i + D$ parameters
- **Output layer**: $D_oD + D_o$ parameters

$$\text{total} = D(D_i + 1) + D_o(D + 1)$$

Worked parameter count

Consider:

- $D_i = 3$ inputs
- $D = 3$ hidden units
- $D_o = 2$ outputs

Then

$$D(D_i + 1) + D_o(D + 1) = 3(3 + 1) + 2(3 + 1)$$

which is 20 parameters.

Where do the 20 parameters come from?

Weights:

$$3 \times 3 + 2 \times 3 = 15$$

Biases:

$$3 + 2 = 5$$

Therefore:

$$15 \text{ weights} + 5 \text{ biases} = 20 \text{ parameters}$$

Why must the activation be nonlinear?

Suppose we remove ReLU. Then

- **Hidden layer**: $\mathbf{h} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$
- **Output layer**: $\mathbf{y} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2$

Linear layers collapse

Substituting,

$$\mathbf{y} = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

Expanding,

$$\mathbf{y} = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2)$$

One affine map again

Define

- $\mathbf{W} = \mathbf{W}_2\mathbf{W}_1$
- $\mathbf{b} = \mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2$

Then

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Depth without nonlinearity

No matter how many affine layers we compose, the entire model is still affine.

$$f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$$

So extra layers without nonlinear activations add parameters, but not nonlinear expressive power.

The activation is the expressive ingredient

The hidden layer performs

$$\mathbf{x} \longmapsto \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 \longmapsto \mathrm{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

ReLU prevents the sequence of layers from collapsing into a single affine transformation.

Terminology: layers

A shallow network consists of:

- an **input layer**
- one **hidden layer**
- an **output layer**

Definition 1 Shallow neural network: a neural network with one hidden layer.

Definition 2 Deep neural network: a neural network with multiple hidden layers.

Hidden units or neurons

Definition 3 Hidden unit / neuron: one of the nodes in a hidden layer.

For example,

$$\mathbf{h} = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_D \end{bmatrix}$$

contains D hidden units.

The term **neuron** is historical terminology; mathematically, each unit simply computes part of a parameterised function.

Pre-activations and activations

Definition 4 Pre-activation: the value at the hidden unit *before* the activation function is applied.

$$z_d = \theta_{d0} + \sum_i \theta_{di} x_i$$

Definition 5 Activation: the value at the hidden unit *after* the activation function is applied.

$$h_d = \mathrm{a}[z_d]$$

Weights

Definition 6 Weights: the parameters associated with connections between units.

In matrix notation,

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

the entries of $\mathbf{W}$ are weights.

They scale and combine values arriving from the previous layer.

Biases

Definition 7 Biases: the offset parameters in a neural network.

In

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b},$$

the entries of $\mathbf{b}$ are biases.

Biases allow activation boundaries to shift rather than being forced through the origin.

Why the bias matters geometrically

The activation boundary is

$$\mathbf{w}^\top \mathbf{x} + b = 0$$

- $\mathbf{w}$ determines the orientation of the hyperplane
- b shifts the hyperplane through the input space

Without the bias, every boundary would be forced through the origin.

Feed-forward networks

Definition 8 **Feed-forward network**: a neural network in which the connections form an **acyclic graph** (a graph with no loops).

$$\text{input} \longrightarrow \text{hidden} \longrightarrow \text{output}$$

Computation flows from earlier layers to later layers without cycling back.

Fully connected networks

Definition 9 **Fully connected network**: a network in which every element in one layer connects to every element in the next.

If

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b},$$

then every entry W_{ji} represents the connection from source unit x_i to target unit z_j.

In modern software libraries, a fully connected layer is often called a **dense layer**.

The terms usually refer to the same structural idea:

- all inputs are available to all outputs
- each connection has its own learned weight

Multi-layer perceptron

For historical reasons, a neural network with at least one hidden layer is also commonly called a **multi-layer perceptron**, or **MLP**.

The term persists even though modern MLPs usually use activation functions such as ReLU rather than the original perceptron threshold rule.

In contemporary usage, “MLP” usually means a feed-forward network built primarily from fully connected layers.

Shallow versus deep

- **Shallow network**: one hidden layer
- **Deep network**: multiple hidden layers

Depth refers to the number of successive learned transformations, not simply the total number of neurons.

Terminology describes different properties

These labels are not interchangeable:

- **shallow/deep** describes depth
- **feed-forward** describes graph directionality
- **fully connected** describes connectivity
- **MLP** names a broad architectural family

A network can be all of these at once.

Why are they called “neural” networks?

The biological connection is historically important but conceptually limited.

- Network diagrams contain densely connected units
- Biological brains also contain densely connected neurons
- The resemblance is largely superficial

For this major, treat a neural network primarily as a **parameterised mathematical function**, not as a faithful model of biological computation.

A very short history I

McCulloch & Pitts (1943) (McCulloch and Pitts 1943)

- proposed an early mathematical model of an artificial neuron
- combined inputs to produce an output
- did not provide a practical learning algorithm

Rosenblatt (1958) (Rosenblatt 1958)

- developed the perceptron
- linearly combined inputs and thresholded the result
- provided an algorithm for learning its weights

A very short history II

Minsky & Papert (1969; reissued 2017) (Minsky and Papert 2017)

- highlighted important limitations of linear perceptrons
- recognised that hidden layers with nonlinear activations could represent more general input/output relations
- noted that the available perceptron learning rule did not train such multilayer models

1980s

- practical backpropagation methods made multilayer networks trainable
- research activity in neural networks accelerated again

Why ReLU is popular

ReLU is especially easy to analyse.

$$\mathrm{ReLU}(z) = \max(0, z)$$

It gives us:

- simple piecewise-linear geometry
- simple derivatives
- clear active/inactive regimes
- efficient computation

ReLU and optimisation

For $z > 0$,

$$\frac{d}{dz}\mathrm{ReLU}(z) = 1$$

so ReLU does not shrink the local gradient on its positive side.

This contrasts with sigmoid-like functions, whose derivatives can become very small when their inputs have large magnitude.

This behaviour is one reason ReLU became important in the training of modern neural networks.

ReLU derivatives

Away from $z = 0$,

$$\frac{d}{dz}\mathrm{ReLU}(z) = \begin{cases} 0, & z < 0 \\ 1, & z > 0 \end{cases}$$

At $z = 0$, the classical derivative is undefined.

In optimisation software, a conventional value is chosen there.

Why the derivative matters

Training methods such as gradient descent rely on derivatives.

For positive pre-activations:

$$\frac{d}{dz}\mathrm{ReLU}(z) = 1$$

so gradient information passes through unchanged.

For negative pre-activations:

$$\frac{d}{dz}\mathrm{ReLU}(z) = 0$$

so the local gradient is blocked.

The dying ReLU problem

If every training example produces a negative pre-activation for one unit, then

$$h_d = 0$$

and locally,

$$\frac{\partial h_d}{\partial z_d} = 0$$

The incoming weights to that unit may stop receiving useful gradient updates.

Leaky ReLU

One modification is to preserve a small negative-side slope:

$$\mathrm{LeakyReLU}(z) = \begin{cases} \alpha z, & z < 0 \\ z, & z \geq 0 \end{cases}$$

with small $\alpha > 0$.

Negative inputs are attenuated rather than completely clipped.

Parametric ReLU

A **parametric ReLU** has the same basic form as leaky ReLU,

$$\mathrm{PReLU}(z) = \begin{cases} \alpha z, & z < 0 \\ z, & z \geq 0, \end{cases}$$

but α is treated as a **learned parameter** rather than a fixed constant.

Concatenated ReLU

A **concatenated ReLU** keeps information from both signs of the input by producing two outputs:

$$\mathrm{CReLU}(z) = \begin{bmatrix} \mathrm{ReLU}(z) \\ \mathrm{ReLU}(-z) \end{bmatrix}$$

One component clips below zero; the other captures the corresponding negative-side magnitude.

Smooth activation functions

A variety of smooth alternatives to ReLU have been proposed, including:

- **SoftPlus**
- **GELU**: Gaussian error linear unit
- **SiLU**: sigmoid linear unit
- **ELU**: exponential linear unit
- **SELU**: scaled exponential linear unit
- **Swish**

Most attempt, in different ways, to retain useful gradients for negative inputs while controlling the size and behaviour of activations.

Swish

A common form of Swish is

$$\text{Swish}(z) = \frac{z}{1 + e^{-\beta z}}$$

where β may be fixed or learned.

Swish is smooth and does not hard-clip all negative inputs to zero.

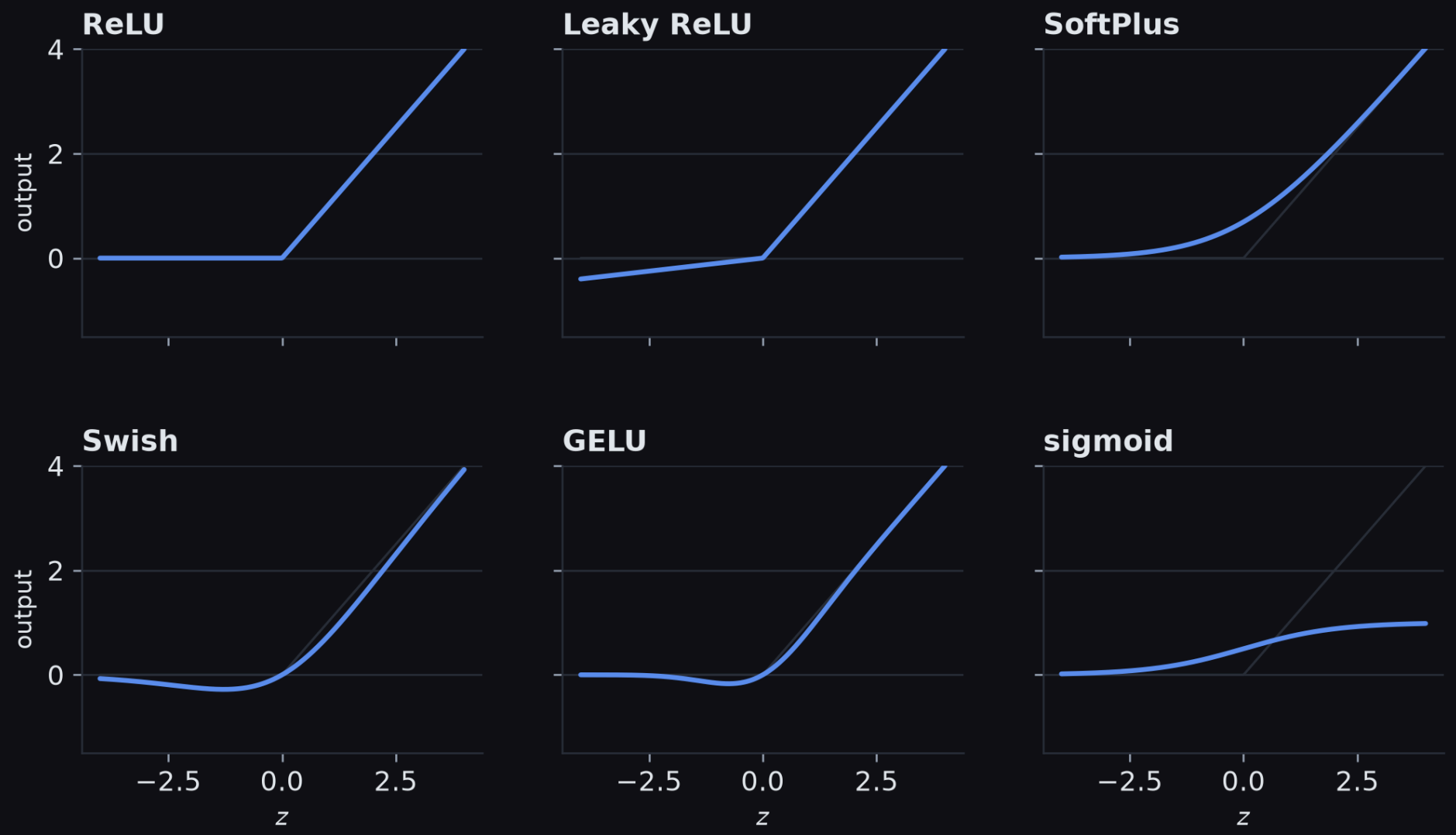
HardSwish

HardSwish approximates Swish with a cheaper piecewise function:

$$\mathrm{HardSwish}(z) = \begin{cases} 0, & z < -3 \\ \dfrac{z(z+3)}{6}, & -3 \le z \le 3 \\ z, & z > 3 \end{cases}$$

It has a similar qualitative shape while being simpler to compute.

Six activation functions



Six activations on shared axes. ReLU is drawn faintly behind each panel.

Is there a best activation function?

There is **no definitive answer** as to which activation function is empirically best in every setting.

- Alternatives can outperform ReLU in particular architectures or tasks
- The differences are often context-dependent
- ReLU remains especially valuable pedagogically because the functions it creates are easy to analyse geometrically

For this reason, we use ReLU as our main activation while developing the theory.

Why not always sigmoid?

The logistic sigmoid is

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

For large positive or negative z, its derivative becomes small.

This saturation can make gradient-based optimisation more difficult in deep networks.

Sigmoid and ReLU derivatives

The derivative of the logistic sigmoid, and of ReLU.

Linear versus affine

Definition 10 A map is **linear** when it satisfies superposition: $f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})$ and $f(c\mathbf{x}) = cf(\mathbf{x})$.

- **Linear**: $f(\mathbf{x}) = \mathbf{W}\mathbf{x}$
- **Affine**: $f(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$, with $\mathbf{b} \neq \mathbf{0}$

A bias breaks linearity, so every "linear layer" in machine learning is really affine.

Why ML often says “linear” anyway

Machine-learning texts often use terms such as:

- linear layer
- linear model
- linear transformation

even when biases are included.

Mathematically, many of these transformations are affine rather than strictly linear.

The distinction is worth knowing even when the terminology is relaxed.

Number of regions: geometric view

Every ReLU hidden unit contributes one activation boundary.

The collection of boundaries partitions the input space.

Each resulting region has:

- one fixed activation pattern
- one affine input/output map

This makes geometry a useful way to reason about expressivity.

Maximum regions in higher dimensions

For D hyperplanes in a D_i-dimensional input space, with $D_i \leq D$, the maximum number of regions is

$$\sum_{j=0}^{D_i} \binom{D}{j}$$

under a general-position arrangement.

A surprisingly large number of regions

Consider a shallow network with

$$D = 500, \qquad D_i = 100.$$

The theoretical maximum number of linear regions is greater than

$$10^{107}.$$

Yet a scalar-output network of this size has only 51,001 parameters.

Even relatively small shallow networks can therefore induce extremely rich geometric partitions in high-dimensional input spaces.

Recovering the 1D result

Set $D_i = 1$:

$$\sum_{j=0}^{1} \binom{D}{j} = \binom{D}{0} + \binom{D}{1}$$

$$= 1 + D$$

So the familiar $D + 1$ result is the one-dimensional special case.

A useful region-count rule of thumb

Shallow neural networks usually have more hidden units than input dimensions:

$$D > D_i.$$

A useful qualitative range is

$$2^{D_i} \lesssim \text{number of regions} \leq 2^D,$$

although the exact attainable count depends on the arrangement of the activation hyperplanes.

The binomial-sum formula gives the tighter geometric upper bound for hyperplanes in general position.

Capacity grows rapidly

In higher dimensions:

- each hidden unit adds a hyperplane
- hyperplanes can intersect
- intersections create many distinct regions
- each region can carry a different affine map

A relatively small network can therefore induce a surprisingly complicated partition of its input space.

But region count is not everything

More regions do not automatically imply a better model.

Expressivity also depends on:

- where the regions are placed
- how the affine pieces are oriented
- how outputs combine hidden features
- whether training finds useful parameter values

One compact equation

A fully connected shallow ReLU network is

$$\boxed{\mathbf{y} = \mathbf{W}_2 \, \mathrm{ReLU}(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2}$$

This single equation encodes the entire architecture.

Read the equation left to right

Expression	What it is
$\mathbf{x}$	input
$\mathbf{W}_1\mathbf{x} + \mathbf{b}_1$	hidden pre-activations
$\mathrm{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$	hidden activations
$\mathbf{W}_2\mathbf{h} + \mathbf{b}_2$	output

Core algebraic picture

A shallow network alternates:

$$\text{affine} \to \text{nonlinear} \to \text{affine}$$

The nonlinearity prevents the affine maps from collapsing into one.

Core geometric picture

Each hidden ReLU:

- introduces an activation boundary
- divides the input space
- is active on one side
- is inactive on the other

The whole network becomes a continuous piecewise-affine function.

Core expressivity picture

More hidden units mean:

- more activation boundaries
- potentially more regions
- more affine pieces
- greater representational capacity

$$\text{affine maps} + \text{nonlinearity} + \text{enough hidden units} \Longrightarrow \text{rich function approximation}$$

Check your understanding: one hidden unit

For $h = \mathrm{ReLU}(2x - 4)$:

- Where is the joint?
- When is the unit active?
- What is its slope when active?

$$2x - 4 = 0 \quad \Longrightarrow \quad x = 2$$

Check your understanding: reading a network

For $y = 1 + 3\,\mathrm{ReLU}(x - 2) - 2\,\mathrm{ReLU}(x - 5)$, identify:

- the joints
- the activation pattern in each region
- the slope in each region
- the role of the output bias

Check your understanding: parameter shapes

Suppose $\mathbf{x} \in \mathbb{R}^5$, $\mathbf{h} \in \mathbb{R}^{20}$, $\mathbf{y} \in \mathbb{R}^3$.

- What are the parameter shapes?
- **First layer**: $\mathbf{W}_1 \in \mathbb{R}^{20 \times 5}$, $\mathbf{b}_1 \in \mathbb{R}^{20}$
- **Second layer**: $\mathbf{W}_2 \in \mathbb{R}^{3 \times 20}$, $\mathbf{b}_2 \in \mathbb{R}^3$

Check your understanding: no activation

Why does stacking these two layers not create a nonlinear model?

- **Hidden layer**: $\mathbf{h} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$
- **Output layer**: $\mathbf{y} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2$

Because the composition of affine maps is still affine.

Summary

A shallow neural network has one hidden layer.

It:

1. computes several affine functions of the input
2. passes each through a nonlinear activation function
3. takes affine combinations of those activations to form the outputs

With ReLU activations, the network divides its input space into regions and implements a different affine function in each region.

With enough hidden units, shallow networks can approximate continuous functions to arbitrary precision on suitable compact domains.

What should you now be able to explain?

- How a hidden ReLU creates a joint
- Why shallow ReLU networks are piecewise linear
- How activation patterns define regions
- How width affects capacity
- What universal approximation actually claims
- How scalar networks generalise to vector inputs and outputs
- Why nonlinear activations are essential

Terminology you should now own

- weight
- bias
- hidden unit / neuron
- pre-activation
- activation
- layer
- shallow network
- deep network
- feed-forward network
- fully connected / dense layer
- multi-layer perceptron

Final picture

A shallow neural network is not mysterious machinery.

It is a carefully structured function:

$$\boxed{\mathbf{y} = \mathbf{W}_2 \, \mathrm{ReLU}(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2}$$

- affine transformation
- nonlinear gating
- affine recombination
- repeated across many hidden units

That simple construction is enough to produce a remarkably expressive family of functions.

Deep neural networks

From shallow to deep

- A **shallow neural network** has one hidden layer
- A **deep neural network** has more than one hidden layer
- With ReLU activations, both describe piecewise-linear mappings
- The key question is therefore not:
  - *can deep networks represent nonlinear functions?*
- It is:
  - *what does depth buy us?*

A shallow network can approximate arbitrarily complex continuous functions given enough hidden units.

But some functions require an impractically large number of hidden units in a shallow network.

Deep networks can often create much richer piecewise-linear structure for a fixed parameter budget.

Today’s questions

We will ask:

- What happens when we compose neural networks?
- Why is composition naturally represented by depth?
- How does each layer increase functional complexity?
- What do **depth**, **width**, and **capacity** mean?
- Why can depth be more parameter-efficient than width?
- Why are deep networks especially useful for structured inputs?

Recap: a shallow ReLU network

A one-hidden-layer network has the form

- **Hidden layer**: $\mathbf{h} = \mathrm{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$
- **Output layer**: $\mathbf{y} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2$

or as one expression,

$$\mathbf{y} = \mathbf{W}_2\,\mathrm{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2.$$

Recap: the geometric view

A shallow ReLU network:

- divides the input space with activation boundaries
- creates an activation pattern in each region
- implements one affine function inside each region

Depth will let us repeatedly transform and repartition those regions.

A terminology reminder

Strictly, once biases are included, the function inside each region is **affine**:

$$f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}.$$

The common machine-learning convention calls these regions **linear**.

We will keep that convention when discussing **linear regions**, while remembering the mathematical distinction.

Reminder: function composition

Suppose

$$y = f(x)$$

and

$$y' = g(y).$$

Then applying f first and g second gives

$$y' = g(f(x)) = (g \circ f)(x).$$

This is **function composition**.

Why composition matters

Composition lets us build a complicated function from simpler functions.

$$x \overset{f}{\longmapsto} y \overset{g}{\longmapsto} y'$$

The output of one computation becomes the input to the next.

Deep neural networks are fundamentally **compositions of learned functions**.

Compose two shallow networks

Consider two shallow networks.

The first maps

$$x \longmapsto y.$$

The second maps

$$y \longmapsto y'.$$

Together:

$$x \longmapsto y \longmapsto y'.$$

The first shallow network

Let the first network have three hidden units:

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$

$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$

$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x]$$

and output

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3.$$

The second shallow network

The second network also has three hidden units:

$$h'_1 = \mathrm{a}\left[\theta'_{10} + \theta'_{11} y\right]$$

$$h'_2 = \mathrm{a}\left[\theta'_{20} + \theta'_{21} y\right]$$

$$h'_3 = \mathrm{a}\left[\theta'_{30} + \theta'_{31} y\right]$$

The second output

Its output is

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3.$$

So the full model computes

$$y' = g(f(x)).$$

Both networks are piecewise linear

With ReLU activations:

- the first network defines a piecewise-linear map $x \mapsto y$
- the second defines a piecewise-linear map $y \mapsto y'$

But the composition can contain substantially more linear regions than either network alone.

A deliberately folded first function

Imagine choosing the first network so that over $x \in [-1, 1]$ it contains three linear regions with alternating slopes.

Different ranges of x can then map onto the same range of y.

For example, several different input values may all map to the same y.

Many inputs can share one intermediate value

Suppose

$$f(x_1) = f(x_2) = f(x_3) = y.$$

Then the second network receives exactly the same value in all three cases.

$$g(f(x_1)) = g(f(x_2)) = g(f(x_3)).$$

The second computation is therefore repeated across several parts of the original input space.

Replicating the second function

If the first network maps three different input regions onto the same range of y, then the second network's piecewise-linear structure is applied to all three regions.

The second function is effectively **duplicated**, possibly flipped and rescaled, across the original input space.

Suppose:

- the first function contributes three relevant linear regions
- the second function contributes three linear regions over the range produced by the first

Then the composition can produce $3 \times 3 = 9$ linear regions.

Compare with a shallow network

Two shallow networks with three hidden units each use six hidden units in total.

A single shallow network with six hidden units and scalar input can create at most

$$6 + 1 = 7$$

linear regions.

But the composition can create nine.

Depth can therefore create structure that is not captured by merely counting total hidden units.

The folding view

A useful geometric interpretation is:

> the first network **folds the input space back onto itself**.

Different inputs are mapped to the same intermediate representation.

The next network then performs the same computation on all locations that were folded together.

Conceptually:

$$\text{input space} \longrightarrow \text{folded representation} \longrightarrow \text{new transformation}$$

When viewed back in the original input coordinates, the new transformation appears repeatedly across the folded regions.

Composition as folding

f, then g, then the composition $g(f(x))$.

Why “folding” is useful intuition

The folding picture explains how depth creates reuse.

A later layer does not need to independently construct the same feature in every original region.

Earlier layers can transform different regions into a shared representation, allowing later layers to process them similarly.

The same idea in two dimensions

Now suppose the first network receives

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

and produces a scalar y.

The first network may partition the 2D input into several linear regions.

A 2D composition example

In this example:

- the first network produces seven linear regions
- one region is flat
- six non-flat regions span the relevant output range
- the second network contributes two linear regions

Each non-flat first-stage region can be divided again by the second stage.

Counting the 2D example

The six non-flat regions are each split into two:

$$6 \times 2 = 12.$$

The original flat region remains one region.

Therefore:

$$12 + 1 = 13$$

linear regions.

Composition compounds complexity

The important pattern is not the exact number 13.

It is that a later network can introduce new subdivisions **inside regions already created by an earlier network**.

Successive layers can therefore compound functional complexity.

From composed networks to a deep network

Composing two shallow networks looks like

$$x \longrightarrow \text{hidden layer} \longrightarrow y \longrightarrow \text{hidden layer} \longrightarrow y'.$$

But there is a redundancy here.

The intermediate output y is produced by an affine operation, and the next network begins with another affine operation.

Affine transformations compose

Recall:

$$y = \phi_0 + \sum_i \phi_i h_i.$$

The next layer begins with something like

$$z'_j = \theta'_{j0} + \theta'_{j1} y.$$

Substitute the first equation into the second.

Substitute the intermediate output

For one second-stage hidden unit,

$$z_1' = \theta_{10}' + \theta_{11}' \left(\phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3\right).$$

Expand:

$$z_1' = \theta_{10}' + \theta_{11}'\phi_0 + \theta_{11}'\phi_1 h_1 + \theta_{11}'\phi_2 h_2 + \theta_{11}'\phi_3 h_3.$$

Define new parameters

Collect the constants:

$$\psi_{10} = \theta'_{10} + \theta'_{11}\phi_0.$$

And define

$$\psi_{11} = \theta'_{11}\phi_1, \qquad \psi_{12} = \theta'_{11}\phi_2, \qquad \psi_{13} = \theta'_{11}\phi_3.$$

Then

$$h'_1 = \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3]\,.$$

Repeat for every second-layer unit

Similarly,

$$h_2' = \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3]$$

and

$$h_3' = \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3]\,.$$

The intermediate output disappears

We no longer need an explicit scalar y between the networks.

The computation becomes

$$x \longrightarrow \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} \longrightarrow \begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix} \longrightarrow y'.$$

This is a network with **two hidden layers**.

Composition is a special case of depth

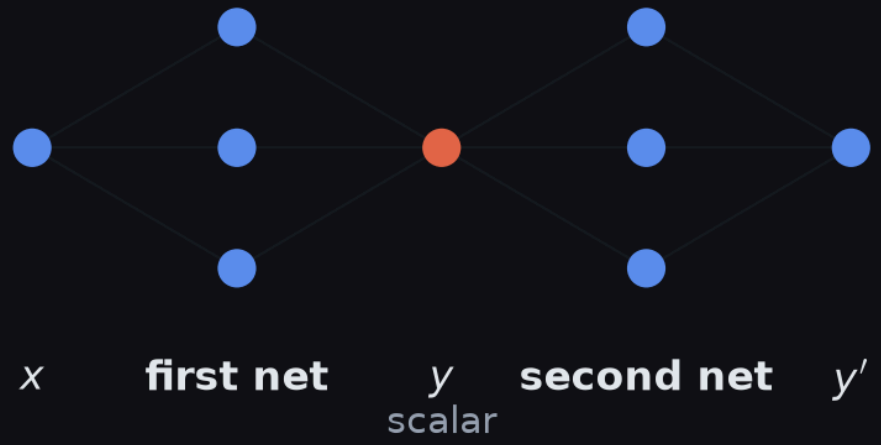
A two-hidden-layer network can represent the functions obtained by composing the two shallow networks.

But it can actually represent **more** functions than that composition construction.

Why?

Because the weights between hidden layers are no longer forced to satisfy the constraints introduced by the scalar intermediate y.

Composed networks and deep networks



Two shallow networks joined through a scalar, and a network with two hidden layers.

The composition imposes structure

In the composed-shallow-network construction, the inter-layer weights have the form

$$\psi_{ji} = \theta'_{j1}\phi_i.$$

So the matrix of inter-layer slopes can be written as an outer product.

$$\mathbf{\Psi} = \begin{bmatrix} \theta'_{11} \\ \theta'_{21} \\ \theta'_{31} \end{bmatrix} \begin{bmatrix} \phi_1 & \phi_2 & \phi_3 \end{bmatrix}.$$

Why the outer-product constraint matters

An outer-product matrix has highly constrained entries.

The nine values

$$\psi_{11}, \psi_{12}, \ldots, \psi_{33}$$

cannot vary independently in the composed construction.

A general two-hidden-layer network allows all nine to be learned independently.

So the deep network represents a broader family of functions.

A two-hidden-layer network

For scalar input x, first hidden layer:

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$

$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$

$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x]\,.$$

The second hidden layer

$$h_1' = \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3]$$

$$h_2' = \mathrm{a}[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3]$$

$$h_3' = \mathrm{a}[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3]\,.$$

The output layer

Finally,

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3.$$

This is now a genuine **deep neural network**.

How does the first layer behave?

The first layer works exactly as in a shallow network.

It:

- computes affine functions of the input
- passes them through ReLU
- creates piecewise-linear hidden activations

Nothing conceptually new happens in the first layer.

Pre-activations in the second layer

Before the second ReLU, we compute new affine combinations of the first hidden activations.

For example,

$$z_1' = \psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3.$$

Because each h_i is already piecewise linear in x, z_1' is also piecewise linear in x.

Shared joints before the second ReLU

The three second-layer pre-activations

$$z_1'(x), \quad z_2'(x), \quad z_3'(x)$$

are different piecewise-linear functions.

But because they are all affine combinations of the same first-layer activations, their existing joints occur at the same input locations.

Then ReLU clips again

Now apply ReLU:

$$h'_j = \mathrm{ReLU}(z'_j).$$

Each z'_j may cross zero **inside an existing linear region**.

When that happens, ReLU introduces a new joint.

This is how later layers refine the partition created by earlier layers.

The clipping view

A second way to think about depth is:

1. create piecewise-linear functions
2. recombine them
3. clip them with ReLU
4. create new joints
5. recombine again

Each layer can create new boundaries inside regions inherited from previous layers.

Folding view versus clipping view

Two complementary intuitions:

Folding view

- emphasises how multiple inputs can share representations
- explains repeated structure

Clipping view

- emphasises how later ReLUs create new joints
- explains subdivision of existing regions

The folding picture highlights dependencies and reuse.

The clipping picture highlights the creation of new regions.

Both are useful, but neither replaces the algebraic definition of the network.

Do not lose sight of the model

However complicated the diagram becomes, a deep network is still just a parameterised function.

$$\mathbf{y} = f(\mathbf{x}; \boldsymbol{\phi}).$$

It maps inputs to outputs according to a nested sequence of affine transformations and nonlinear activations.

The nested scalar expression

For the two-hidden-layer scalar example, substituting everything produces a large nested expression.

Schematically:

$$y' = \phi'_0 + \sum_j \phi'_j \mathrm{a}\left[\psi_{j0} + \sum_i \psi_{ji} \mathrm{a}[\theta_{i0} + \theta_{i1} x]\right].$$

Why layer notation matters

The fully expanded equation is correct, but difficult to reason about.

Layer notation exposes the repeated structure:

$$\text{affine} \to \text{activation} \to \text{affine} \to \text{activation} \to \cdots$$

This is one reason matrix notation becomes essential for deep networks.

Extending to more layers

There is no special reason to stop at two hidden layers.

A deep network may contain:

- a few hidden layers
- tens of hidden layers
- hundreds of hidden layers

Modern architectures can contain very large numbers of units across these layers.

Depth

Definition 11 Depth: the number of hidden layers in the network.

We denote the number of hidden layers by

$$K.$$

So:

- $K = 1$: shallow network
- $K > 1$: deep network

Some papers count depth differently. We follow Prince's convention.

Width

Definition 12 Width: the number of hidden units in a layer.

If hidden layer k contains D_k hidden units, then

$$D_k$$

is the width of that layer.

Different layers need not have the same width.

Capacity

The total number of hidden units is one measure of the network's **capacity**.

Roughly:

- more units
- more parameters
- larger family of possible functions

But depth and width influence that capacity in different ways.

Parameters versus hyperparameters

Parameters are learned from data:

- weights
- biases

Definition 13 Hyperparameters: quantities chosen before we learn the model parameters.

For the networks considered here, architectural hyperparameters include:

- the number of hidden layers
- the number of hidden units in each layer

$$K, \quad D_1, D_2, \ldots, D_K$$

A family of families

Fixing the hyperparameters gives us one architecture.

That architecture defines a **family of functions**, indexed by its learned parameters.

Changing the hyperparameters changes the family itself.

So neural-network design can be viewed as choosing from a **family of families of functions**.

Matrix notation: why now?

With multiple layers and multiple units, scalar equations quickly become cumbersome.

Matrix notation gives us one repeated pattern:

$$\text{next layer} = \mathrm{a}[\text{bias} + \text{weight matrix} \times \text{previous layer}] \, .$$

Activations act elementwise

When $\mathrm{a}[\cdot]$ receives a vector, the activation function is applied separately to every element.

For example,

$$\mathrm{a}\left[\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}\right] = \begin{bmatrix} \mathrm{a}[z_1] \\ \mathrm{a}[z_2] \\ \mathrm{a}[z_3] \end{bmatrix}.$$

For ReLU, each component is independently clipped at zero.

A concrete deep network

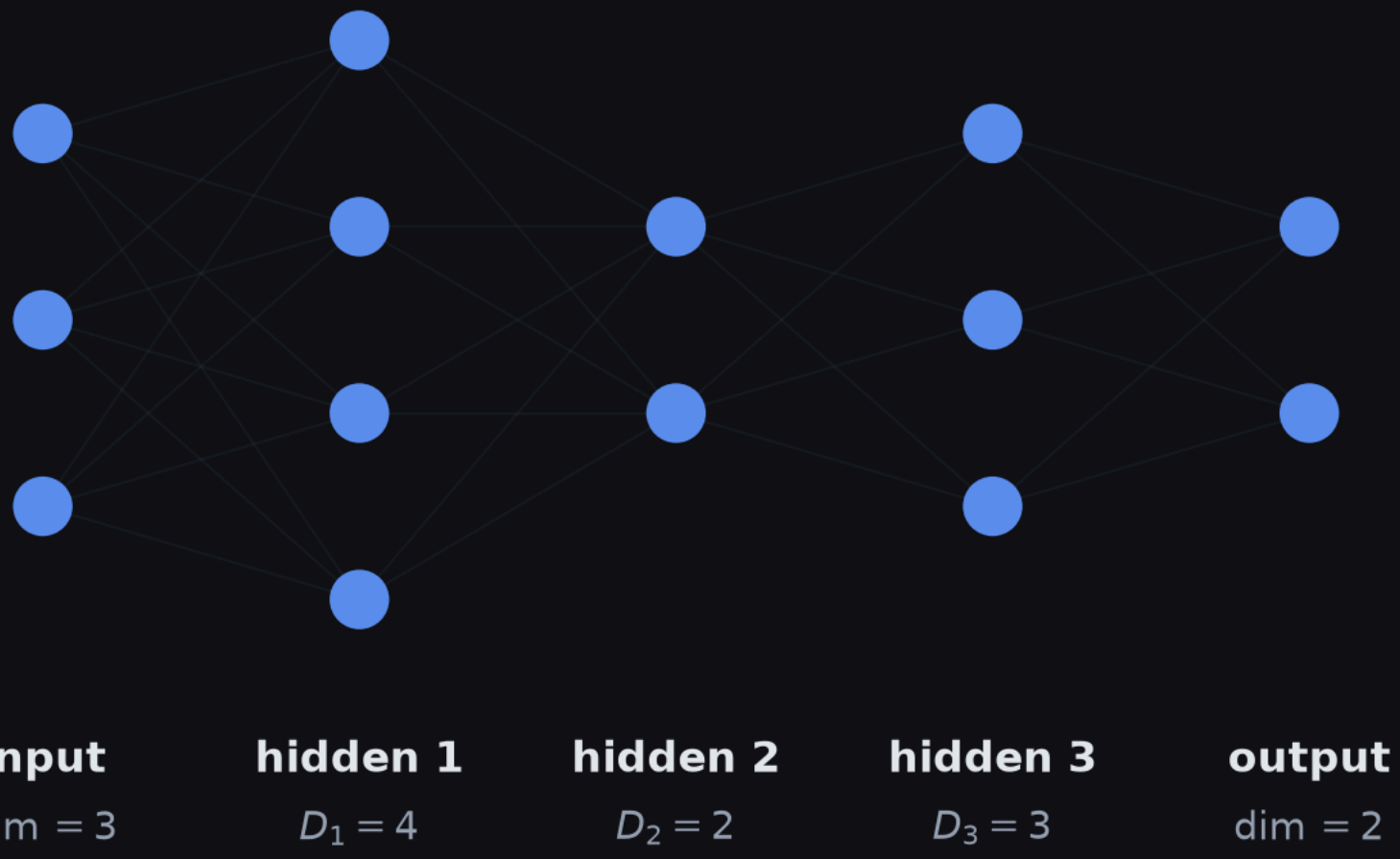
Consider:

$$D_i = 3, \qquad D_o = 2, \qquad K = 3.$$

Let the hidden-layer widths be

$$D_1 = 4, \qquad D_2 = 2, \qquad D_3 = 3.$$

The three-hidden-layer network



Three inputs, hidden layers of 4, 2 and 3 units, two outputs.

Layer 1

The first hidden layer is

$$\mathbf{h}_1 = \mathrm{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}] \,.$$

Shapes:

$$\mathbf{x} \in \mathbb{R}^3, \qquad \mathbf{h}_1 \in \mathbb{R}^4.$$

Therefore

$$\boldsymbol{\Omega}_0 \in \mathbb{R}^{4\times 3}, \qquad \boldsymbol{\beta}_0 \in \mathbb{R}^4.$$

Layer 2

The second hidden layer is

$$\mathbf{h}_2 = \mathrm{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1] \,.$$

Shapes:

$$\mathbf{h}_1 \in \mathbb{R}^4, \qquad \mathbf{h}_2 \in \mathbb{R}^2.$$

Therefore

$$\boldsymbol{\Omega}_1 \in \mathbb{R}^{2\times 4}, \qquad \boldsymbol{\beta}_1 \in \mathbb{R}^2.$$

Layer 3

The third hidden layer is

$$\mathbf{h}_3 = \mathrm{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2 \mathbf{h}_2] \,.$$

Therefore

$$\boldsymbol{\Omega}_2 \in \mathbb{R}^{3\times 2}, \qquad \boldsymbol{\beta}_2 \in \mathbb{R}^3.$$

Output layer

The output is

$$\mathbf{y} = \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3 \mathbf{h}_3.$$

Since

$$\mathbf{y} \in \mathbb{R}^2,$$

we require

$$\boldsymbol{\Omega}_3 \in \mathbb{R}^{2\times 3}, \qquad \boldsymbol{\beta}_3 \in \mathbb{R}^2.$$

General layer notation

For a network with K hidden layers:

$$\mathbf{h}_1 = \mathrm{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}]$$

$$\mathbf{h}_2 = \mathrm{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1 \mathbf{h}_1]$$

$$\vdots$$

Last hidden layer and output

$$\mathbf{h}_K = \mathrm{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{h}_{K-1}]$$

and

$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{h}_K.$$

General deep-network definition

A deep neural network is a composition of affine transformations and nonlinear activation functions.

Let

$$\mathbf{h}_0 \equiv \mathbf{x}.$$

Then for the hidden layers,

$$\mathbf{h}_{k+1} = \mathrm{a}[\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k], \qquad k = 0, \ldots, K-1.$$

For the architecture considered here, the output layer is affine:

$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathbf{h}_K.$$

What is in the parameter set?

The learned parameters are all weight matrices and bias vectors:

$$\phi = \beta_k, \Omega_k{}_{k=0}^{K}.$$

The architecture fixes their shapes.

Training learns their numerical values.

General parameter shapes

If hidden layer k has width D_k:

$$\boldsymbol{\beta}_{k-1} \in \mathbb{R}^{D_k}.$$

The first weight matrix has shape

$$\boldsymbol{\Omega}_0 \in \mathbb{R}^{D_1 \times D_i}.$$

Intermediate weight shapes

For intermediate layers,

$$\boldsymbol{\Omega}_k \in \mathbb{R}^{D_{k+1} \times D_k}.$$

The matrix has:

- one column per source unit
- one row per target unit

Output shapes

The final weight matrix has shape

$$\boldsymbol{\Omega}_K \in \mathbb{R}^{D_o \times D_K}.$$

The final bias has shape

$$\boldsymbol{\beta}_K \in \mathbb{R}^{D_o}.$$

The whole network as one function

Substituting layer by layer:

$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathrm{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1} \mathrm{a}[\cdots]] \,.$$

More explicitly:

$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathrm{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1} \mathrm{a}[\cdots \mathrm{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}]]] \,.$$

One repeated computational motif

Every hidden layer repeats:

$$\mathbf{z}^{(k)} = \boldsymbol{\beta}_{k-1} + \boldsymbol{\Omega}_{k-1}\mathbf{h}_{k-1}$$

$$\mathbf{h}_k = \mathrm{a}\left[\mathbf{z}^{(k)}\right].$$

with

$$\mathbf{h}_0 \equiv \mathbf{x}.$$

Shallow versus deep

We now compare:

Shallow

- one hidden layer
- potentially very wide

Deep

- multiple hidden layers
- composition across layers

Both can be universal approximators.

The practical question is how efficiently they represent useful functions.

Can a deep network approximate arbitrary functions?

Yes, given sufficient capacity.

A deep network can reproduce a shallow network when its additional computation implements a suitable identity mapping.

Therefore, if a shallow network can approximate a target function, a sufficiently capable deep network can too.

The identity function simply returns its input:

$$I(x) = x.$$

If an added part of the network computes the identity, it leaves the preceding function unchanged.

This lets a deep architecture contain a shallow architecture as a special case.

Universal approximation still applies

Given enough capacity, deep networks can approximate suitable continuous functions arbitrarily closely.

So universality alone does **not** explain why depth is useful.

Both shallow and deep networks can be universal approximators.

We need a stronger comparison.

Compare regions per parameter

For scalar input and scalar output, take a shallow ReLU network with $D > 2$ hidden units.

- **Linear regions, at most**: $D + 1$
- **Parameters**: $3D + 1$

For each hidden unit:

- one input weight
- one hidden bias
- one output weight

That gives

$$3D.$$

Then add one output bias:

$$3D + 1.$$

A deep scalar network

Now consider:

- scalar input
- scalar output
- K hidden layers
- $D > 2$ units in every hidden layer

A construction exists that can create up to

$$(D+1)^K$$

linear regions.

Parameter count for the deep network

The parameter count is

$$3D + 1 + (K - 1)D(D + 1).$$

The first and last parts resemble the shallow network.

The extra term accounts for the fully connected transformations between hidden layers.

Why depth can win

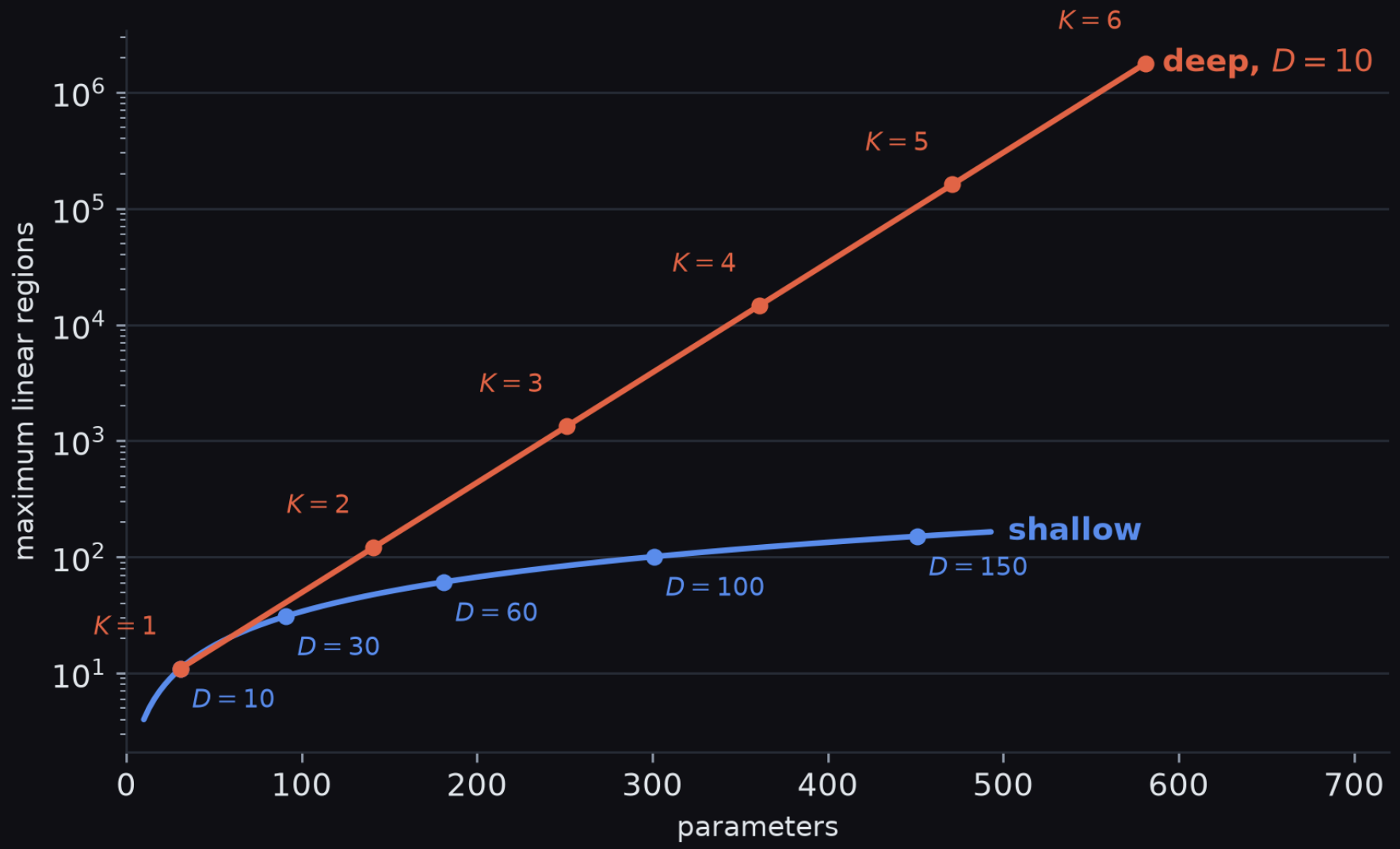
The parameter count grows roughly linearly with K when width is fixed.

But the number of regions in this construction grows exponentially with K:

$$(D+1)^K.$$

This is the basic source of the **regions-per-parameter advantage** of depth.

Regions per parameter



Maximum linear regions against parameter count, log scale. Each marked point is labelled with the width or depth that produced it.

Concrete example: five layers

Take $K = 5$ hidden layers with $D = 10$ units in each.

- **Parameters**: 471
- **Linear regions, at most**: $161\,051$

$$(D + 1)^K = 11^5$$

which is 161,051.

A shallow network with a similar parameter budget cannot produce remotely as many scalar linear regions.

Higher dimensions magnify the effect

The disparity becomes even greater as the input dimension increases.

One example uses $D_i = 10$, $K = 5$ hidden layers and $D = 50$ units each.

That network has 10,801 parameters.

The high-dimensional region count

That network can create more than

$$10^{41}$$

linear regions.

The number is enormous compared with the parameter count.

But we must be careful about what that does—and does not—tell us.

More regions are not automatically better

The flexibility of a network is still constrained by its parameters.

Deep networks can create huge numbers of regions, but those regions have:

- dependencies
- symmetries
- shared parameter structure

The regions are not independently adjustable pieces.

The same learned transformation is reused across parts of the input space that earlier layers fold together.

That creates:

- parameter efficiency
- but also structural dependencies

So more regions are especially useful when the target function contains corresponding structure.

When might depth help most?

Two cases are particularly plausible:

1. the real-world function contains useful symmetries or repeated structure
2. the target mapping is naturally a composition of simpler functions

This is a stronger argument for depth than raw region count alone.

Depth efficiency

Definition 14 Depth efficiency: the phenomenon that some functions can be approximated by deep networks much more efficiently than by shallow networks.

For certain functions, a shallow network requires an **exponentially larger number of hidden units** to achieve an equivalent approximation to a deep network.

Both shallow and deep networks may be universal.

But universality asks only:

> can the function be approximated?

Depth efficiency asks:

> how many computational resources are needed to approximate it?

Representation efficiency is often more practically important than mere representability.

But there is an important caveat

Mathematics can construct functions for which depth gives an exponential advantage.

That does not automatically prove that the functions encountered in real-world machine-learning tasks have exactly this structure.

The theoretical advantage is real; its relevance to every practical problem is not guaranteed.

Large, structured inputs

So far we have discussed fully connected networks.

These become problematic for very large inputs.

Consider an image containing roughly

$$10^6$$

pixels.

Connecting every input independently to every hidden unit can require an enormous number of parameters.

Images contain structure

An image is not merely a million unrelated numbers.

Nearby pixels are related.

Objects can occur in many positions.

We do not want to learn a completely independent detector for the same visual pattern at every possible image location.

Local-to-global processing

A better strategy is:

- process local regions
- reuse similar computations across locations
- combine local information
- progressively integrate information over larger regions

This kind of local-to-global hierarchy is naturally expressed using multiple layers.

Depth and structured computation

For structured inputs, depth is useful for more than raw expressive power.

It lets us encode a **sequence of computational stages**, progressively integrating information from larger parts of the input.

An illustrative image hierarchy might look like:

- local patterns
- larger motifs
- object parts
- whole objects

This hierarchy is an intuition for local-to-global processing; the precise features learned are determined by the model and data.

Training deep versus shallow networks

Another possible advantage is optimisation.

It is usually easier to train **moderately deep networks** than shallow ones.

One possible explanation is that over-parameterised deep models may contain a large family of roughly equivalent solutions that are comparatively easy to find.

This is a possible explanation rather than a settled result.

As more hidden layers are added, training eventually becomes more difficult again.

Many techniques have been developed to mitigate this problem.

We will return to the mechanics of training and the methods used to stabilise deep networks later in the major.

Generalisation

Deep neural networks also often generalise better in practice than comparable shallow alternatives.

The strongest empirical results in many tasks have historically come from models with:

- tens of layers
- hundreds of layers

Why deep, over-parameterised networks generalise as well as they do is not fully understood.

How deep and shallow networks differ

Deep and shallow networks differ in several important ways.

1. both can approximate arbitrary suitable functions
2. deep networks can create more linear regions per parameter
3. some functions are much more efficiently represented with depth
4. large structured inputs benefit from multistage processing
5. deep networks often work better in practice

Do not overstate the conclusion

Depth is not magic.

More depth can bring:

- representational efficiency
- useful compositional structure
- better inductive biases in specialised architectures

But it can also bring:

- harder optimisation
- more implementation complexity
- more opportunities for instability

Summary: composition and deep networks

We began by composing two shallow networks.

The first can fold the input space so that different inputs share the same intermediate representation.

The second then applies its piecewise-linear function to that folded representation.

The result appears repeatedly across the original input space.

The composition of shallow networks is a special case of a network with multiple hidden layers.

A general deep network is more flexible because its inter-layer weights are learned independently rather than constrained by the composition construction.

Summary: layers and clipping

Each layer:

- forms new affine combinations
- applies nonlinear activations
- creates or refines activation regions

With ReLU:

successive layers repeatedly **recombine and clip** piecewise-linear functions.

Summary: hyperparameters

Architectural hyperparameters include:

- number of hidden layers K
- width of each hidden layer D_k

Learned parameters include:

- weight matrices
- bias vectors

Hyperparameters choose the architecture; parameters choose a function within that architecture.

Summary: why depth?

Compared with shallow networks, deep networks can:

- generate more linear regions for a fixed parameter budget
- represent some functions exponentially more efficiently
- express hierarchical processing naturally
- perform very well empirically

Depth changes not just how much a network can represent, but **how efficiently and structurally it represents it**.

Historical note: “deep learning”

The idea of networks with multiple hidden layers is old.

The term **deep learning** appears as early as Dechter (1986) (Dechter 1986) – though there it names a constraint-satisfaction search strategy, not anything to do with layered networks.

For many years, however, practical interest was limited because deep networks were difficult to train effectively.

The striking image-classification improvements reported by **Krizhevsky et al. (2012)** (Krizhevsky et al. 2017) are a key event in the modern deep-learning era.

It highlights a confluence of four factors:

- larger training datasets
- improved processing power for training
- ReLU activations
- stochastic gradient descent

The success of deep learning was therefore not the result of depth alone.

Region-count upper bound

For a deep ReLU network with a total of D hidden units, one general upper bound on the number of linear regions is

$$2^D.$$

This follows from the fact that D ReLU units have at most 2^D distinct binary activation patterns.

But not every activation pattern is attainable

The upper bound

$$2^D$$

is usually loose.

Geometric constraints and dependencies between layers prevent arbitrary activation patterns from necessarily corresponding to distinct reachable input regions.

Counting regions in realistic deep networks is a difficult combinatorial problem.

Sharper bounds and exact counting

Later work gives tighter region-count bounds, including:

- Arora et al. (2016) (Arora et al. 2016)
- Serra et al. (2018) (Serra et al. 2017)

Serra et al. also give an algorithm for counting the regions of a particular network exactly.

Exact counting is computationally practical only for very small networks.

A constructive lower bound

For a deep ReLU network with:

- D_i-dimensional input
- K hidden layers
- $D \geq D_i$ hidden units per layer

Montúfar et al. (2014) (Montúfar et al. 2014) give a construction whose number of regions grows on the order of

$$\Omega\left(\left(\frac{D}{D_i}\right)^{(K-1)D_i} D^{D_i}\right).$$

The important feature is the strong dependence on depth.

A specialised constructive bound

When every hidden layer has width D, and D is an integer multiple of D_i, the expression is

$$N_r = \left(\frac{D}{D_i} + 1\right)^{D_i(K-1)} \cdot \sum_{j=0}^{D_i} \binom{D}{j}.$$

Treat this as a **constructive lower bound on the maximal number of regions**, not as the exact maximum in general.

Interpreting the constructive bound

The first factor,

$$\left(\frac{D}{D_i}+1\right)^{D_i(K-1)},$$

captures repeated folding through the first $K-1$ layers in the construction.

The second factor,

$$\sum_{j=0}^{D_i}\binom{D}{j},$$

is the familiar shallow-network hyperplane-region term contributed at the final stage.

Deep universal approximation: width view

One route to universality is simply to make hidden layers sufficiently wide.

Because a deep network can emulate a shallow universal approximator, the usual width-based universal approximation result carries over.

Depth is therefore not required for universality.

Deep universal approximation: bounded width

Lu et al. (2017) (Lu et al. 2017) prove that for any **Lebesgue-integrable** function

$$f : \mathbb{R}^{D_i} \to \mathbb{R}$$

and any desired L^1 approximation error, there exists a sufficiently deep fully connected ReLU network with width at most

$$D_i + 4$$

that achieves that accuracy.

This result is stronger in function class than the continuous-on-a-compact-domain theorem, but it uses the L^1 notion of approximation.

What the bounded-width theorem says conceptually

Classical width-based universality:

> keep the network shallow, but make the hidden layer sufficiently wide.

Bounded-width universality:

> keep the network relatively narrow, but allow sufficient depth.

Both width and depth can provide routes to expressive power.

Depth-efficiency results

Theory gives explicit examples where deep networks are exponentially more efficient.

The general form of the claim is:

A function can be represented by a moderately sized deep network but requires an exponentially larger shallow—or substantially shallower—network for comparable accuracy.

Several families of results bear on this:

- **Telgarsky (2016)** (Telgarsky 2016): explicit depth-separation constructions
- **Eldan & Shamir (2016)** (Eldan and Shamir 2015): a function representable by a three-layer network for which a two-layer network requires exponential capacity in the input dimension
- **Cohen et al. (2016)** (Cohen et al. 2015) and related work: further depth-efficiency results
- **Liang & Srikant (2016)** (Liang and Srikant 2016): exponential shallow-network costs for approximation over a broad class of functions

What “depth separation” means

A **depth-separation result** proves that one depth can represent or approximate some function efficiently, while a shallower network requires dramatically more width or parameters.

It is evidence that layers are not always interchangeable with extra width.

Width efficiency

There is a complementary question:

> are there functions represented efficiently by wide shallow networks that require much greater depth when the network is narrow?

This is known as **width efficiency**.

Some wide shallow networks can require narrow networks to become deeper.

But the known lower bounds for reducing width are generally less severe than the exponential penalties seen in many depth-separation results.

This suggests that, in these theoretical comparisons, depth can be a particularly powerful resource.

A later width-efficiency result

A later result from **Vardi et al. (2022)** (Vardi et al. 2022):

for ReLU networks, reducing the width can be compensated for with only a **linear increase in depth** under their construction.

This reinforces the broad message that depth is often a particularly effective representational resource.

Theory versus practice

Region counts, universal approximation, and depth-separation theorems tell us about **representational possibility**.

They do not directly tell us:

- whether gradient descent will find the representation
- how much data is required
- whether it will generalise
- whether the theoretical hard functions resemble real tasks

Expressivity theory is one part of the story, not the whole story.

Check your understanding: composition

Suppose $f(x)$ has three relevant linear regions, and $g(y)$ has three over the range f produces.

- Why can $g(f(x))$ have nine regions?

Each of f's three regions is mapped onto the whole range g sees, so g's three-region structure is replicated inside every one of them: $3 \times 3 = 9$.

Check your understanding: folding

What does it mean to say that the first network “folds” the input space?

Several distinct input regions are mapped onto the same intermediate region, so a later computation is reused across them.

Check your understanding: depth

Which network is deeper?

- **A**: one hidden layer of $10{,}000$ units
- **B**: ten hidden layers of 100 units each

Network B is deeper; Network A is wider.

Check your understanding: hyperparameters

Which are hyperparameters?

- number of hidden layers
- hidden-layer width
- individual weight values
- individual bias values

The layer counts and widths are hyperparameters; weights and biases are learned parameters.

Check your understanding: matrix shapes

Suppose $D_1 = 8$ and $D_2 = 5$.

- What is the shape of the matrix mapping layer 1 to layer 2?

$\boldsymbol{\Omega}_1 \in \mathbb{R}^{5 \times 8}$

Check your understanding: universality

Does the universal approximation theorem imply that deep networks are always better than shallow networks?

No. Both can be universal, and the interesting differences concern efficiency, structure, optimisation, and generalisation.

Check your understanding: regions

Why is “more linear regions” not equivalent to “better model”?

Because the regions are constrained by shared parameters and may not match the structure of the target function or data.

Terminology you should now own

- deep neural network
- depth
- width
- capacity
- hyperparameter
- composition
- hidden layer
- folding
- linear region
- depth efficiency
- width efficiency
- depth separation

One equation to keep

A general deep network with K hidden layers can be written recursively as

$$\mathbf{h}_1 = \mathrm{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}]$$

$$\mathbf{h}_{k+1} = \mathrm{a}[\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k]$$

$$\mathbf{y} = \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K \mathbf{h}_K.$$

One mental model to keep

Each layer:

$$\boxed{\text{recombine} \rightarrow \text{activate} \rightarrow \text{repartition}}$$

Earlier layers create a representation.

Later layers operate on that representation and can refine it further.

Final picture

A deep neural network is a composition of simple operations:

$$\mathbf{x} \to \text{affine} \to \text{ReLU} \to \text{affine} \to \text{ReLU} \to \cdots \to \mathbf{y}.$$

Depth lets these simple operations build on one another.

That composition can produce highly complex functions using far fewer parameters than a comparable shallow construction.

Reading

- **Prince, chapters 3 and 4**: shallow networks, then deep – activation functions, linear regions, composition and depth efficiency (Prince 2023)

Where we go next

We now have the **model**:

$$\mathbf{y} = f(\mathbf{x}; \boldsymbol{\phi}).$$

To learn its parameters, we still need:

- a **loss function** that measures mismatch between predictions and targets
- an **optimisation procedure** that searches for parameters that reduce that loss

The next stage is therefore to move from **what neural networks can represent** to **how neural networks learn**.

References

Arora, Raman, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. 2016. *Understanding Deep Neural Networks with Rectified Linear Units*. https://arxiv.org/abs/1611.01491.

Cohen, Nadav, Or Sharir, and Amnon Shashua. 2015. *On the Expressive Power of Deep Learning: A Tensor Analysis*. https://arxiv.org/abs/1509.05009.

Cybenko, G. 1989. "Approximation by Superpositions of a Sigmoidal Function." *Mathematics of Control, Signals, and Systems* 2 (4): 303–14. https://doi.org/10.1007/bf02551274.

Dechter, Rina. 1986. "Learning While Searching in Constraint-Satisfaction-Problems." *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence (AAAI-86)*, 178–83. https://cdn.aaai.org/AAAI/1986/AAAI86-029.pdf.

Eldan, Ronen, and Ohad Shamir. 2015. *The Power of Depth for Feedforward Neural Networks*. https://arxiv.org/abs/1512.03965.

Hornik, Kurt. 1991. "Approximation Capabilities of Multilayer Feedforward Networks." *Neural Networks* 4 (2): 251–57. https://doi.org/10.1016/0893-6080(91)90009-t.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. 2017. "ImageNet Classification with Deep Convolutional Neural Networks." *Communications of the ACM* 60 (6): 84–90. https://doi.org/10.1145/3065386.

Liang, Shiyu, and R. Srikant. 2016. *Why Deep Neural Networks for Function Approximation?* https://arxiv.org/abs/1610.04161.

Lu, Zhou, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. 2017. *The Expressive Power of Neural Networks: A View from the Width*. https://arxiv.org/abs/1709.02540.

McCulloch, Warren S., and Walter Pitts. 1943. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *The Bulletin of Mathematical Biophysics* 5 (4): 115–33. https://doi.org/10.1007/bf02478259.

Minsky, Marvin, and Seymour A. Papert. 2017. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press. https://doi.org/10.7551/mitpress/11301.001.0001.

Montúfar, Guido, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. 2014. *On the Number of Linear Regions of Deep Neural Networks*. https://arxiv.org/abs/1402.1869.

Prince, Simon J. D. 2023. *Understanding Deep Learning*. MIT Press. https://udlbook.github.io/udlbook/.

Questions?

Eoin O’Brien · eoin@eoin.ai