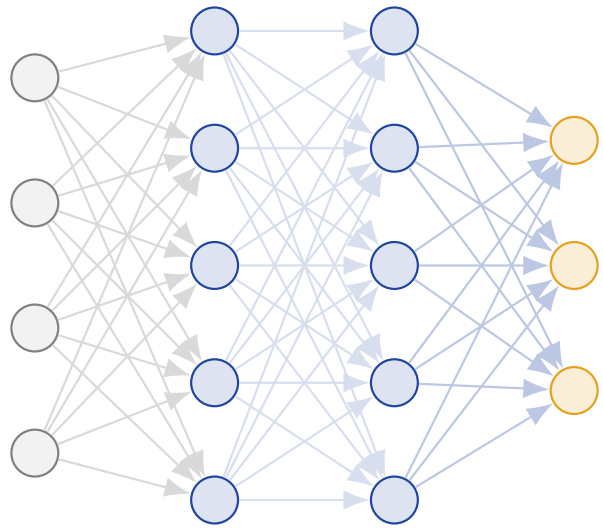
Deep Feedforward Networks

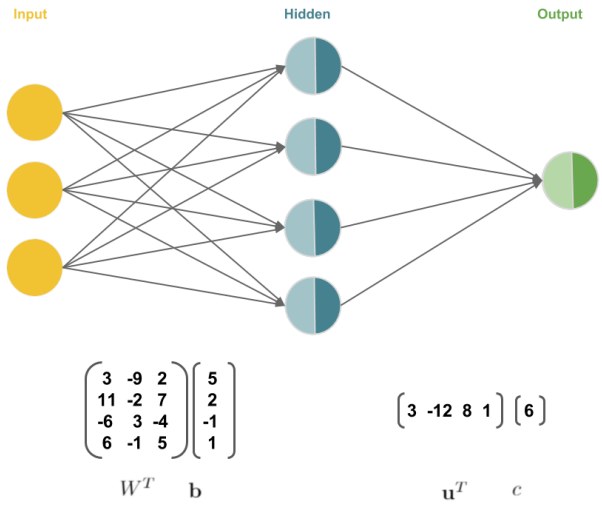
Matrix notation, many layers, many classes – and one theorem

This deck embeds the LMU *Introduction to Deep Learning* MLP chunks in full (matrix notation → deep networks → multi-class softmax → universal approximation), with our own notes sprinkled in.



Deep Learning

MLP – Matrix Notation



Learning goals

- Compact representation of neural network equations
- Vector notation for neuron layers
- Vector and matrix notation of bias and weight parameters

SINGLE HIDDEN LAYER NETWORKS: NOTATIONS

- The input $\mathbf{x}$ is a column vector with dimensions $p \times 1$.
- $\mathbf{W}$ is a weight matrix with dimensions $p \times m$, where m is the amount of hidden neurons:

$$\mathbf{W} = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,m} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{p,1} & w_{p,2} & \cdots & w_{p,m} \end{pmatrix}$$

SINGLE HIDDEN LAYER NETWORKS: NOTATIONS / 2

Hidden layer:

- For example, to obtain z_1, we pick the first column of W:

$$\mathbf{W}_1 = \begin{pmatrix} w_{1,1} \\ w_{2,1} \\ \vdots \\ w_{p,1} \end{pmatrix}$$

and compute

$$z_1 = \sigma(\mathbf{W}_1^T \mathbf{x} + b_1) \ ,$$

where b_1 is the bias of the first hidden neuron and $\sigma : \mathbb{R} \to \mathbb{R}$ is an activation function.

SINGLE HIDDEN LAYER NETWORKS: NOTATION

- The network has m hidden neurons $z_1, \ldots, z_m$ with

$$z_j = \sigma(\mathbf{W}_j^T \mathbf{x} + b_j)$$

  - $z_{j,in} = \mathbf{W}_j^T \mathbf{x} + b_j$
  - $z_{j,out} = \sigma(z_{j,in}) = \sigma(\mathbf{W}_j^T \mathbf{x} + b_j)$

  for $j \in \{1, \ldots, m\}$.

- Vectorized notation:
  - $\mathbf{z}_{in} = (z_{1,in}, \ldots, z_{m,in})^T = \mathbf{W}^T \mathbf{x} + \mathbf{b}$
    (Note: $\mathbf{W}^T \mathbf{x} = (\mathbf{x}^T \mathbf{W})^T$)
  - $\mathbf{z} = \mathbf{z}_{out} = \sigma(\mathbf{z}_{in}) = \sigma(\mathbf{W}^T \mathbf{x} + \mathbf{b})$, where the (hidden layer) activation function σ is applied element-wise to $\mathbf{z}_{in}$.

SINGLE HIDDEN LAYER NETWORKS: NOTATION / 2

- **Bias term**:
  - We sometimes omit the bias term by adding a constant feature to the input $\tilde{\mathbf{x}} = (1, x_1, ..., x_p)$ and by adding the bias term to the weight matrix

$$\tilde{\mathbf{W}} = (\mathbf{b}, \mathbf{W}_1, ..., \mathbf{W}_p).$$

  - **Note**: For simplification purposes, we will not explicitly represent the bias term graphically in the following. However, the above "trick" makes it straightforward to represent it graphically.

SINGLE HIDDEN LAYER NETWORKS: NOTATION / 3

Output layer:

- For regression or binary classification: one output unit f where
  - $f_{in} = \mathbf{u}^T\mathbf{z} + c$, i.e. a linear combination of derived features plus the bias term c of the output neuron, and
  - $f(\mathbf{x}) = f_{out} = \tau(f_{in}) = \tau(\mathbf{u}^T\mathbf{z} + c)$, where τ is the output activation function.
- For regression τ is the identity function.
- For binary classification, τ is a sigmoid function.
- **Note**: The purpose of the hidden-layer activation function σ is to introduce non-linearities so that the network is able to learn complex functions whereas the purpose of τ is merely to get the final score to the same range as the target.

SINGLE HIDDEN LAYER NETWORKS: NOTATION / 4

Multiple inputs:

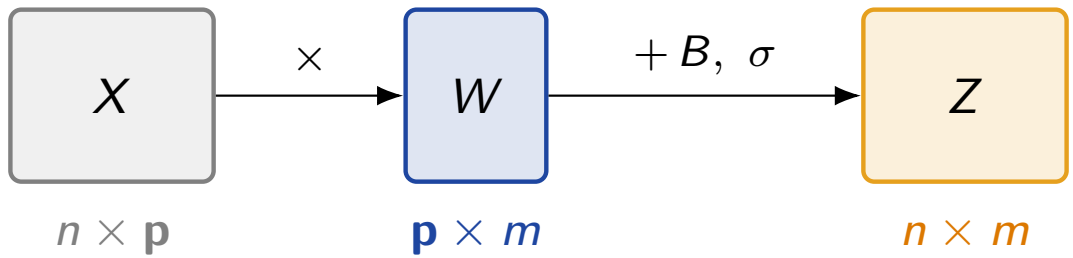
- It is possible to feed multiple inputs to a neural network simultaneously.
- The inputs $\mathbf{x}^{(i)}$, for $i \in \{1, \ldots, n\}$, are arranged as rows in the **design matrix X**.
  - **X** is a $(n \times p)$-matrix.
- The weighted sum in the hidden layer is now computed as $\mathbf{XW} + \boldsymbol{B}$, where,
  - **W**, as usual, is a $(p \times m)$ matrix, and,
  - $\boldsymbol{B}$ is a $(n \times m)$ matrix containing the bias vector **b** (duplicated) as the rows of the matrix.
- The *matrix* of hidden activations $\boldsymbol{Z} = \sigma(\mathbf{XW} + \boldsymbol{B})$
  - $\boldsymbol{Z}$ is a $(n \times m)$ matrix.

SINGLE HIDDEN LAYER NETWORKS: NOTATION / 5

- The final output of the network, which contains a prediction for each input, is $\tau(\boldsymbol{Z}\mathbf{u} + \boldsymbol{C})$, where
  - $\mathbf{u}$ is the vector of weights of the output neuron, and,
  - $\boldsymbol{C}$ is a $(n \times 1)$ matrix whose elements are the (scalar) bias c of the output neuron.

Improving the notation: track the shapes

LMU's notation is compact but abstract – pure symbols. The one thing to hold onto is that **the shapes must line up**: a whole batch of n inputs flows through the layer as matrix multiplies.

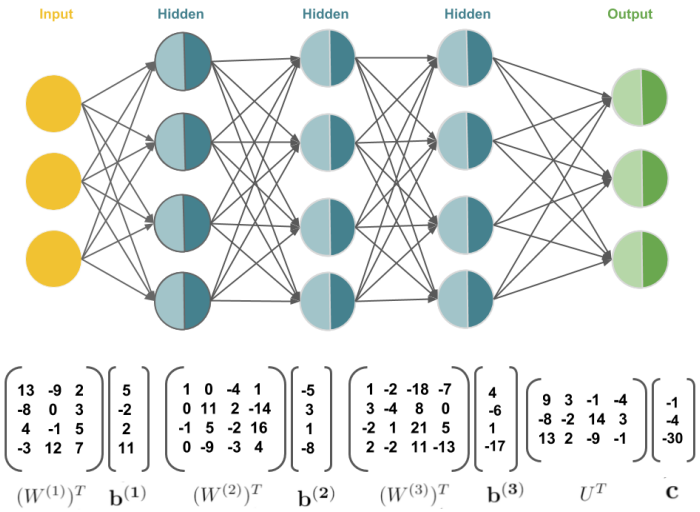


- Hidden layer: $Z = \sigma(XW + B)$ has shape $(n \times m)$ – the **inner** p's must match (X's columns $=$ W's rows), and they cancel.
- Output layer: $f = \tau(Zu + C)$ has shape $(n \times 1)$ – one prediction per input.

Whenever the algebra feels abstract, **write the shape under each symbol**. Two adjacent inner dimensions that don't match = an illegal multiply – the fastest bug-catch in deep learning.

Deep Learning

MLP – Multi-Layer Feedforward Neural Networks



Learning goals

- Architectures of deep neural networks
- Deep neural networks as chained functions

FEEDFORWARD NEURAL NETWORKS

- We will now extend the model class once again, such that we allow an arbitrary amount l of hidden layers.

- The general term for this model class is (multi-layer) **feedforward networks** (inputs are passed through the network from left to right, no feedback-loops are allowed)

FEEDFORWARD NEURAL NETWORKS / 2

- We can characterize those models by the following chain structure:

$$f = \tau \circ \phi \circ \sigma^{(l)} \circ \phi^{(l)} \circ \sigma^{(l-1)} \circ \phi^{(l-1)} \circ \ldots \circ \sigma^{(1)} \circ \phi^{(1)}$$

  where $\sigma^{(i)}$ and $\phi^{(i)}$ are the activation function and the weighted sum of hidden layer i, respectively. τ and ϕ are the corresponding components of the output layer.

- Each hidden layer has:
  - an associated weight matrix $\mathbf{W}^{(i)}$, bias $\mathbf{b}^{(i)}$, and activations $\mathbf{z}^{(i)}$ for $i \in \{1 \ldots l\}$.
  - $\mathbf{z}^{(i)} = \sigma^{(i)}(\phi^{(i)}) = \sigma^{(i)}(\mathbf{W}^{(i)T}\mathbf{z}^{(i-1)} + \mathbf{b}^{(i)})$, where $\mathbf{z}^{(0)} = \mathbf{x}$.

- Again, without non-linear activations in the hidden layers, the network can only learn linear decision boundaries.

FEEDFORWARD NEURAL NETWORKS / 3

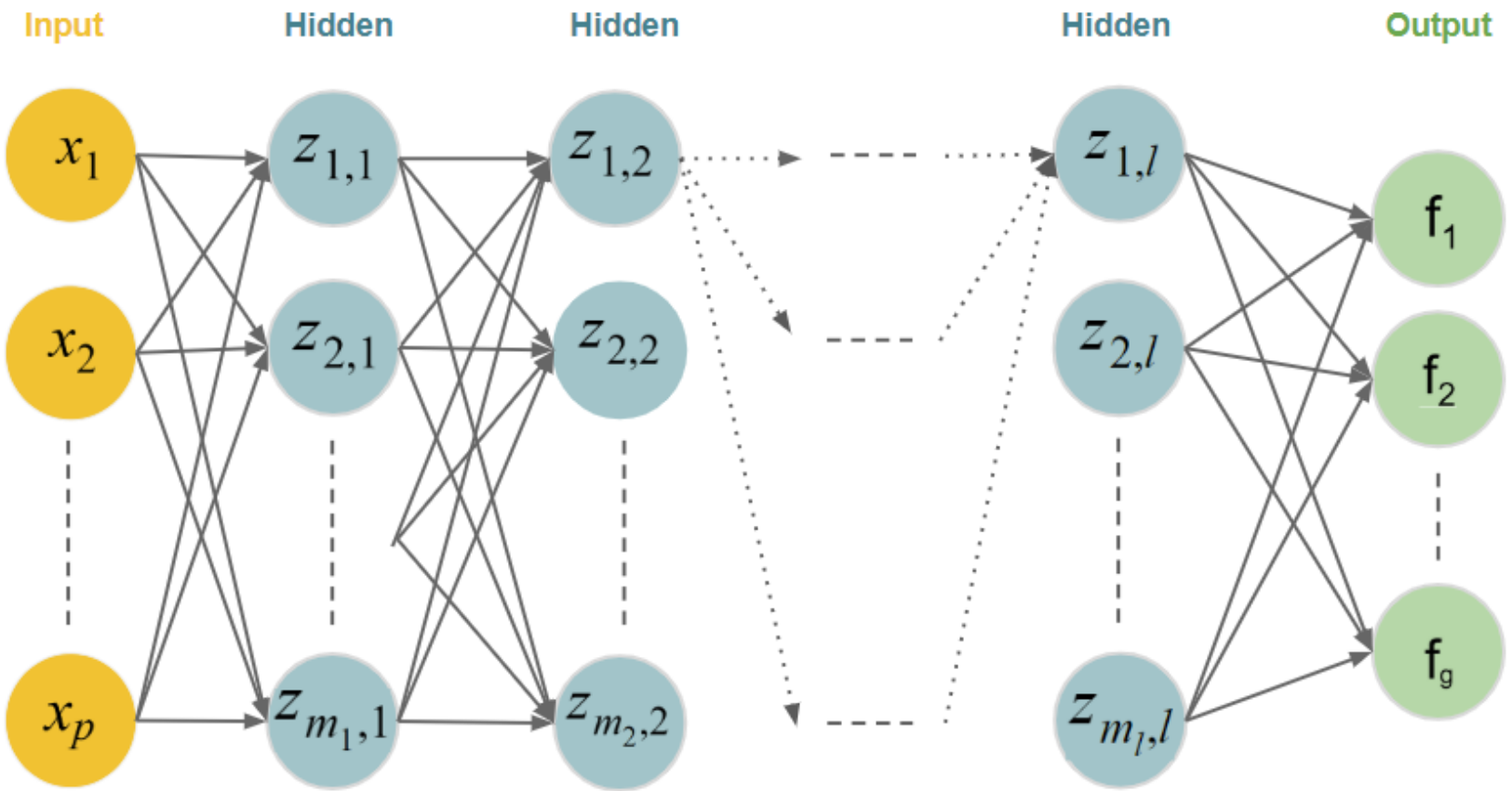
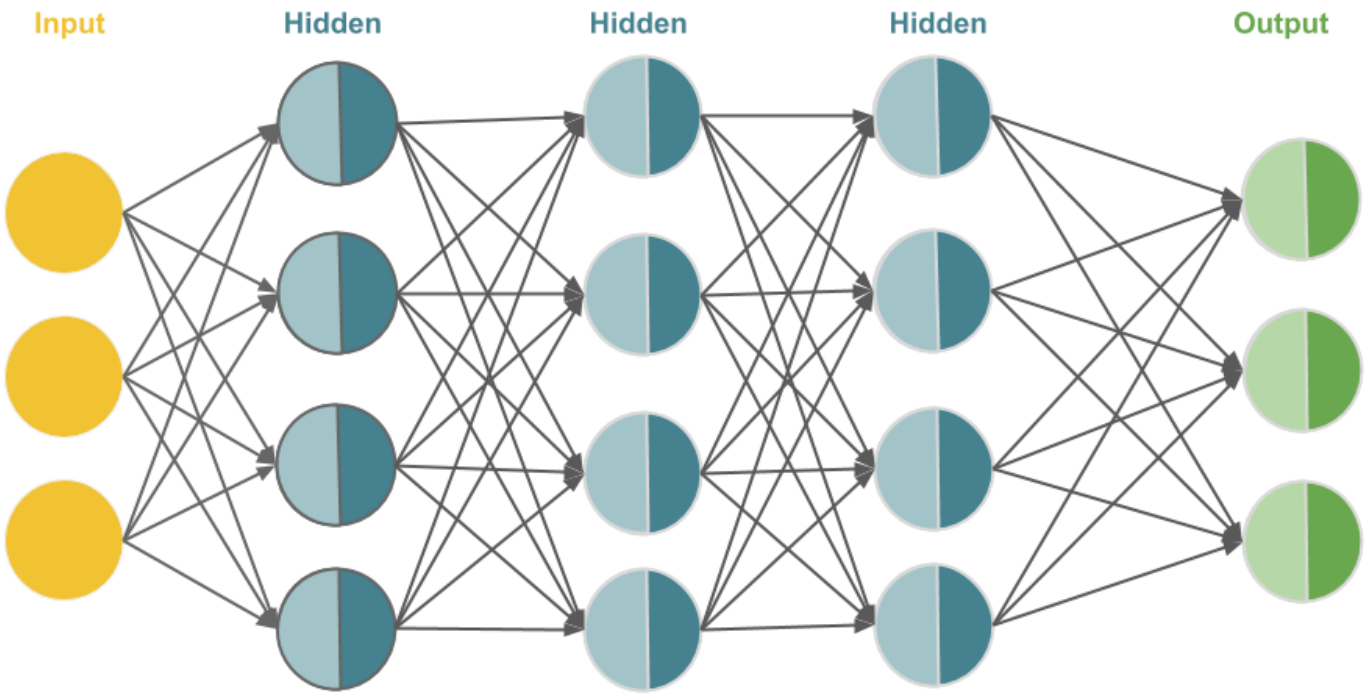


Figure: Structure of a deep neural network with *l* hidden layers (bias terms omitted).

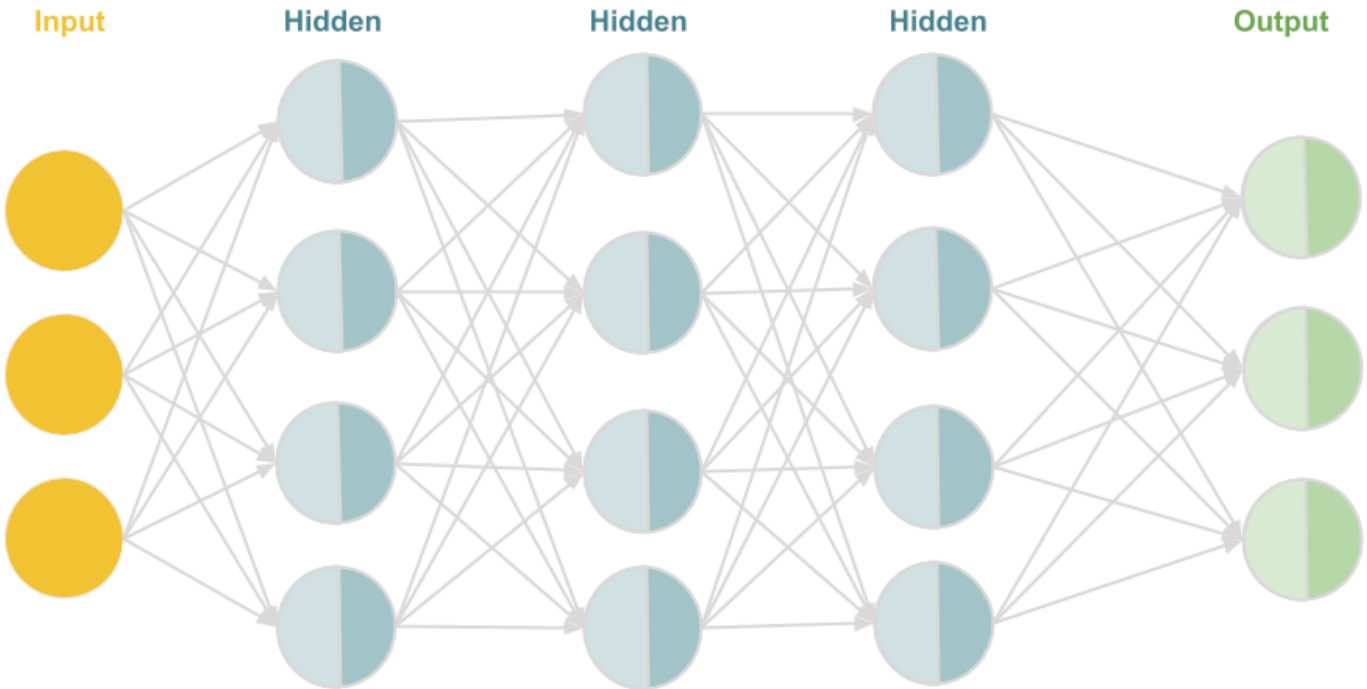
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



$$\begin{pmatrix} 13 & -9 & 2 \\ -8 & 0 & 3 \\ 4 & -1 & 5 \\ -3 & 12 & 7 \end{pmatrix} \begin{pmatrix} 5 \\ -2 \\ 2 \\ 11 \end{pmatrix} \begin{pmatrix} 1 & 0 & -4 & 1 \\ 0 & 11 & 2 & -14 \\ -1 & 5 & -2 & 16 \\ 0 & -9 & -3 & 4 \end{pmatrix} \begin{pmatrix} -5 \\ 3 \\ 1 \\ -8 \end{pmatrix} \begin{pmatrix} 1 & -2 & -18 & -7 \\ 3 & -4 & 8 & 0 \\ -2 & 1 & 21 & 5 \\ 2 & -2 & 11 & -13 \end{pmatrix} \begin{pmatrix} 4 \\ -6 \\ 1 \\ -17 \end{pmatrix} \begin{pmatrix} 9 & 3 & -1 & -4 \\ -8 & -2 & 14 & 3 \\ 13 & 2 & -9 & -1 \end{pmatrix} \begin{pmatrix} -1 \\ -4 \\ -30 \end{pmatrix}$$

$$(W^{(1)})^T \quad \mathbf{b}^{(1)} \quad (W^{(2)})^T \quad \mathbf{b}^{(2)} \quad (W^{(3)})^T \quad \mathbf{b}^{(3)} \quad U^T \quad \mathbf{c}$$

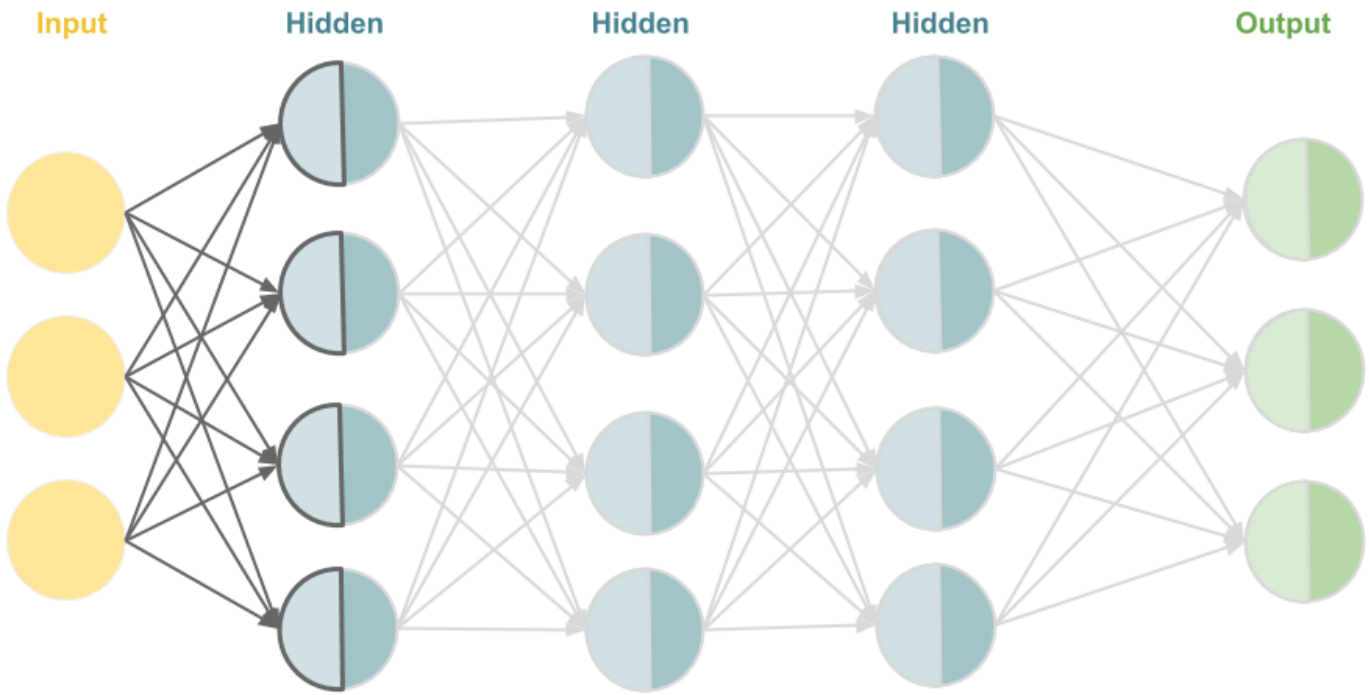
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



7
1
-4

$\mathbf{x}$

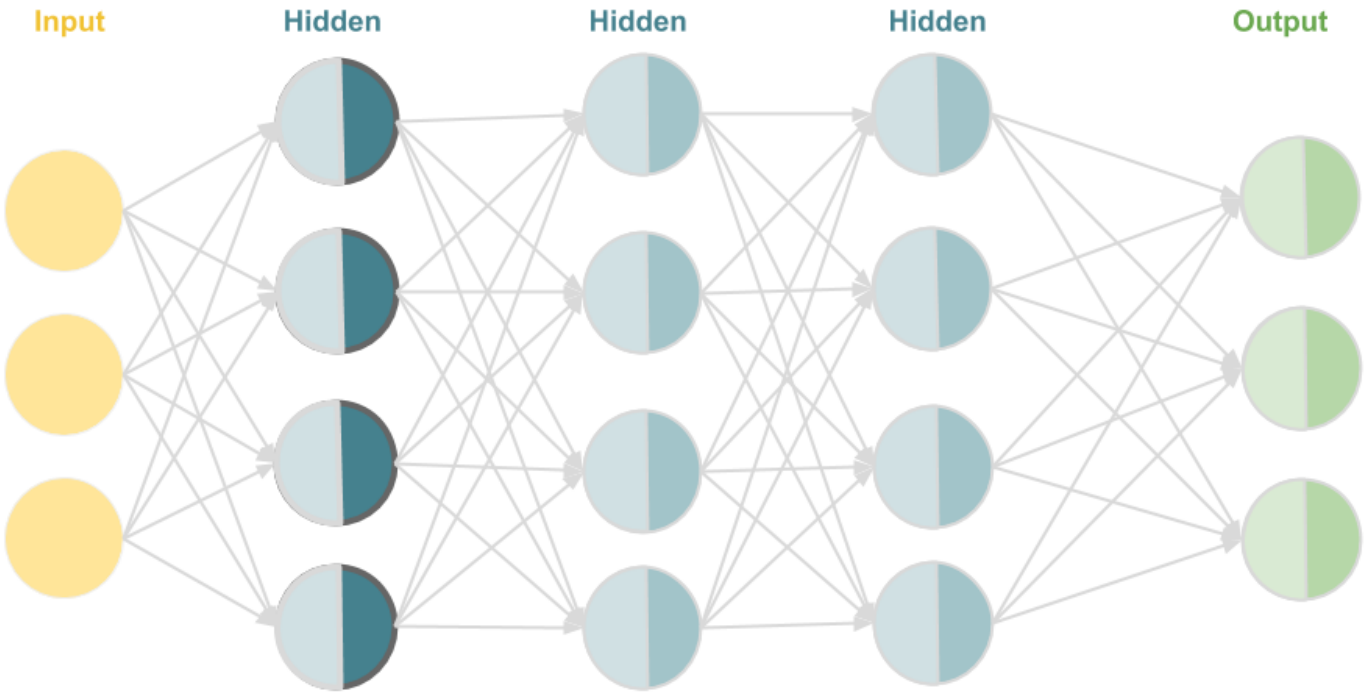
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



$$\begin{matrix} 7 \\ 1 \\ -4 \end{matrix} \begin{pmatrix} 7*13 + 1*(-9) + (-4)*2 + 5 \\ 7*(-8) + 1*0 + (-4)*3 + (-2) \\ 7*4 + 1*(-1) + (-4)*5 + 2 \\ 7*(-3) + 1*12 + (-4)*7 + 11 \end{pmatrix}$$

$\mathbf{x}$ $\quad \mathbf{z}_{in}^{(1)} = W^{(1)T}\mathbf{x} + \mathbf{b}^{(1)}$

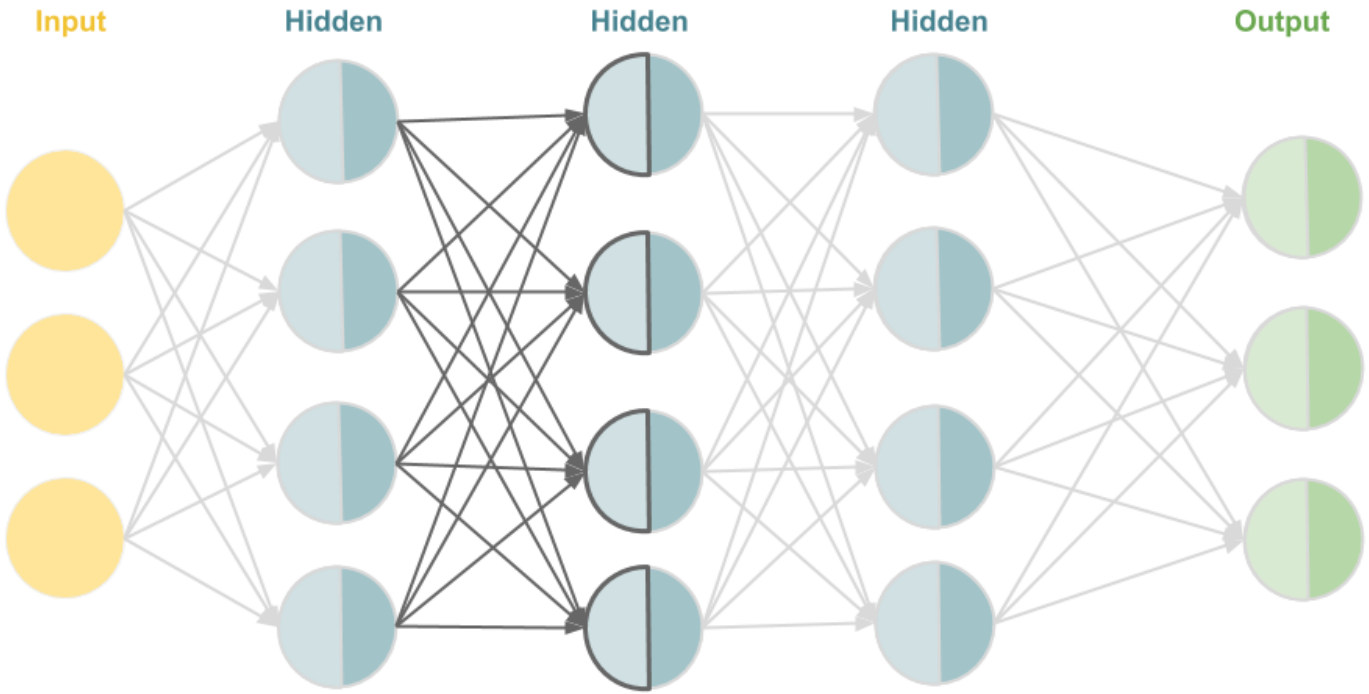
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



$$\begin{bmatrix} 7 \\ 1 \\ -4 \end{bmatrix} \quad \begin{bmatrix} 79 \\ -70 \\ 9 \\ -26 \end{bmatrix} \quad \begin{bmatrix} \max(0, 79) \\ \max(0, -70) \\ \max(0, 9) \\ \max(0, -26) \end{bmatrix}$$

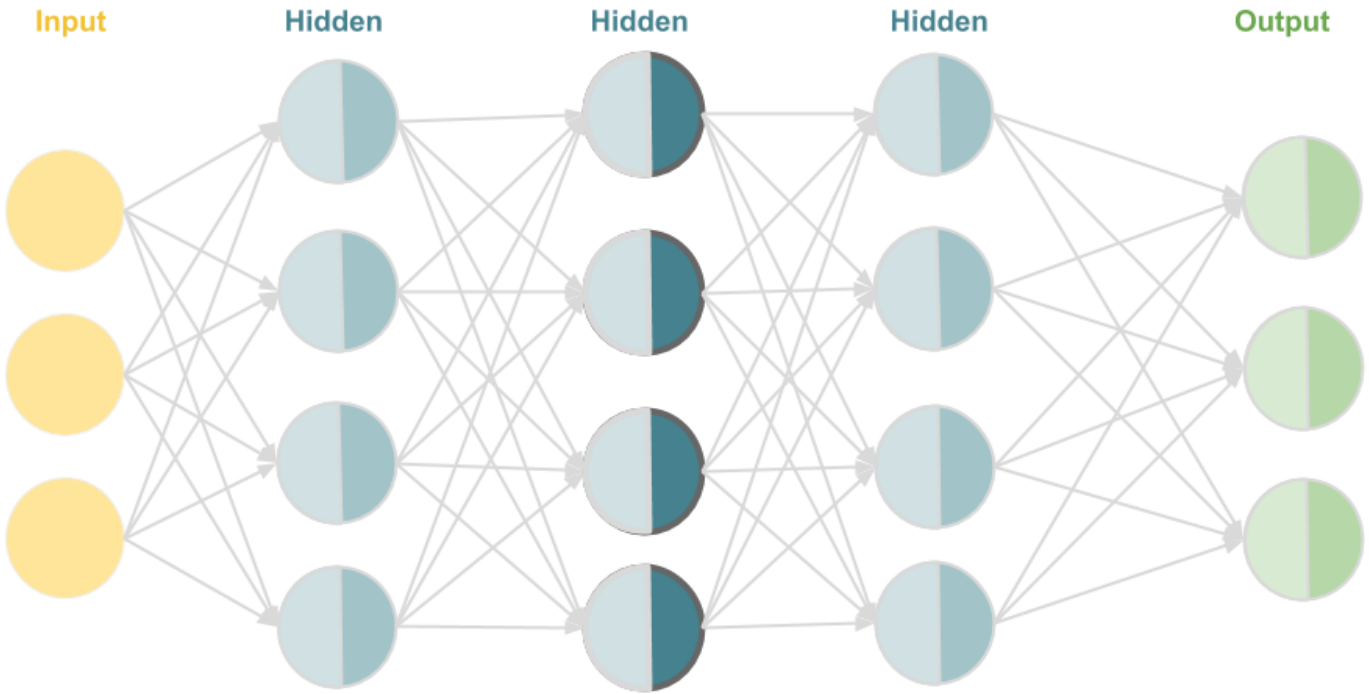
$\mathbf{x} \quad \mathbf{z}_{in}^{(1)} \quad \mathbf{z}^{(1)} = \mathbf{z}_{out}^{(1)} = \sigma(\mathbf{z}_{in}^{(1)})$

FEEDFORWARD NEURAL NETWORKS: EXAMPLE



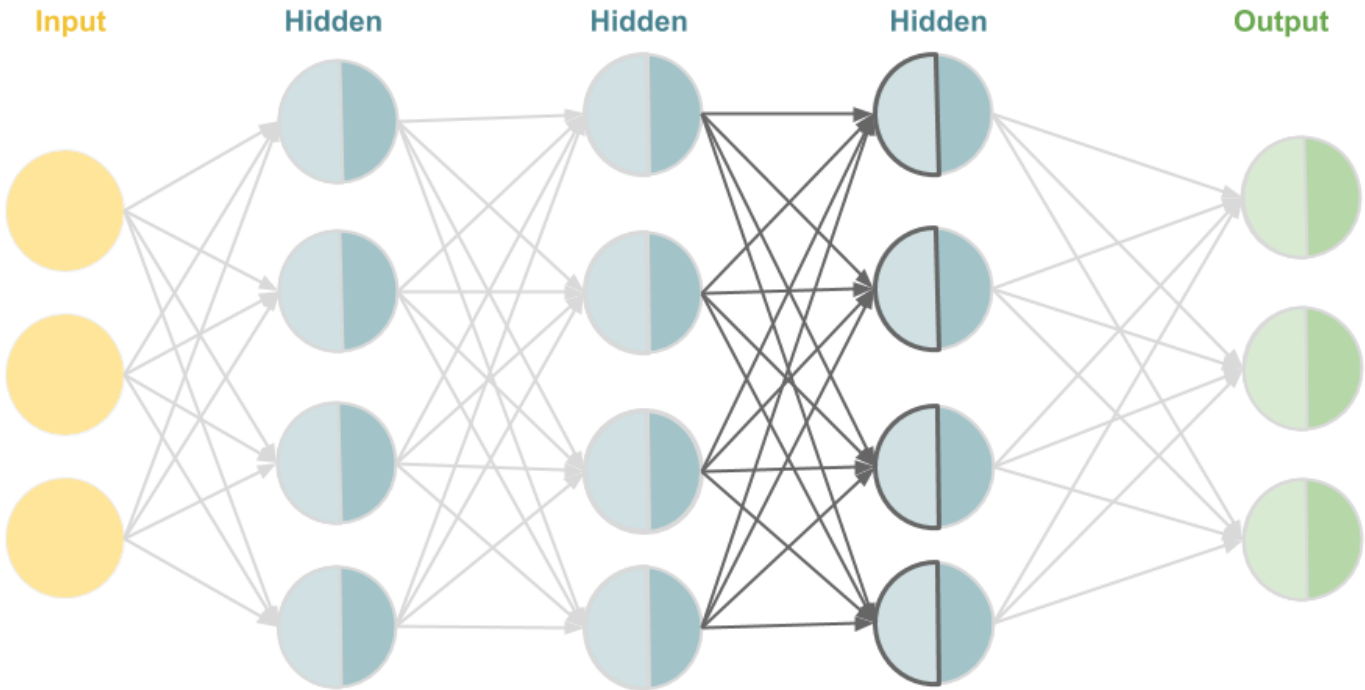
$\mathbf{x}$	$\mathbf{z}_{in}^{(1)}$	$\mathbf{z}^{(1)}$	$\mathbf{z}_{in}^{(2)} = W^{(2)T}\mathbf{z}^{(1)} + \mathbf{b}^{(2)}$
7 1 -4	79 -70 9 -26	79 0 9 0	79*1 + 0*0 + 9*(-4) + 0*1 + (-5) 79*0 + 0*11 + 9*2 + 0*(-14) + 3 79*(-1) + 0*5 + 9*(-2) + 0*16 + 1 79*0 + 0*(-9) + 9*(-3) + 0*4 + (-8)

FEEDFORWARD NEURAL NETWORKS: EXAMPLE

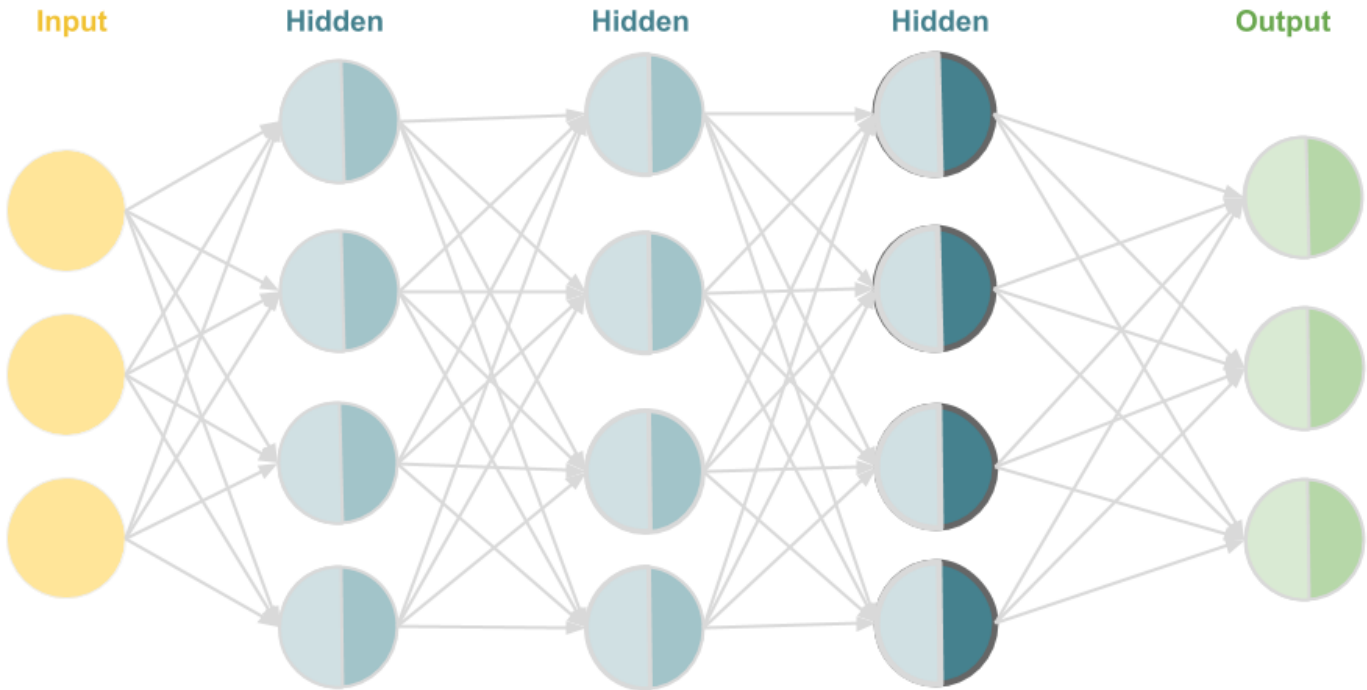


$\mathbf{x}$	$\mathbf{z}_{in}^{(1)}$	$\mathbf{z}^{(1)}$	$\mathbf{z}_{in}^{(2)}$	$\mathbf{z}^{(2)} = \mathbf{z}_{out}^{(2)} = \sigma(\mathbf{z}_{in}^{(2)})$
7	79	79	38	max(0, 38)
1	-70	0	21	max(0, 21)
-4	9	9	-96	max(0, -96)
	-26	0	-35	max(0, -36)

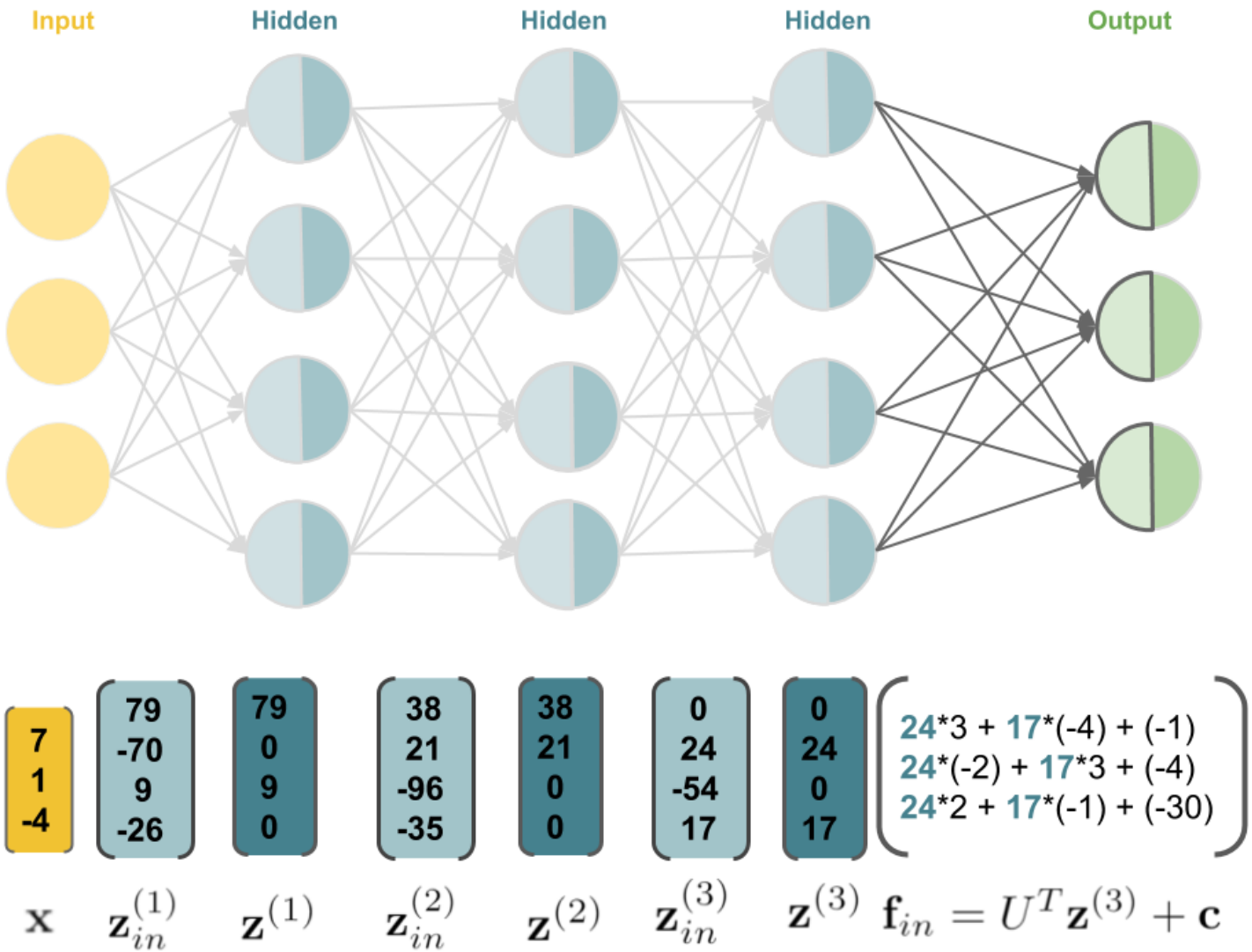
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



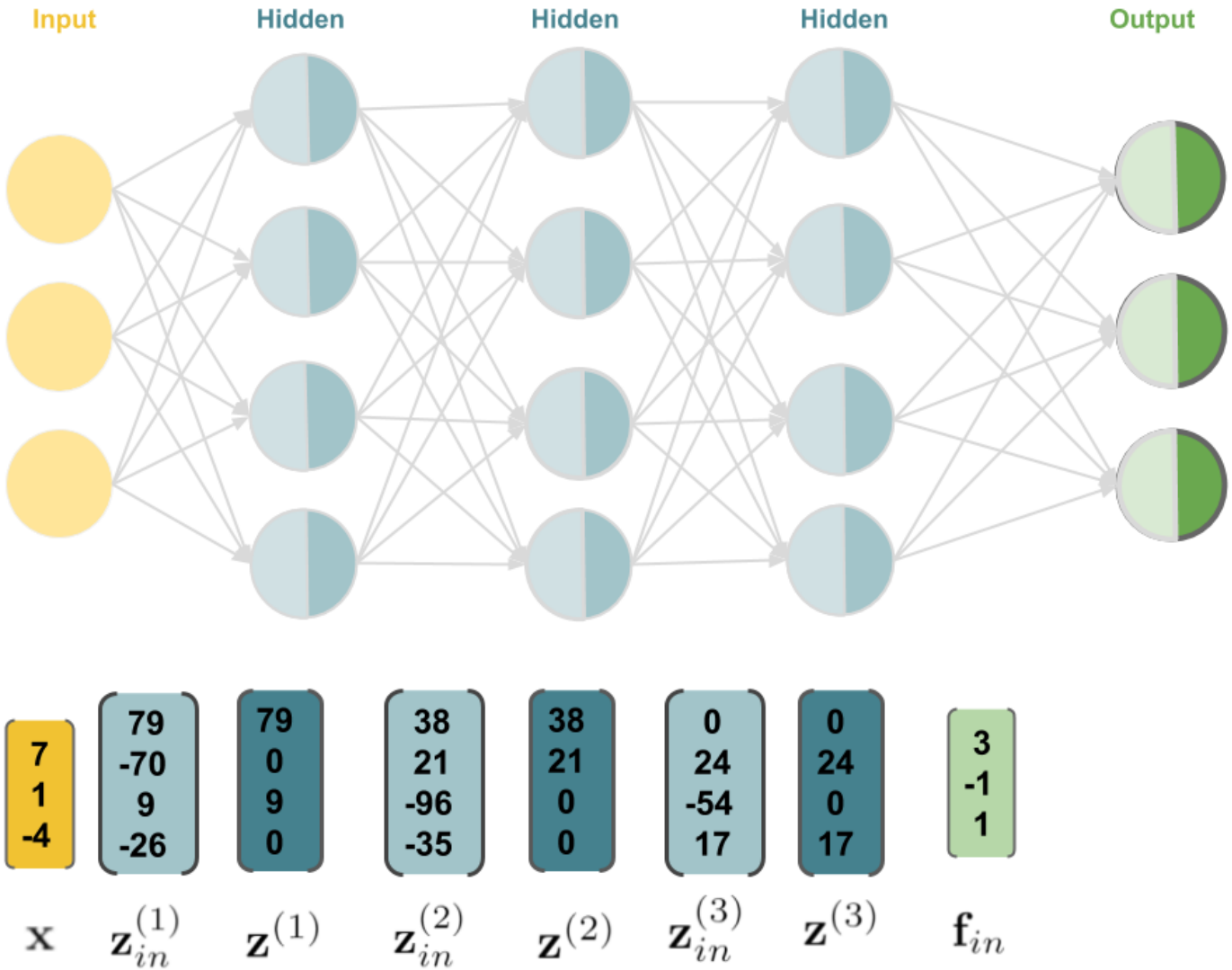
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



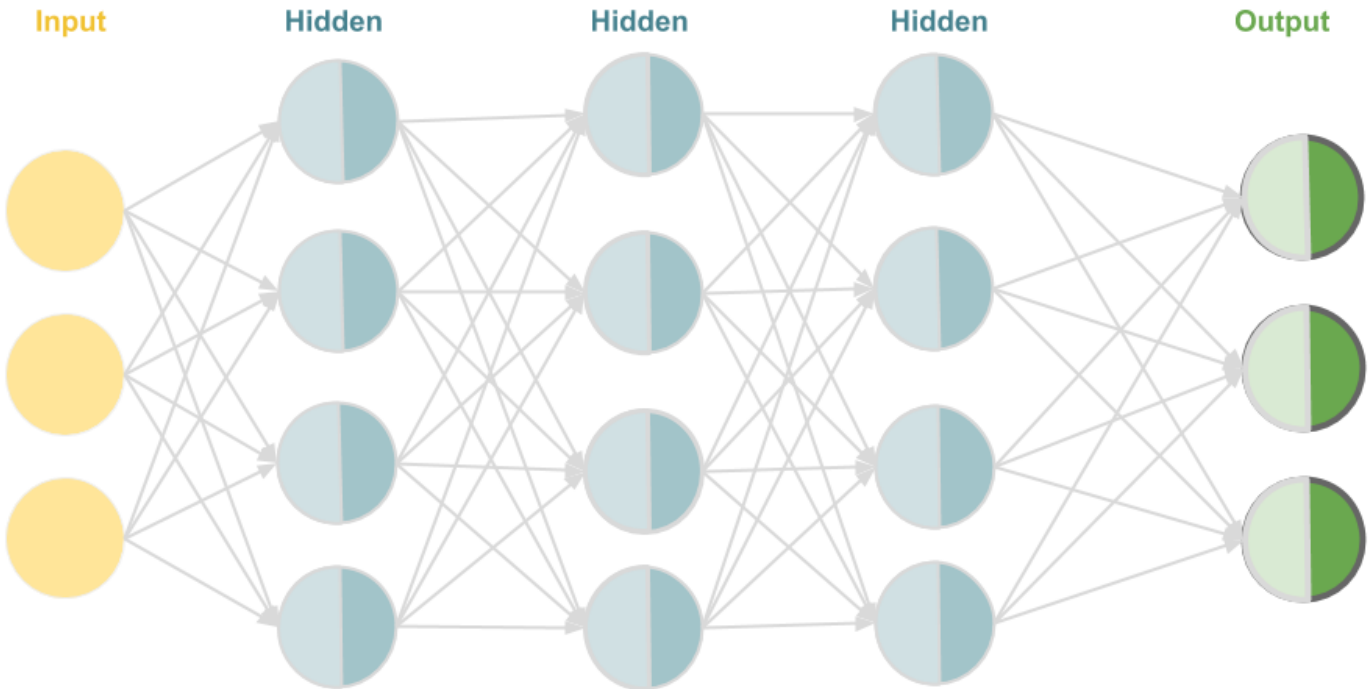
FEEDFORWARD NEURAL NETWORKS: EXAMPLE



FEEDFORWARD NEURAL NETWORKS: EXAMPLE



FEEDFORWARD NEURAL NETWORKS: EXAMPLE



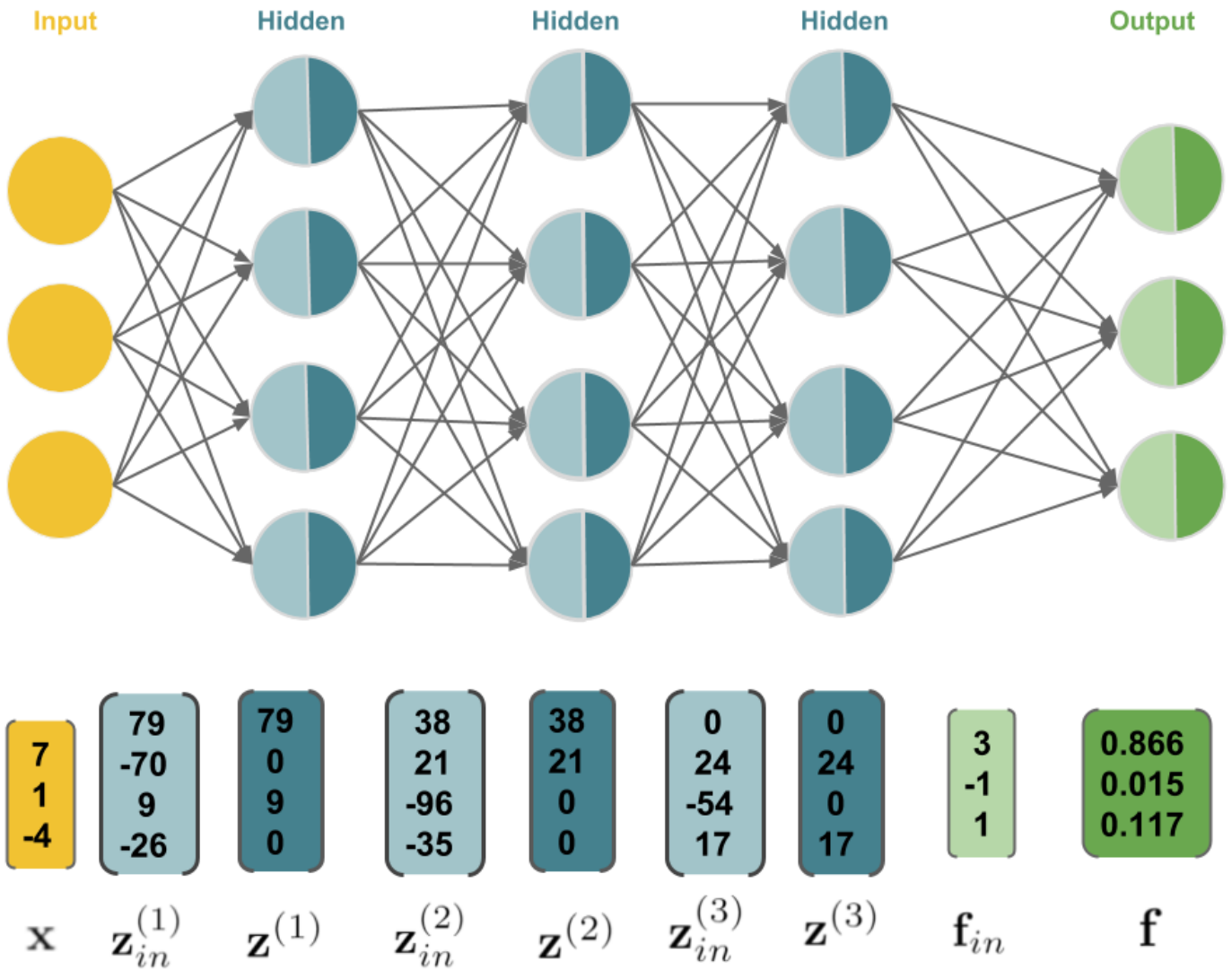
$$\begin{bmatrix} 3 \\ -1 \\ 1 \end{bmatrix}$$

$\mathbf{f}_{in}$

$$\begin{bmatrix} \frac{\exp(3)}{\exp 3+\exp(-1)+\exp(1)} \\ \frac{\exp(-1)}{\exp 3+\exp(-1)+\exp(1)} \\ \frac{\exp(1)}{\exp 3+\exp(-1)+\exp(1)} \end{bmatrix}$$

$$\mathbf{f} = \mathbf{f}_{out} = \tau(\mathbf{f}_{in})$$

FEEDFORWARD NEURAL NETWORKS: EXAMPLE



WHY ADD MORE LAYERS?

- Multiple layers allow for the extraction of more and more abstract representations.
- Each layer in a feed-forward neural network adds its own degree of non-lnearity to the model.

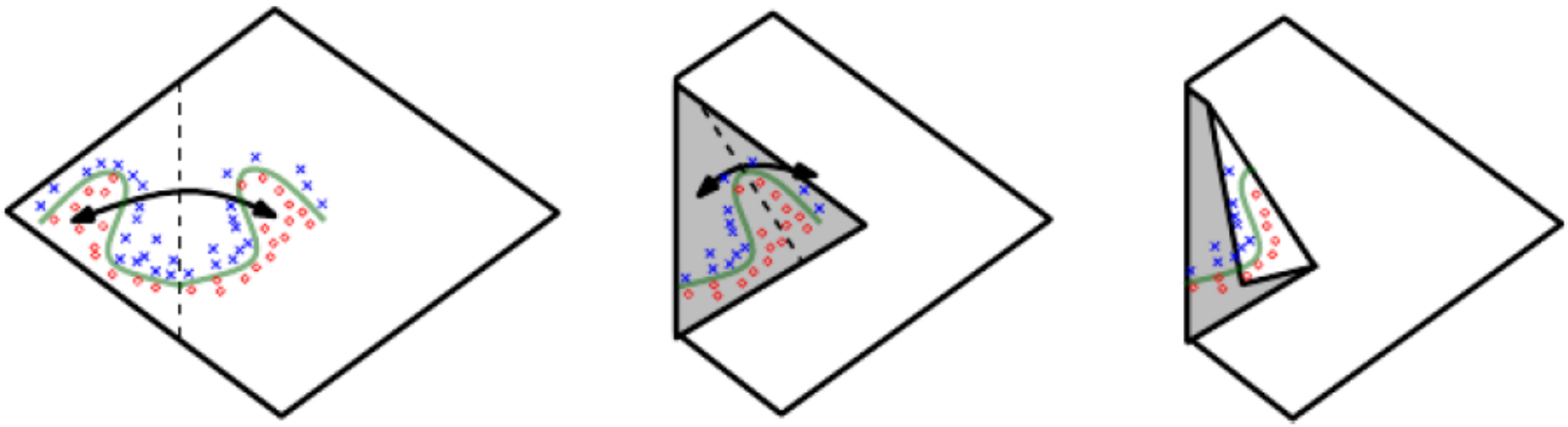


Figure: An intuitive, geometric explanation of the exponential advantage of deeper networks formally (Montúfar et al., 2014).

DEEP NEURAL NETWORKS

Neural networks today can have hundreds of hidden layers. The greater the number of layers, the "deeper" the network. Historically DNNs were very challenging to train and not popular until the late ’00s for several reasons:

- The use of sigmoid activations (e.g., logistic sigmoid and tanh) significantly slowed down training due to a phenomenon known as “vanishing gradients”. The introduction of the ReLU activation largely solved this problem.
- Training DNNs on CPUs was too slow to be practical. Switching over to GPUs cut down training time by more than an order of magnitude.
- When dataset sizes are small, other models (such as SVMs) and techniques (such as feature engineering) often outperform them.

DEEP NEURAL NETWORKS / 2

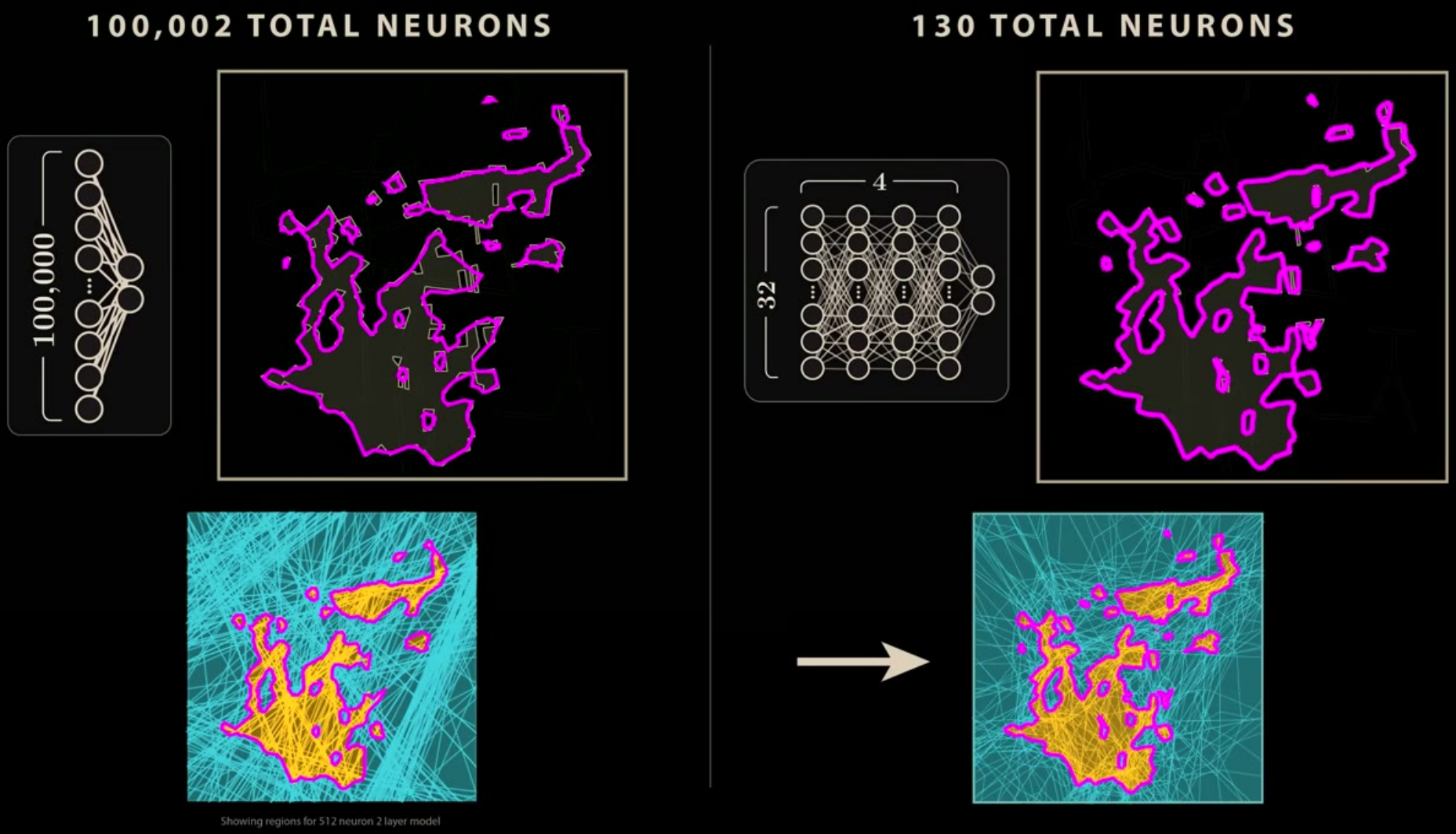
- The availability of large datasets and novel architectures that are capable of handling even complex tensor-shaped data (e.g. CNNs for image data), faster hardware, and better optimization and regularization methods made it feasible to successfully implement deep neural networks.

- An increase in depth often translates to an increase in performance on a given task. State-of-the-art neural networks, however, are much more sophisticated than the simple architectures we have encountered so far.

The term "**deep learning**" encompasses all of these developments and refers to the field as a whole.

REFERENCES

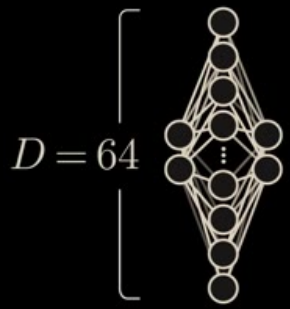
Montúfarr, G., Pascanu, R., Cho, K., & Bengio, Y. (2014). *On the Number of Linear Regions of Deep Neural Networks*.



Welch Labs (2025), *Why Deep Learning Works Unreasonably Well*: youtu.be/qx7hirqgfuU

These are theoretical *upper bounds*, and loose ones: trained deep networks do not show this exponential growth in regions in practice.

SHALLOW

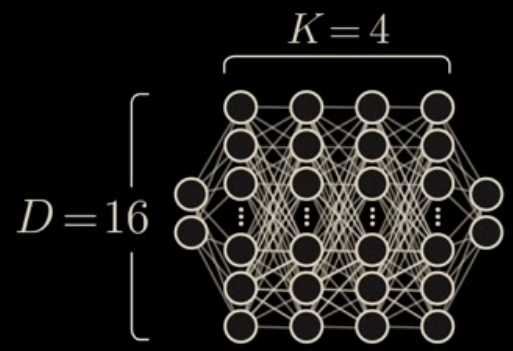


$$N_r = \frac{D^2 + D + 2}{2}$$

$$= \frac{64^2 + 64 + 2}{2}$$

$$\boxed{= 2081}$$

DEEP



$$N_r = \left(\frac{D}{D_i} + 1\right)^{D_i(K-1)} \left(\frac{D^2 + D + 2}{2}\right)$$

$$= \left(\frac{16}{2} + 1\right)^{2(4-1)} \left(\frac{16^2 + 16 + 2}{2}\right)$$

$$= 9^6 \cdot 137 = \boxed{72,807,417}$$

Welch Labs (2025), *Why Deep Learning Works Unreasonably Well*: youtu.be/qx7hirqgfuU

Reading the last slide: 2081 vs 72,807,417

Both networks take a **2D input** and have **64 ReLU neurons** in total. N_r is the **most linear regions** (flat pieces) the network can cut the input plane into.

Shallow: one layer, $D = 64$. Each neuron adds one fold line. 64 lines cut a plane into at most

$$1 + 64 + \binom{64}{2} = 1 + 64 + 2016 = 2081 \text{ pieces} = \frac{D^2+D+2}{2}.$$

Polynomial in D: double the neurons, roughly $4\times$ the regions.

Deep: $K = 4$ layers of $D = 16$. Each of the first $K - 1 = 3$ layers folds the plane onto itself, so the next layer's cuts are copied into every fold, multiplying the count by up to $(16/2 + 1)^2 = 81$:

$$\underbrace{81^3 = 9^6 = 531{,}441}_{\text{3 folding layers}} \times \underbrace{\frac{16^2+16+2}{2} = 137}_{\text{last layer's cuts}} = 72{,}807{,}417.$$

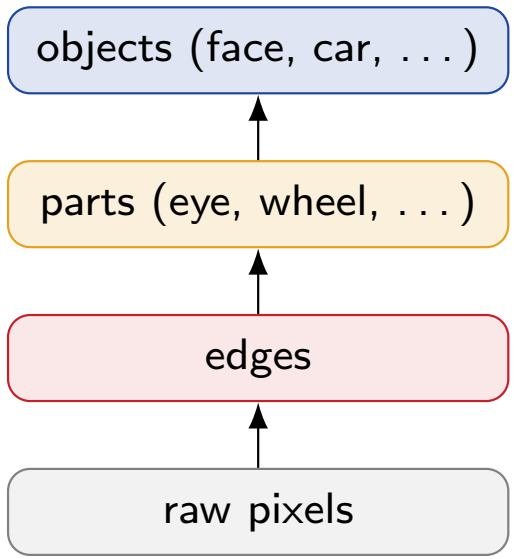
Exponential in K: each extra layer multiplies.

Same 64 neurons, $\sim$35,000$\times$ more possible regions: the usual "depth beats width" argument. But these are best-case counts; trained networks come nowhere near them.

Depth = a hierarchy of features

Folding explains *how many* regions depth buys. Here is *what* those extra layers tend to represent: each layer composes the one below into something more abstract.

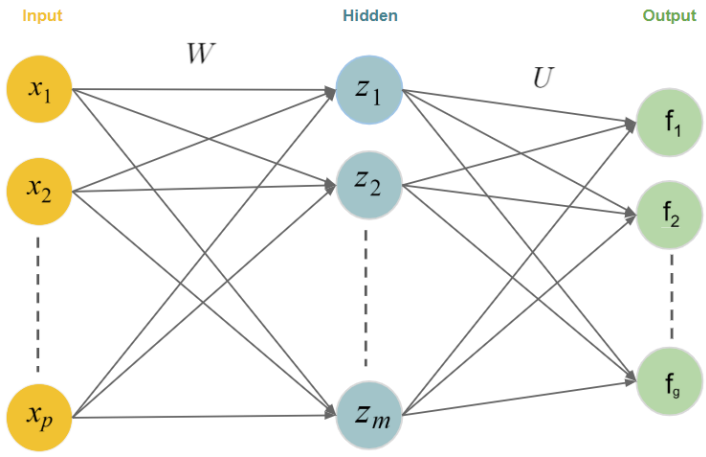
On images, a trained deep net learns roughly this progression **on its own** – nobody programs the edges or the parts. That is **representation learning**.



deeper layer = more abstract feature

Deep Learning

Single Hidden Layer Networks for Multi-Class Classification



Learning goals

- Neural network architectures for multi-class classification
- Softmax activation function
- Softmax loss

MULTI-CLASS CLASSIFICATION

- We have only considered regression and binary classification problems so far.
- How can we get a neural network to perform multiclass classification?

MULTI-CLASS CLASSIFICATION

- The first step is to add additional neurons to the output layer.
- Each neuron in the layer will represent a specific class (number of neurons in the output layer = number of classes).

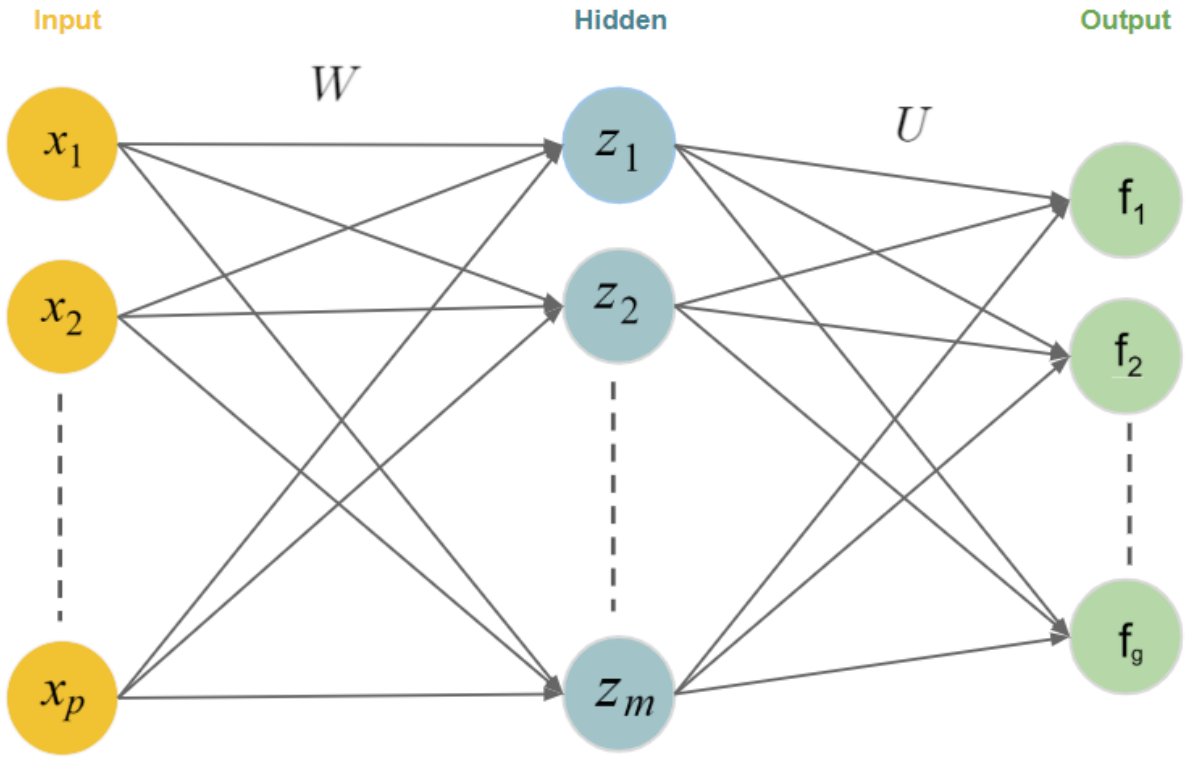


Figure: Structure of a single hidden layer, feed-forward neural network for g-class classification problems (bias term omitted).

MULTI-CLASS CLASSIFICATION

Notation:

- For g-class classification, g output units:

$$\mathbf{f} = (f_1, \ldots, f_g)$$

- m hidden neurons $z_1, \ldots, z_m$, with

$$z_j = \sigma(\mathbf{W}_j^T \mathbf{x}), \quad j = 1, \ldots, m.$$

- Compute linear combinations of derived features z:

$$f_{in,k} = \boldsymbol{U}_k^T \mathbf{z}, \quad \mathbf{z} = (z_1, \ldots, z_m)^T, \quad k = 1, \ldots, g$$

MULTI-CLASS CLASSIFICATION

- The second step is to apply a **softmax** activation function to the output layer.

- This gives us a probability distribution over g different possible classes:

$$f_{out,k} = \tau_k(f_{in,k}) = \frac{\exp(f_{in,k})}{\sum_{k'=1}^{g} \exp(f_{in,k'})}$$

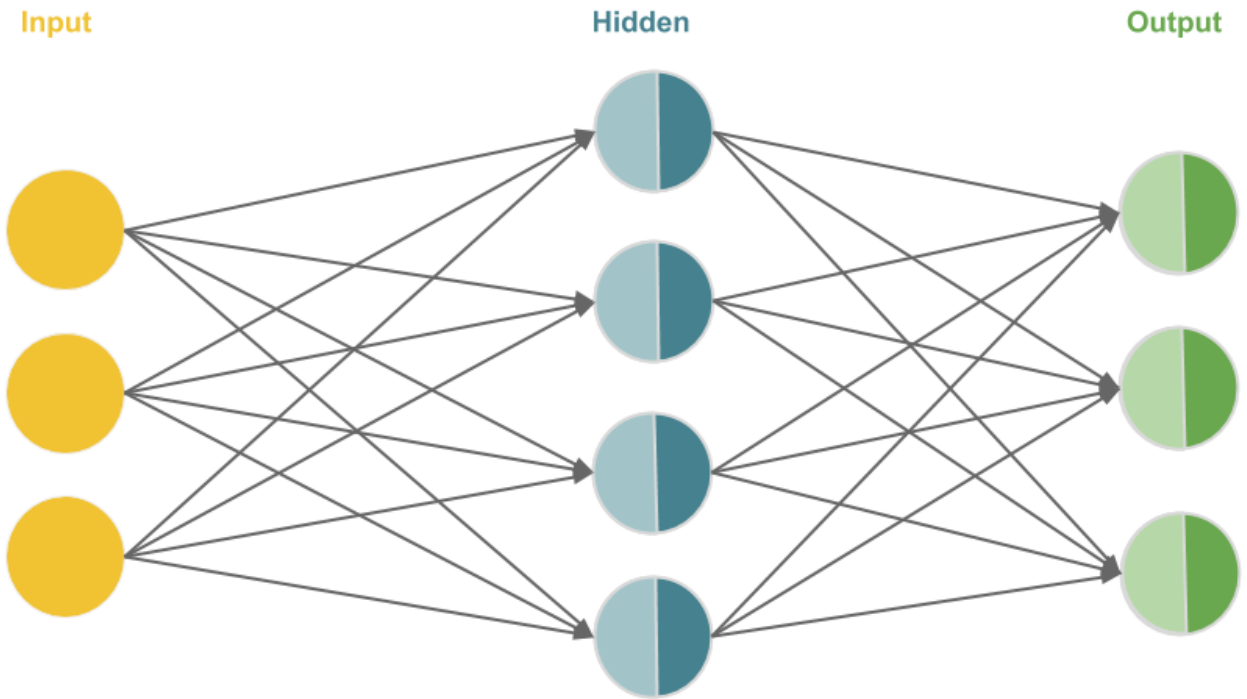
- This is the same transformation used in softmax regression!

- Derivative $\frac{\partial \tau(\mathbf{f}_{in})}{\partial \mathbf{f}_{in}} = \text{diag}(\tau(\mathbf{f}_{in})) - \tau(\mathbf{f}_{in})\tau(\mathbf{f}_{in})^T$

- It is a "smooth" approximation of the argmax operation, so $\tau((1, 1000, 2)^T) \approx (0, 1, 0)^T$ (picks out 2nd element!).

MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



$$\begin{pmatrix} 3 & -9 & 2 \\ 11 & -2 & 7 \\ -6 & 3 & -4 \\ 6 & -1 & 5 \end{pmatrix} \begin{pmatrix} 5 \\ 2 \\ -1 \\ 1 \end{pmatrix}$$

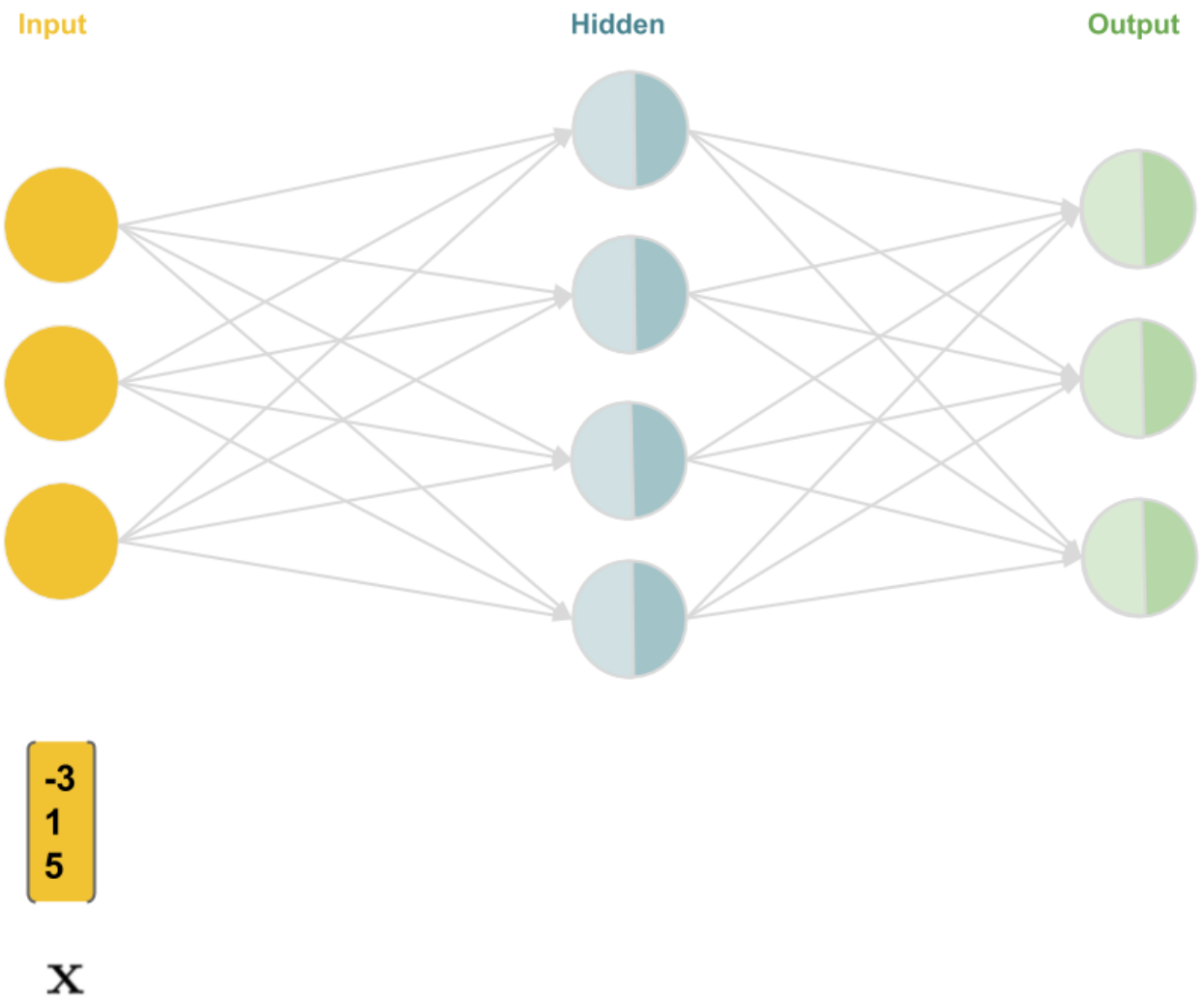
W^T $\mathbf{b}$

$$\begin{pmatrix} 3 & -12 & 8 & 1 \\ 2 & -3 & 9 & 1 \\ -5 & 1 & -1 & 7 \end{pmatrix} \begin{pmatrix} 6 \\ 0 \\ -8 \end{pmatrix}$$

U^T $\mathbf{c}$

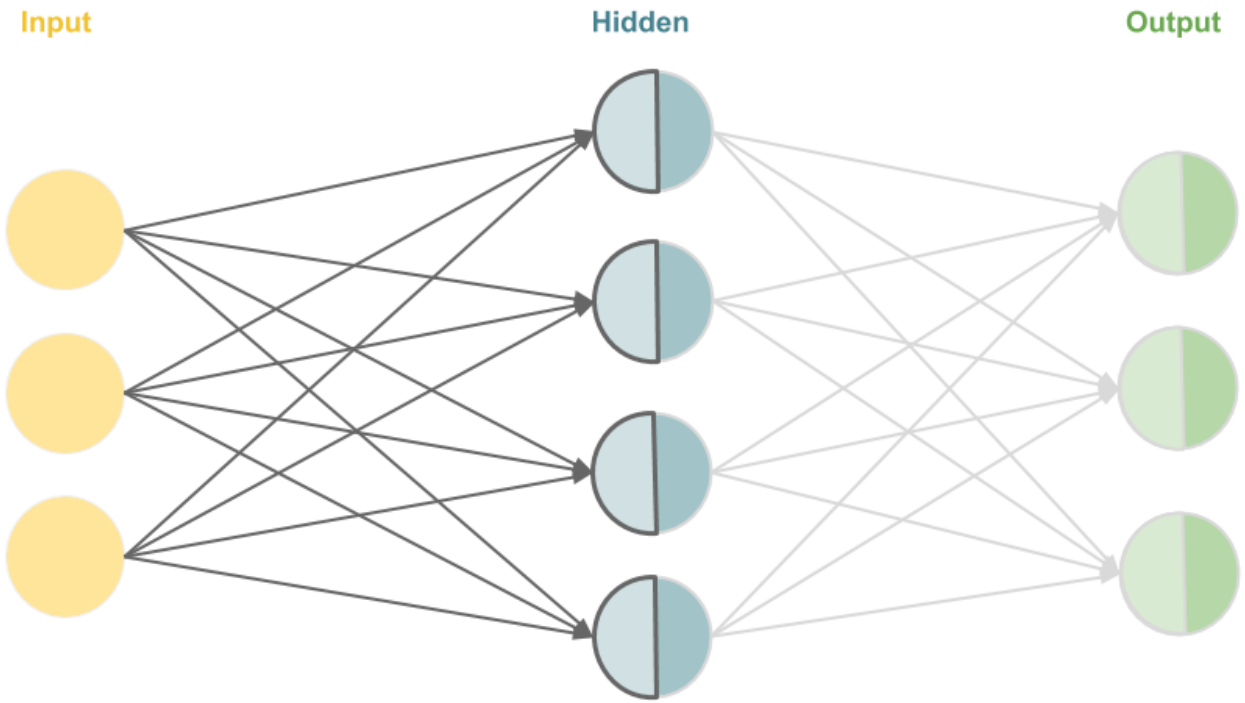
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).

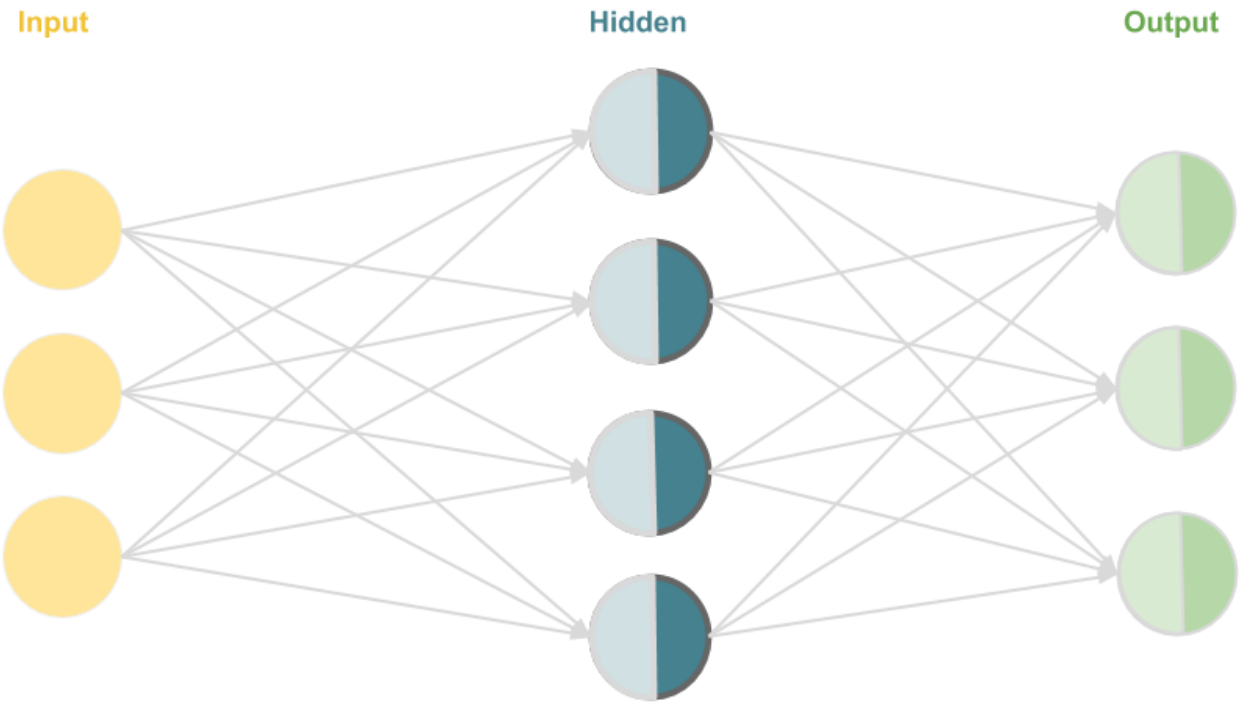


$$\begin{bmatrix} -3 \\ 1 \\ 5 \end{bmatrix} \quad \begin{pmatrix} (-3)*3 + 1*(-9) + 5*2 + 5 \\ (-3)*11 + 1*(-2) + 5*7 + 2 \\ (-3)*(-6) + 1*3 + 5*(-4) + (-1) \\ (-3)*6 + 1*(-1) + 5*5 + 1 \end{pmatrix}$$

$$\mathbf{x} \qquad \mathbf{z}_{in} = \mathbf{w}^T\mathbf{x} + \mathbf{b}$$

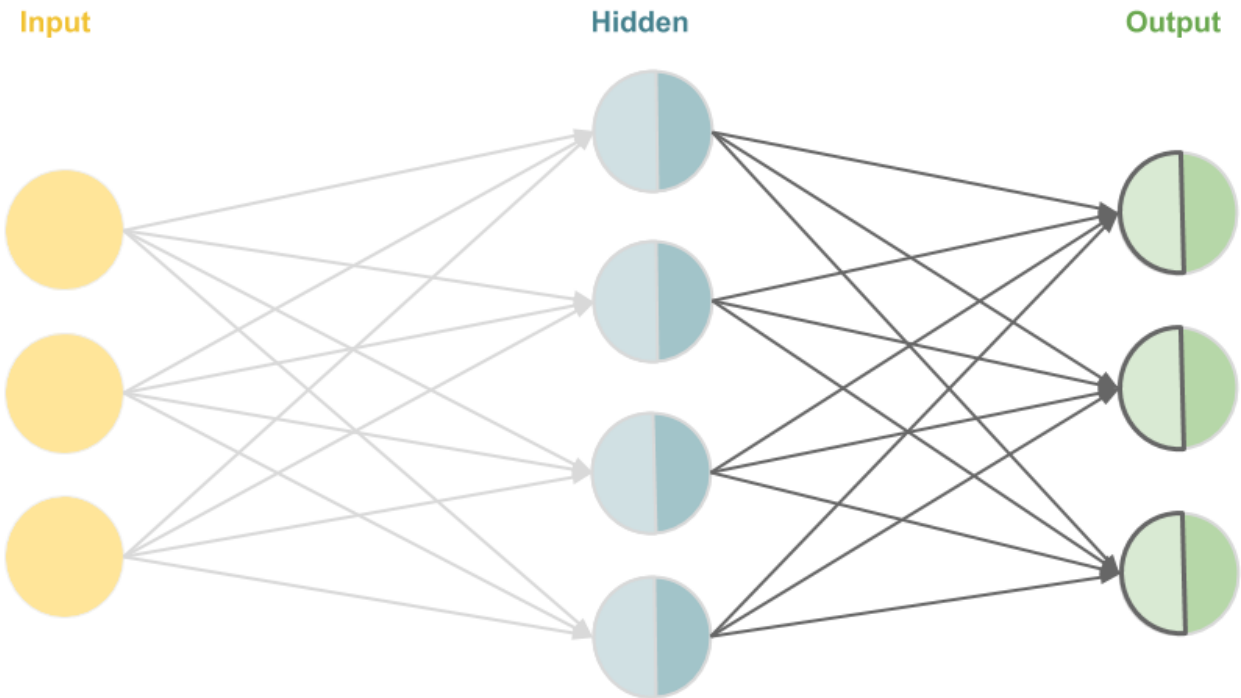
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



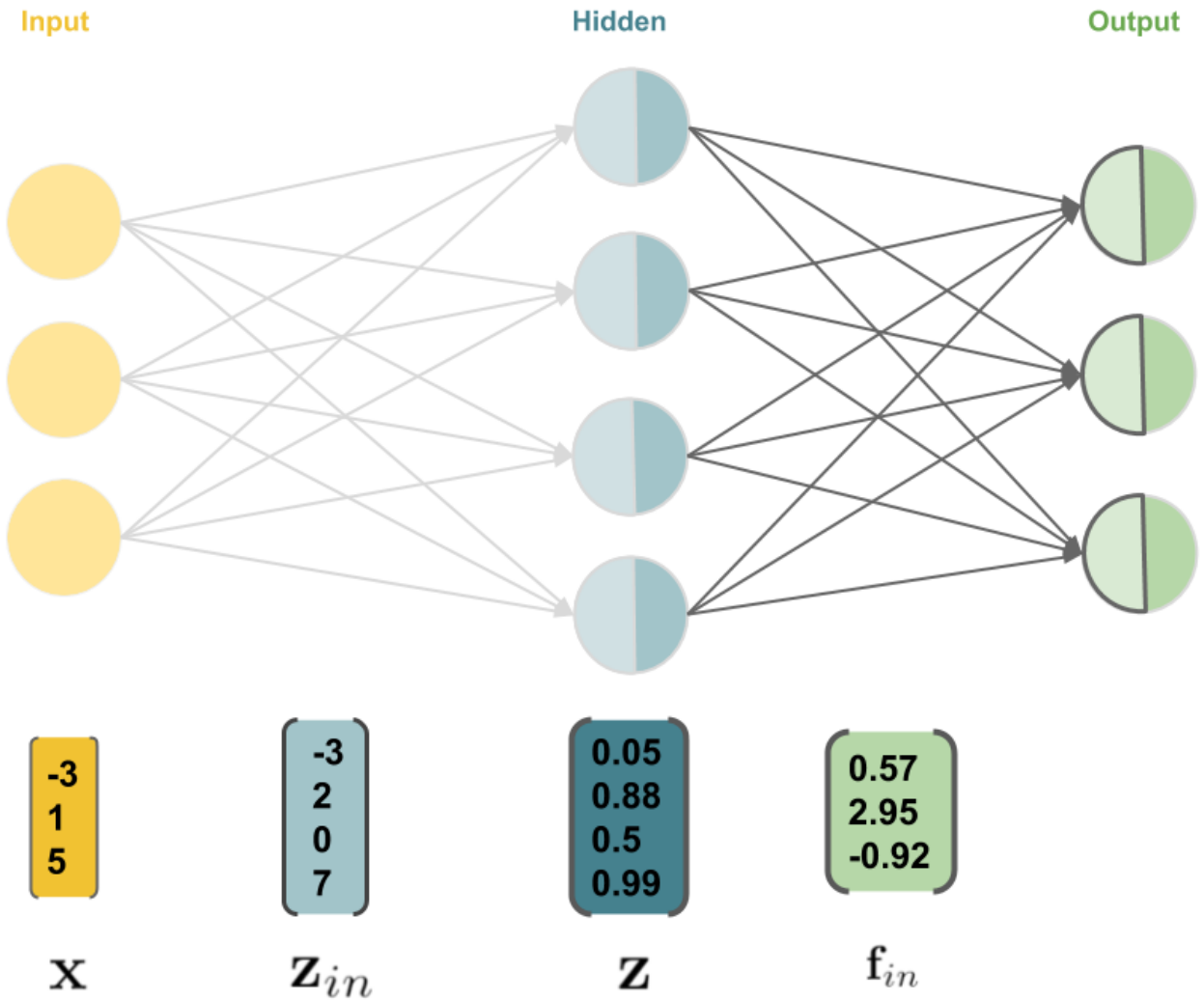
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



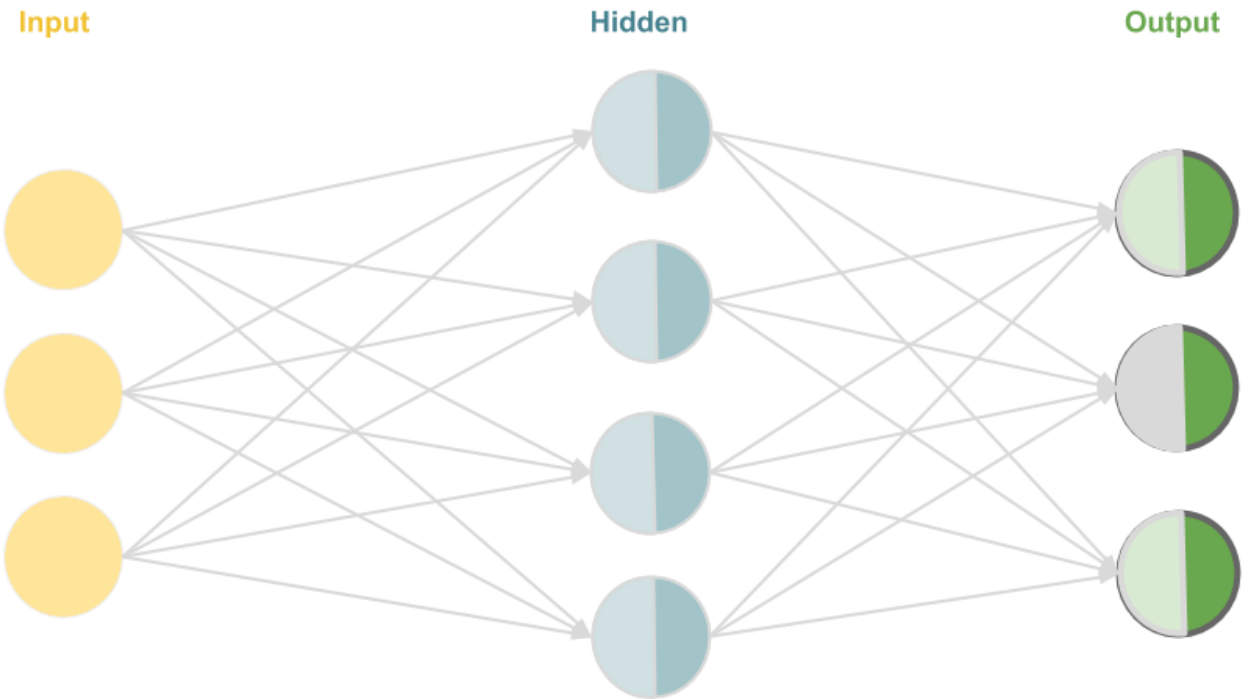
MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



0.57
2.95
-0.92

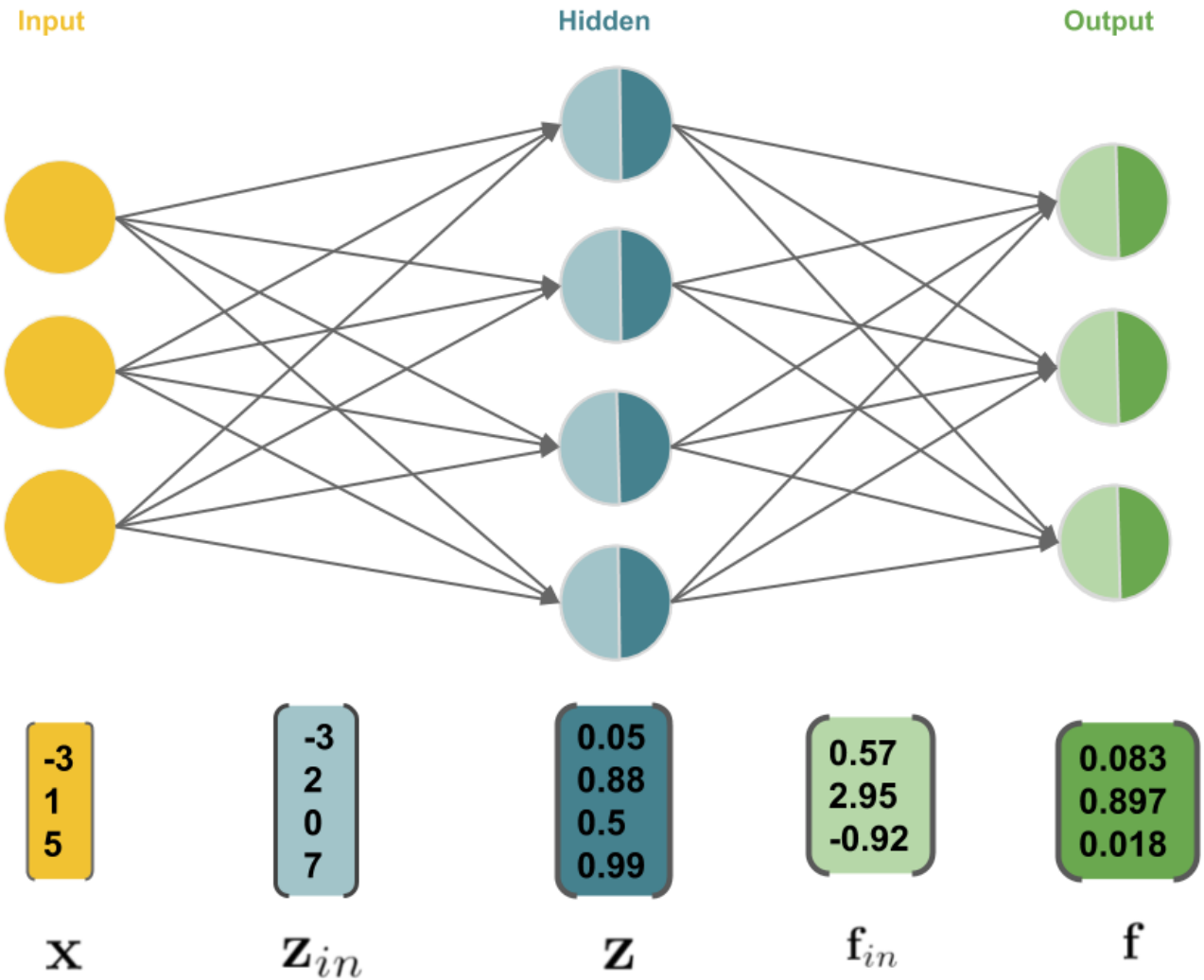
$\mathbf{f}_{in}$

$$\begin{pmatrix} \frac{\exp(0.57)}{\exp(0.57)+\exp(2.95)+\exp(-0.92)} \\ \frac{\exp(2.95)}{\exp(0.57)+\exp(2.95)+\exp(-0.92)} \\ \frac{\exp(-0.92)}{\exp(0.57)+\exp(2.95)+\exp(-0.92)} \end{pmatrix}$$

$$\mathbf{f} = \mathbf{f}_{out} = \tau(\mathbf{f}_{in})$$

MULTI-CLASS CLASSIFICATION: EXAMPLE

Forward pass (Hidden: Sigmoid, Output: Softmax).



OPTIMIZATION: SOFTMAX LOSS

- The loss function for a softmax classifier is

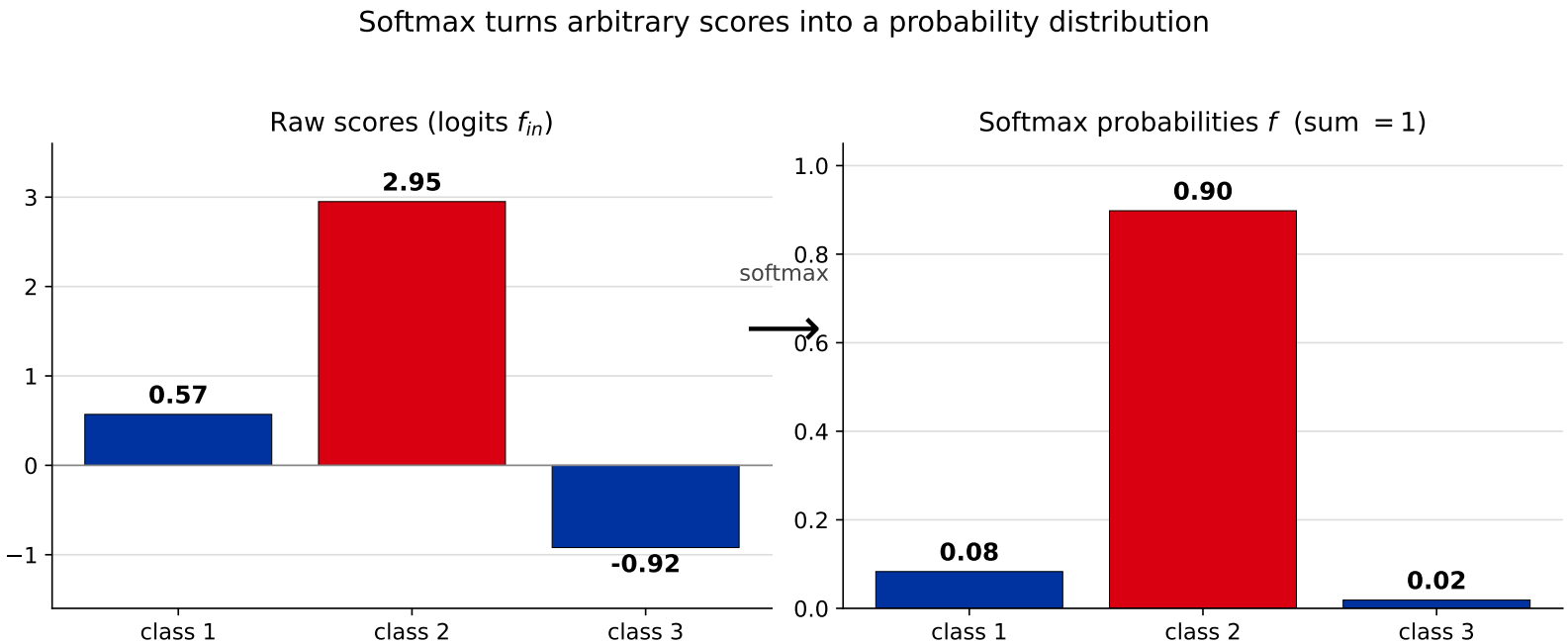
$$L(y, f(\mathbf{x})) = -\sum_{k=1}^{g} [y = k] \log \left(\frac{\exp(f_{in,k})}{\sum_{k'=1}^{g} \exp(f_{in,k'})} \right)$$

where $[y = k] = \begin{cases} 1 & \text{if } y = k \\ 0 & \text{otherwise} \end{cases}$.

- This is equivalent to the cross-entropy loss when the label vector $\boldsymbol{y}$ is one-hot coded (e.g. $\mathbf{y} = (0, 0, 1, 0)^T$).
- Optimization: Again, there is no analytic solution.

Softmax, as a picture

LMU's example ran a forward pass to the output scores $f_{in} = (0.57, 2.95, -0.92)$. Softmax turns those raw scores into a **probability distribution over the classes**:



Exponentiate, then normalise so the bars sum to 1. It is a **smooth argmax**: the largest score dominates (class 2, 0.90), but every class keeps a nonzero probability – the same softmax as multinomial logistic regression.

An old joke – and a warning about what's coming

Sherlock Holmes and Dr. Watson are lost in a hot-air balloon. They drift down and Holmes calls to a man in the field below: *"Excuse me – where are we?"*

After a long pause, the answer floats back: *"You are in a hot-air balloon, about thirty feet above this field."*

"That man," Holmes remarks to Watson, "is a **mathematician**."

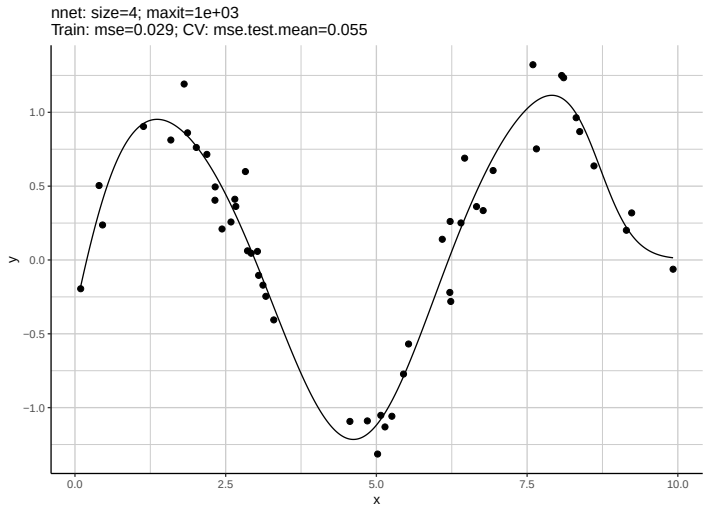
"How can you tell?"

"His answer was **slow**, **perfectly correct** – and **completely useless**."

Hold that thought for the theorem you are *about to see*: "a network can represent *any* continuous function" is perfectly correct – and, on its own, useless. It names no layer size, no training recipe, and no guarantee it will generalize.

Deep Learning

Universal Approximation



Learning goals

- Universal approximation theorem for one-hidden-layer neural networks
- The pros and cons of a low approximation error

UNIVERSAL APPROXIMATION PROPERTY

Theorem. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, non-constant, bounded, and monotonically increasing function. Let $C \subset \mathbb{R}^p$ be compact, and let $\mathcal{C}(C)$ denote the space of continuous functions $C \to \mathbb{R}$. Then, given a function $g \in \mathcal{C}(C)$ and an accuracy $\varepsilon > 0$, there exists a hidden layer size $m \in \mathbb{N}$ and a set of coefficients $W_j \in \mathbb{R}^p$, $u_j, b_j \in \mathbb{R}$ (for $j \in \{1, \ldots, m\}$), such that

$$f : C \to \mathbb{R}\,; \quad f(\mathbf{x}) = \sum_{j=1}^{m} u_j \cdot \sigma\left(W_j^T \mathbf{x} + b_j\right)$$

is an ε-approximation of g, that is,

$$\|f - g\|_\infty := \max_{x \in C} |f(\mathbf{x}) - g(\mathbf{x})| < \varepsilon \ .$$

The theorem extends trivially to multiple outputs.

UNIVERSAL APPROXIMATION PROPERTY / 2

Corollary. Neural networks with a single sigmoidal hidden layer and linear output layer are universal approximators.

- This means that for a given target function g there exists a sequence of networks $\left(f_k\right)_{k\in\mathbb{N}}$ that converges (pointwise) to g.
- Usually, as the networks come closer and closer to g, they will need more and more hidden neurons.
- A network with fixed layer sizes can only model a subspace of all continuous functions.
- The continuous functions form an infinite dimensional vector space. Therefore arbitrarily large hidden layer sizes are needed.

UNIVERSAL APPROXIMATION PROPERTY / 3

- Why is universal approximation a desirable property?
- Recall the definition of a Bayes optimal hypothesis $f^* : X \to Y$. It is the best possible hypothesis (model) for the given problem: it has minimal loss averaged over the data generating distribution.
- So ideally we would like the neural network (or any other learner) to approximate the Bayes optimal hypothesis.
- Usually we do not manage to learn f^*.
- This is because we do not have enough (infinite) data. We have no control over this, so we have to live with this limitation.
- But we do have control over which model class we use.

UNIVERSAL APPROXIMATION PROPERTY / 4

- Universal approximation $\Rightarrow$ approximation error tends to zero as hidden layer size tends to infinity.

- Positive approximation error implies that no matter how big the data set, we cannot find the optimal model.

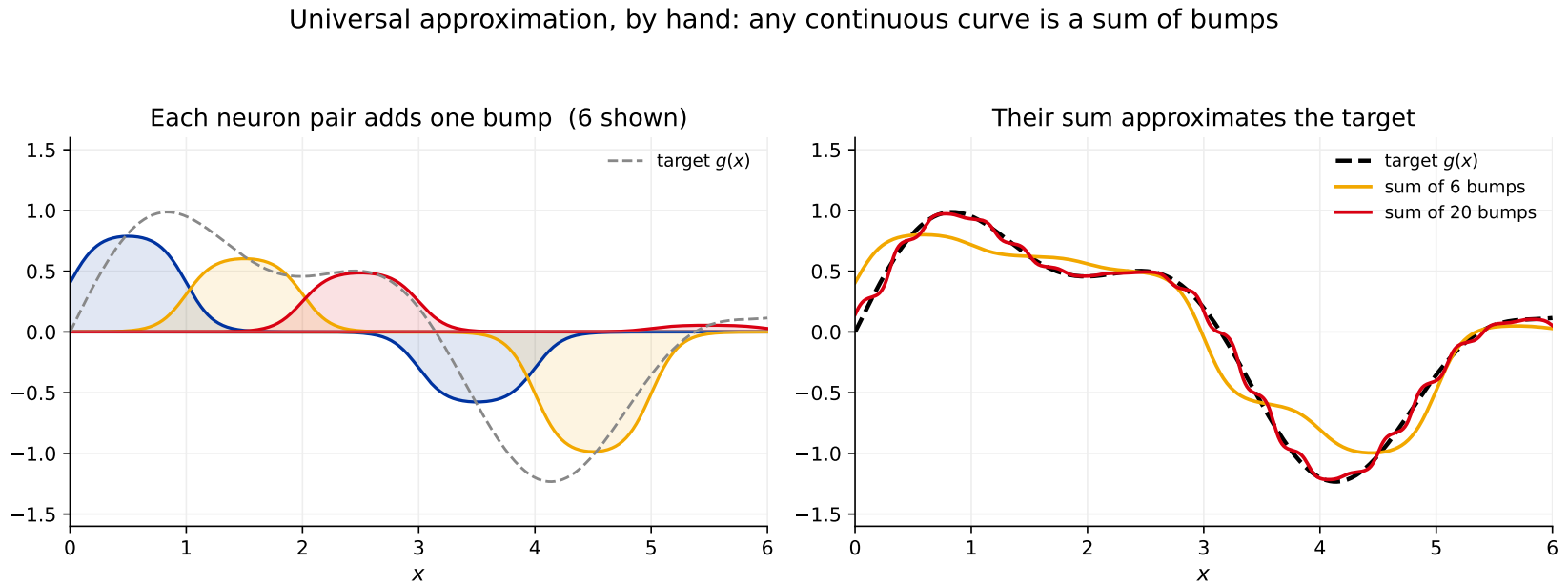
- This bears the risk of systematic under-fitting, which can be avoided with a universal model class.

UNIVERSAL APPROXIMATION PROPERTY / 5

- As we know, there are also good reasons for restricting the model class.
- This is because a flexible model class with universal approximation ability often results in over-fitting, which is no better than under-fitting.
- Thus, “universal approximation $\Rightarrow$ low approximation error”, but at the risk of a substantial learning error.
- In general, models of intermediate flexibility give the best predictions. For neural networks this amounts to a reasonably sized hidden layer.

Why it works: any curve is a sum of bumps

The theorem promises an approximation *exists* – but not *why*. The intuition: each hidden neuron contributes one localized **bump**, and their sum can trace any continuous curve. More neurons $\Rightarrow$ closer fit.

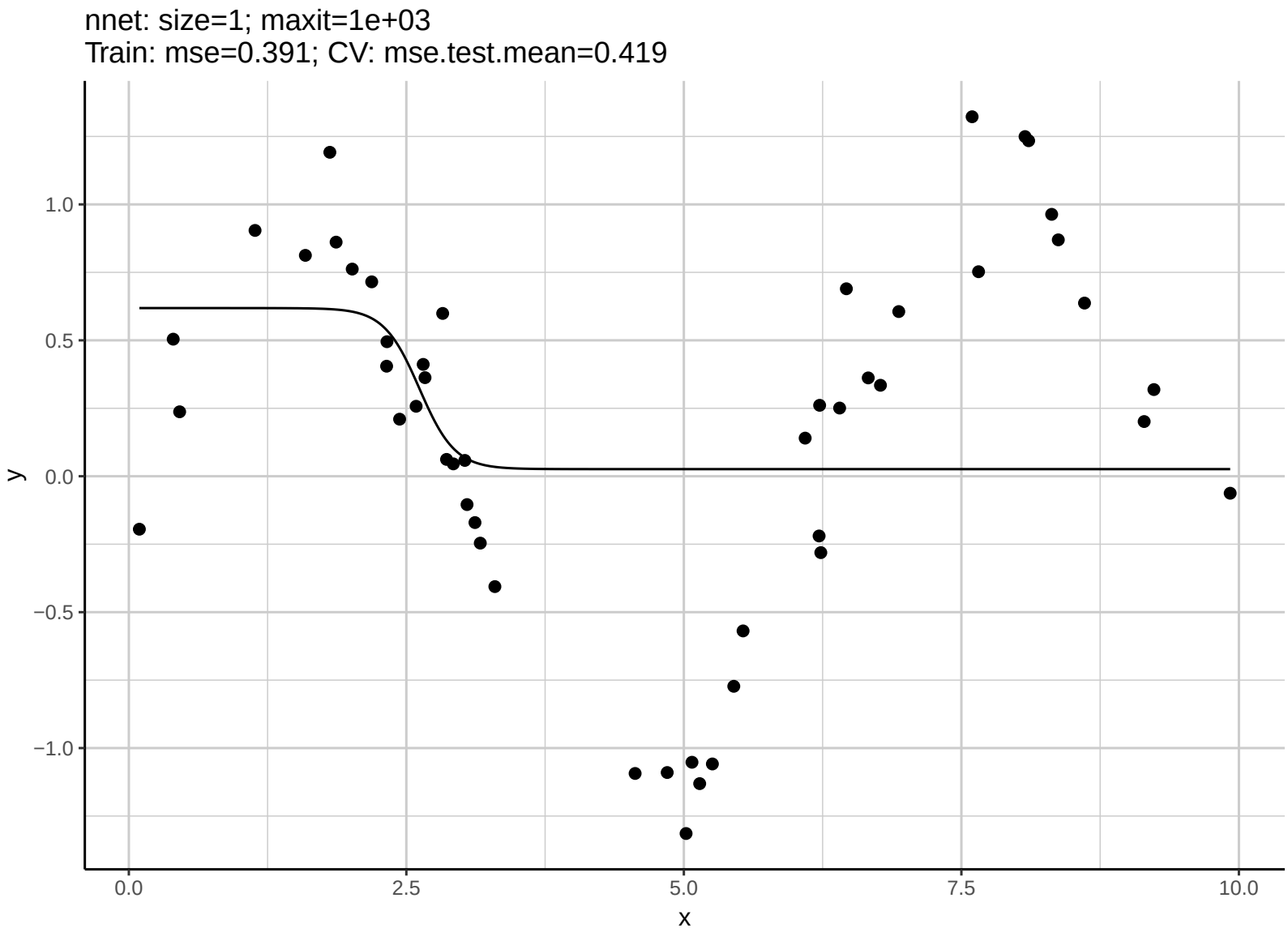


Each sigmoid pair makes one bump (left). Summing 6 already follows the target; 20 nearly nails it (right). "$m \to \infty$" in the theorem is just **"add more bumps"** – and the LMU examples on the next slides show exactly this, as the hidden-layer size grows.

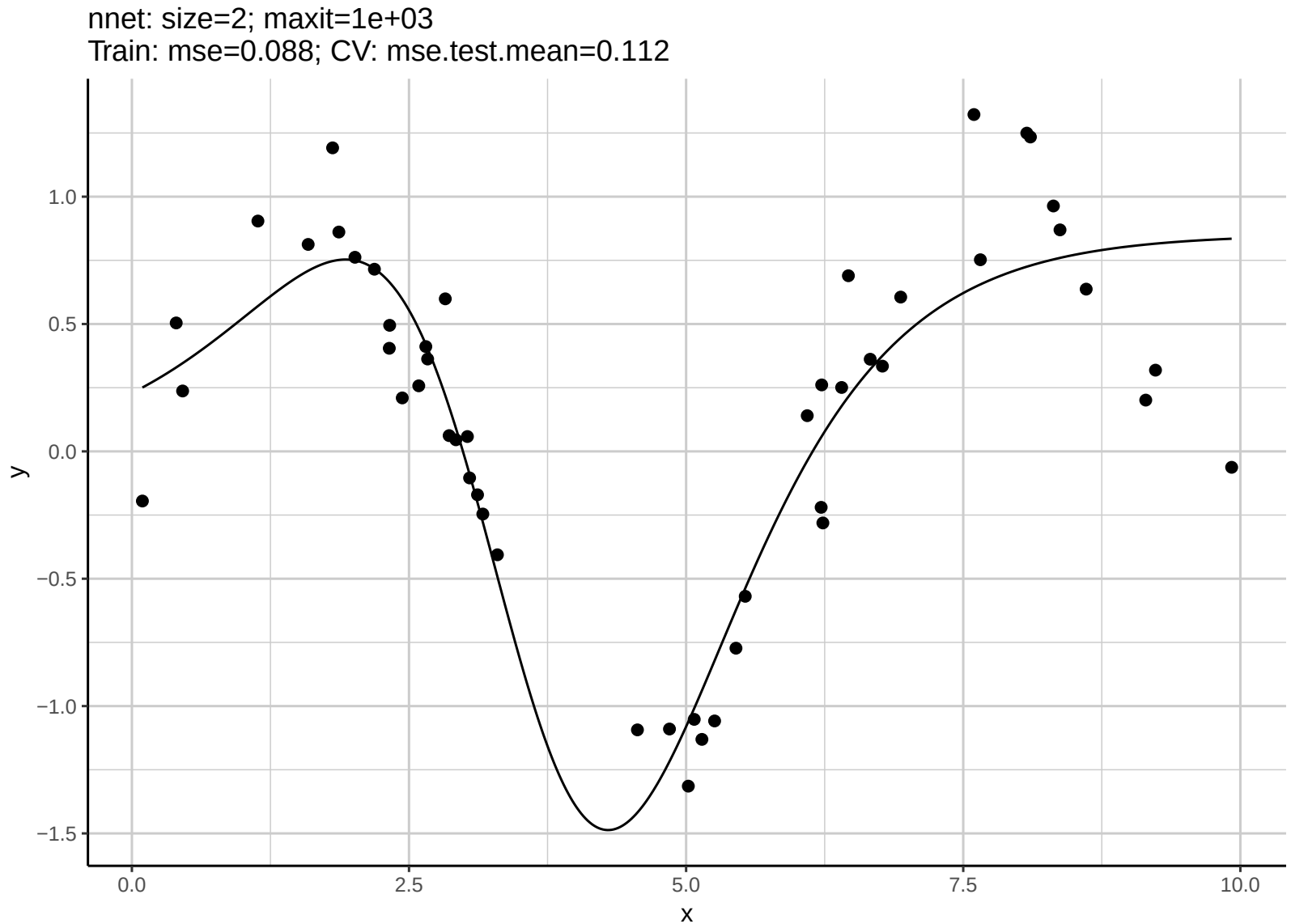
EXAMPLE : REGRESSION/CLASSIFICATION

- Let's look at a few examples of the types of functions and decisions boundaries learnt by neural networks (with a **single** hidden layer) of various sizes.
- "size" here refers to the number of neurons in the hidden layer.
- The number of "iterations" in the following slides corresponds to the number of steps of the applied iterative optimization algorithm (stochastic gradient descent).

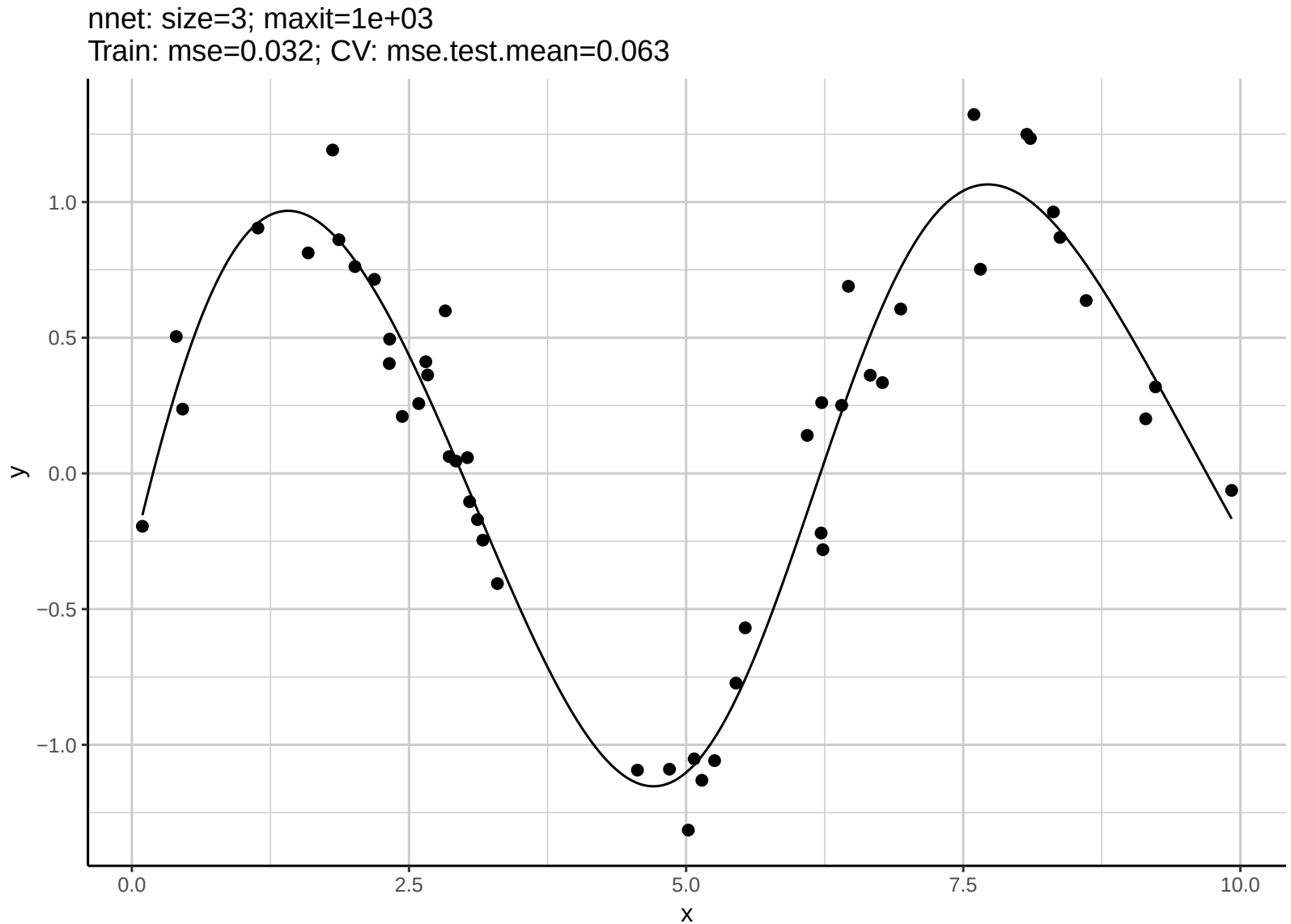
REGRESSION EX.: 1000 TRAINING ITERATIONS



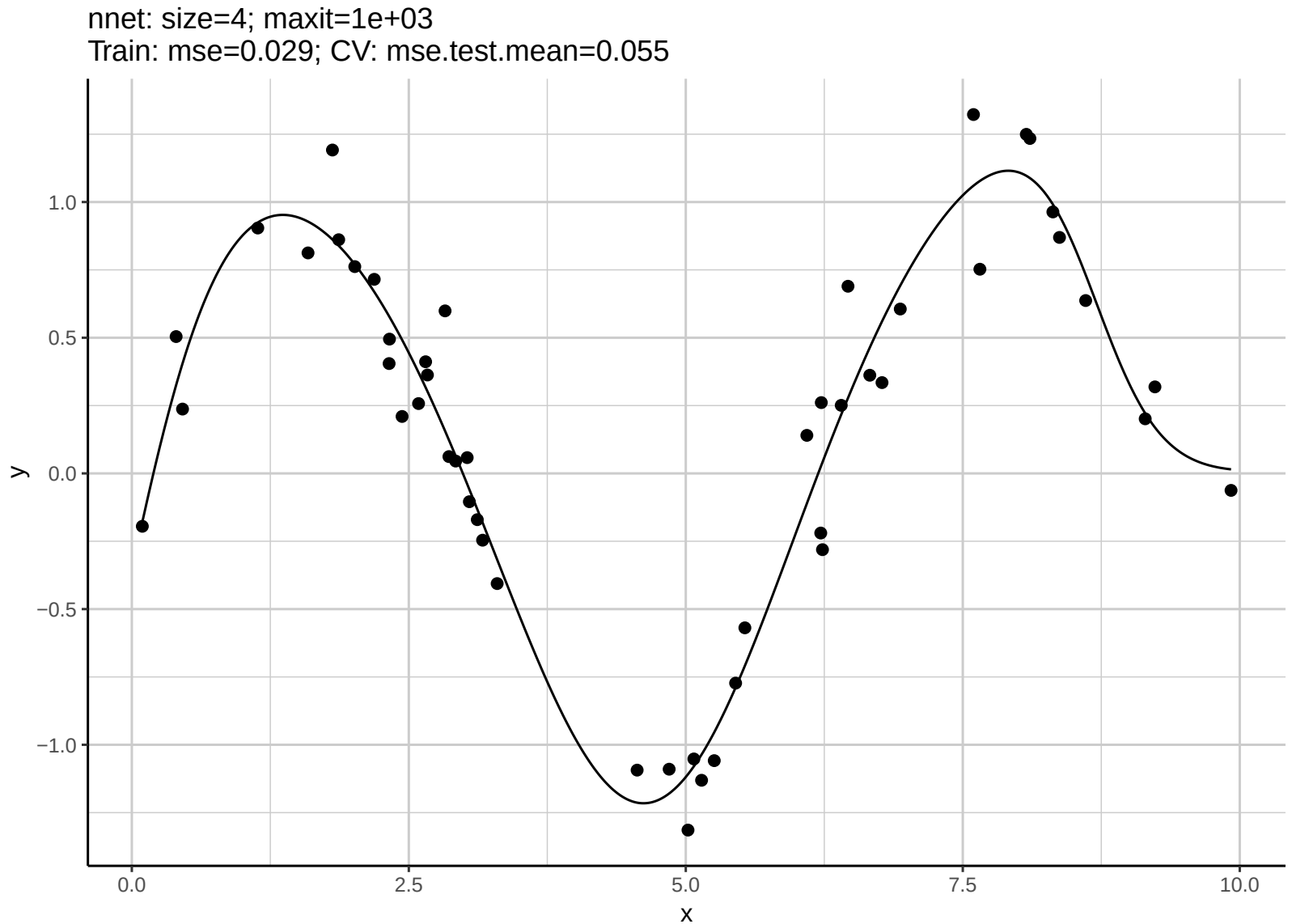
REGRESSION EX.: 1000 TRAINING ITERATIONS / 2



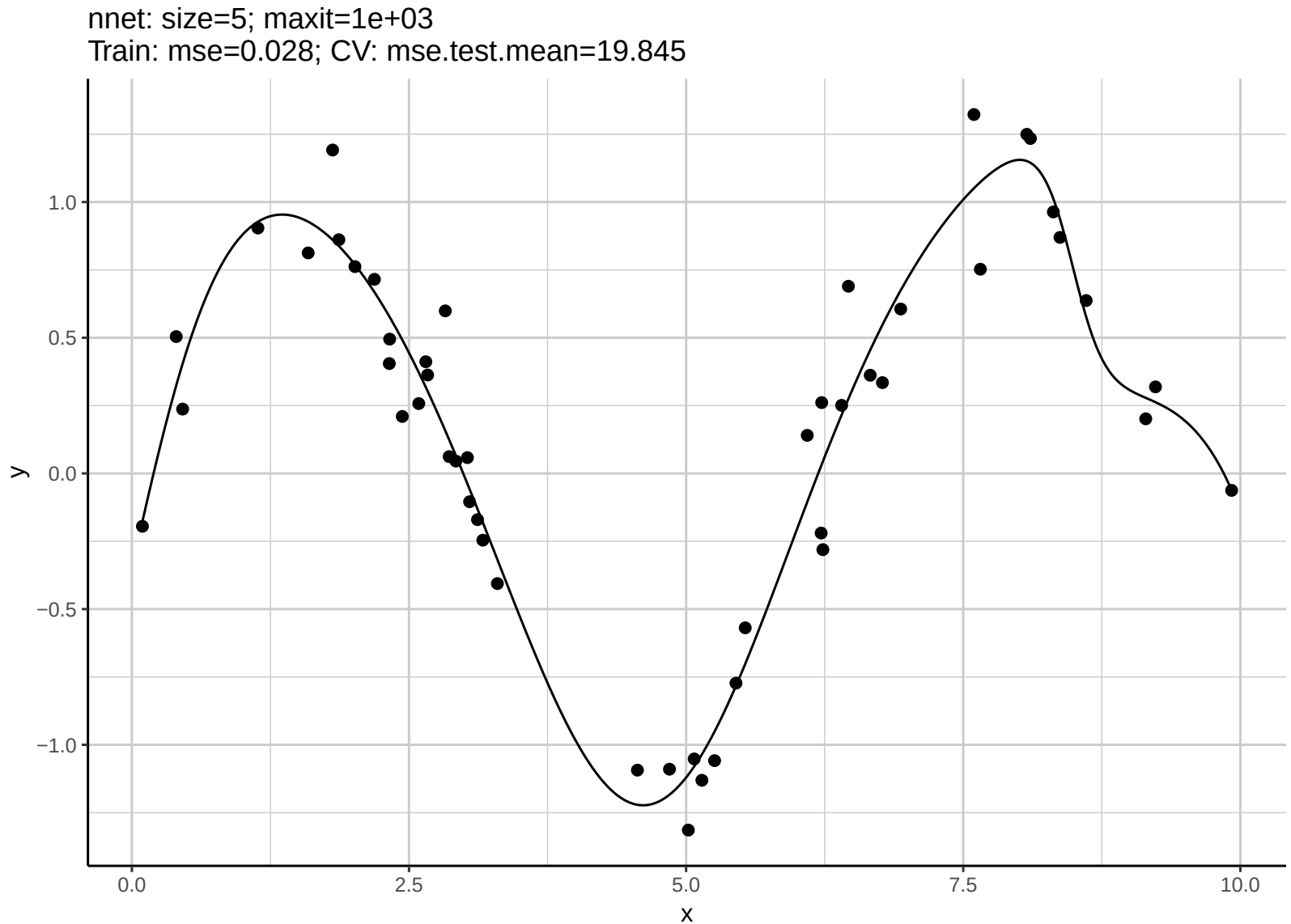
REGRESSION EX.: 1000 TRAINING ITERATIONS / 3



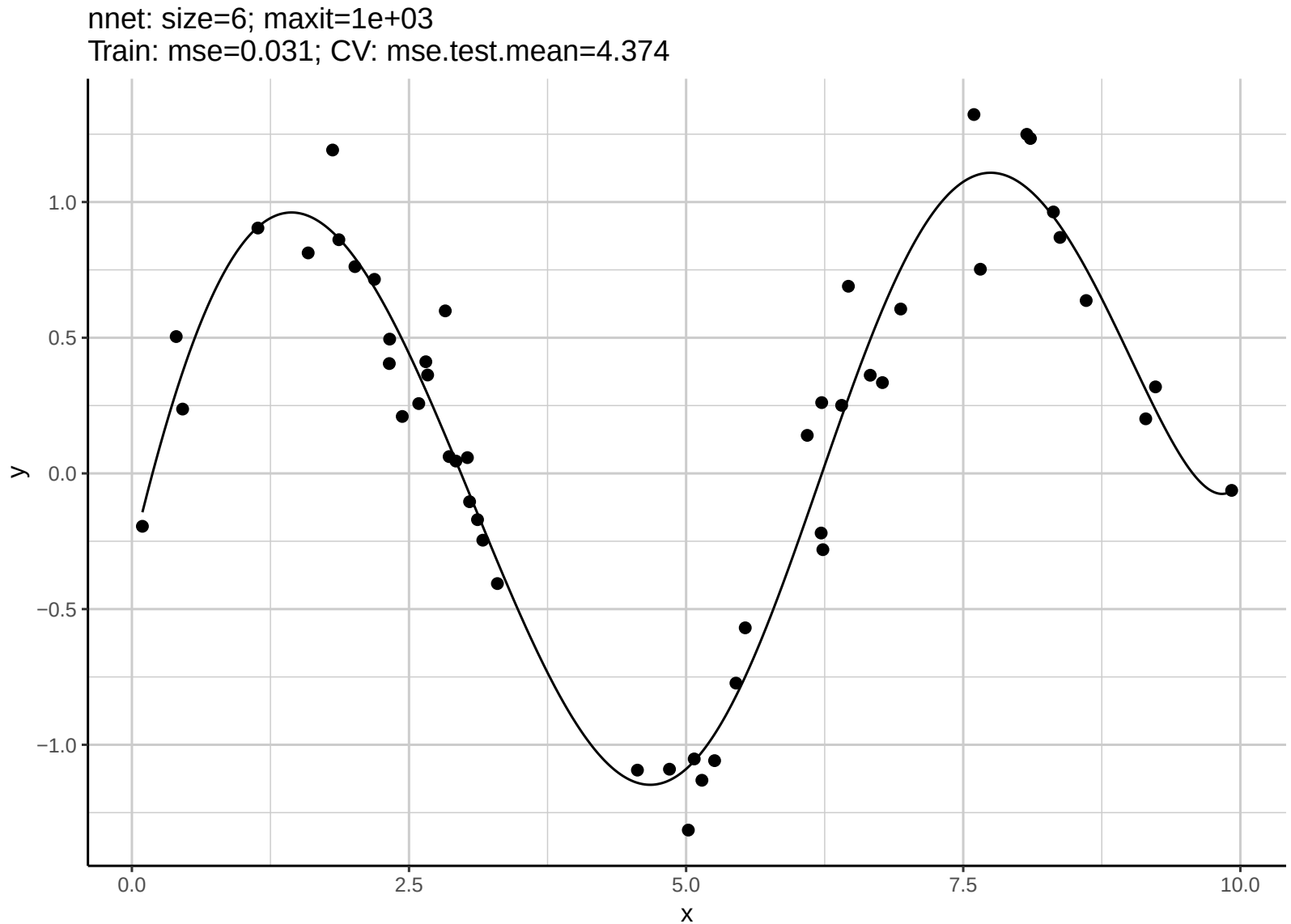
REGRESSION EX.: 1000 TRAINING ITERATIONS / 4



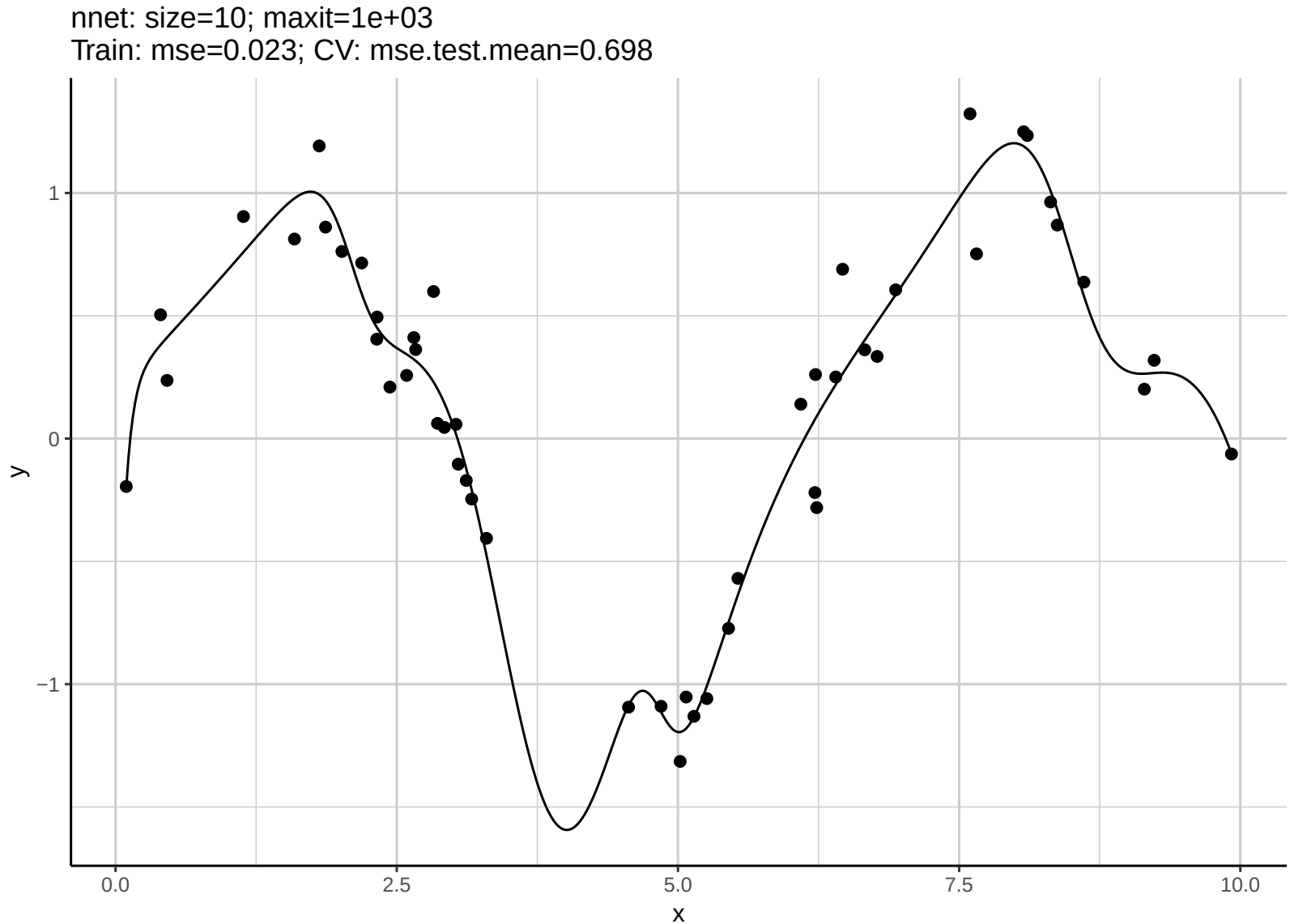
REGRESSION EX.: 1000 TRAINING ITERATIONS / 5



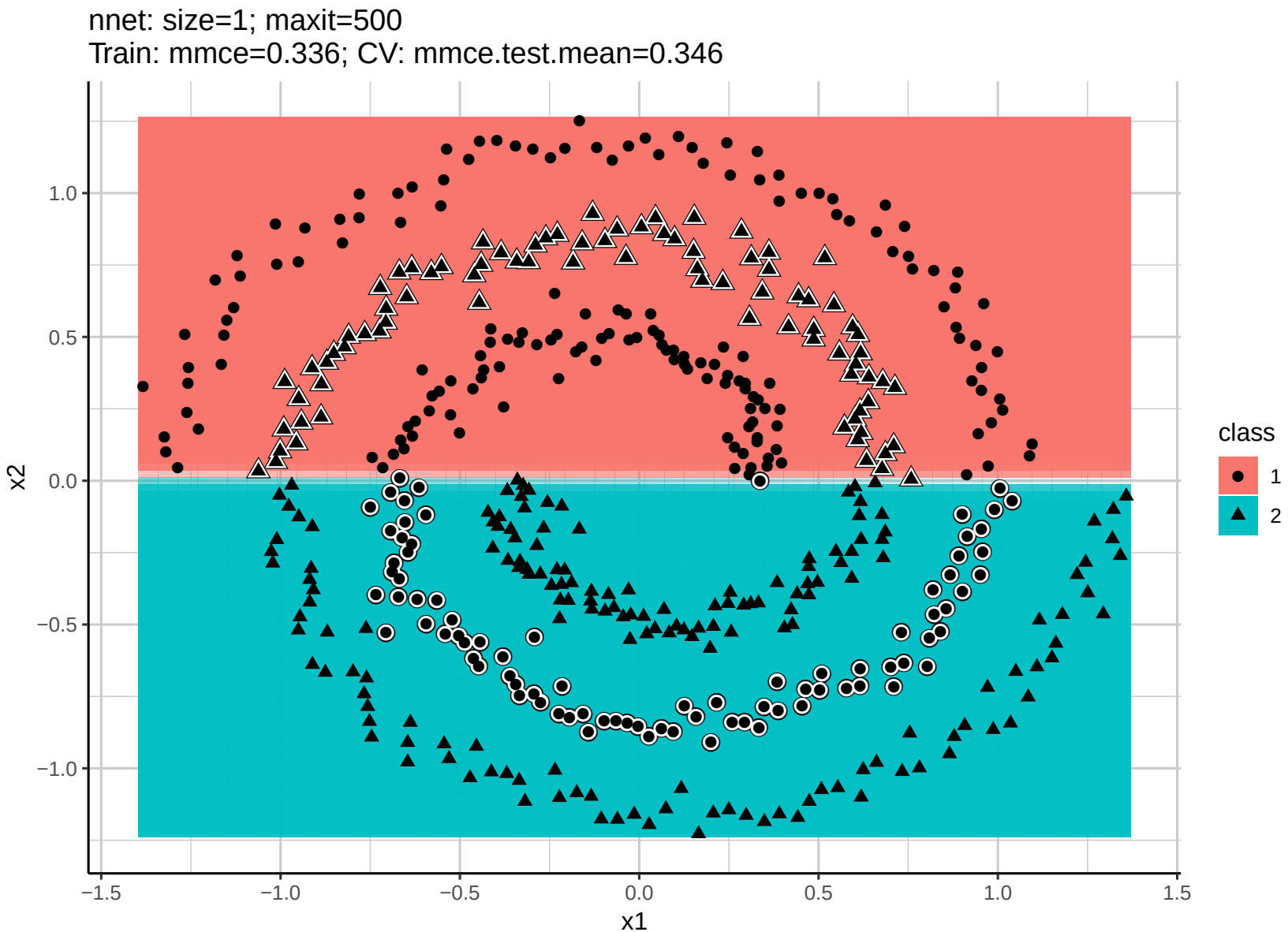
REGRESSION EX.: 1000 TRAINING ITERATIONS / 6



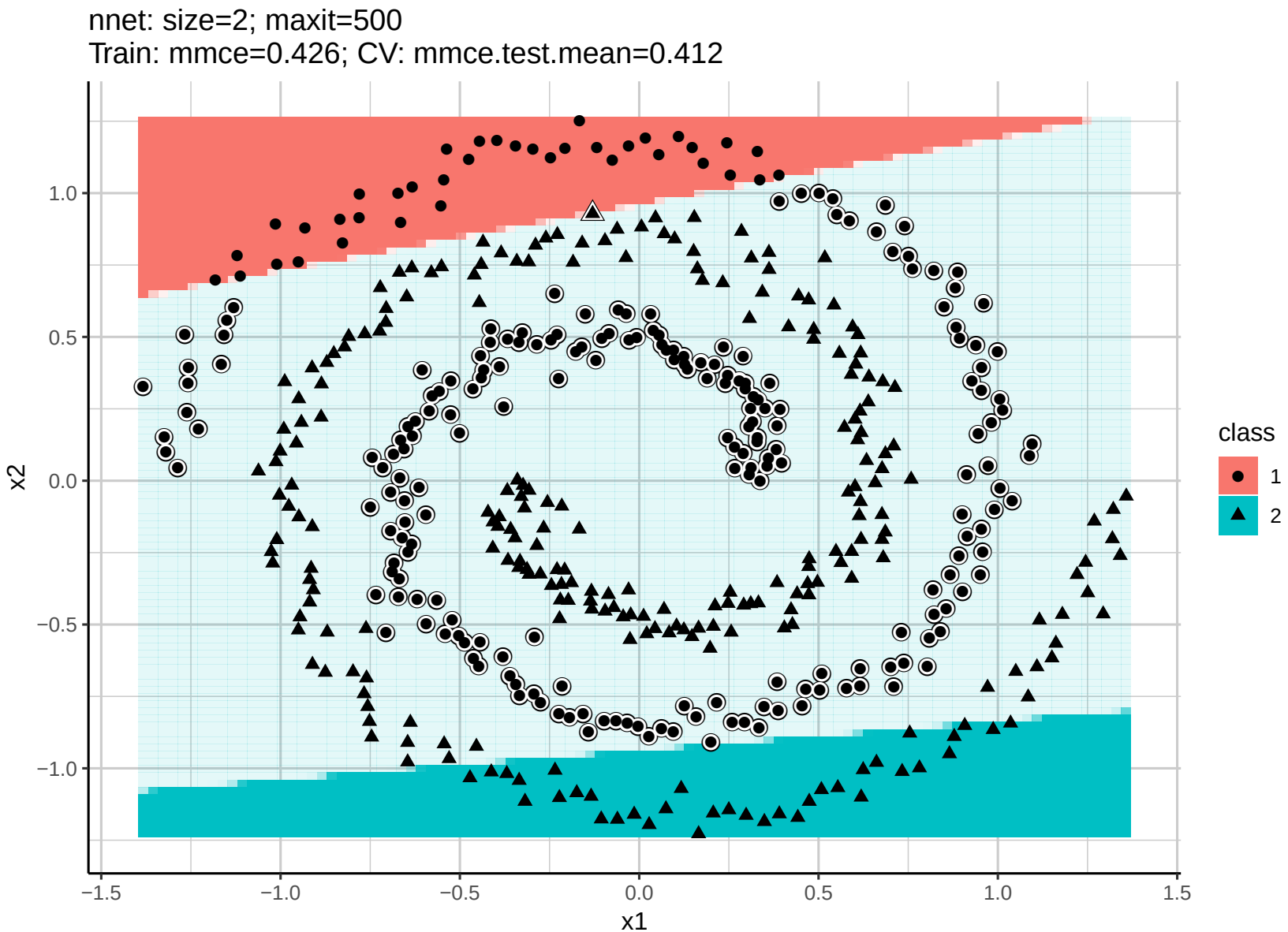
REGRESSION EX.: 1000 TRAINING ITERATIONS / 7



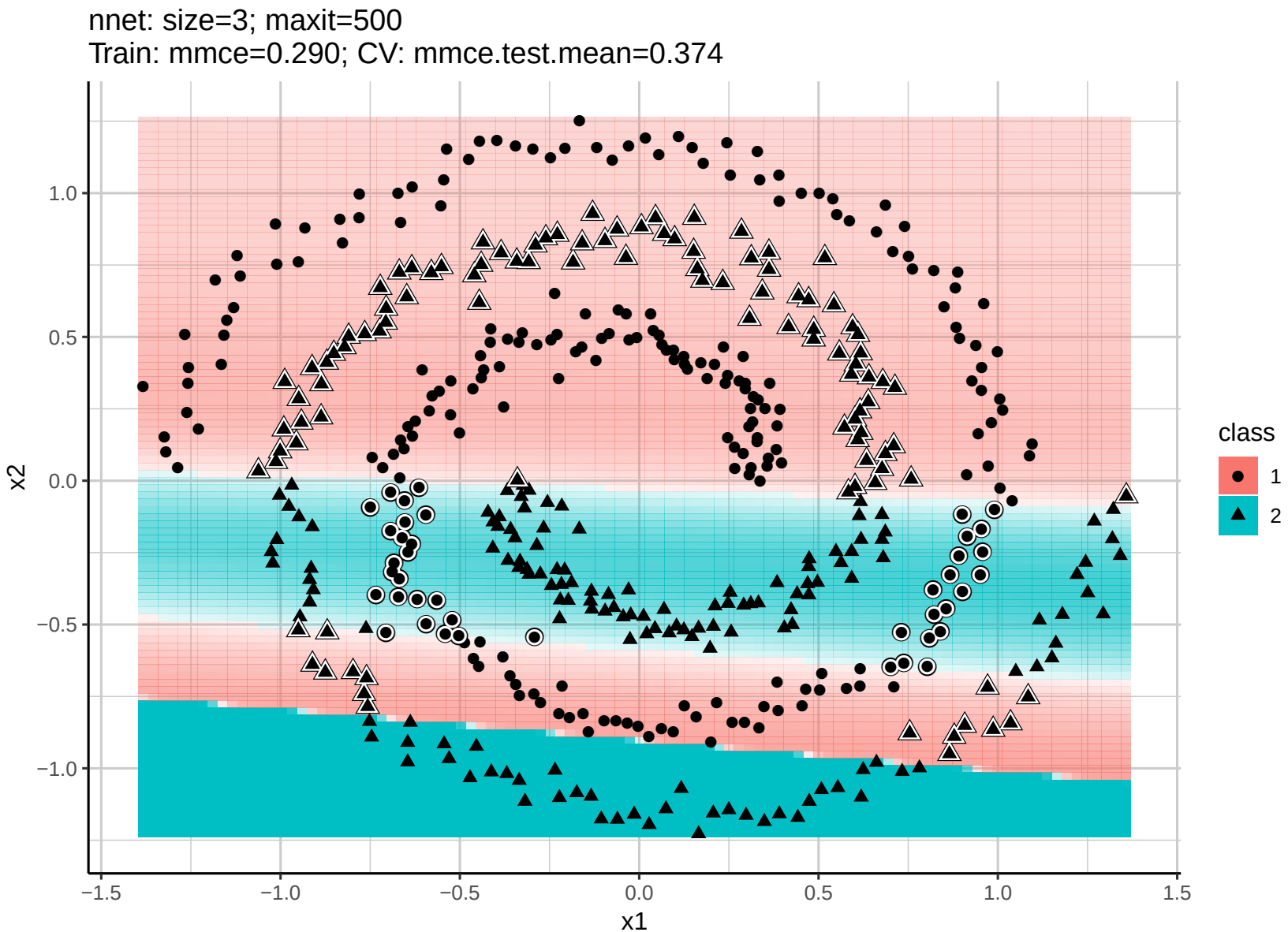
CLASSIFICATION: 500 TRAINING ITERATIONS



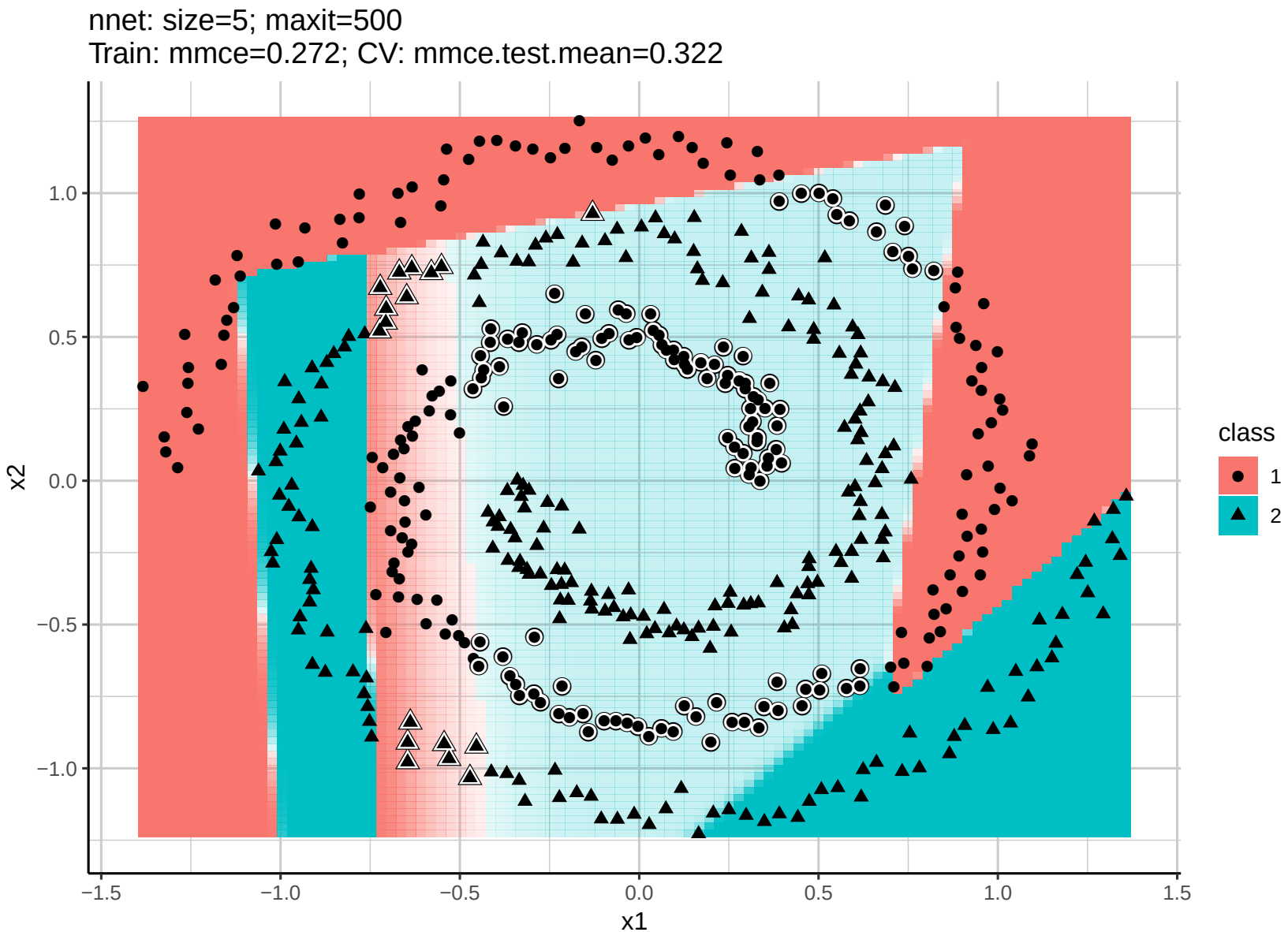
CLASSIFICATION: 500 TRAINING ITERATIONS



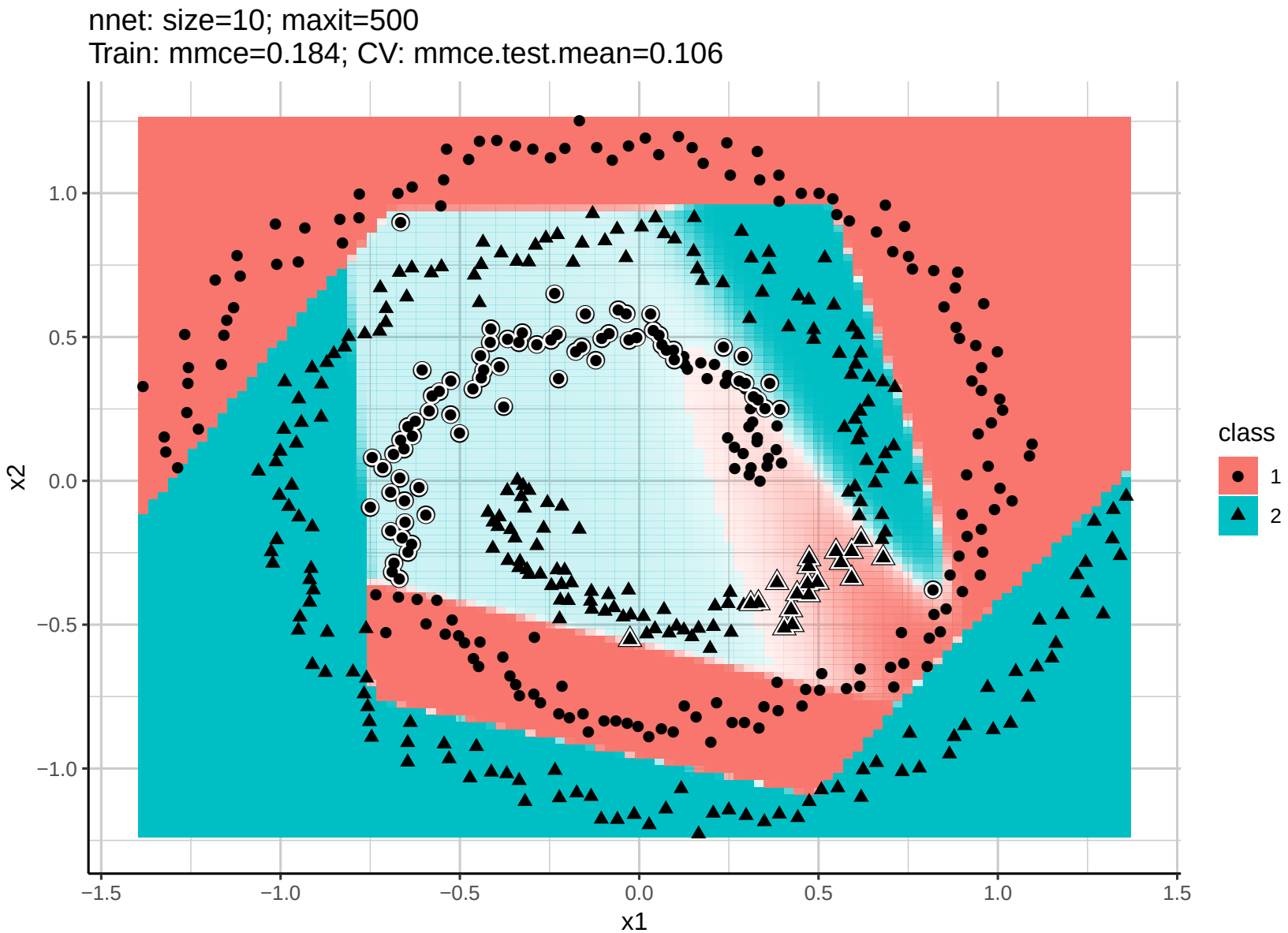
CLASSIFICATION: 500 TRAINING ITERATIONS



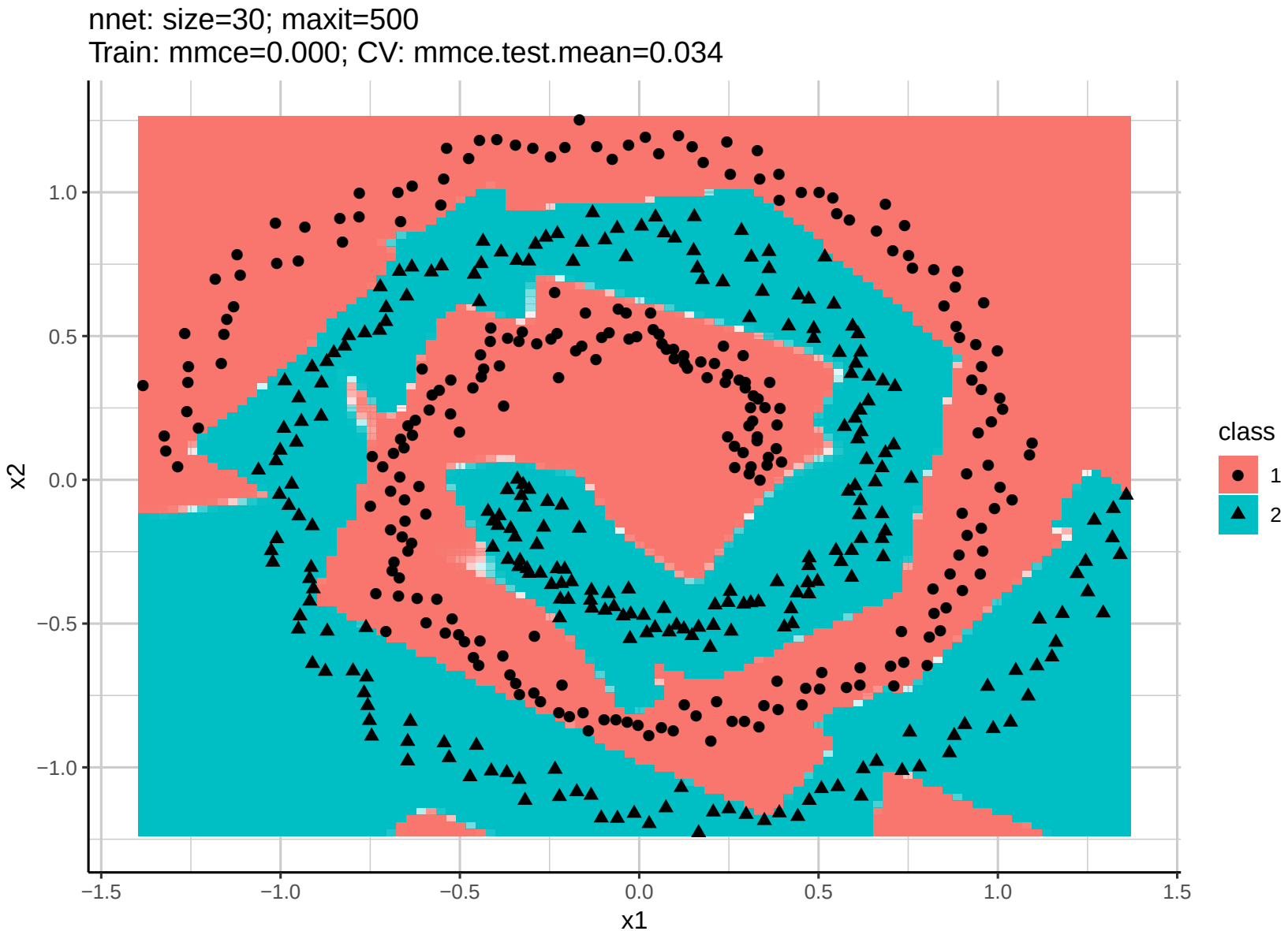
CLASSIFICATION: 500 TRAINING ITERATIONS



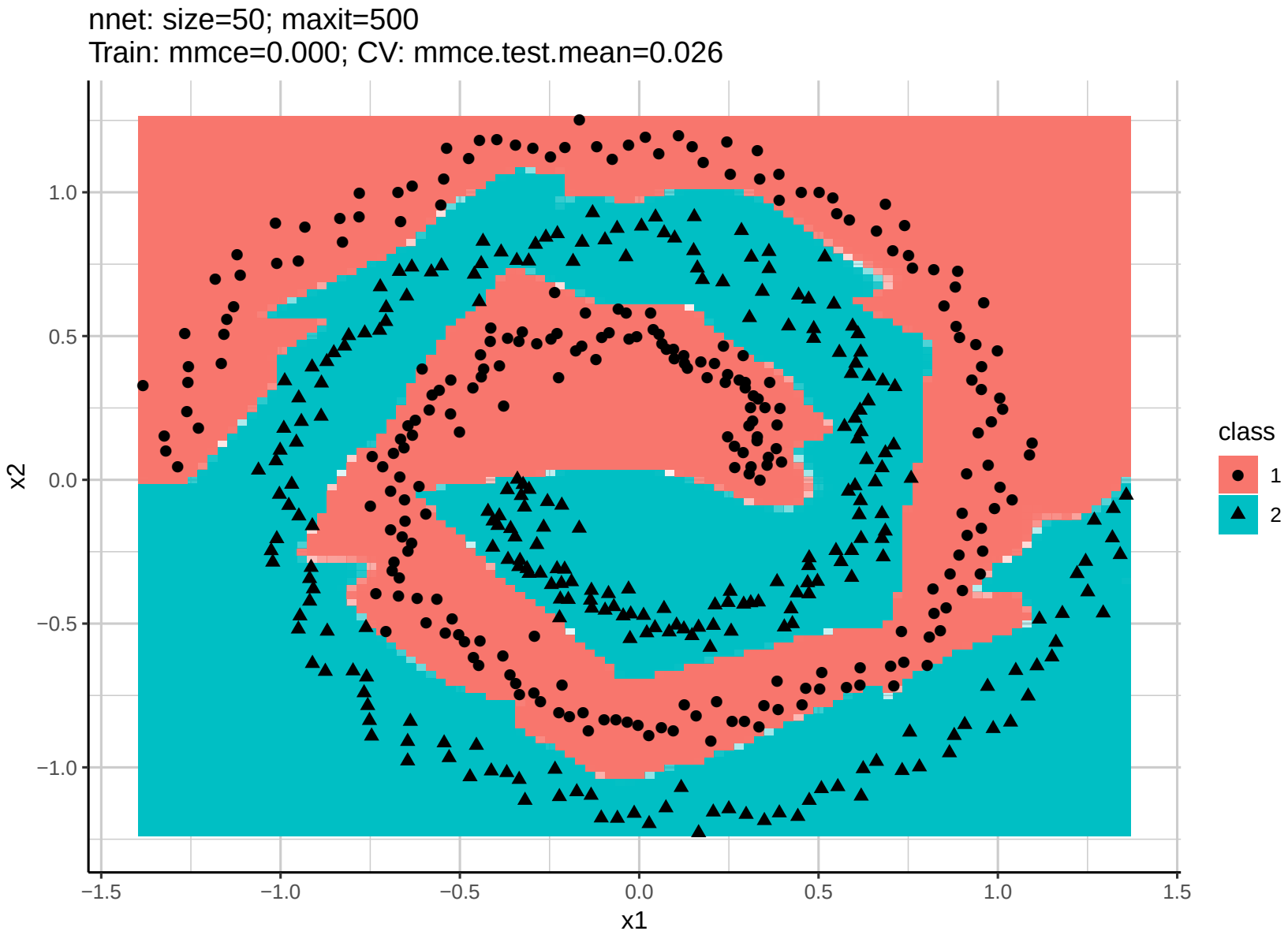
CLASSIFICATION: 500 TRAINING ITERATIONS



CLASSIFICATION: 500 TRAINING ITERATIONS



CLASSIFICATION: 500 TRAINING ITERATIONS



Universal approximation: read the small print

The theorem is reassuring but easy to over-read. Three cautions:

- **Existence, not construction.** It says good weights *exist* – not that gradient descent will *find* them. Training can still get stuck.
- **One layer can be absurdly wide.** "Some m" may mean an astronomically large hidden layer; **depth** reaches the same functions far more cheaply (previous sprinkle).
- **Approximating $\neq$ generalizing.** The LMU example showed it: a size-5 net nailed the training data but its *test* error exploded (CV mse ≈ 20). Capacity you cannot control overfits.

Takeaway. Universal approximation is a *floor* on what a big-enough net *could* represent – not a promise about training or test performance. In practice: a **reasonably sized** hidden layer, plus everything in the training chunk.

Putting it together

We can now write down a network of any depth, width, and number of classes – and know it is expressive enough.

Recap: the model is now fully specified

- **Matrix notation:** a layer is $\mathbf{z} = \sigma(\mathbf{W}^\top \mathbf{x} + \mathbf{b})$; a whole batch is $Z = \sigma(XW + B)$ – one matrix multiply per layer.
- **Depth:** stack l layers into a chain $f = \tau \circ \ldots \circ \sigma^{(1)}$. Each layer builds more abstract features; depth is **exponentially** more expressive than width (folding).
- **Many classes:** a **softmax** output layer turns scores into a probability distribution – the same softmax as multinomial logistic regression.
- **Universal approximation:** one hidden layer can approximate any continuous function in principle – but trainability and generalization are separate questions.

Everything is still "a stack of affine maps and nonlinearities". **One thing is still missing: how to learn the weights.**

Next: learning the weights

Every network so far had its weights **given** to us. With the model fully specified, the last piece is **training** – finding good weights automatically from data.

- **Gradient descent:** nudge every weight downhill on the loss.
- **Backpropagation:** get the gradient for *all* weights in one backward pass – what makes training a million-weight net feasible.

That is the next chunk (LMU's *backpropagation* week). It turns these static architectures into models that actually learn.