Neural Networks

Langauge and Computation I, Fall 2025

Zhenghao Herbert Zhou, Oct 9th, 2025

Recap from Classification

Logistic Regression (from lecture on Sep 18th)

$$z = \sum_{i=1}^{n} w_i x_i + b \qquad \text{Sigmoid: } \sigma(z) = \frac{1}{1 + e^{-z}} \qquad \text{decision}(\mathbf{x}) = \begin{cases} 1, & \text{if } \sigma(z) > 0.5 \\ 0, & \text{otherwise} \end{cases}$$

Recap from Classification

Logistic Regression (from lecture on Sep 18th)

$$z = \sum_{i=1}^{n} w_i x_i + b \qquad \text{Sigmoid: } \sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{decision}(\mathbf{x}) = \begin{cases} 1, & \text{if } \sigma(z) > 0.5 \\ 0, & \text{otherwise} \end{cases}$$

- **Logistic Regression**: learning to make a binary decision / classification based on a set of input features.

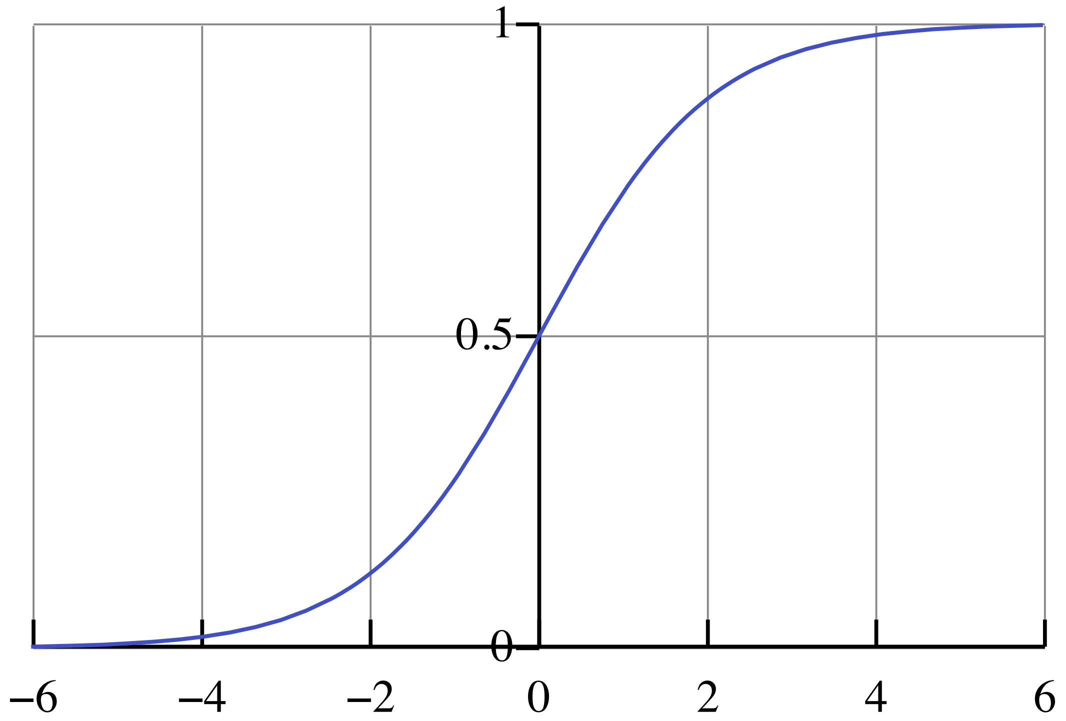
Recap from Classification

Logistic Regression (from lecture on Sep 18th)

$$z = \sum_{i=1}^{n} w_i x_i + b \qquad \text{Sigmoid: } \sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{decision}(\mathbf{x}) = \begin{cases} 1, & \text{if } \sigma(z) > 0.5 \\ 0, & \text{otherwise} \end{cases}$$

- **Logistic Regression**: learning to make a binary decision / classification based on a set of input features.
    - Given a set of input features x_i, learn a set of weight w_i and a bias term b;

Recap from Classification

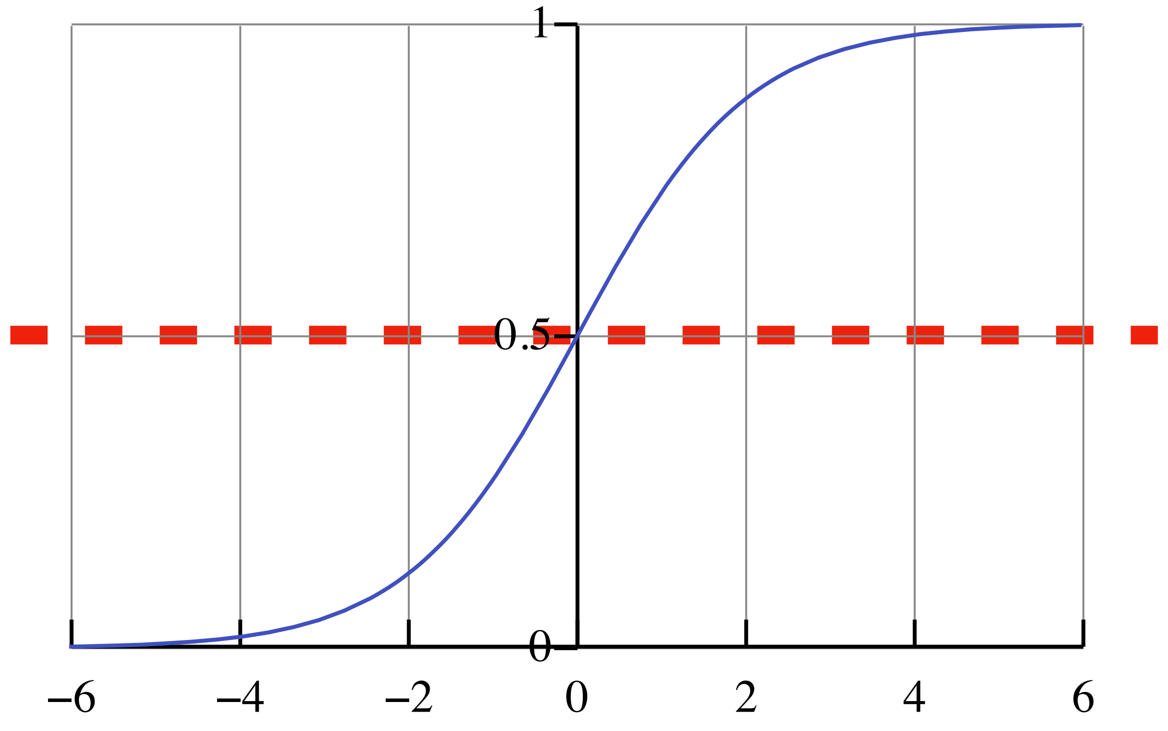


Logistic Regression (from lecture on Sep 18th)

$$z = \sum_{i=1}^{n} w_i x_i + b \qquad \text{Sigmoid: } \sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{decision}(\mathbf{x}) = \begin{cases} 1, & \text{if } \sigma(z) > 0.5 \\ 0, & \text{otherwise} \end{cases}$$

- **Logistic Regression**: learning to make a binary decision / classification based on a set of input features.
  - Given a set of input features x_i, learn a set of weight w_i and a bias term b;
  - Transform the weighted sum z into range $[0, 1]$ through Sigmoid function;

Recap from Classification



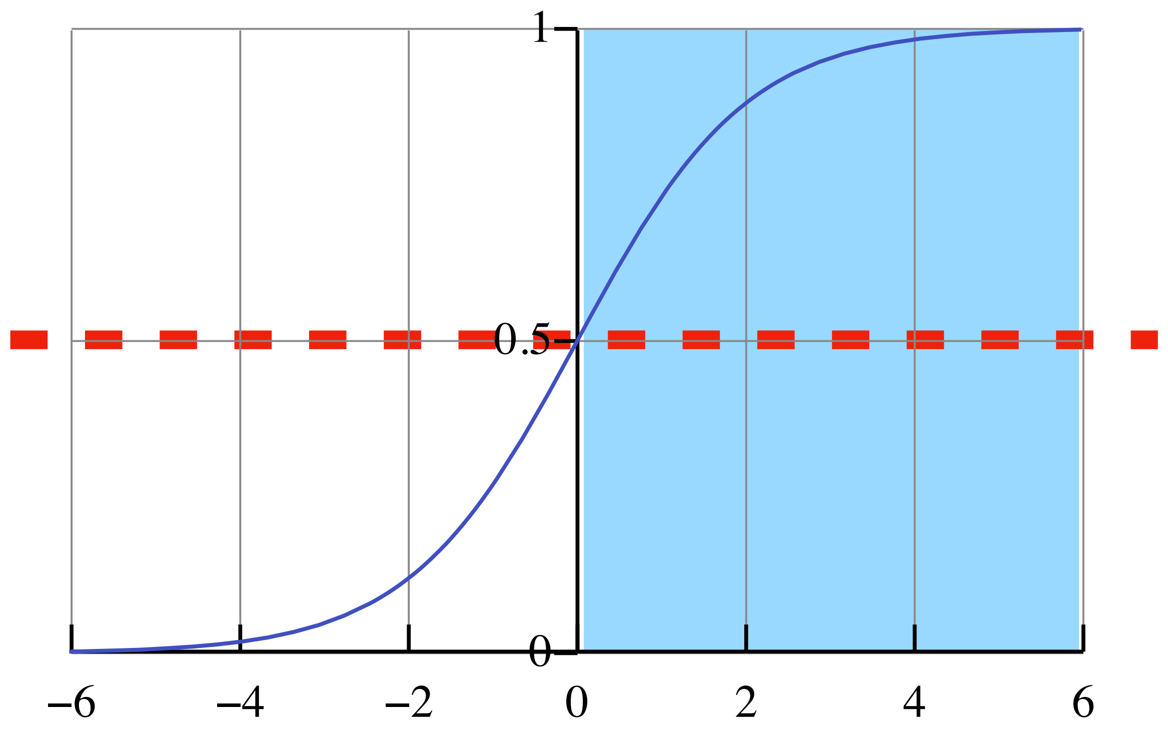
Logistic Regression (from lecture on Sep 18th)

$$z = \sum_{i=1}^{n} w_i x_i + b \qquad \text{Sigmoid: } \sigma(z) = \frac{1}{1 + e^{-z}} \qquad \text{decision}(\mathbf{x}) = \begin{cases} 1, & \text{if } \sigma(z) > 0.5 \\ 0, & \text{otherwise} \end{cases}$$

- **Logistic Regression**: learning to make a binary decision / classification based on a set of input features.
    - Given a set of input features x_i, learn a set of weight w_i and a bias term b;
    - Transform the weighted sum z into range $[0, 1]$ through Sigmoid function;
    - Classification: use $\sigma(z) = 0.5$ as the decision boundary.

Recap from Classification

Logistic Regression (from lecture on Sep 18th)

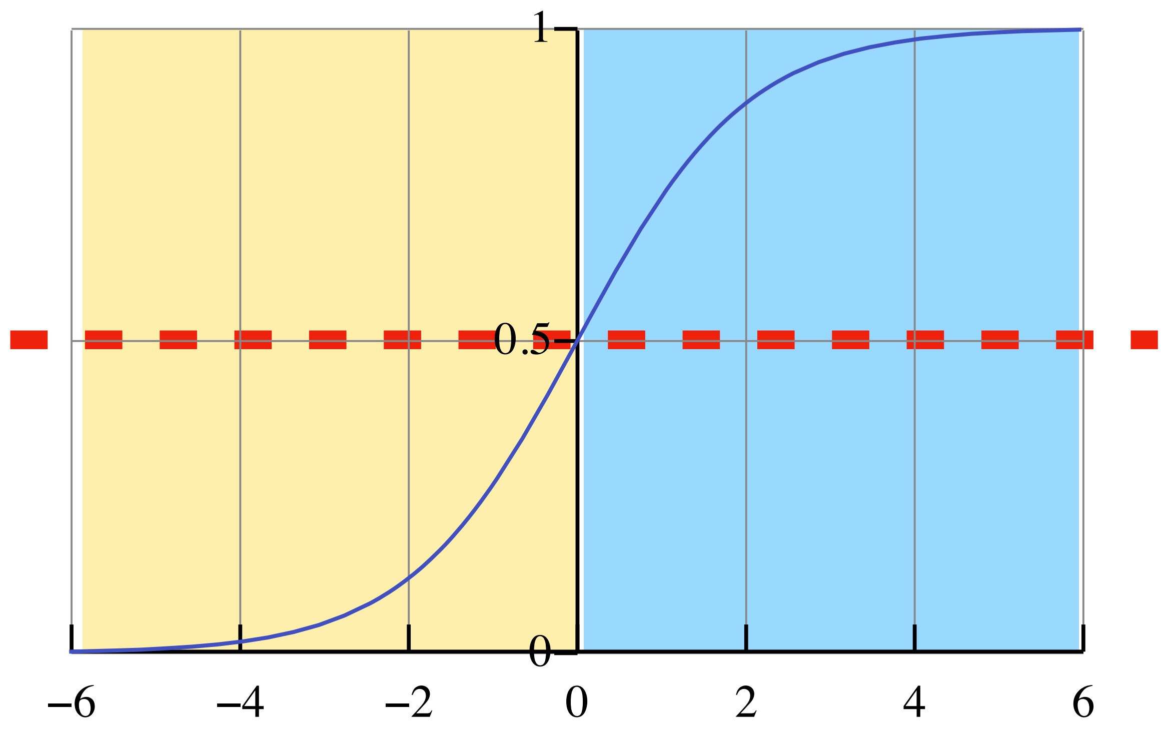


$$z = \sum_{i=1}^{n} w_i x_i + b \qquad \text{Sigmoid: } \sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{decision}(\mathbf{x}) = \begin{cases} 1, & \text{if } \sigma(z) > 0.5 \\ 0, & \text{otherwise} \end{cases}$$

- **Logistic Regression**: learning to make a binary decision / classification based on a set of input features.
  - Given a set of input features x_i, learn a set of weight w_i and a bias term b;
  - Transform the weighted sum z into range $[0, 1]$ through Sigmoid function;
  - Classification: use $\sigma(z) = 0.5$ as the decision boundary.

Recap from Classification

Logistic Regression (from lecture on Sep 18th)



$$z = \sum_{i=1}^{n} w_i x_i + b \qquad \text{Sigmoid: } \sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{decision}(\mathbf{x}) = \begin{cases} 1, & \text{if } \sigma(z) > 0.5 \\ 0, & \text{otherwise} \end{cases}$$

- **Logistic Regression**: learning to make a binary decision / classification based on a set of input features.
  - Given a set of input features x_i, learn a set of weight w_i and a bias term b;
  - Transform the weighted sum z into range $[0, 1]$ through Sigmoid function;
  - Classification: use $\sigma(z) = 0.5$ as the decision boundary.

Another Way of Representing Logistic Regression

An example on decising whether you pass a class

Another Way of Representing Logistic Regression

An example on decising whether you pass a class

- Suppose you want a logistic regression model outputting a binary decision of **{passing, failing}** a class.

Another Way of Representing Logistic Regression

An example on decising whether you pass a class

- Suppose you want a logistic regression model outputing a binary decision of **{passing, failing}** a class.

- Features available:
  - HW 1 (weight = 20%)
  - HW 2 (weight = 30%)
  - Exam (weight = 40%)
  - Attendence (weight = 10%)

Another Way of Representing Logistic Regression

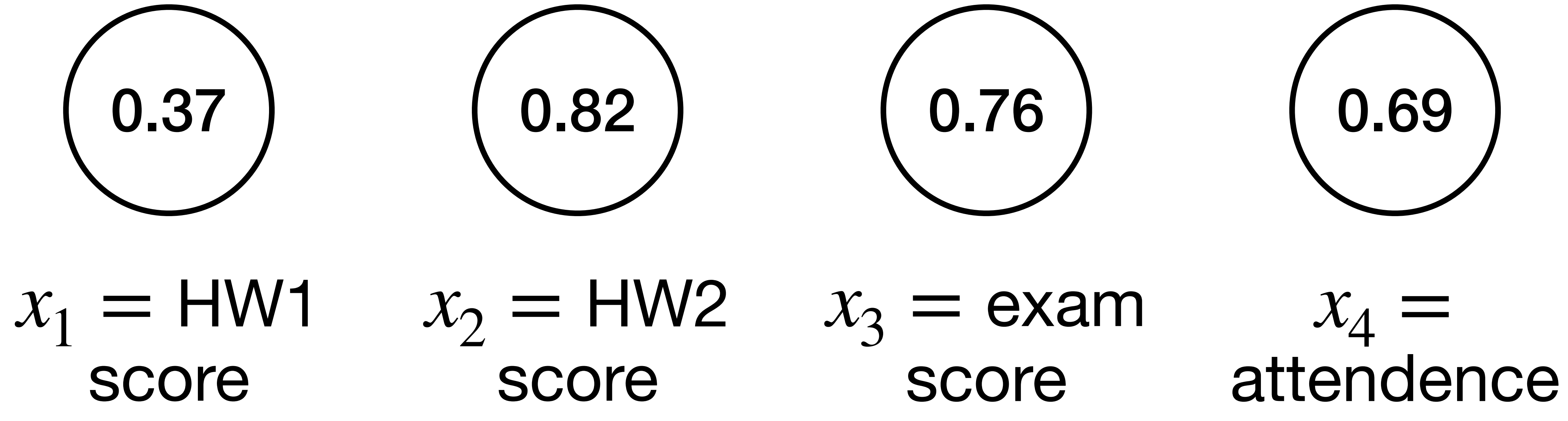
An example on decising whether you pass a class

- Suppose you want a logistic regression model outputting a binary decision of **{passing, failing}** a class.
- Features available:
  - HW 1 (weight = 20%)
  - HW 2 (weight = 30%)
  - Exam (weight = 40%)
  - Attendence (weight = 10%)
- Pass if final grade > 60%.

Another Way of Representing Logistic Regression

An example on decising whether you pass a class

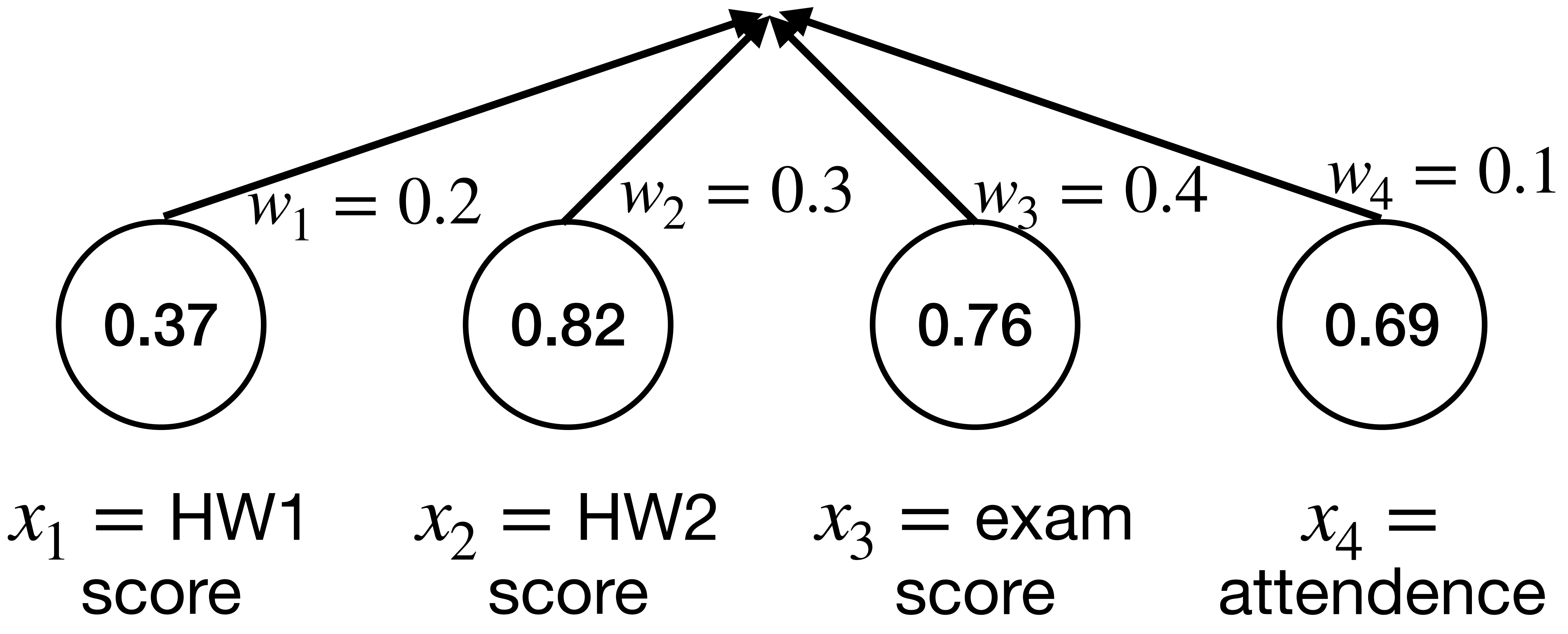
- Suppose you want a logistic regression model outputing a binary decision of **{passing, failing}** a class.
- Features available:
  - HW 1 (weight = 20%)
  - HW 2 (weight = 30%)
  - Exam (weight = 40%)
  - Attendence (weight = 10%)
- Pass if final grade > 60%.



Another Way of Representing Logistic Regression

An example on decising whether you pass a class

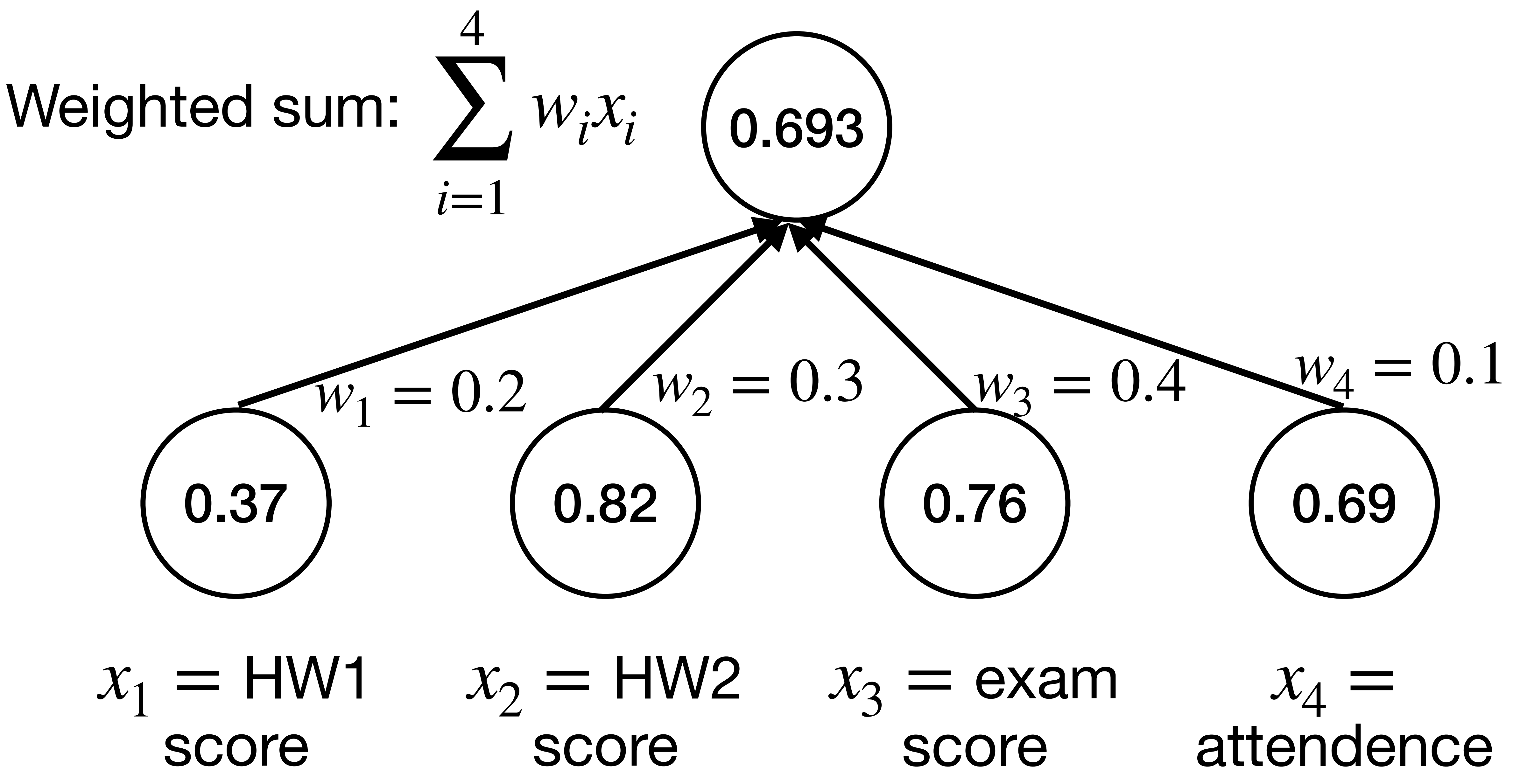
- Suppose you want a logistic regression model outputing a binary decision of **{passing, failing}** a class.
- Features available:
  - HW 1 (weight = 20%)
  - HW 2 (weight = 30%)
  - Exam (weight = 40%)
  - Attendence (weight = 10%)
- Pass if final grade > 60%.



Another Way of Representing Logistic Regression

An example on decising whether you pass a class

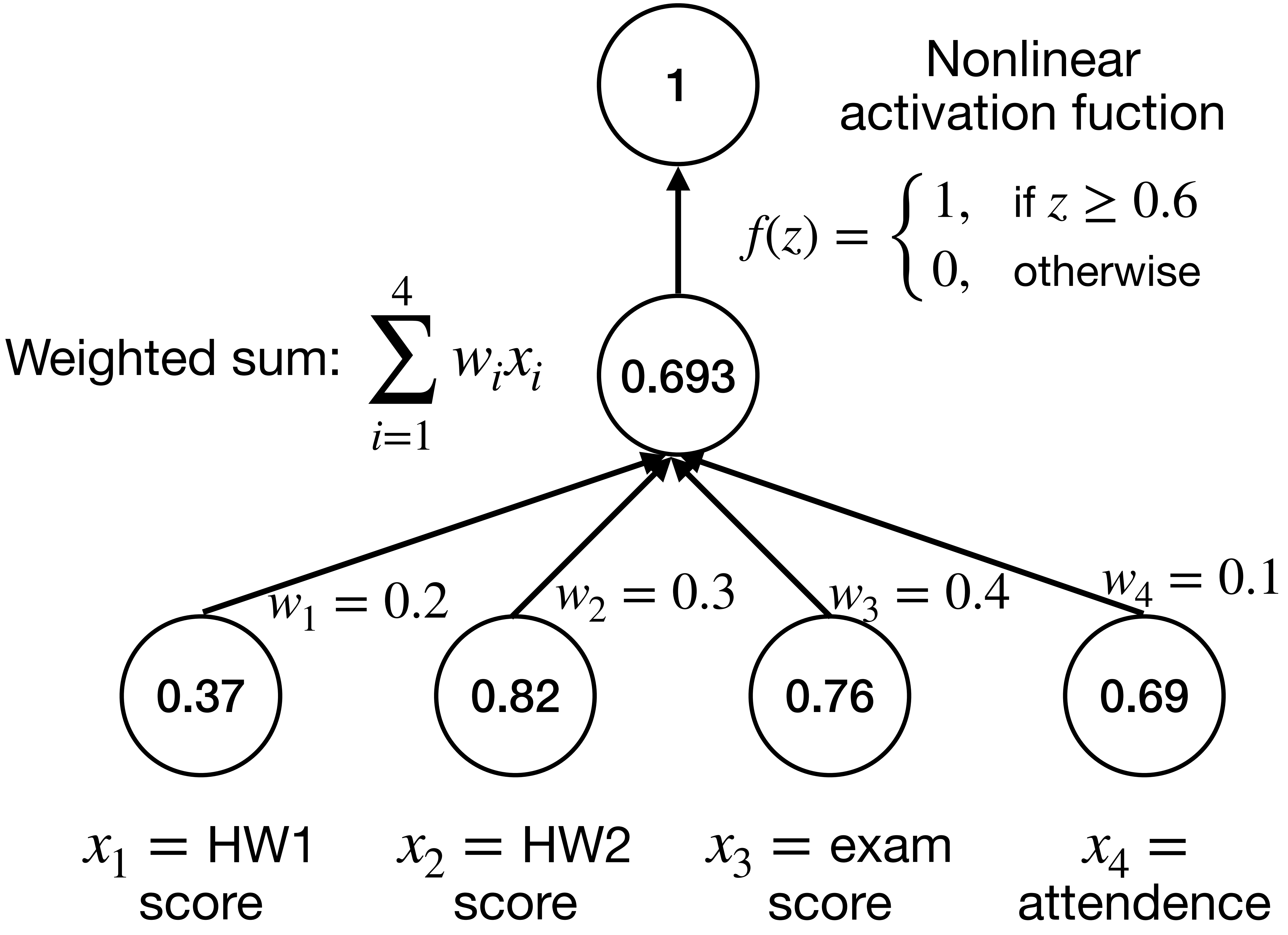
- Suppose you want a logistic regression model outputing a binary decision of **{passing, failing}** a class.
- Features available:
  - HW 1 (weight = 20%)
  - HW 2 (weight = 30%)
  - Exam (weight = 40%)
  - Attendence (weight = 10%)
- Pass if final grade > 60%.



Another Way of Representing Logistic Regression

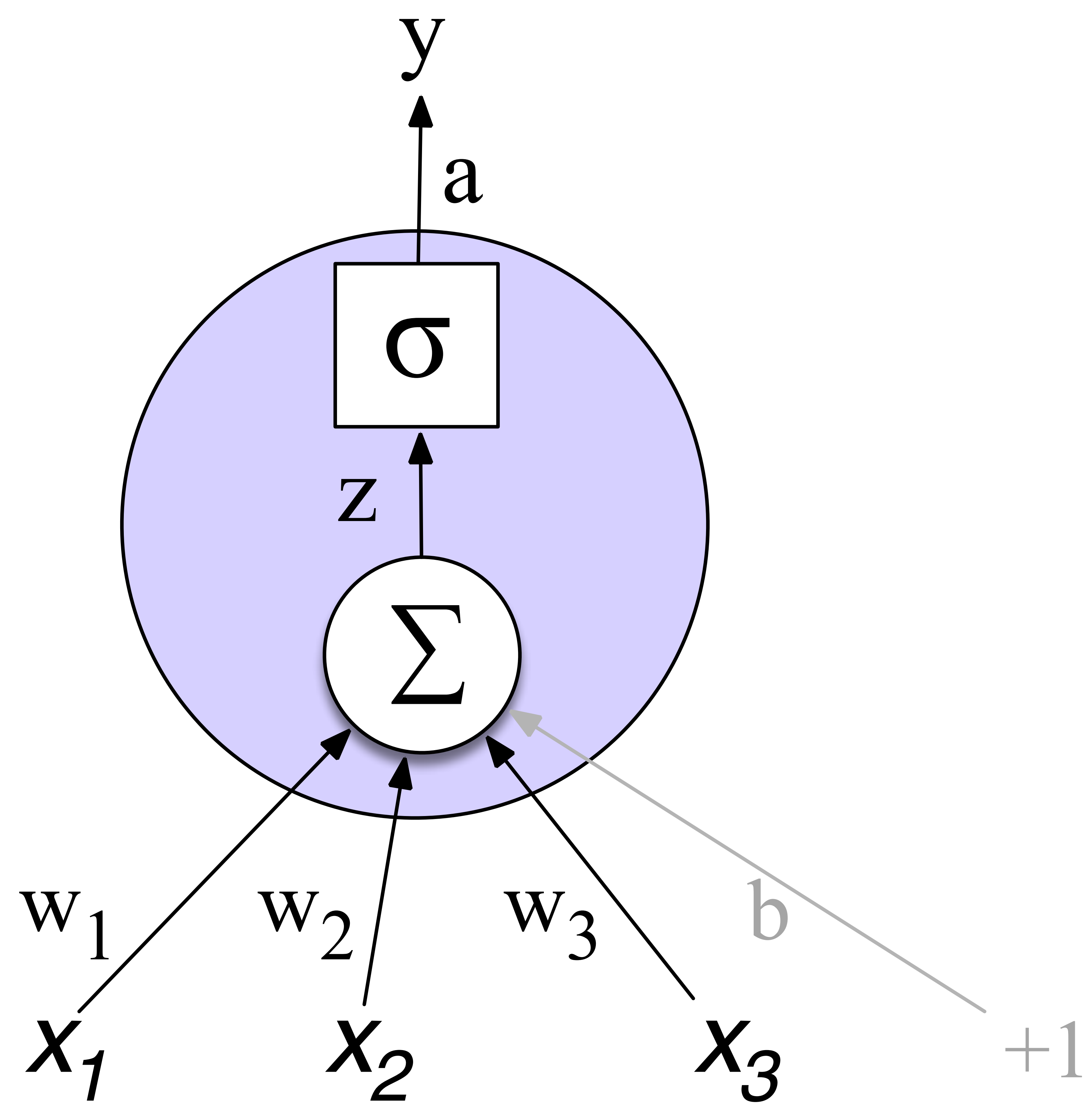
An example on decising whether you pass a class

- Suppose you want a logistic regression model outputing a binary decision of **{passing, failing}** a class.
- Features available:
    - HW 1 (weight = 20%)
    - HW 2 (weight = 30%)
    - Exam (weight = 40%)
    - Attendence (weight = 10%)
- Pass if final grade > 60%.



This becomes an Artificial Neuron!

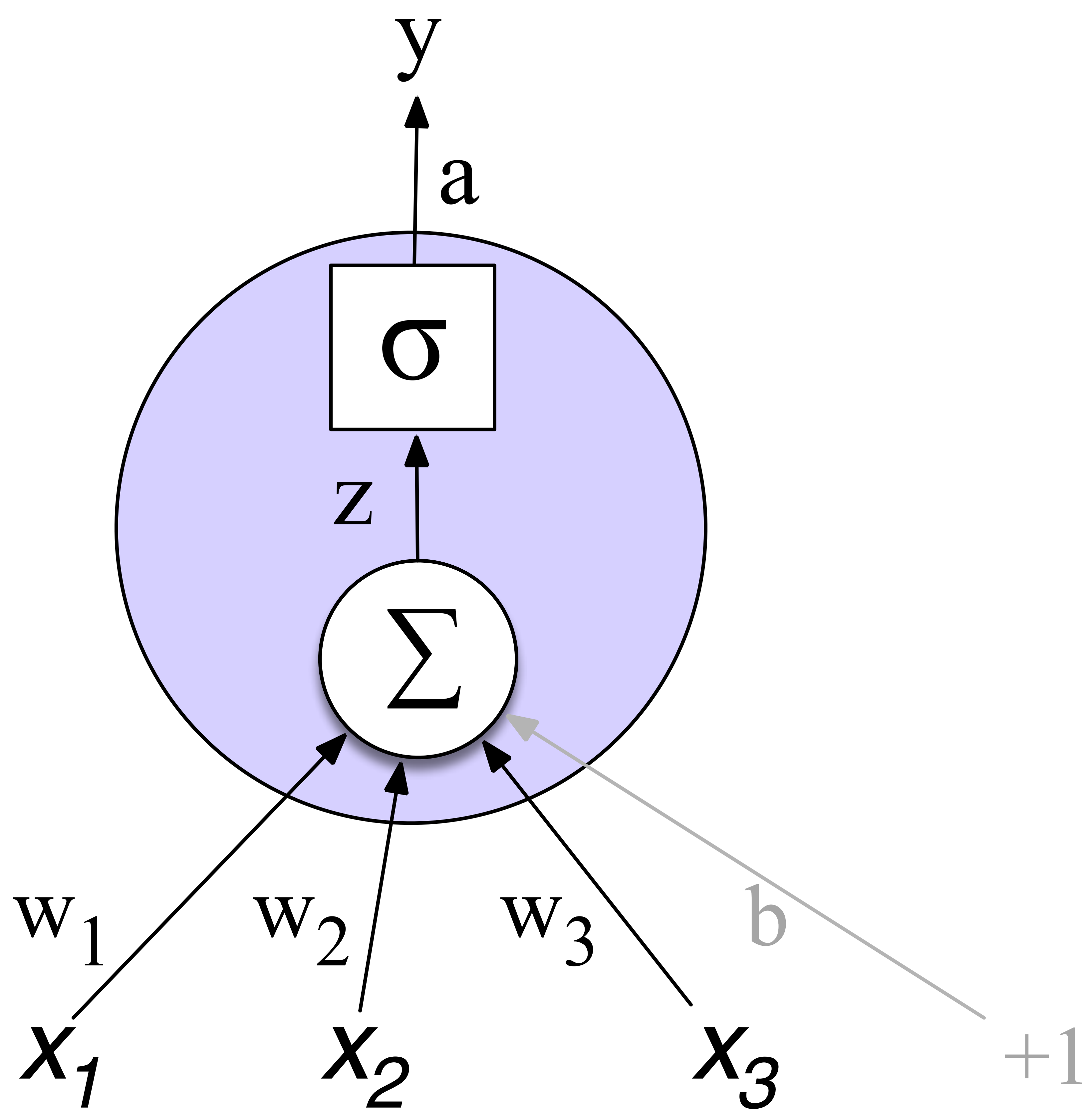
a.k.a. Perceptron



This becomes an Artificial Neuron!

a.k.a. Perceptron

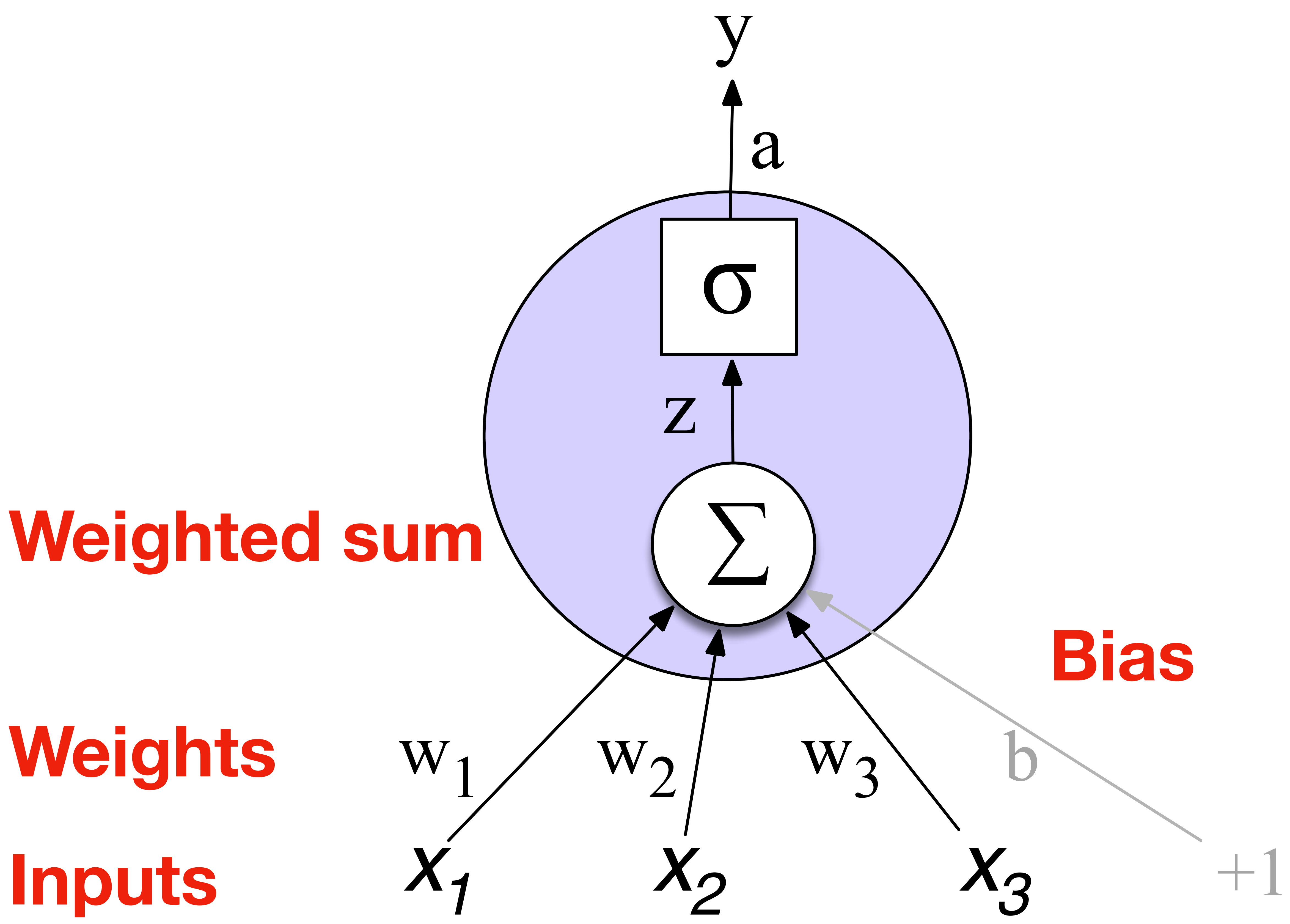
- A **perceptron** is the simplest type of artificial neuron:



This becomes an Artificial Neuron!

a.k.a. Perceptron

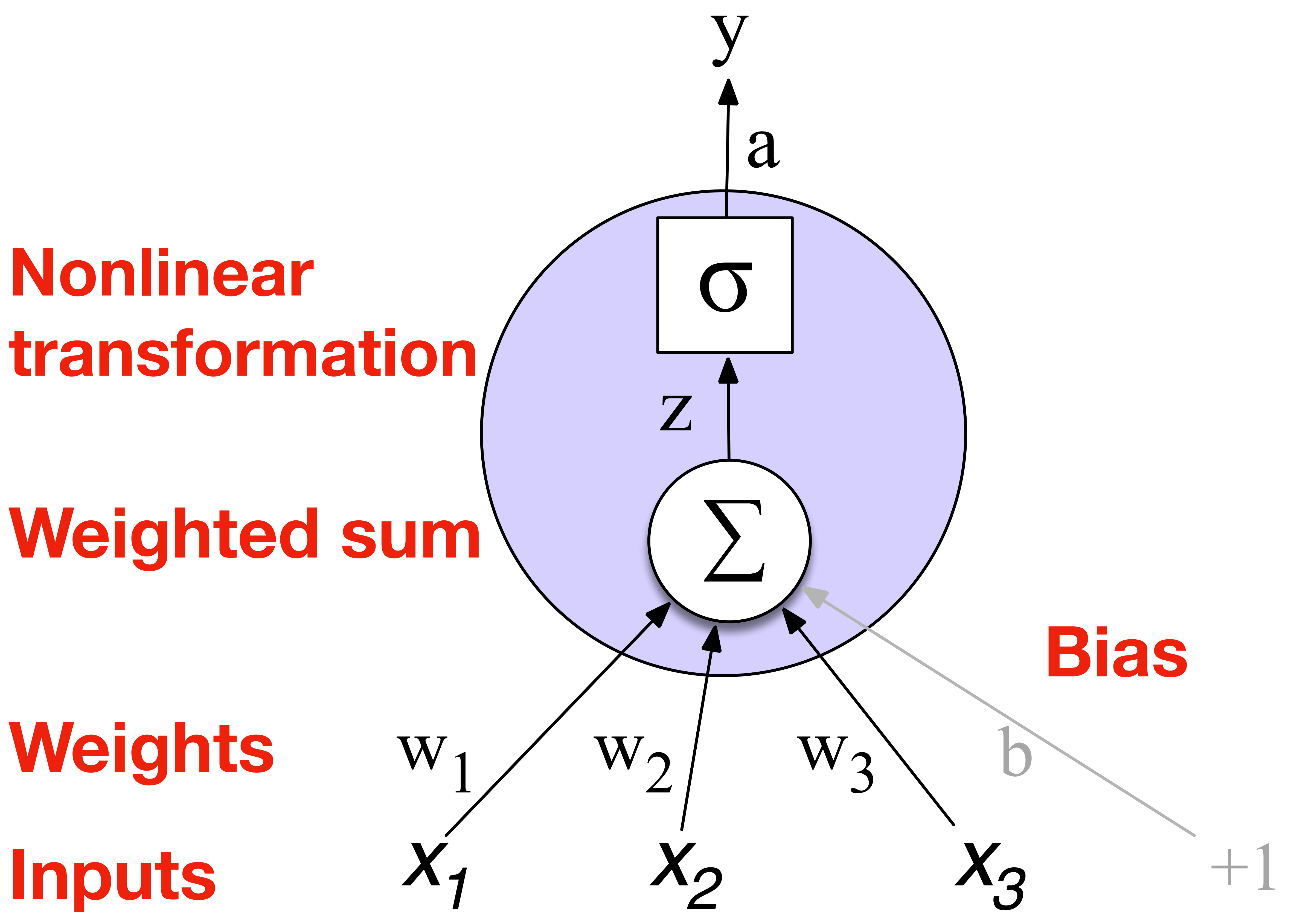
- A **perceptron** is the simplest type of artificial neuron:
    - ✳ It is a linear classifier that computes a weighted sums of its inputs;



This becomes an Artificial Neuron!

a.k.a. Perceptron

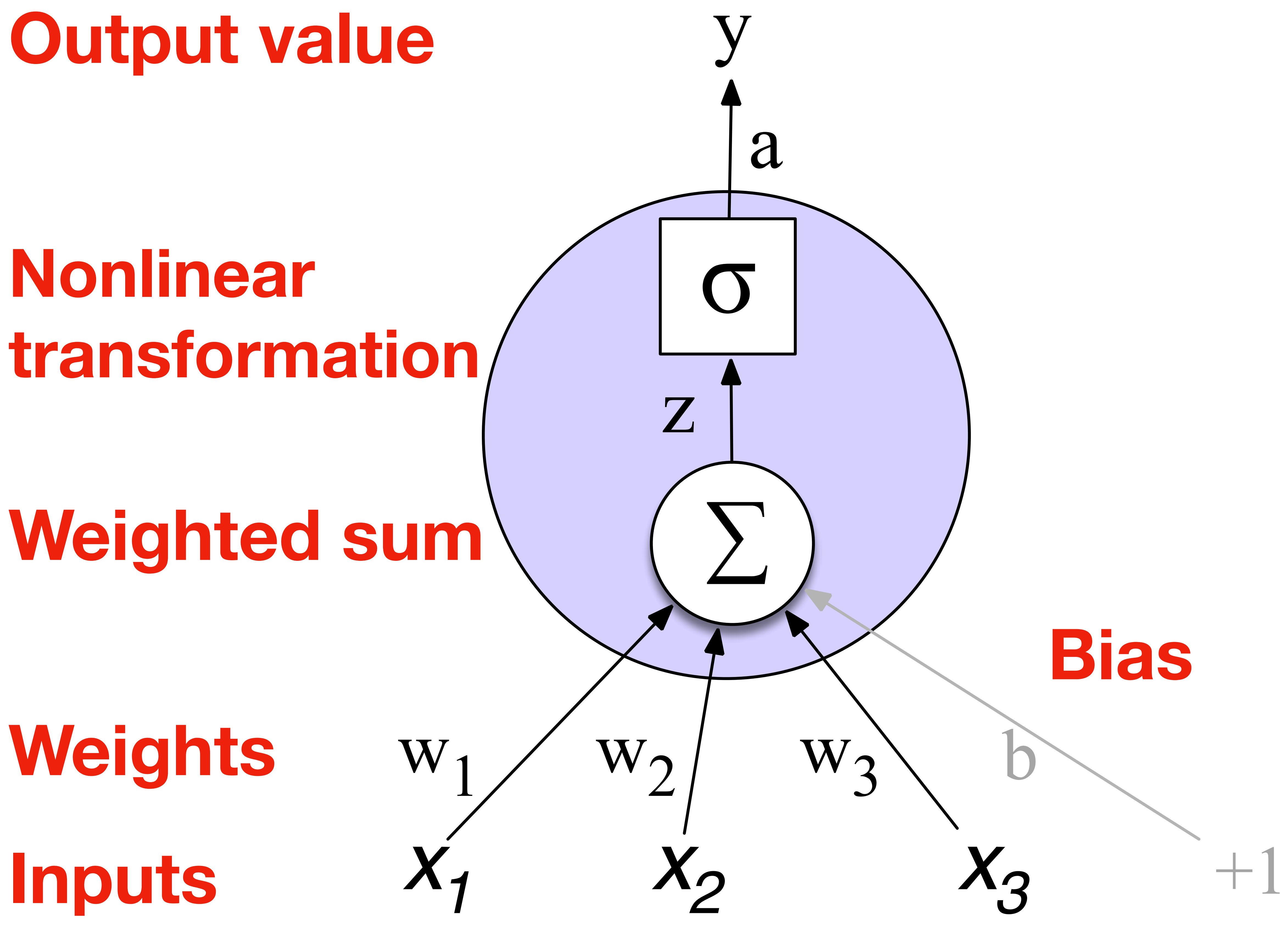
- A **perceptron** is the simplest type of artificial neuron:
  - ✴ It is a linear classifier that computes a weighted sums of its inputs;
  - ✴ It then applies a (nonlinear) activation function — deciding whether it activates;



This becomes an Artificial Neuron!

a.k.a. Perceptron

- A **perceptron** is the simplest type of artificial neuron:
  - ✳ It is a linear classifier that computes a weighted sums of its inputs;
  - ✳ It then applies a (nonlinear) activation function — deciding whether it activates;
  - ✳ And outputs a binary decision.



The Perceptron / Artificial Neuron

1958 - 1968

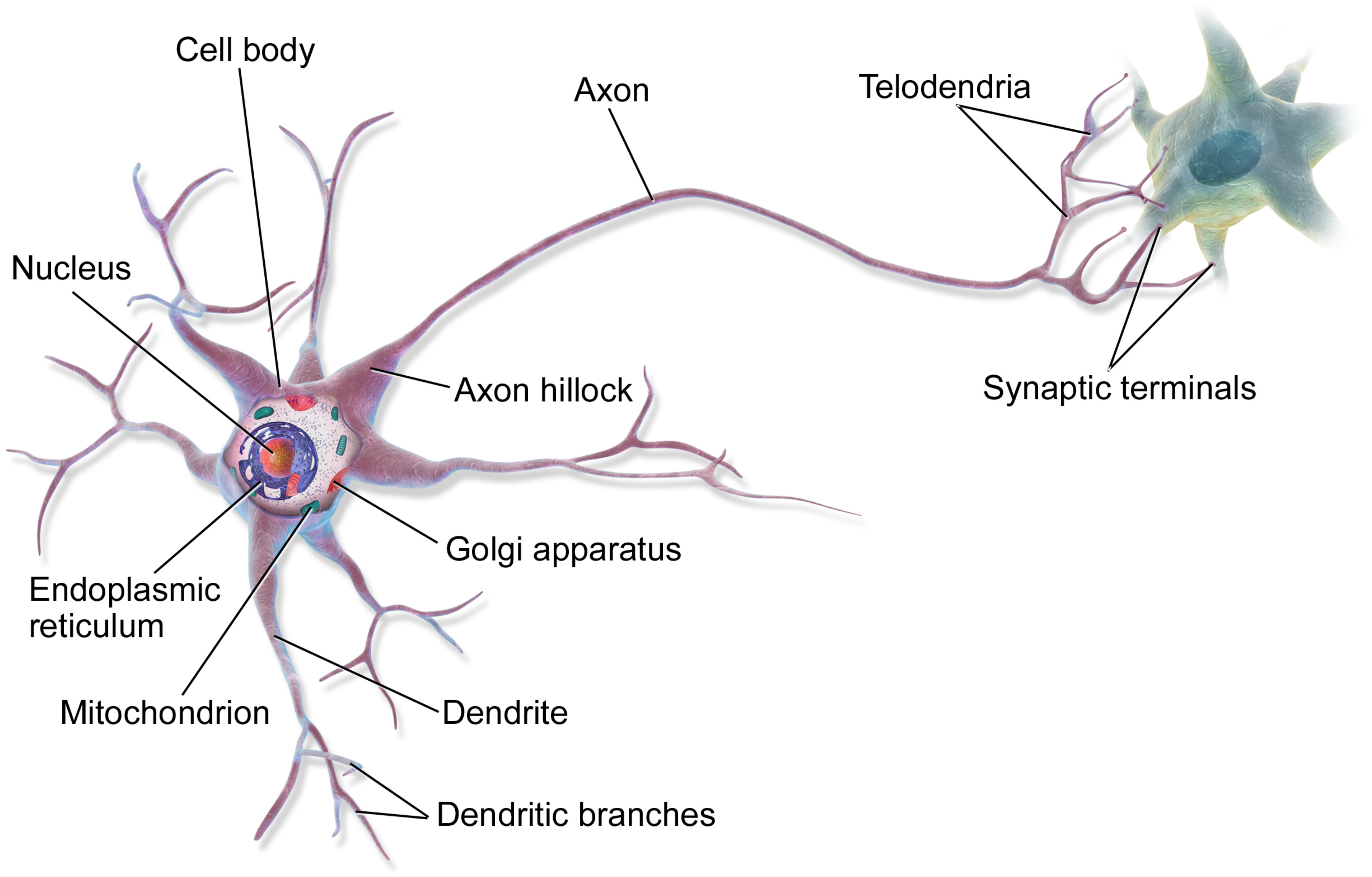
From Brain Neurons to the Perceptron

Inspiration: multiple inputs, weighted connections, one firing output.

(Left) By BruceBlaus - Own work, CC BY 3.0,
https://commons.wikimedia.org/w/index.php?curid=28761830;
(Bottom right) Mark I Perceptron: https://www.researchgate.net/figure/Frank-Rosenblatt-with-his-Mark-I-perceptronleft-and-a-graphical-representation-of_fig2_345813508

From Brain Neurons to the Perceptron

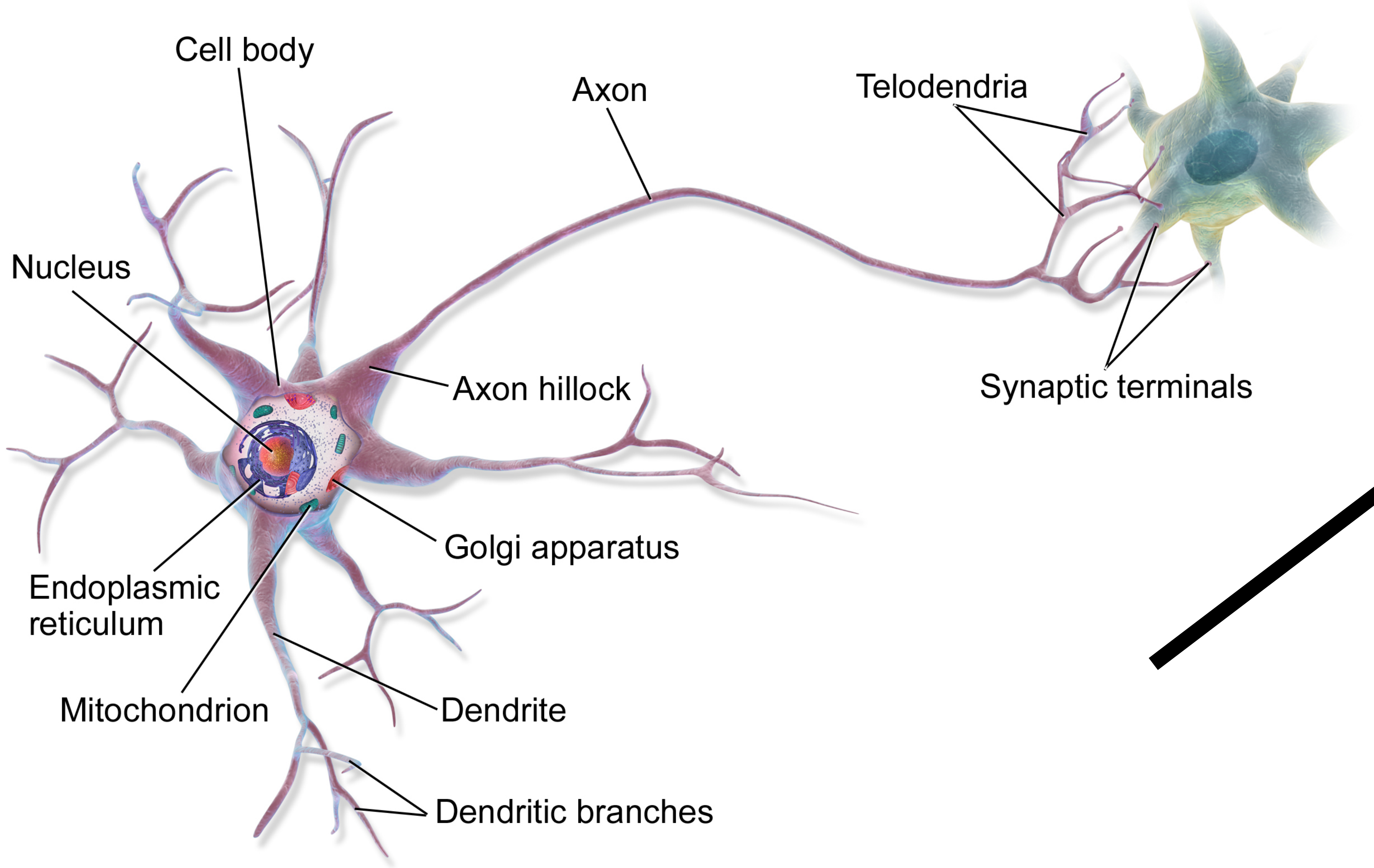
Inspiration: multiple inputs, weighted connections, one firing output.

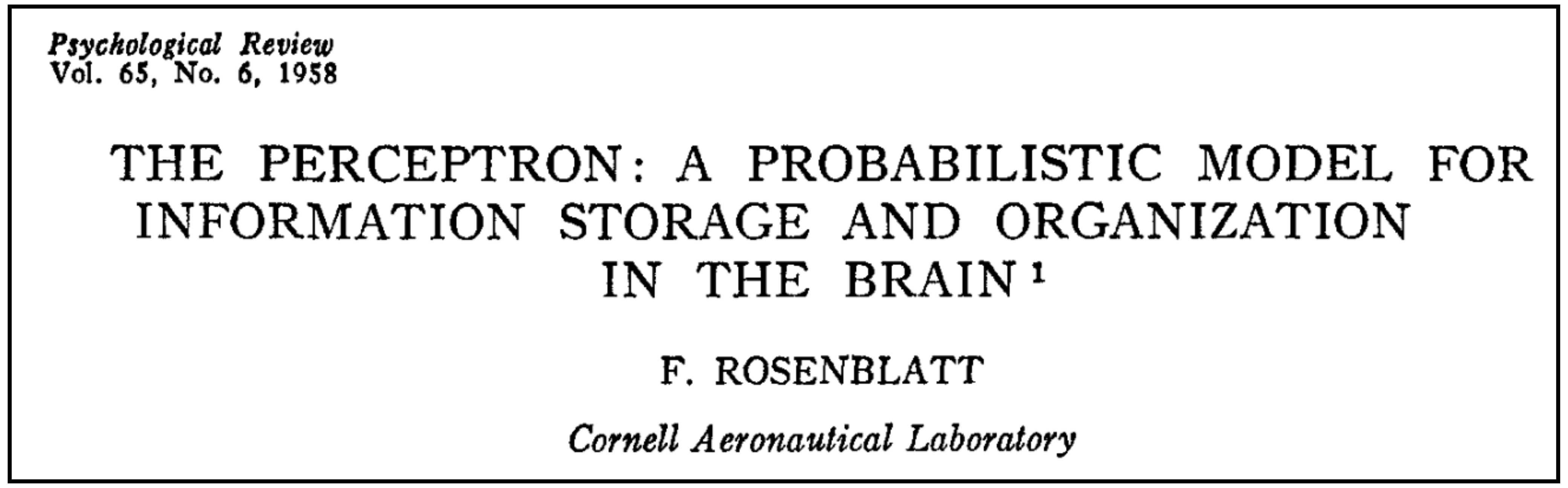


(Left) By BruceBlaus - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=28761830;
(Bottom right) Mark I Perceptron: https://www.researchgate.net/figure/Frank-Rosenblatt-with-his-Mark-I-perceptronleft-and-a-graphical-representation-of_fig2_345813508

From Brain Neurons to the Perceptron

Inspiration: multiple inputs, weighted connections, one firing output.




Psychological Review
Vol. 65, No. 6, 1958

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

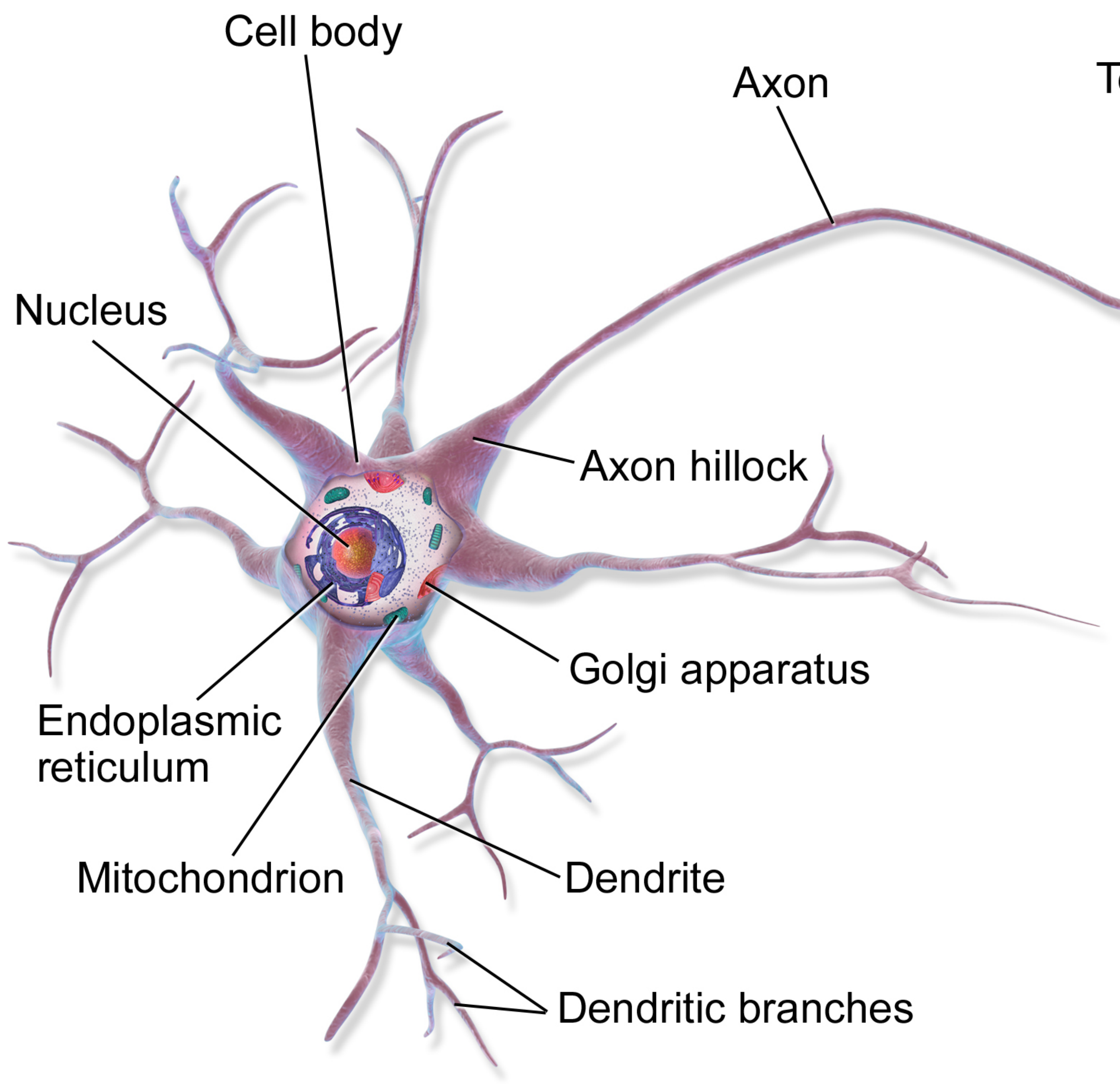
Cornell Aeronautical Laboratory

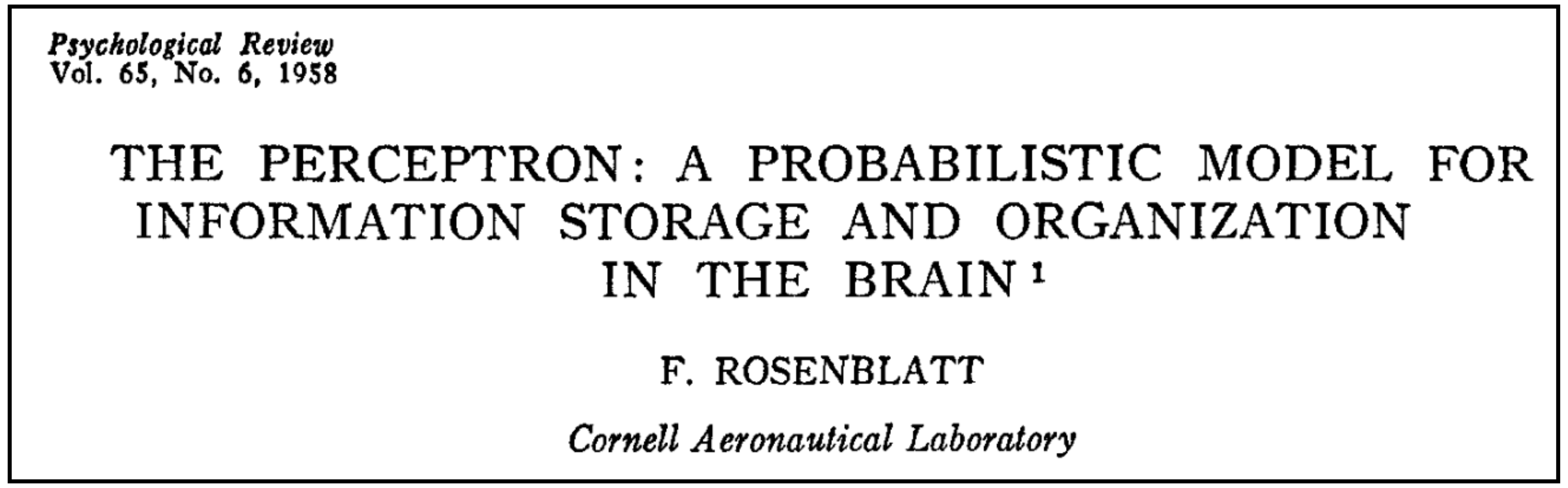
Frank Rosenblatt 1958

(Left) By BruceBlaus - Own work, CC BY 3.0,
https://commons.wikimedia.org/w/index.php?curid=28761830;
(Bottom right) Mark I Perceptron: https://www.researchgate.net/figure/Frank-Rosenblatt-with-his-Mark-I-perceptronleft-and-a-graphical-representation-of_fig2_345813508

From Brain Neurons to the Perceptron

Inspiration: multiple inputs, weighted connections, one firing output.



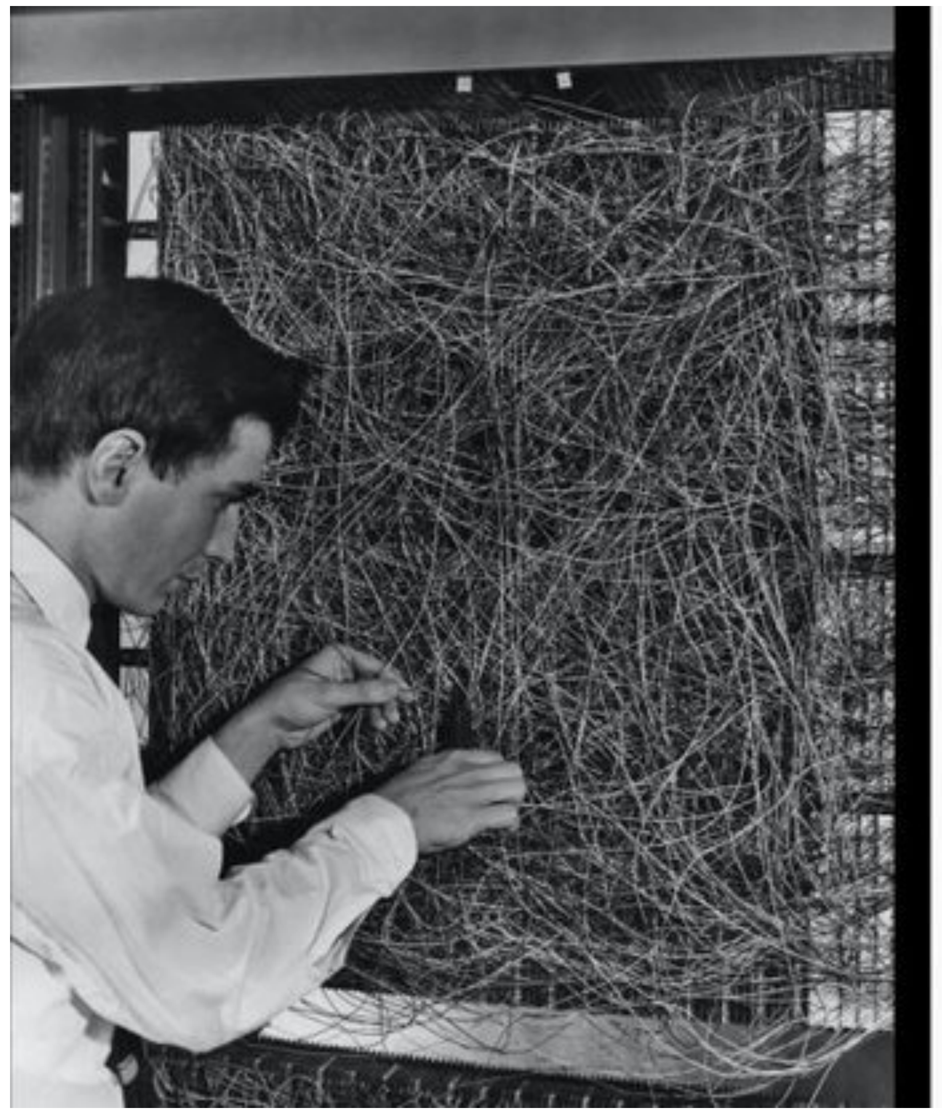

Psychological Review
Vol. 65, No. 6, 1958

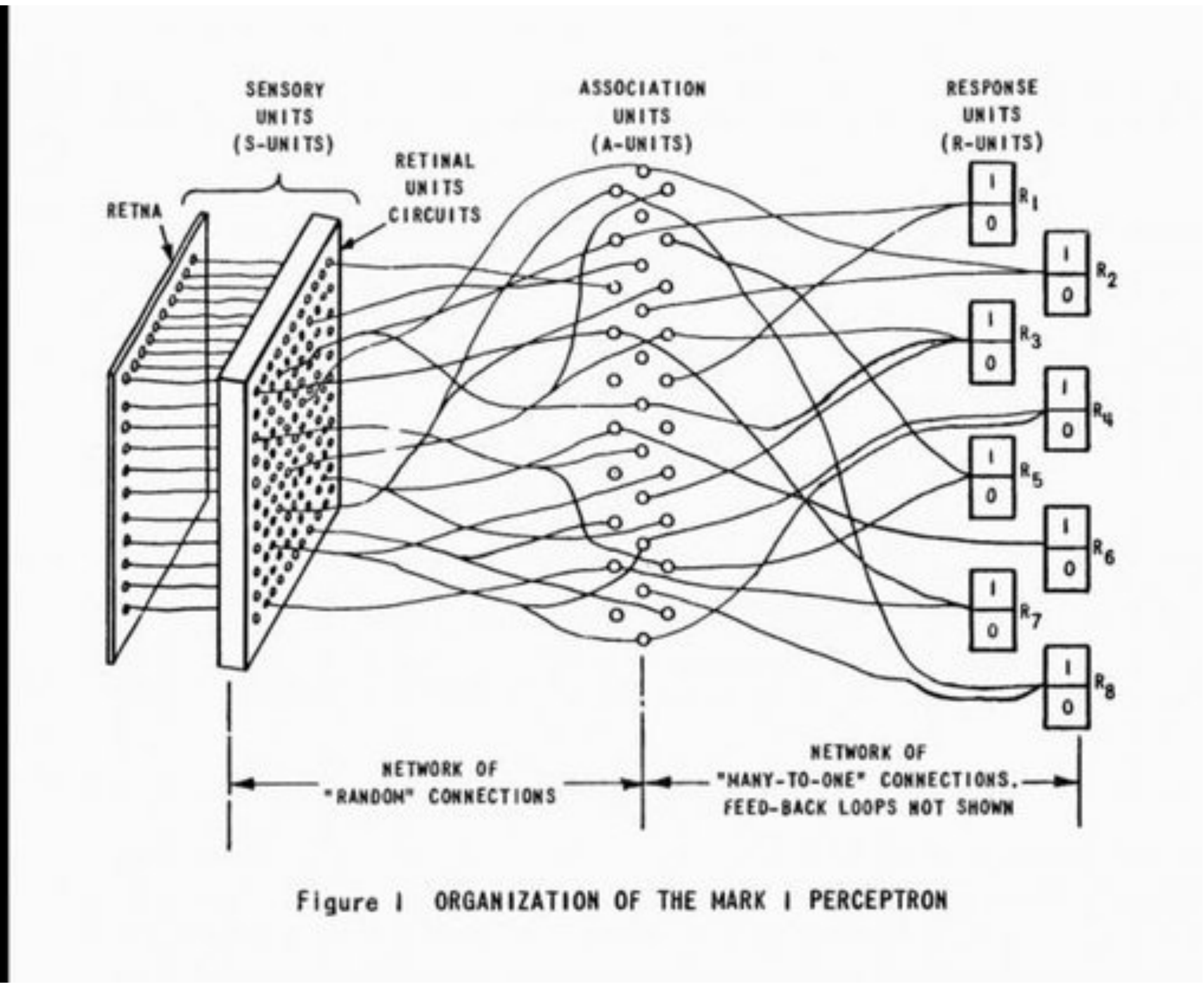
THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

Frank Rosenblatt 1958





(Left) By BruceBlaus - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=28761830;
(Bottom right) Mark I Perceptron: https://www.researchgate.net/figure/Frank-Rosenblatt-with-his-Mark-I-perceptronleft-and-a-graphical-representation-of_fig2_345813508

Some Notations

$$\boxed{y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + exp(-(\mathbf{w} \cdot \mathbf{x} + b))}}$$

Some Notations

$$\boxed{y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + exp(-(\mathbf{w} \cdot \mathbf{x} + b))}}$$

- Represent the input feature values and the weights as vectors, $\mathbf{x}$ and $\mathbf{w}$;
  - For the grading example, $\mathbf{w} = [w_1, w_2, w_3, w_4] = [0.2, 0.3, 0.5, 0.1]$;

Some Notations

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + exp(-(\mathbf{w} \cdot \mathbf{x} + b))}$$

- Represent the input feature values and the weights as vectors, $\mathbf{x}$ and $\mathbf{w}$;
  - For the grading example, $\mathbf{w} = [w_1, w_2, w_3, w_4] = [0.2, 0.3, 0.5, 0.1]$;
- Therefore, $\sum_{i=1}^{n} w_i x_i = \mathbf{w} \cdot \mathbf{x}$, where $\cdot$ represents dot product (element-wise product between two vectors of the same size).

Some Notations

$$\boxed{y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + exp(-(\mathbf{w} \cdot \mathbf{x} + b))}}$$

- Represent the input feature values and the weights as vectors, $\mathbf{x}$ and $\mathbf{w}$;
    - For the grading example, $\mathbf{w} = [w_1, w_2, w_3, w_4] = [0.2, 0.3, 0.5, 0.1]$;
- Therefore, $\sum_{i=1}^{n} w_i x_i = \mathbf{w} \cdot \mathbf{x}$, where $\cdot$ represents dot product (element-wise product between two vectors of the same size).
- In practice, we incorporate the bias term into the vector representation, where bias b always gets weight = 1.0;
    - This gives us $\mathbf{w} = [w_1, w_2, w_3, w_4, w_b] = [0.2, 0.3, 0.5, 0.1, 1.0]$

Notes on Nonlinear Activation Functions

Beyond sigmoid

Notes on Nonlinear Activation Functions

Beyond sigmoid

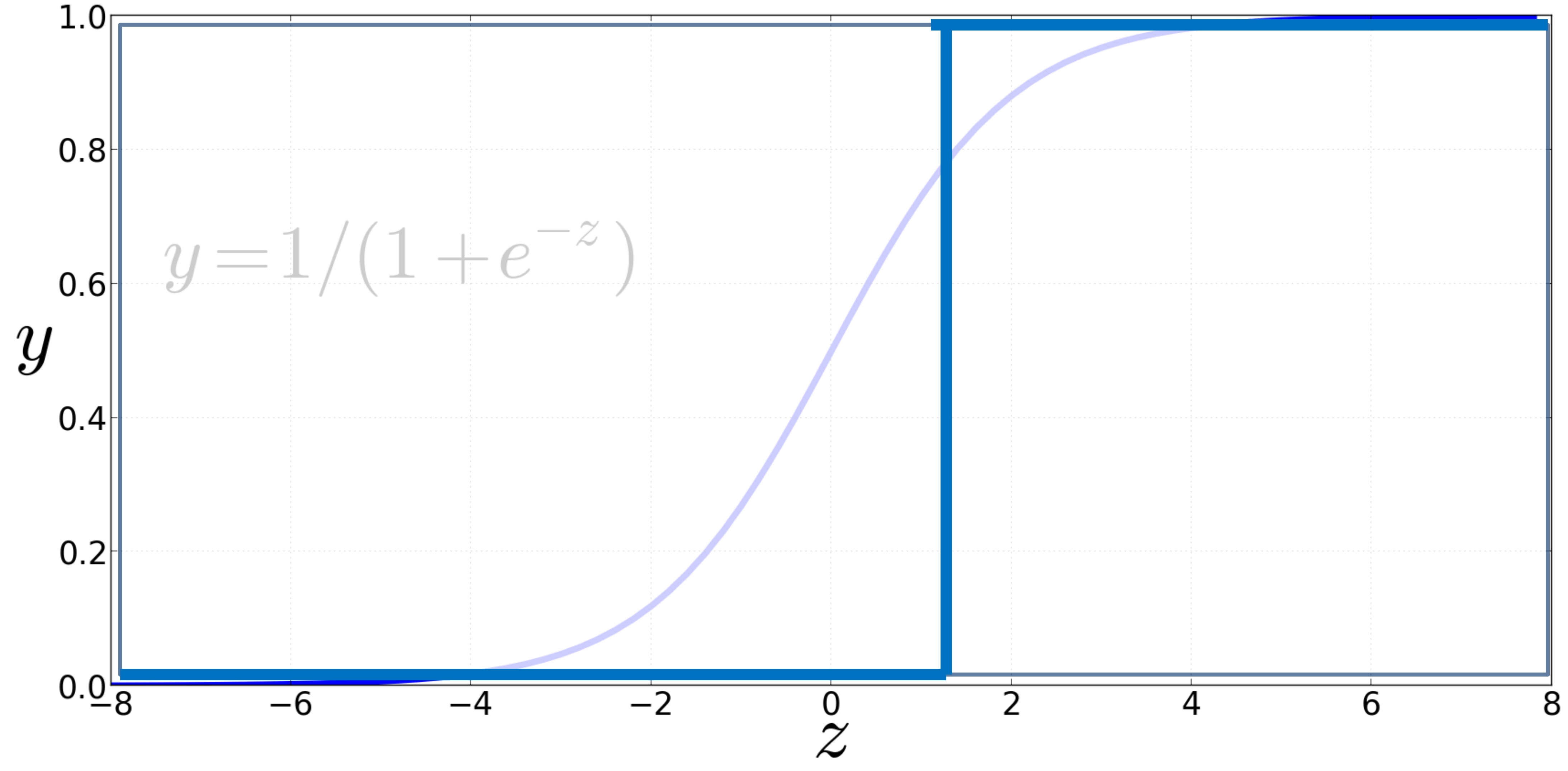
- In standard logistic regression, we use Sigmoid (σ) as the activation function.
  - We want the map *any* real value (from weighted sum) to $[0, 1]$ to be interpreted as probability.

Notes on Nonlinear Activation Functions

Beyond sigmoid

- In standard logistic regression, we use Sigmoid (σ) as the activation function.
  - We want the map *any* real value (from weighted sum) to $[0, 1]$ to be interpreted as probability.
- In the grading example, we use a step function:

$$f(z) = \begin{cases} 1, & \text{if } z \geq 0.6 \\ 0, & \text{otherwise} \end{cases}$$



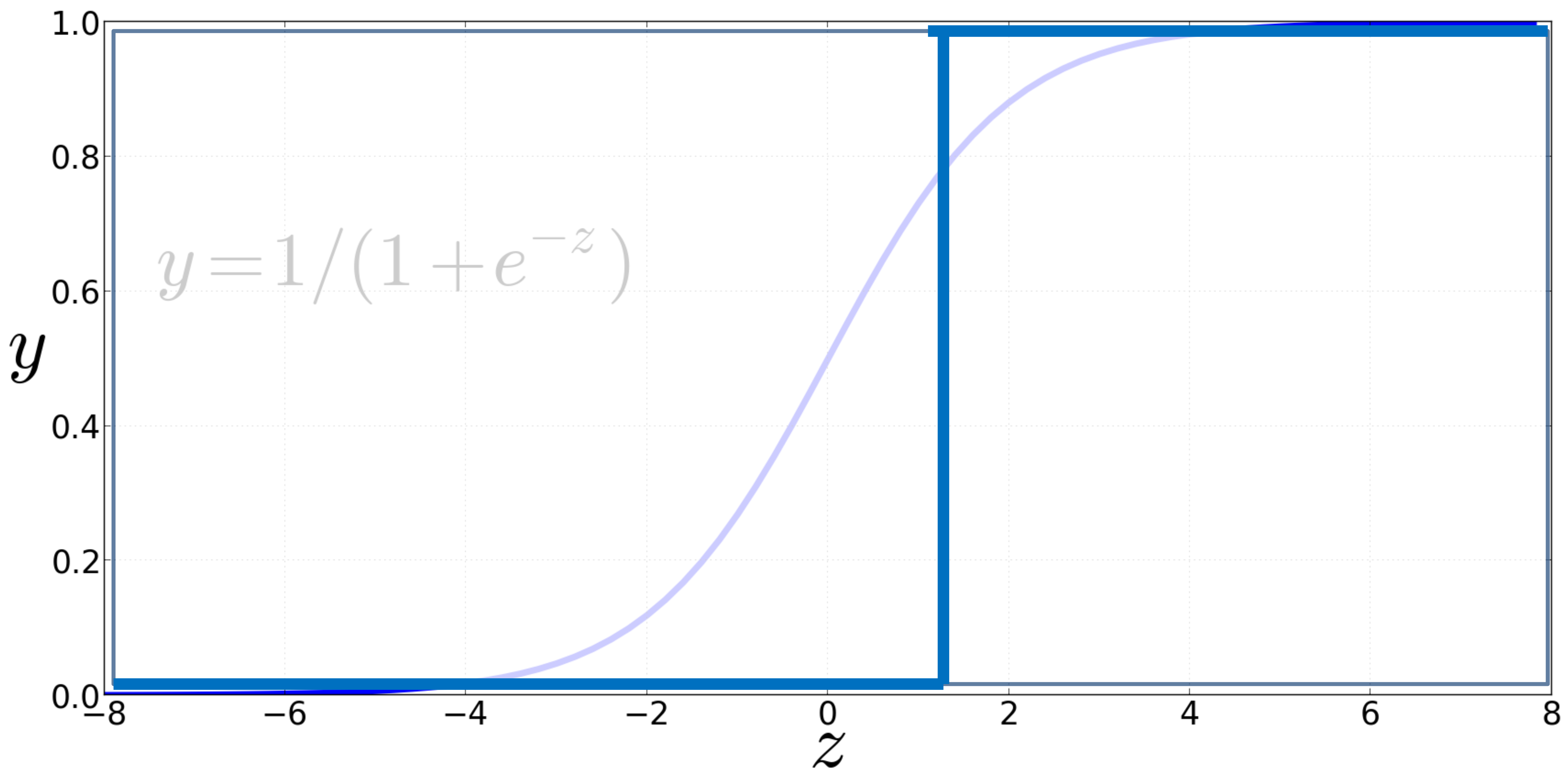
Notes on Nonlinear Activation Functions

Beyond sigmoid

- In standard logistic regression, we use Sigmoid (σ) as the activation function.
    - We want the map *any* real value (from weighted sum) to $[0, 1]$ to be interpreted as probability.
- In the grading example, we use a step function: $$f(z) = \begin{cases} 1, & \text{if } z \geq 0.6 \\ 0, & \text{otherwise} \end{cases}$$

Intuition: nonlinearity enables the model to represent relations beyond linear functions (i.e., cannot be represented by matrix operations)!

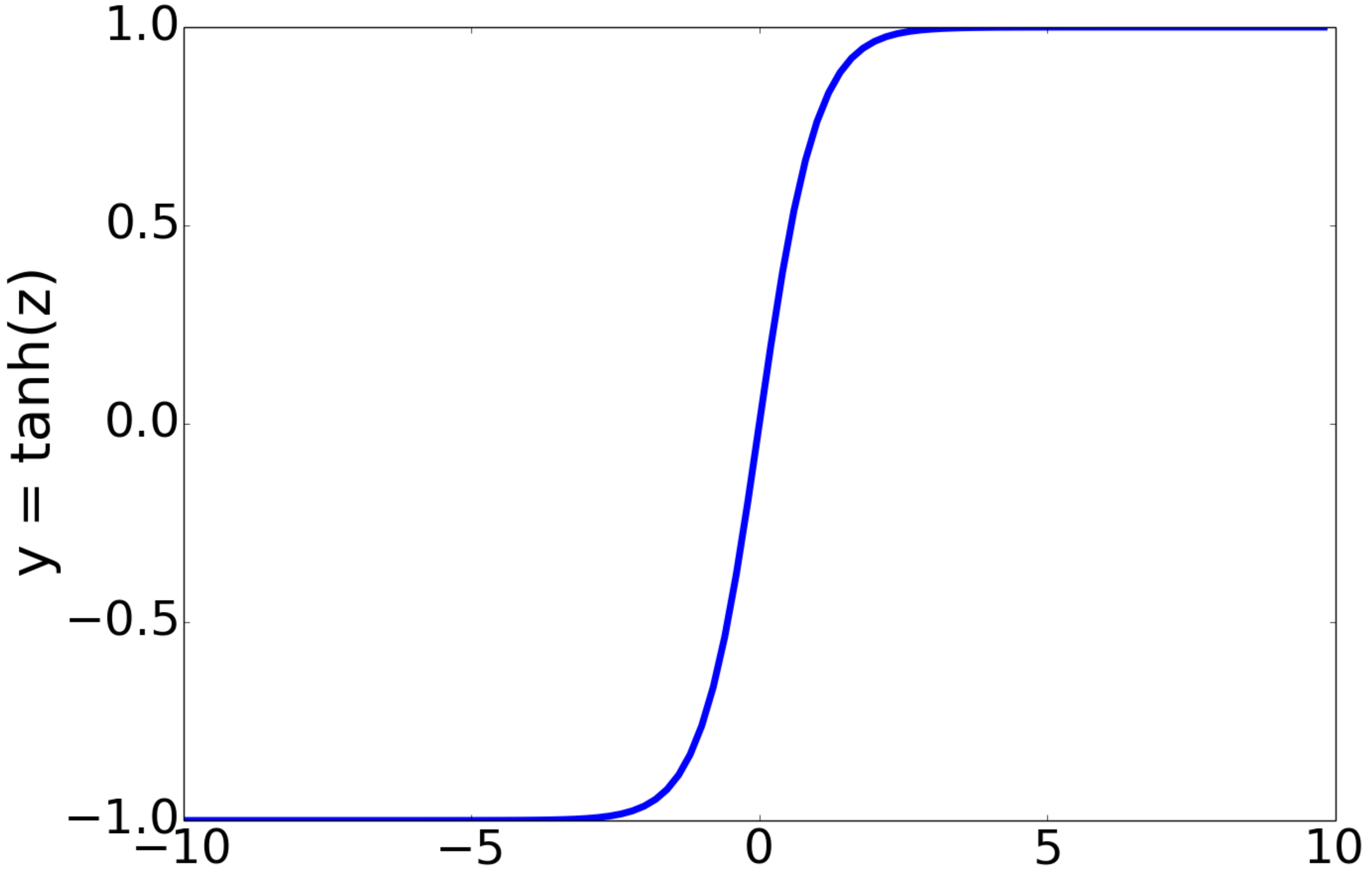
* *will come back to it later*



Other Nonlinear Activation Functions

Choose activation functions wisely given your problem setting

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

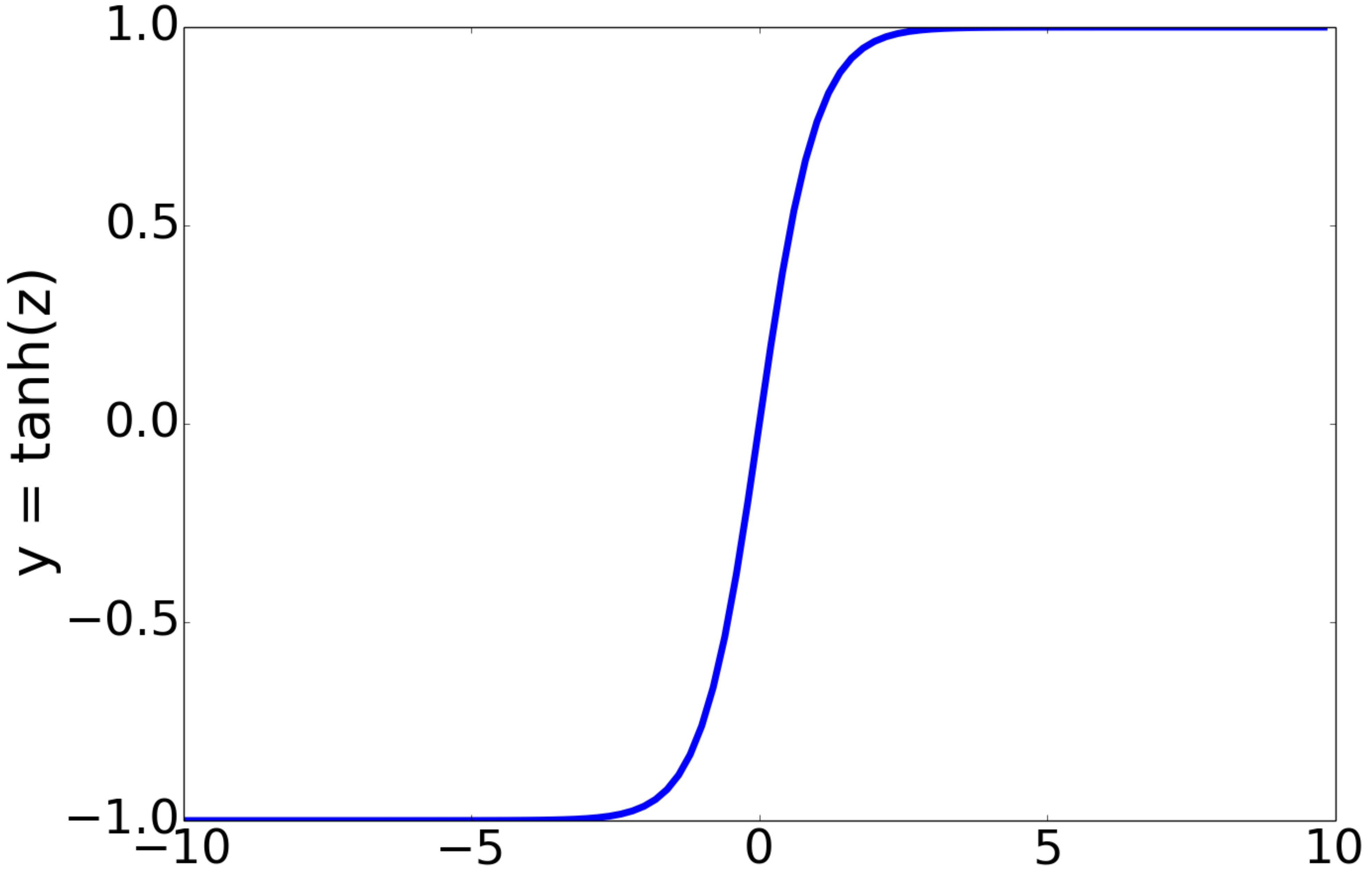


tanh

Other Nonlinear Activation Functions

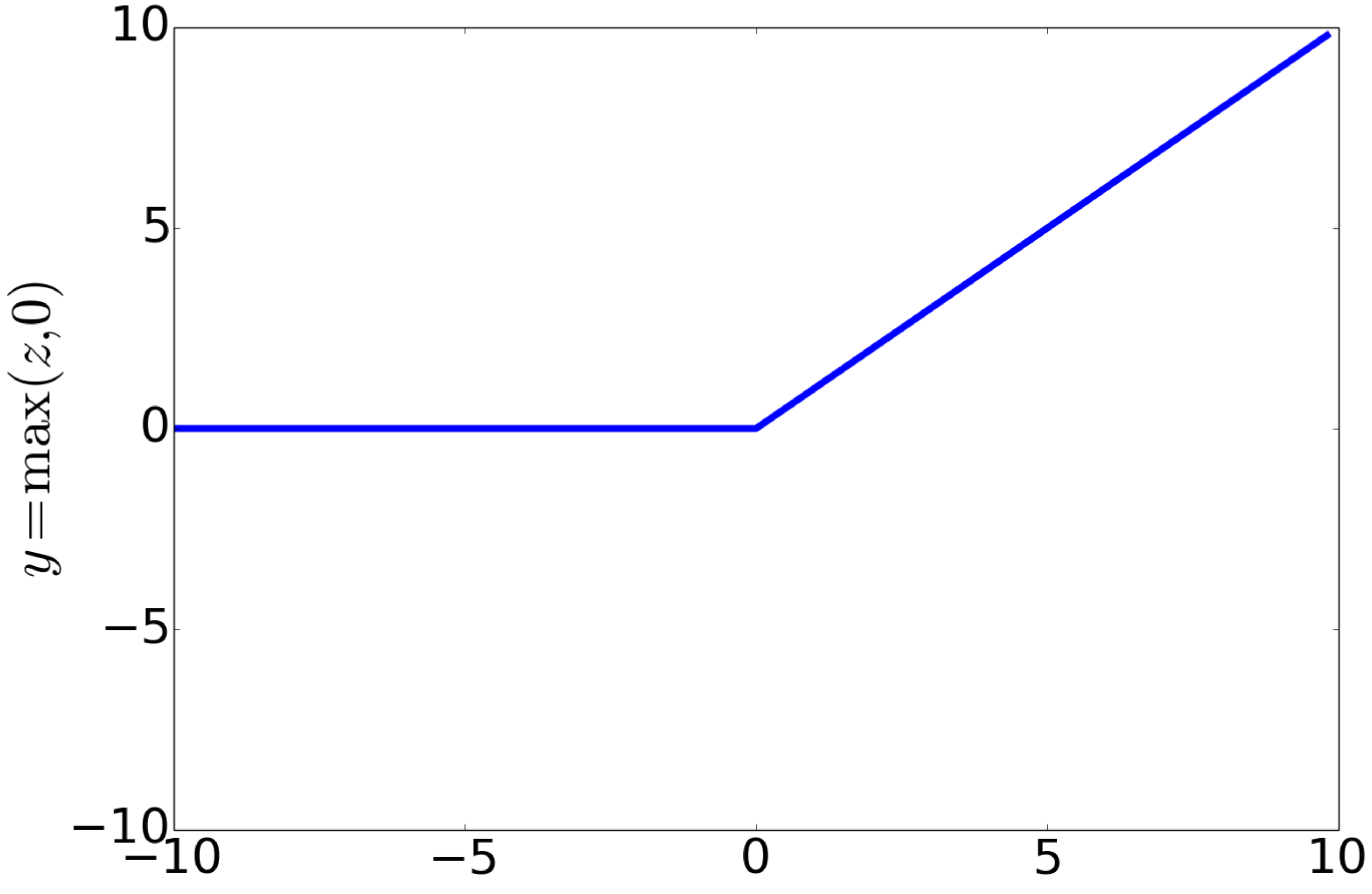
Choose activation functions wisely given your problem setting

$$\text{tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



tanh

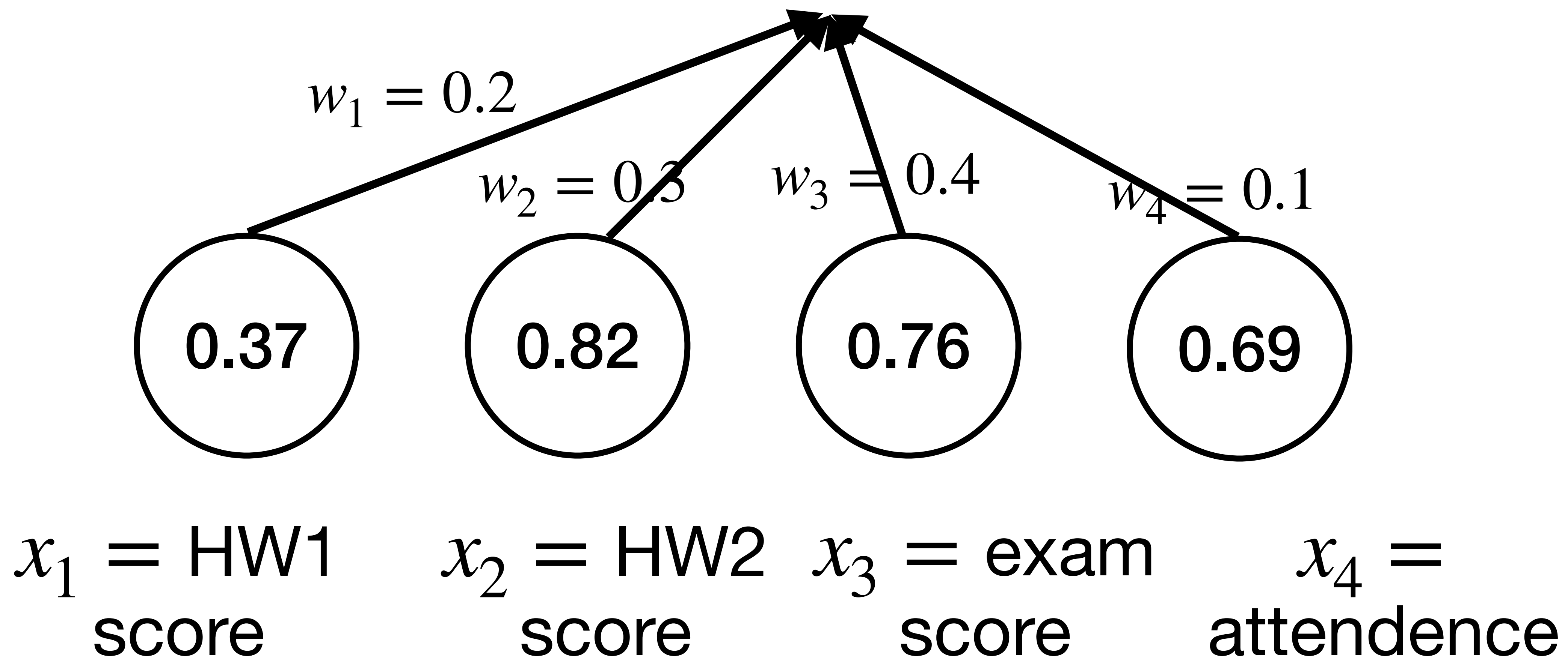
$$\text{ReLu}(z) = max(z,\ 0)$$



ReLU
(Rectified Linear Unit)

An Example Computation

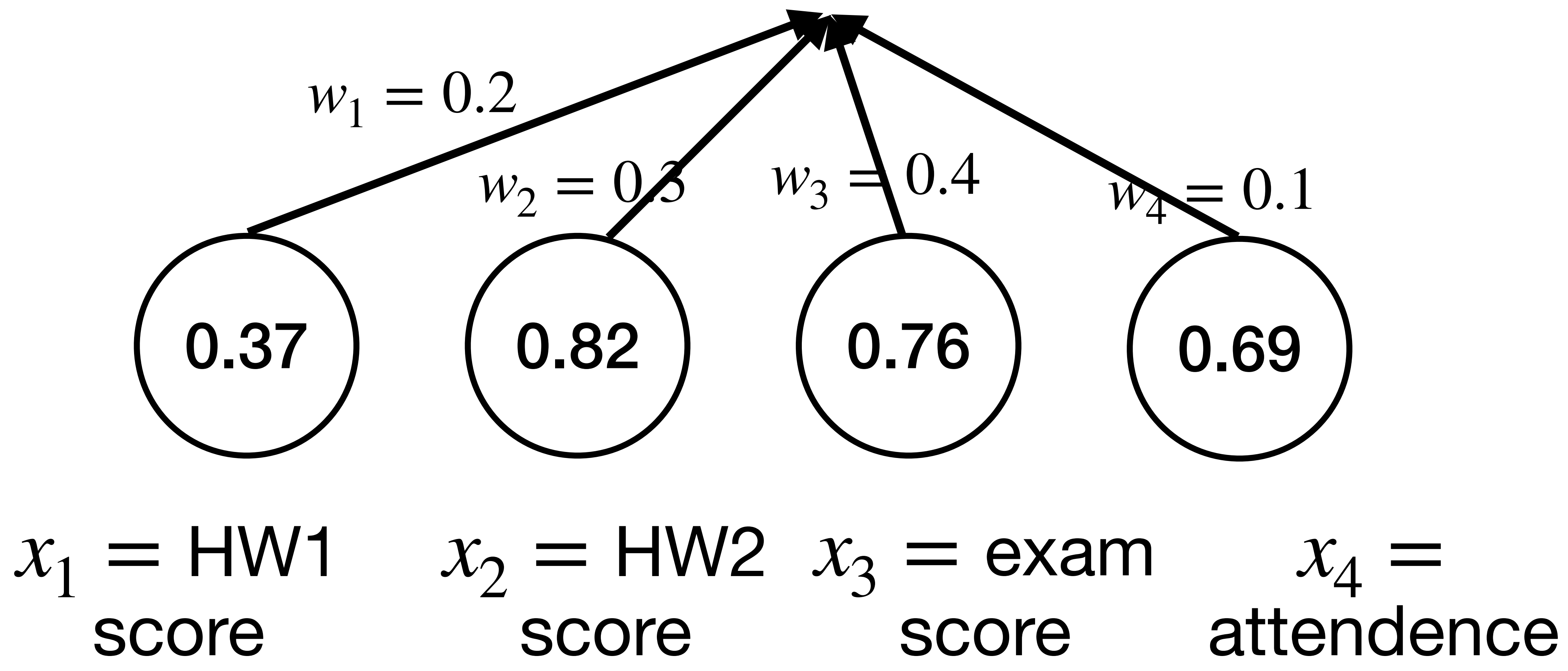
Back to the grading example



An Example Computation

Back to the grading example

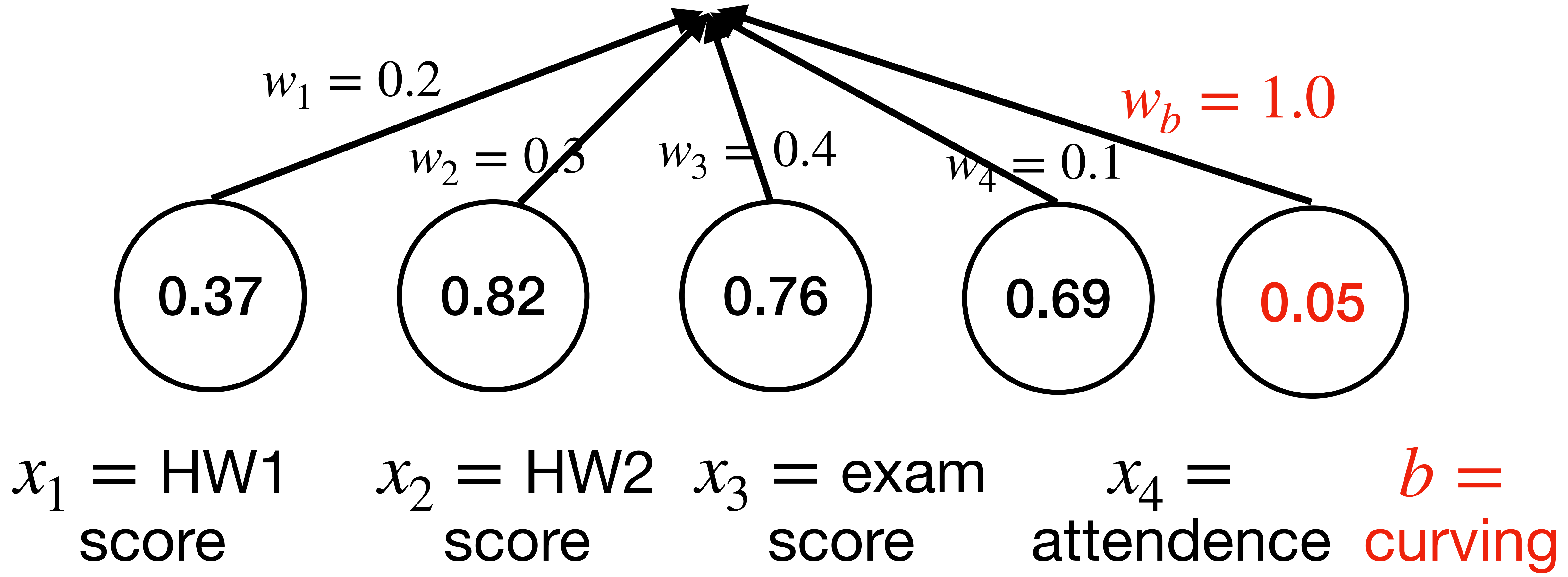
- Suppose we applied a curve and adds 5 points (i.e., 0.05%) to the grade.



An Example Computation

Back to the grading example

- Suppose we applied a curve and adds 5 points (i.e., 0.05%) to the grade.

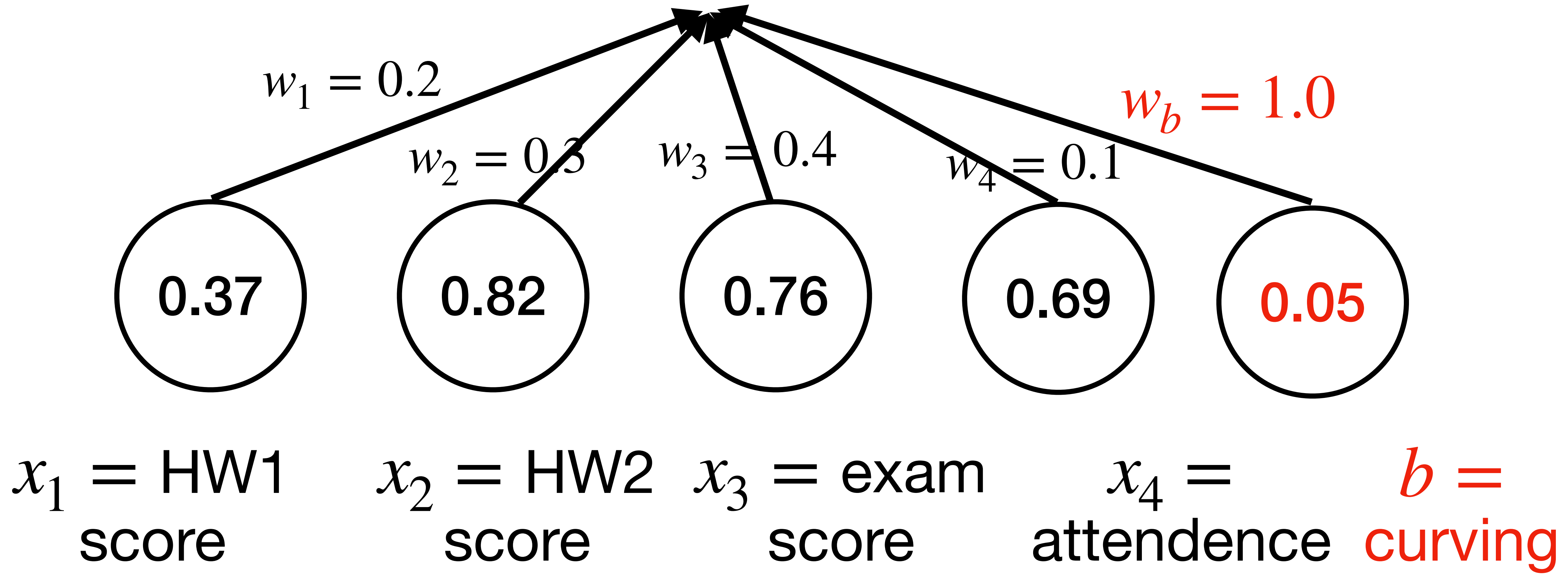


An Example Computation

Back to the grading example

- Suppose we applied a curve and adds 5 points (i.e., 0.05%) to the grade.

- Then, we have vector representations:
  - $\mathbf{w}$ = [0.2, 0.3, 0.4, 0.1, 1.0];
  - $\mathbf{b}$= [0.37, 0.82, 0.76, 0.69, 0.05];



An Example Computation

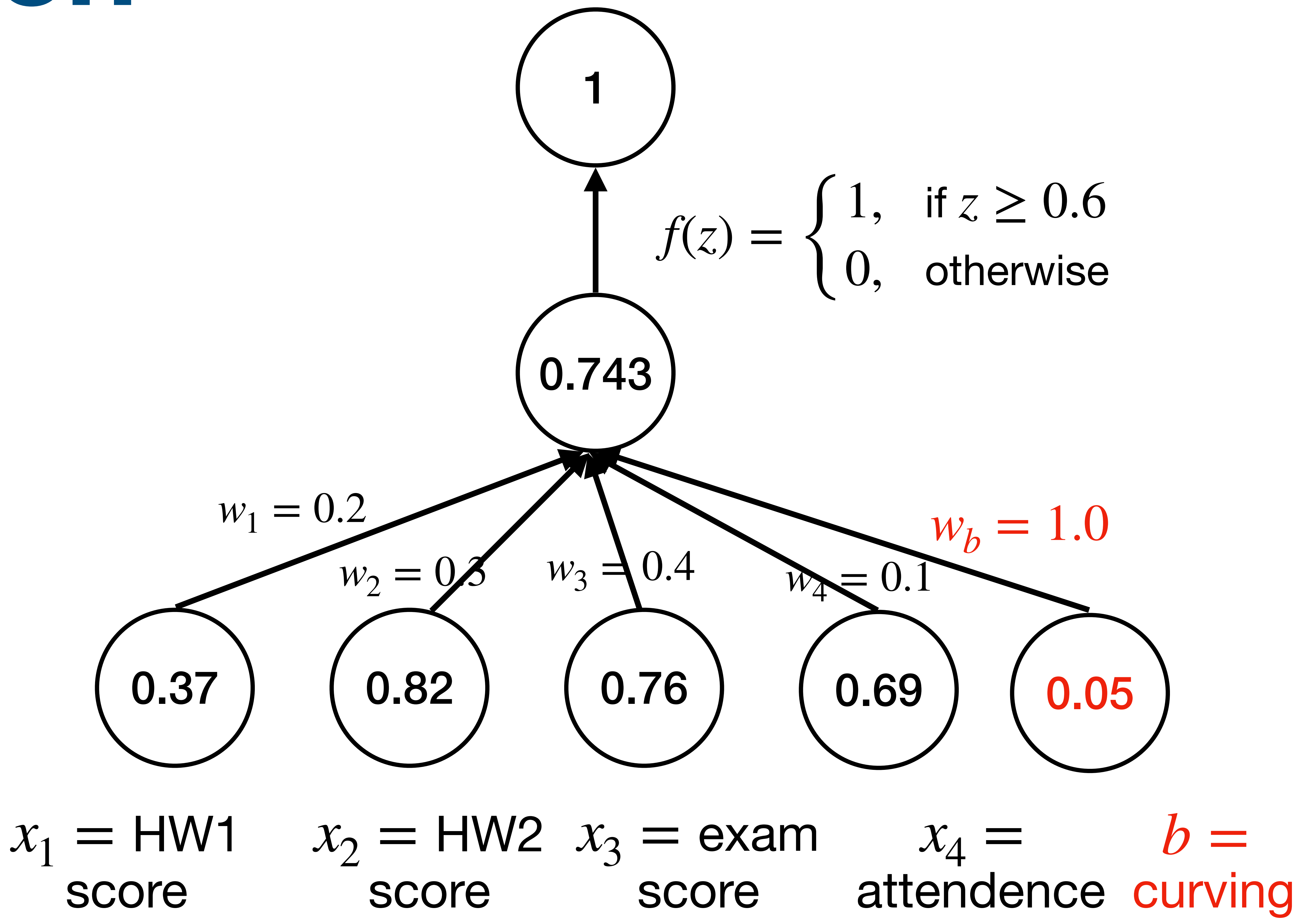
Back to the grading example

- Suppose we applied a curve and adds 5 points (i.e., 0.05%) to the grade.

- Then, we have vector representations:
  - $\mathbf{w}$ = [0.2, 0.3, 0.4, 0.1, 1.0];
  - $\mathbf{b}$= [0.37, 0.82, 0.76, 0.69, 0.05];

- Then:

$$y = f(\mathbf{w} \cdot \mathbf{x} + b)$$
$$= f(0.2 \times 0.37 + 0.3 \times 0.82 + 0.4 \times 0.76 + 0.1 \times 0.69 + 1.0 \times 0.05)$$
$$= f(0.743) = Boolean(0.743 \geq 0.6) = 1$$



Wait…. Where do weights come from?

The Perceptron Learning Rule

You need to do error-driven learning in order to "perceive" and "adapt"

$$\mathbf{w} := \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$$

The Perceptron Learning Rule

You need to do error-driven learning in order to "perceive" and "adapt"

$$\mathbf{w} := \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$$

- Given an input-output pair $(\mathbf{x}, y)$ where $\mathbf{x}$ is a vector and $y \in \{0,1\}$:
  - $\hat{y}$ is the **predicted label** from the current set of weights;
  - η is the **learning rate**: how much to adjust each weight given a datapoint.

The Perceptron Learning Rule

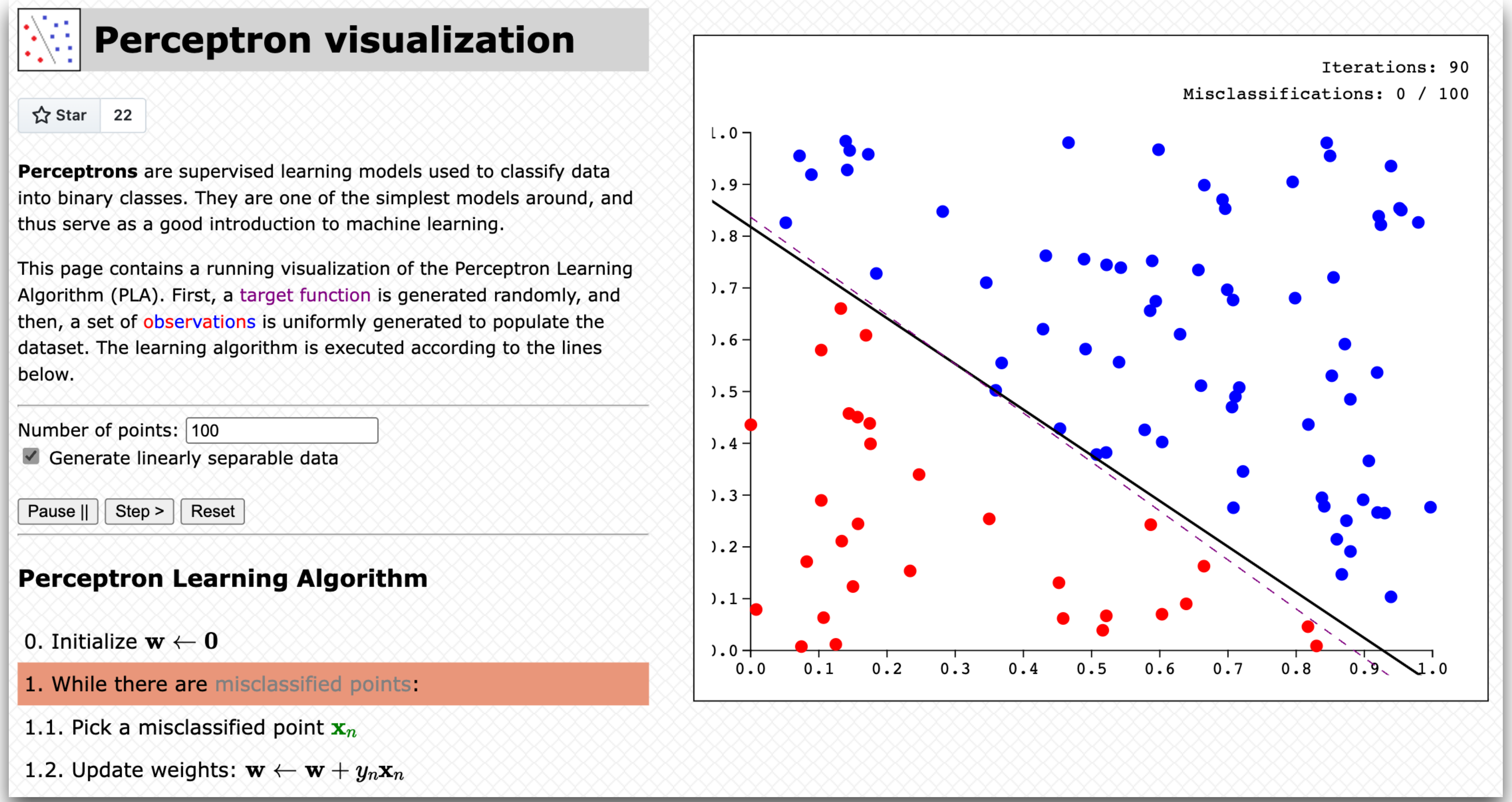
You need to do error-driven learning in order to "perceive" and "adapt"

$$\mathbf{w} := \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$$

- Given an input-output pair $(\mathbf{x}, y)$ where $\mathbf{x}$ is a vector and $y \in \{0,1\}$:
  - $\hat{y}$ is the **predicted label** from the current set of weights;
  - η is the **learning rate**: how much to adjust each weight given a datapoint.
- **Intuition (very important!)**:
  - If I made an error ($\hat{y}$ differs from y, so that $(y - \hat{y}) = \pm 1$), then update each weight w_i with the amount ηx_i towards the correct direction (sign of $y - \hat{y}$).
  - Each value of x_i determines, for this example, how much is w_i *faulty / responsible* for this mis-classification!

A Real-Time Illustration

Perceptron is really “perceiving and adapting” — aka learning!



Interium Summary 1

Logistic Regression → Artifical Neuron / Perceptron → Learning

Interium Summary 1

Logistic Regression → Artifical Neuron / Perceptron → Learning

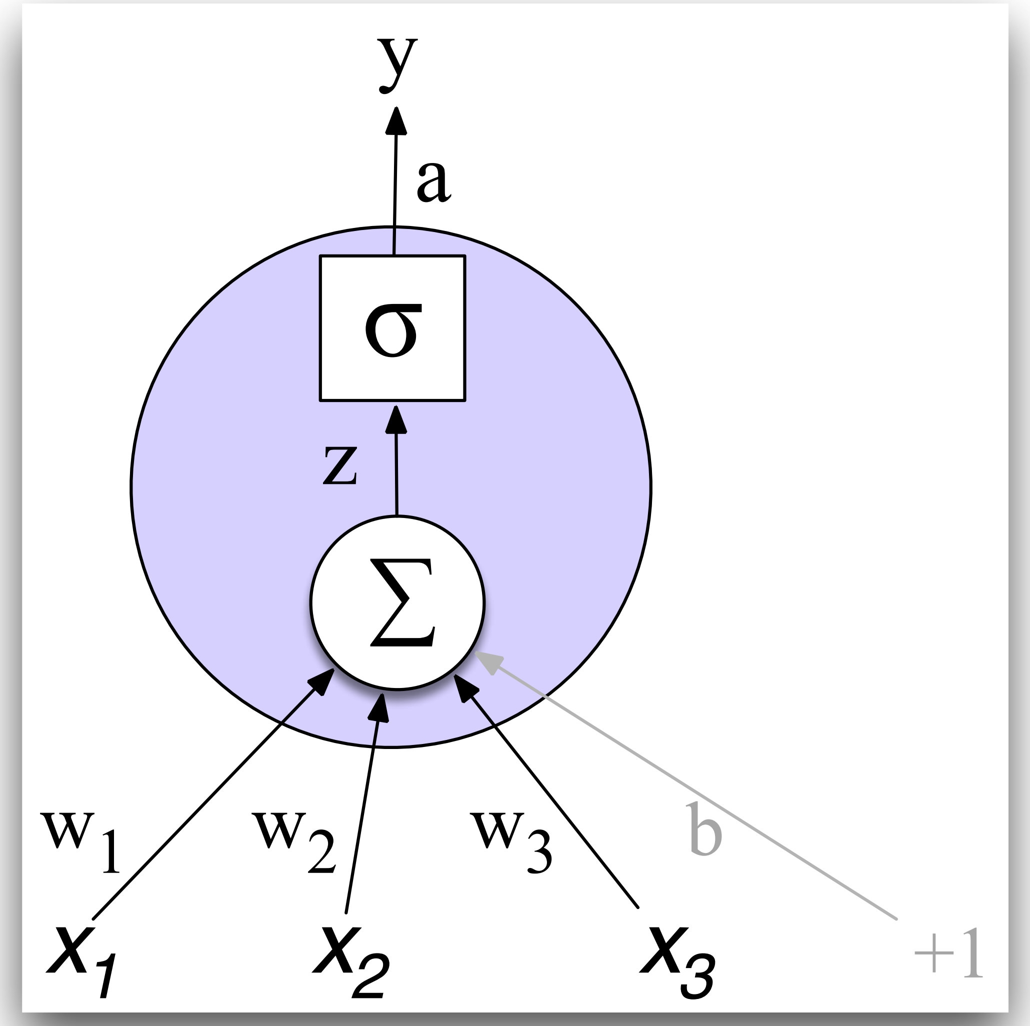
- **Logistic Regression is discriminative**: unlike Naive Bayes, it doesn't care how $\mathbf{x}$ is produced (i.e., not targeting to model the distribution of $\mathbf{x}$). It focuses on learning the mapping from $\mathbf{x}$ to y.

Interium Summary 1

Logistic Regression → Artifical Neuron / Perceptron → Learning

- **Logistic Regression is discriminative**: unlike Naive Bayes, it doesn't care how $\mathbf{x}$ is produced (i.e., not targeting to model the distribution of $\mathbf{x}$). It focuses on learning the mapping from $\mathbf{x}$ to y.

- **Logistic regressoin is a specific version of Perceptron (an artificial neuron)**: where the nonlinear function is Sigmoid, and there is only one output node for binary classification.



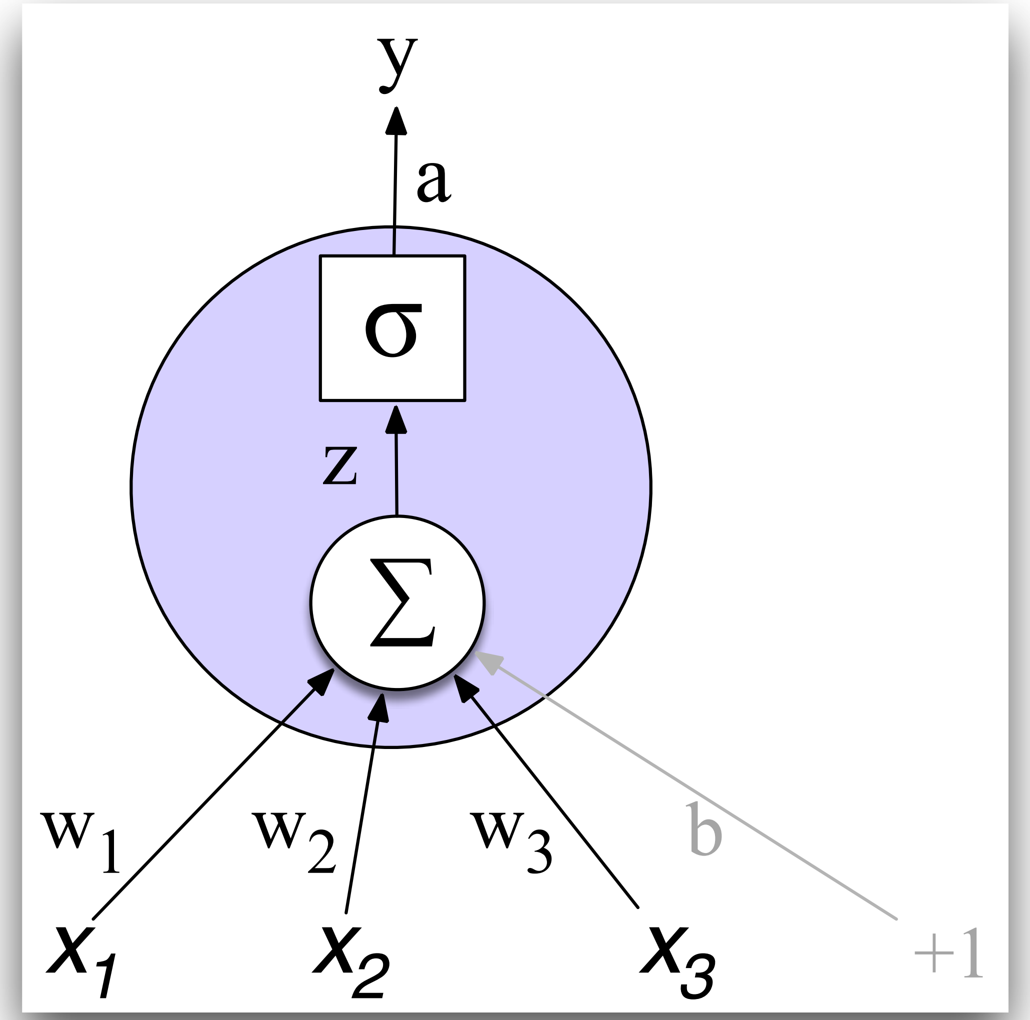
Interium Summary 1

Logistic Regression → Artifical Neuron / Perceptron → Learning

- **Logistic Regression is discriminative**: unlike Naive Bayes, it doesn't care how $\mathbf{x}$ is produced (i.e., not targeting to model the distribution of $\mathbf{x}$). It focuses on learning the mapping from $\mathbf{x}$ to y.

- **Logistic regressoin is a specific version of Perceptron (an artificial neuron)**: where the nonlinear function is Sigmoid, and there is only one output node for binary classification.

- **Perceptron** is inspired by brain neurons and was initially invented to model "information detection, storage, and recognition" (Rosenblatt 1958);



Interium Summary 1

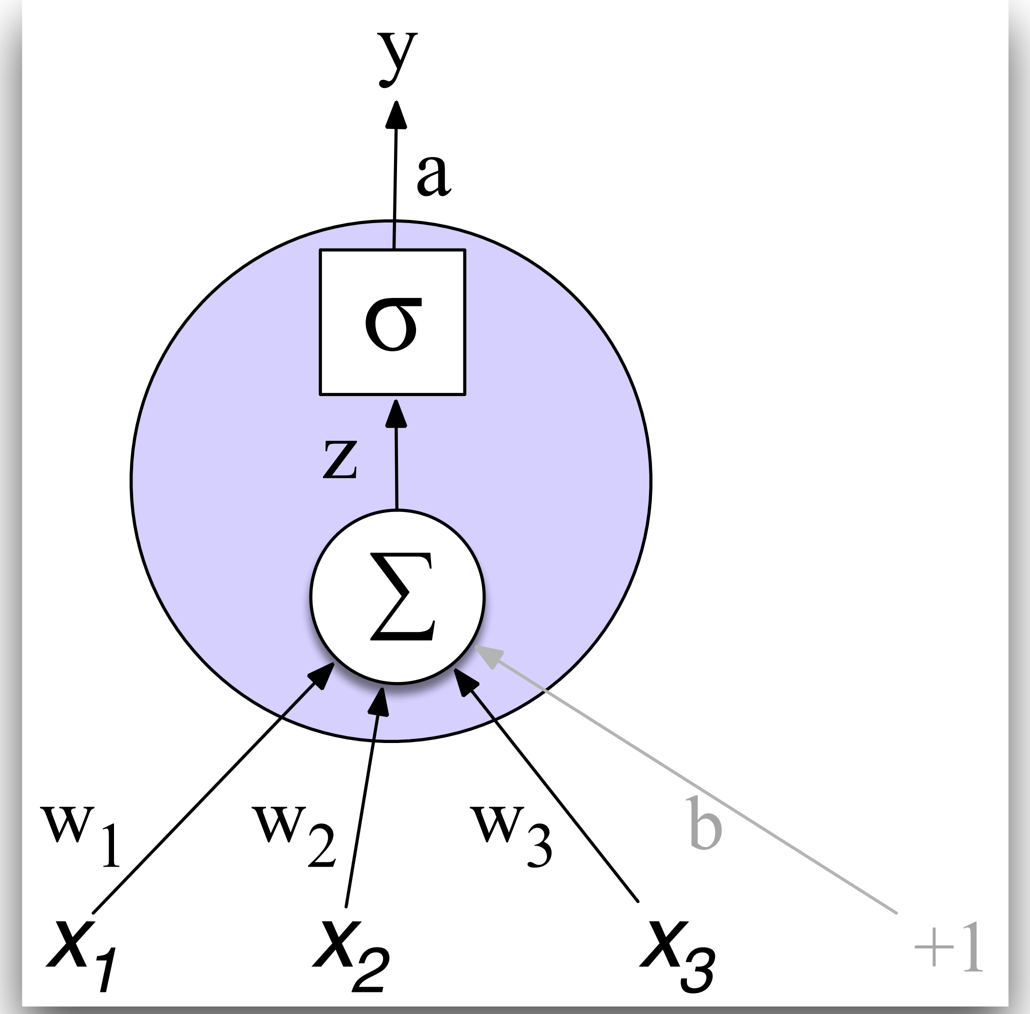
Logistic Regression → Artifical Neuron / Perceptron → Learning

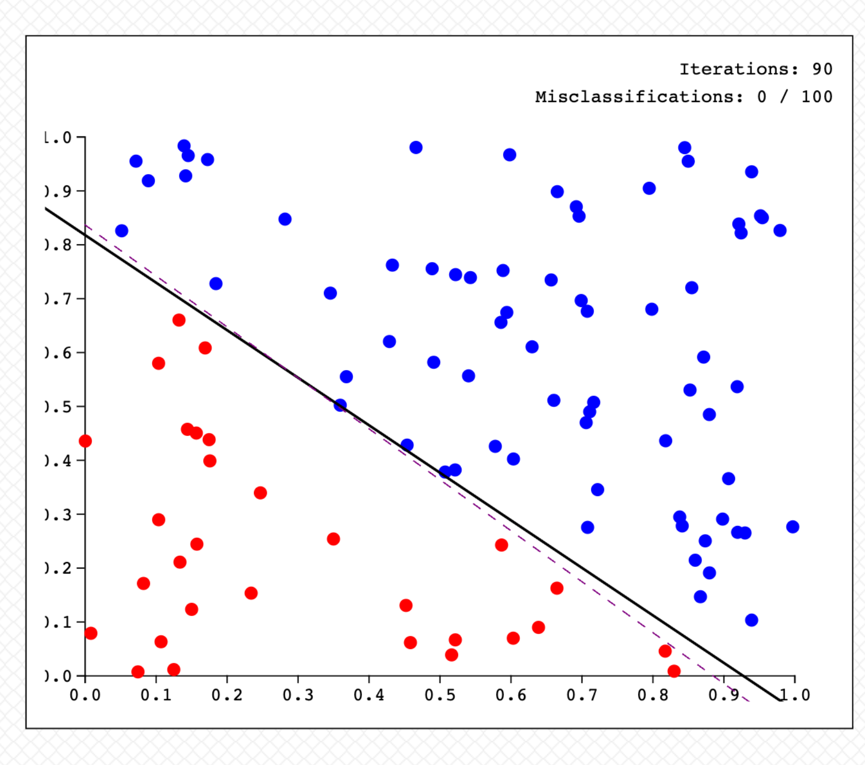
- **Logistic Regression is discriminative**: unlike Naive Bayes, it doesn't care how $\mathbf{x}$ is produced (i.e., not targeting to model the distribution of $\mathbf{x}$). It focuses on learning the mapping from $\mathbf{x}$ to y.

- **Logistic regressoin is a specific version of Perceptron (an artificial neuron)**: where the nonlinear function is Sigmoid, and there is only one output node for binary classification.

- **Perceptron** is inspired by brain neurons and was initially invented to model "information detection, storage, and recognition" (Rosenblatt 1958);

- Perceptron ***iteratively*** **learns a linear decision boundary** for binary classification — we will come back to this intuition later~





The XOR Problem and The First AI Winter

1969 - 1980~ish

How Expressive is a Perceptron?

[Exercise] Let's try representing logical gates!

- Can a Perceptron compute simple functions of input?
- Assume two inputs x_1 and x_2, use the following activation function:

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

AND

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

OR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

How Expressive is a Perceptron?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

Sample Answer

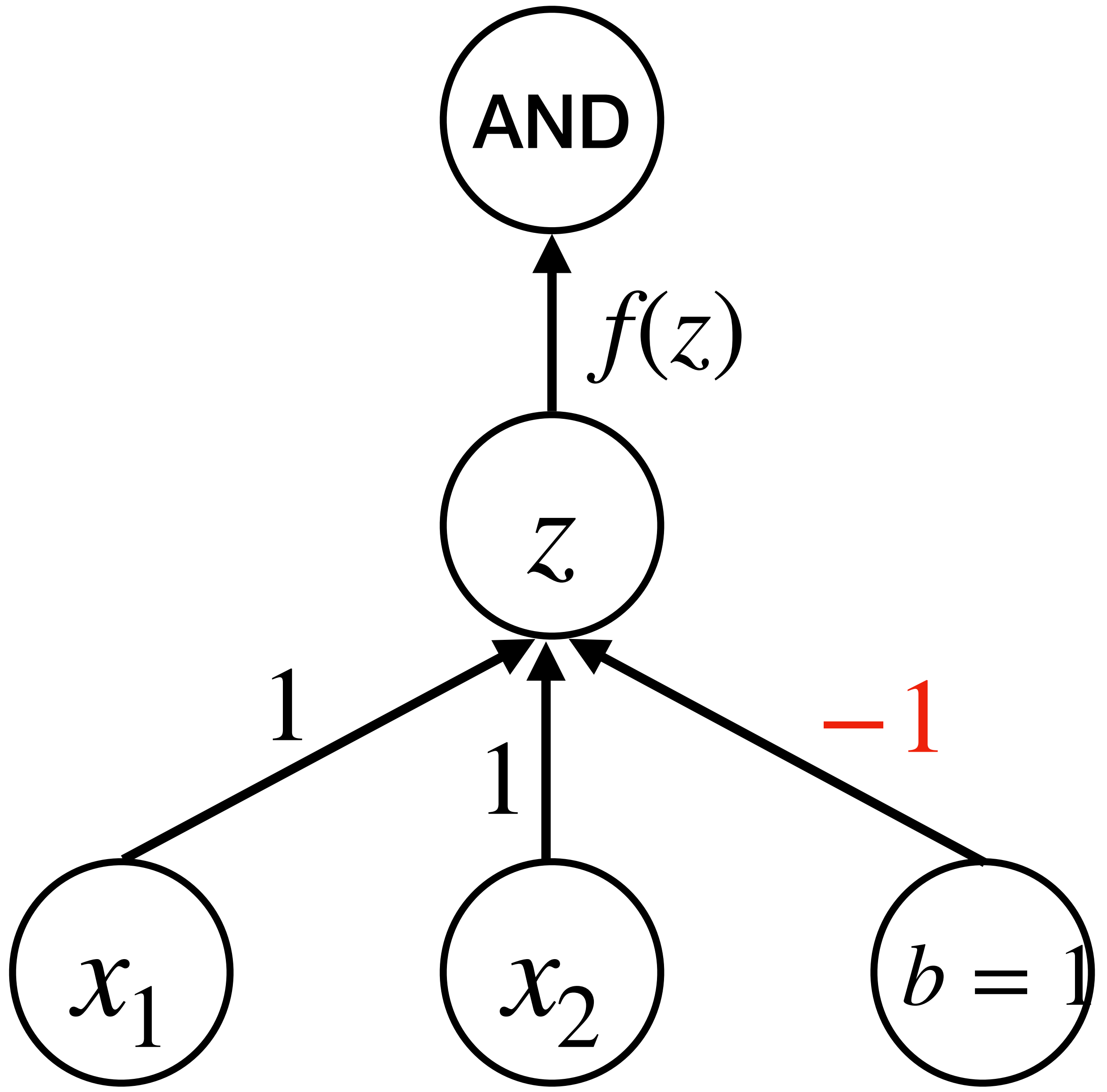
AND

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

How Expressive is a Perceptron?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

Sample Answer



AND

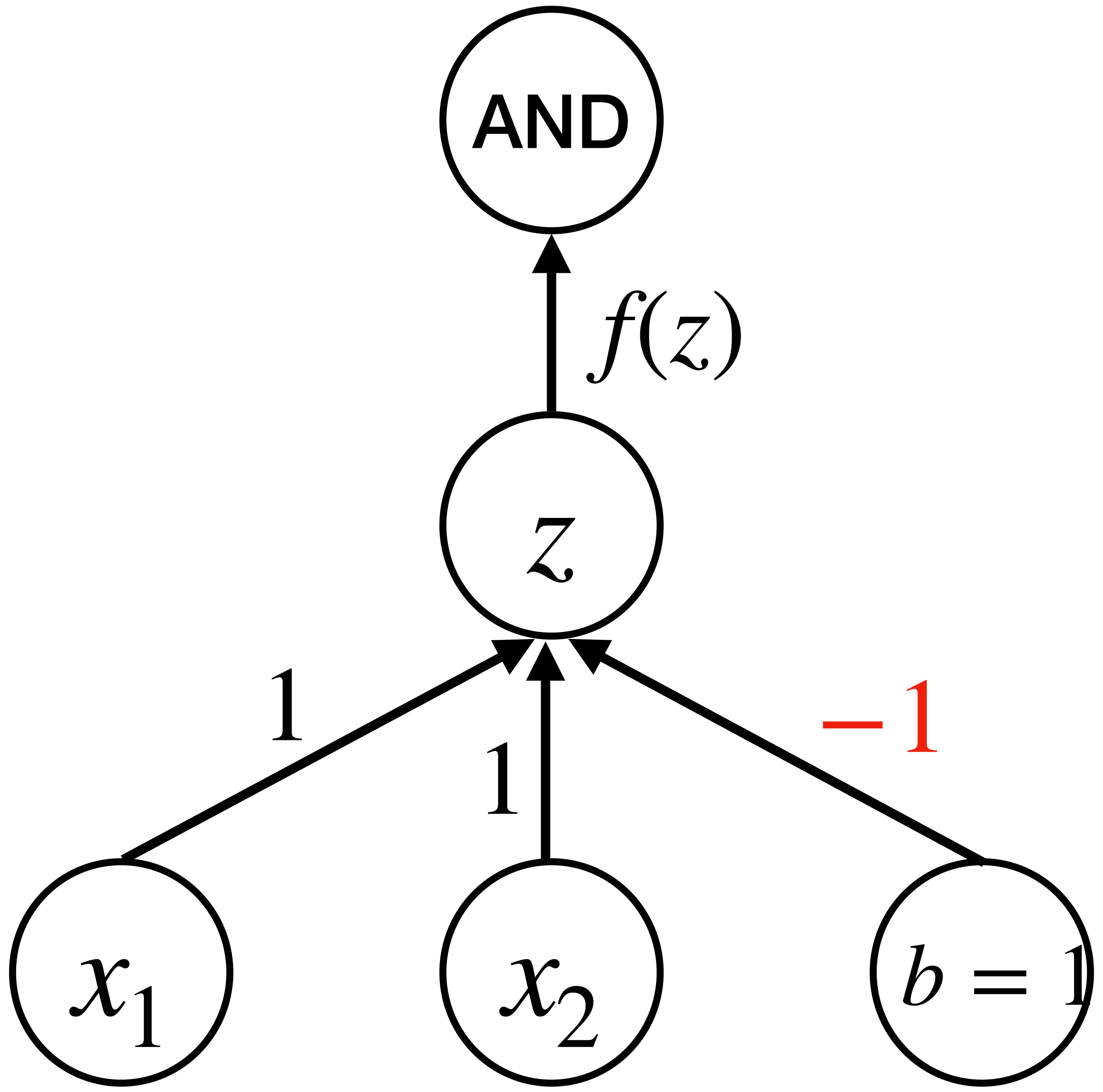
x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

How Expressive is a Perceptron?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

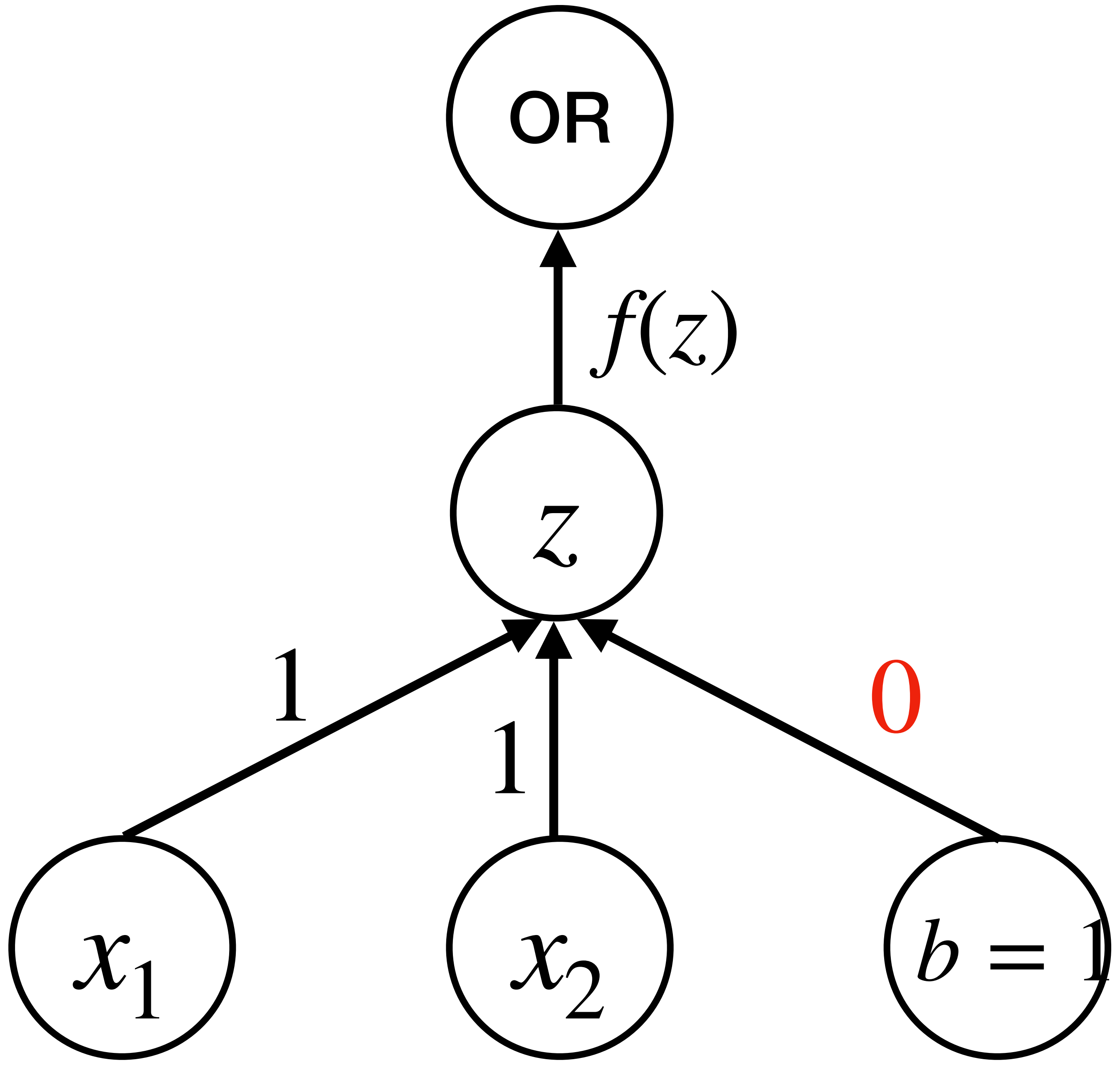
Sample Answer

AND



x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

OR



x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

Introducing another operator: XOR

[Exercise] Let's try representing the XOR gate!

- **XOR** = exclusive OR: **`x1 OR x2`, but not both**
- Formally: **`(x1 OR x2) AND (NOT (x1 AND x2))`**

Introducing another operator: XOR

[Exercise] Let's try representing the XOR gate!

- **XOR** = exclusive OR: **`x1 OR x2`, but not both**
- Formally: **`(x1 OR x2) AND (NOT (x1 AND x2))`**

OR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

Introducing another operator: XOR

[Exercise] Let's try representing the XOR gate!

- **XOR** = exclusive OR: **x1 OR x2, but not both**
- Formally: **(x1 OR x2) AND (NOT (x1 AND x2))**

OR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

Introducing another operator: XOR

[Exercise] Let's try representing the XOR gate!

- **XOR** = exclusive OR: **x1 OR x2, but not both**
- Formally: **(x1 OR x2) AND (NOT (x1 AND x2))**

OR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

Perceptrons are Linear Classifiers

Learning decision boundaries

Perceptrons are Linear Classifiers

Learning decision boundaries

- Given the current step-wise activation function, our perceptrons define **an equation of a line in a 2D space** (2D for 2 inputs):
  - $w_1x_1 + w_2x_2 + b = 0$
  - In standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b)/w_2$

Perceptrons are Linear Classifiers

Learning decision boundaries

- Given the current step-wise activation function, our perceptrons define **an equation of a line in a 2D space** (2D for 2 inputs):
  - $w_1x_1 + w_2x_2 + b = 0$
  - In standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b)/w_2$
- In the 2D space we are familiar with, this defines a **decision boundary**.
  - Output = 0 if the input point is on one side of the line;
  - Output = 1 if the input point is on the other side of the line.

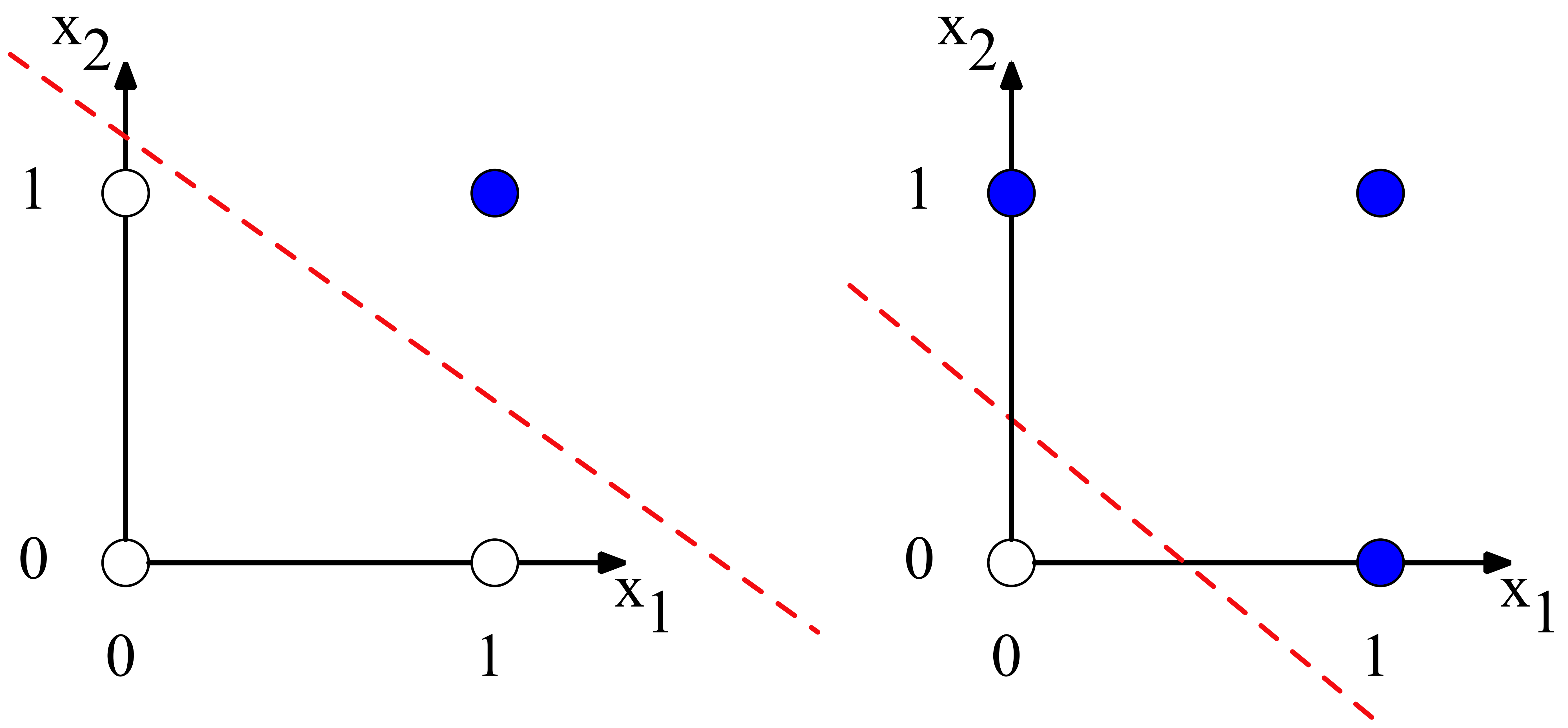
Perceptrons are Linear Classifiers

Learning decision boundaries

- Given the current step-wise activation function, our perceptrons define **an equation of a line in a 2D space** (2D for 2 inputs):
    - $w_1x_1 + w_2x_2 + b = 0$
    - In standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b)/w_2$
- In the 2D space we are familiar with, this defines a **decision boundary**.
    - Output = 0 if the input point is on one side of the line;
    - Output = 1 if the input point is on the other side of the line.
- A good perceptron should be able to find a decision boundary that perfectly separates the 0 points from the 1 points — **linearly separability**!

Perceptron’s Decision Boundaries

Visualizing the decision boundaries for AND and OR

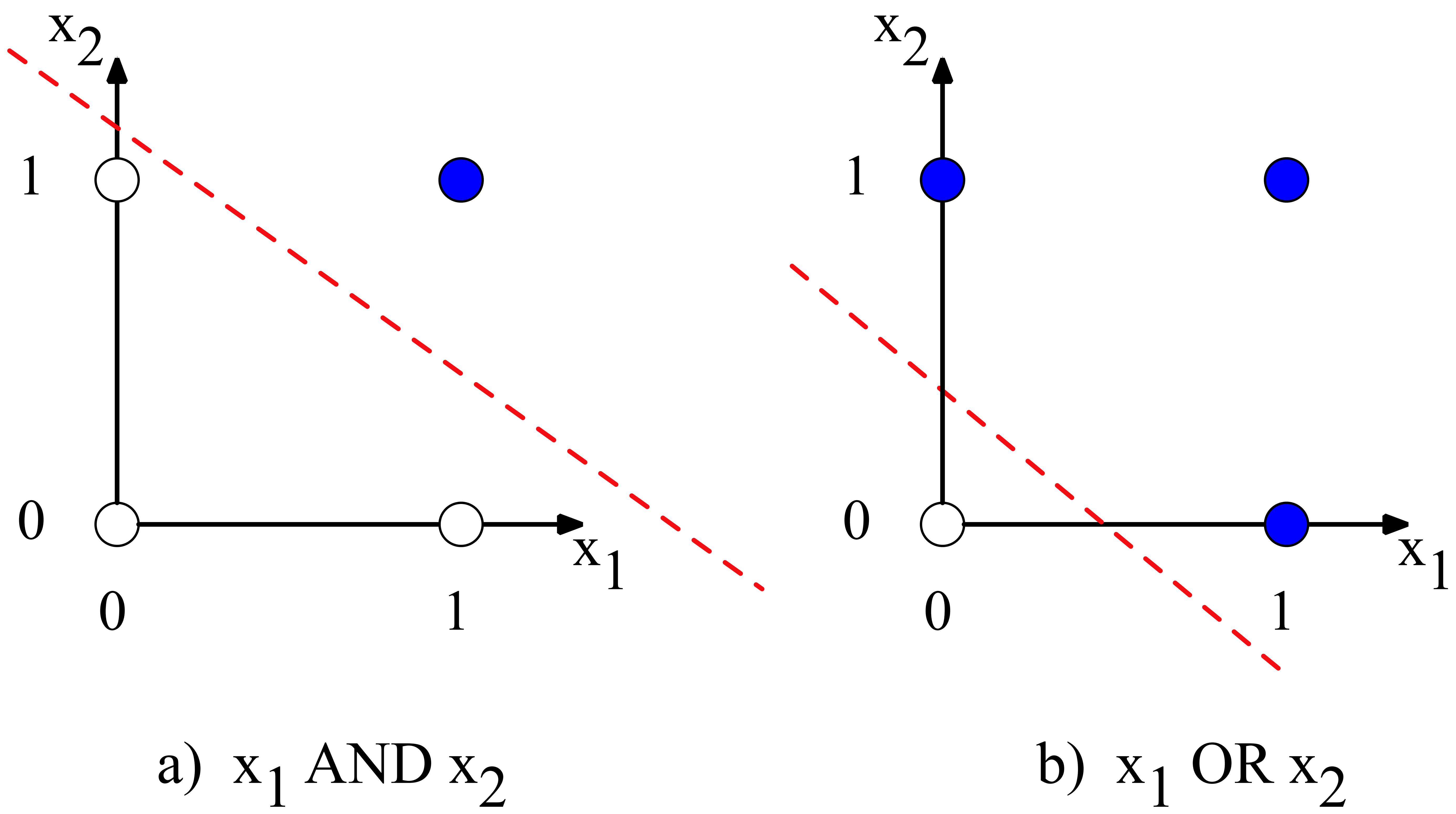


a) x_1 AND x_2

b) x_1 OR x_2

Perceptron's Decision Boundaries

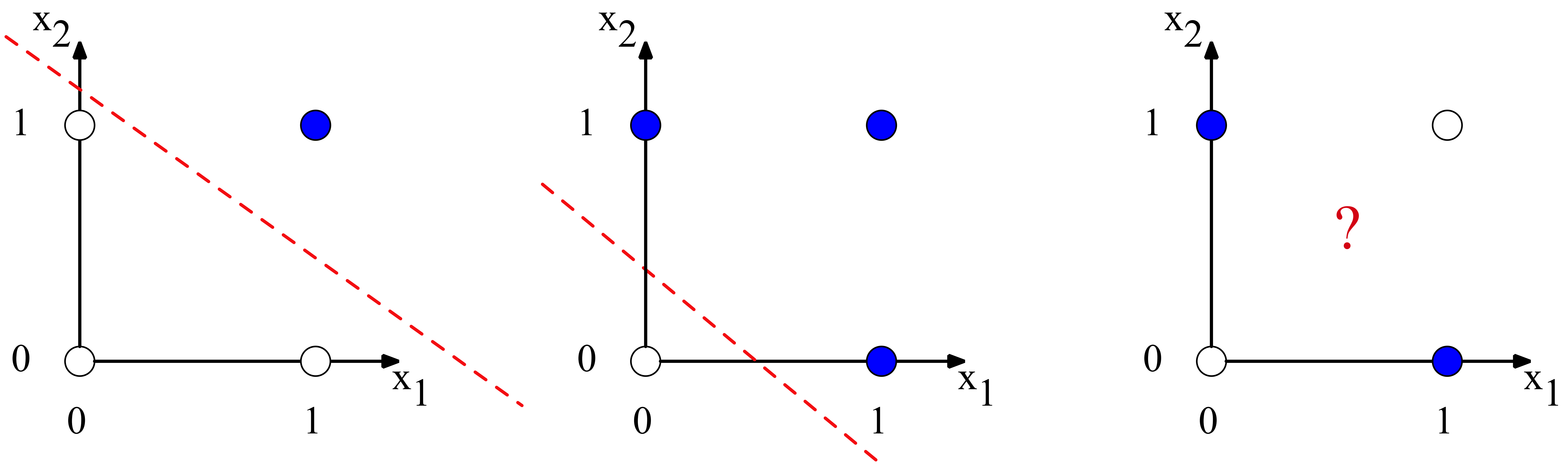
Visualizing the decision boundaries for AND and OR



AND and OR are linearly separable!

Perceptron's Decision Boundaries

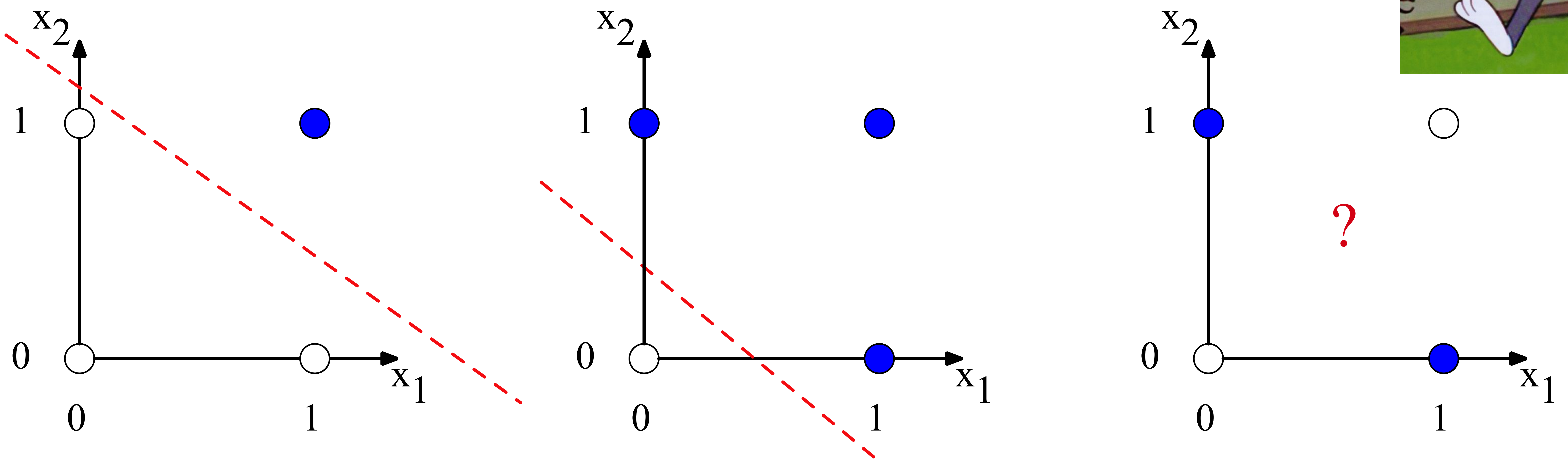
Visualizing the decision boundaries for XOR



a) x_1 AND x_2 b) x_1 OR x_2 c) x_1 XOR x_2

Perceptron's Decision Boundaries

Visualizing the decision boundaries for XOR



a) x_1 AND x_2 b) x_1 OR x_2 c) x_1 XOR x_2

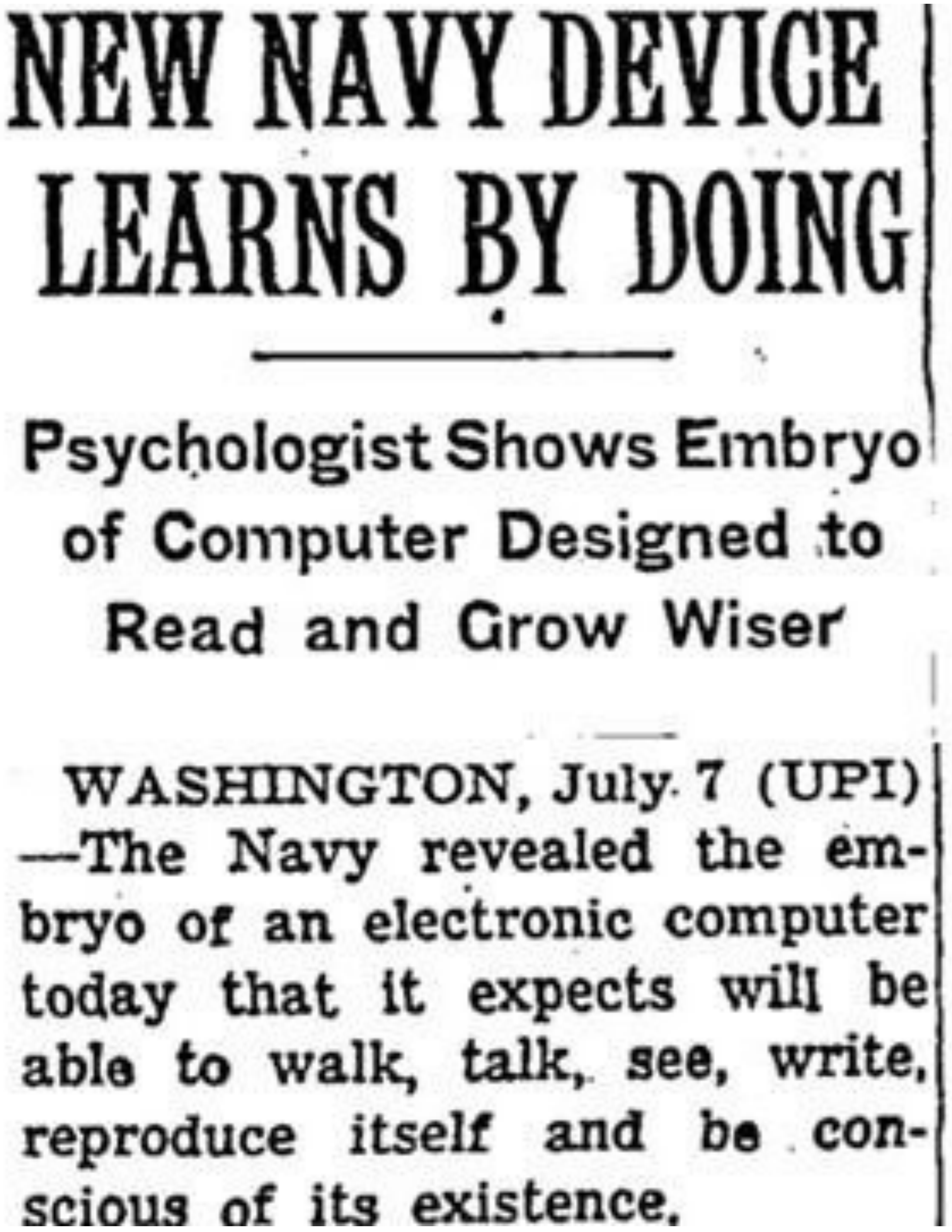
`XOR` is NOT linearly separable!

The AI Winter

10 years from prosperity to fall

- When Perceptron was introduced, it was quite exciting to have a machine that could learn from experience.

- This resulted in some vintage AI hype!
  - ✳ Optimism ran wild: newspapers proclaimed that the perceptron would one day "walk, talk, see, and be conscious of its own existence."
  - ✳ Lots of research fundings.



NEW NAVY DEVICE LEARNS BY DOING

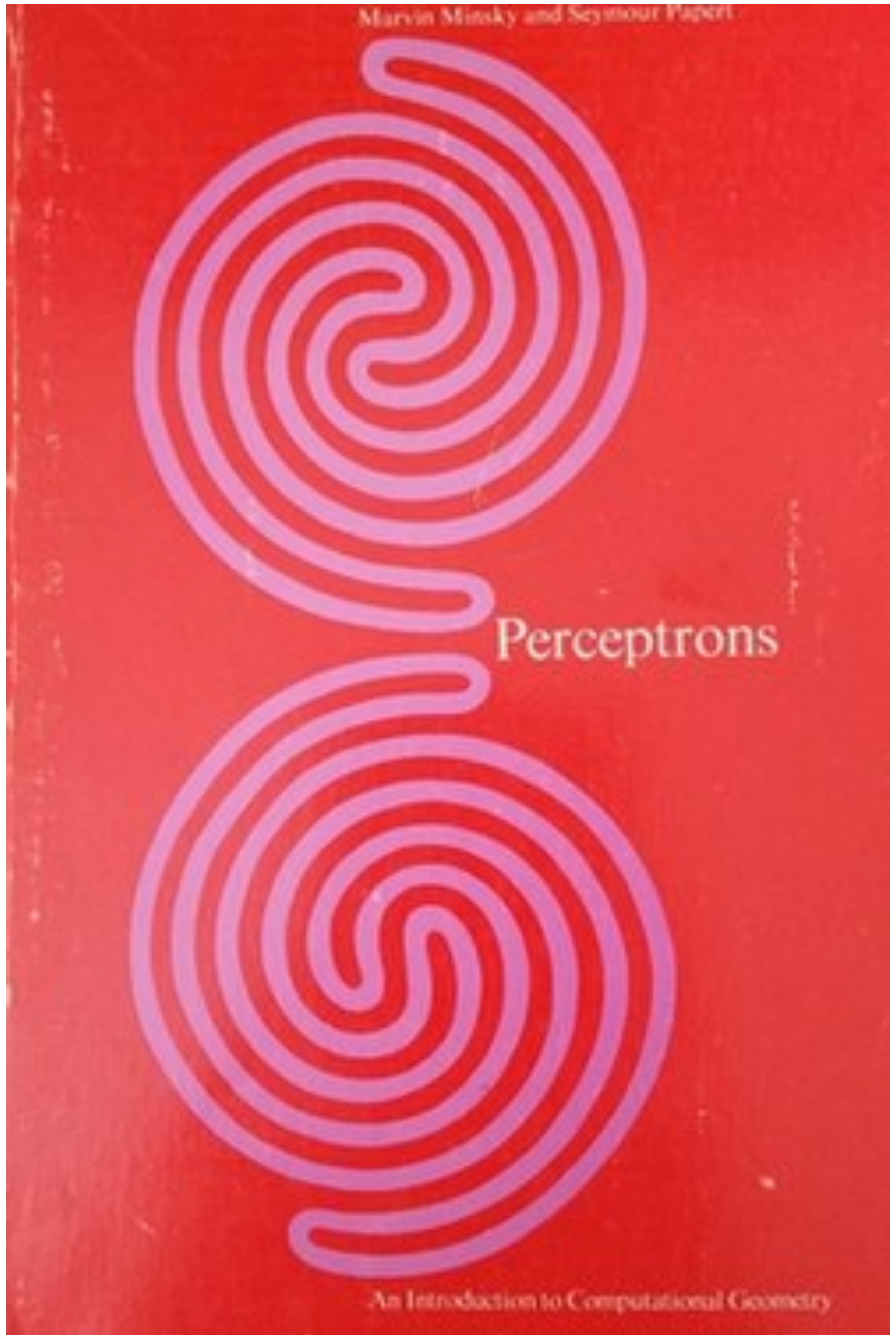
Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

NYT, via StefanoErmon on Twitter
https://x.com/StefanoErmon/status/936396977218056192?lang=en

The AI Winter

10 years from prosperity to fall

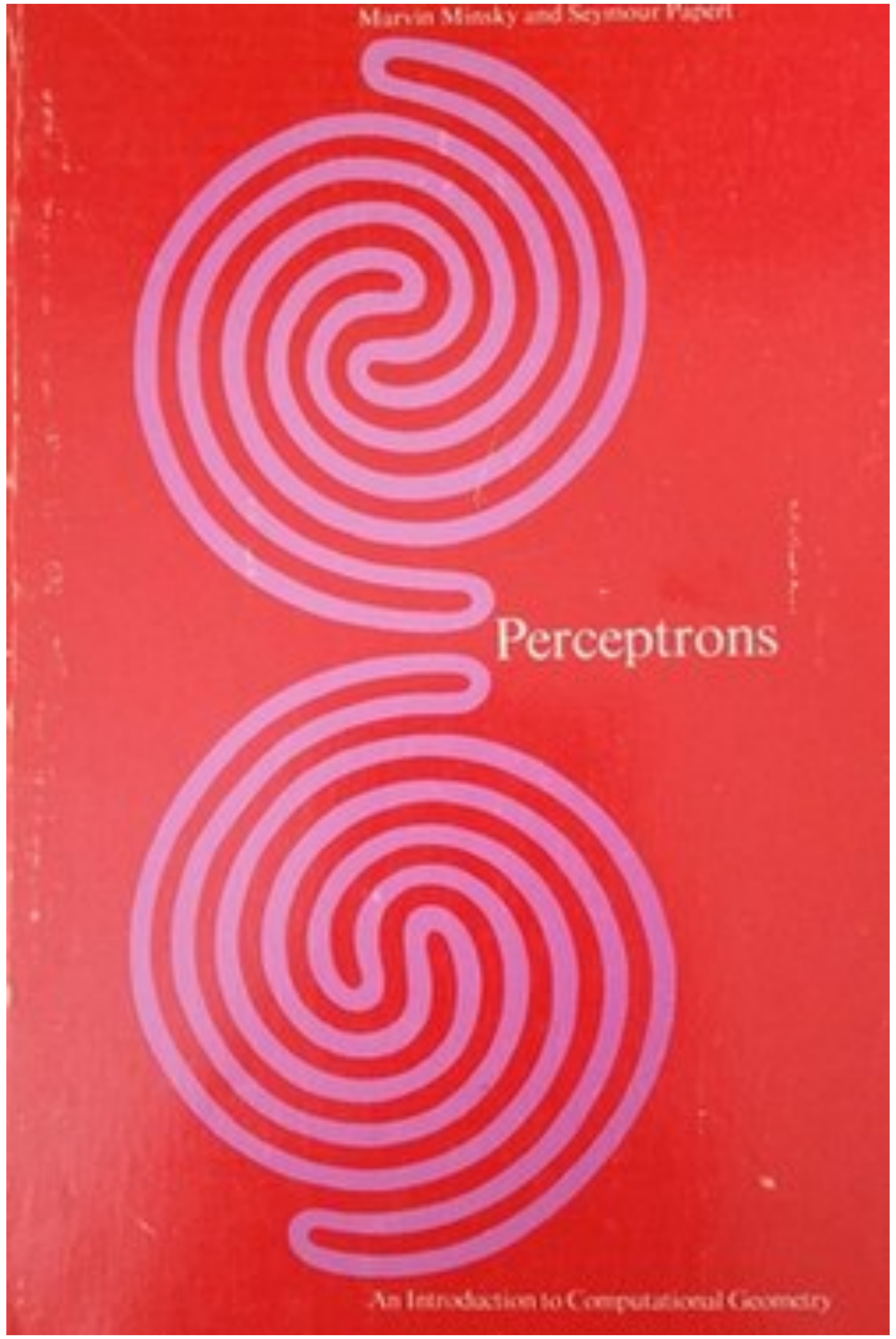


Minsky & Papert 1969

The AI Winter

10 years from prosperity to fall

- Minsky & Papert published a book named *Perceptrons* in 1969, a rigorous analysis showing that single-layer perceptrons could only learn linearly separable functions.
    - Highlighting the XOR problem.

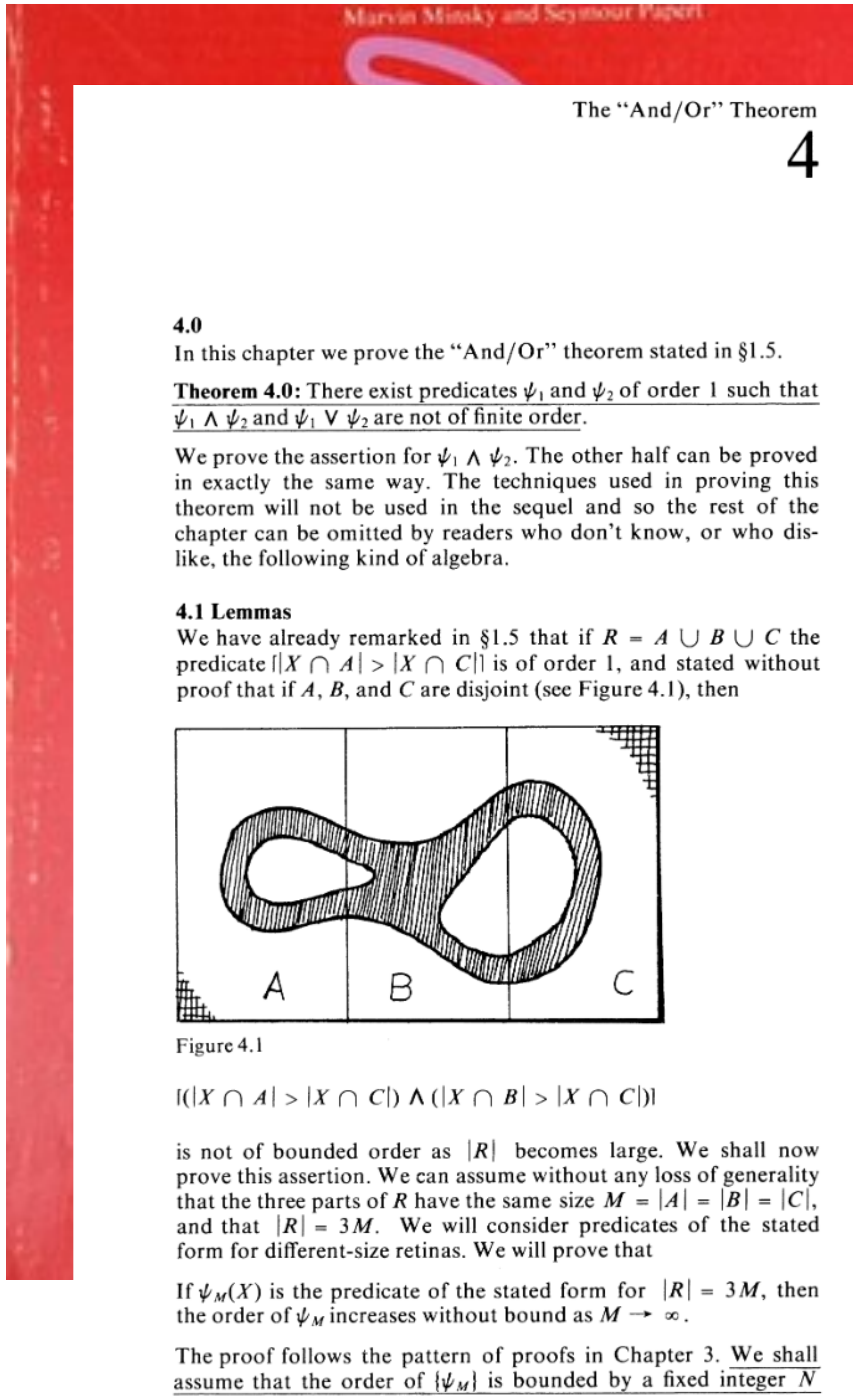


Minsky & Papert 1969

The AI Winter

10 years from prosperity to fall

- Minsky & Papert published a book named *Perceptrons* in 1969, a rigorous analysis showing that single-layer perceptrons could only learn linearly separable functions.
    - Highlighting the XOR problem.



Marvin Minsky and Seymour Papert

The "And/Or" Theorem

4

4.0

In this chapter we prove the "And/Or" theorem stated in §1.5.

Theorem 4.0: There exist predicates ψ_1 and ψ_2 of order 1 such that $\psi_1 \wedge \psi_2$ and $\psi_1 \vee \psi_2$ are not of finite order.

We prove the assertion for $\psi_1 \wedge \psi_2$. The other half can be proved in exactly the same way. The techniques used in proving this theorem will not be used in the sequel and so the rest of the chapter can be omitted by readers who don't know, or who dislike, the following kind of algebra.

4.1 Lemmas

We have already remarked in §1.5 that if $R = A \cup B \cup C$ the predicate $\lceil |X \cap A| > |X \cap C| \rceil$ is of order 1, and stated without proof that if A, B, and C are disjoint (see Figure 4.1), then

A B C

Figure 4.1

$$\lceil (|X \cap A| > |X \cap C|) \wedge (|X \cap B| > |X \cap C|) \rceil$$

is not of bounded order as $|R|$ becomes large. We shall now prove this assertion. We can assume without any loss of generality that the three parts of R have the same size $M = |A| = |B| = |C|$, and that $|R| = 3M$. We will consider predicates of the stated form for different-size retinas. We will prove that

If $\psi_M(X)$ is the predicate of the stated form for $|R| = 3M$, then the order of ψ_M increases without bound as $M \to \infty$.

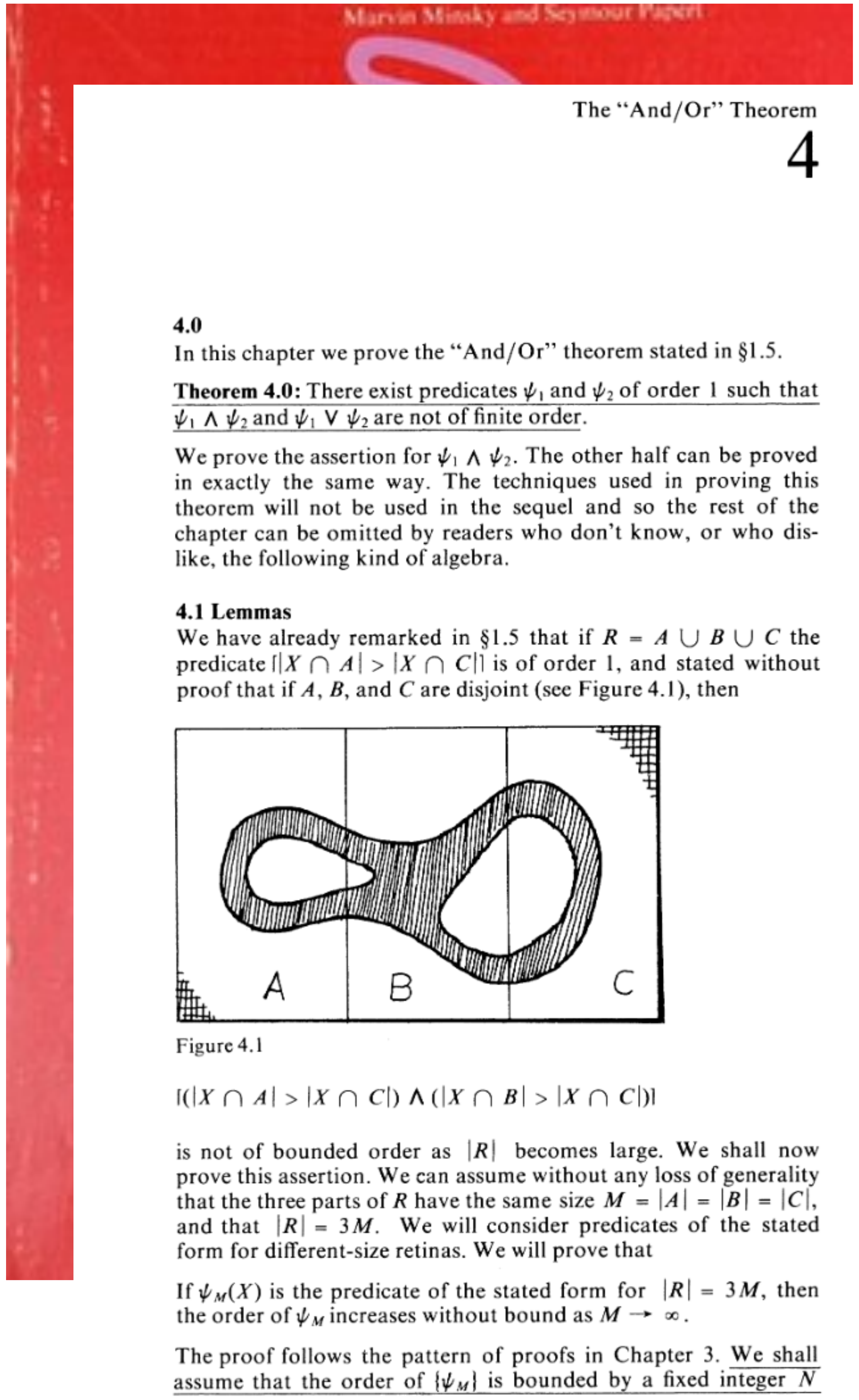
The proof follows the pattern of proofs in Chapter 3. We shall assume that the order of $\{\psi_M\}$ is bounded by a fixed integer N

Minsky & Papert 1969

The AI Winter

10 years from prosperity to fall

- Minsky & Papert published a book named *Perceptrons* in 1969, a rigorous analysis showing that single-layer perceptrons could only learn linearly separable functions.
  - Highlighting the XOR problem.
- This resulted in the first AI winter:
  - ✳ Research funding agencies concluded that neural networks had hit a theoretical dead end.
  - ✳ AI research shifted towards symbolic logic, expert systems, rule-based reasoning (n-gram models were born in this stage!)


Marvin Minsky and Seymour Papert

The "And/Or" Theorem

4

4.0

In this chapter we prove the "And/Or" theorem stated in §1.5.

Theorem 4.0: There exist predicates ψ_1 and ψ_2 of order 1 such that $\psi_1 \wedge \psi_2$ and $\psi_1 \vee \psi_2$ are not of finite order.

We prove the assertion for $\psi_1 \wedge \psi_2$. The other half can be proved in exactly the same way. The techniques used in proving this theorem will not be used in the sequel and so the rest of the chapter can be omitted by readers who don't know, or who dislike, the following kind of algebra.

4.1 Lemmas

We have already remarked in §1.5 that if $R = A \cup B \cup C$ the predicate $\lceil |X \cap A| > |X \cap C| \rceil$ is of order 1, and stated without proof that if A, B, and C are disjoint (see Figure 4.1), then

Figure 4.1

$$\lceil (|X \cap A| > |X \cap C|) \wedge (|X \cap B| > |X \cap C|) \rceil$$

is not of bounded order as $|R|$ becomes large. We shall now prove this assertion. We can assume without any loss of generality that the three parts of R have the same size $M = |A| = |B| = |C|$, and that $|R| = 3M$. We will consider predicates of the stated form for different-size retinas. We will prove that

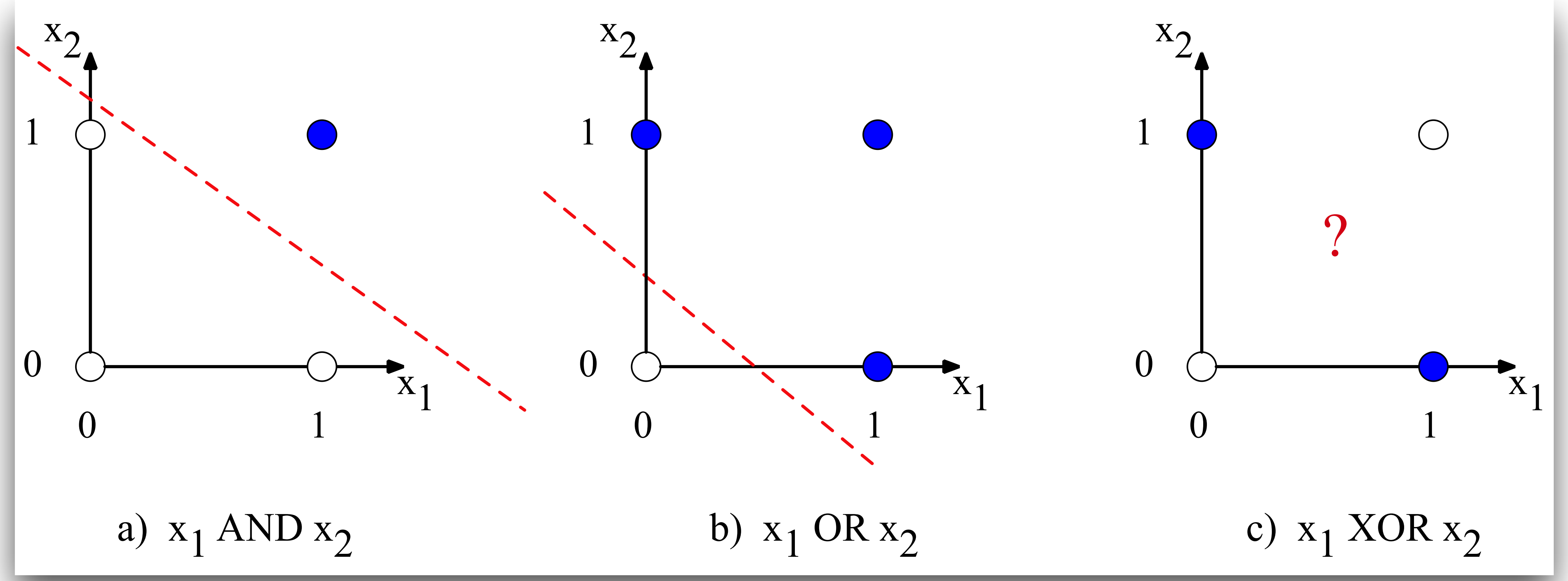
If $\psi_M(X)$ is the predicate of the stated form for $|R| = 3M$, then the order of ψ_M increases without bound as $M \rightarrow \infty$.

The proof follows the pattern of proofs in Chapter 3. We shall assume that the order of $\{\psi_M\}$ is bounded by a fixed integer N

Minsky & Papert 1969

Interium Summary 2

- **The XOR Problem** demonstrates a crucial limit of a single-neuron perceptron.
- Perceptron can only learn linearly separable patterns.
- This triggers **the first AI winter** after the initial AI hype for 10 years.



The Revival of Neural Networks

1980s

One Solution to the XOR Problem

What if we use more than one layer?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

One Solution to the XOR Problem

What if we use more than one layer?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

- Here is one way to think about it. By definition:

One Solution to the XOR Problem

What if we use more than one layer?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

- Here is one way to think about it. By definition:
  - **`XOR`** = exclusive **`OR`**: **`x1 OR x2`, but not both**

One Solution to the XOR Problem

What if we use more than one layer?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

- Here is one way to think about it. By definition:
  - **XOR** = exclusive **OR**: **x1 OR x2, but not both**
  - Formally: **(x1 OR x2) AND (NOT (x1 AND x2))**

One Solution to the XOR Problem

What if we use more than one layer?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

- Here is one way to think about it. By definition:
  - **XOR** = exclusive **OR**: **x1 OR x2, but not both**
  - Formally: **(x1 OR x2) AND (NOT (x1 AND x2))**
- Is it possible to do it compositionally?

One Solution to the XOR Problem

What if we use more than one layer?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

- Here is one way to think about it. By definition:
  - **XOR** = exclusive **OR**: **x1 OR x2, but not both**
  - Formally: **(x1 OR x2) AND (NOT (x1 AND x2))**
- Is it possible to do it compositionally?
  - Let **h1** = **(x1 OR x2)**, **h2** = **(x1 AND x2)**

One Solution to the XOR Problem

What if we use more than one layer?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

- Here is one way to think about it. By definition:
  - **XOR** = exclusive **OR**: **x1 OR x2, but not both**
  - Formally: **(x1 OR x2) AND (NOT (x1 AND x2))**
- Is it possible to do it compositionally?
  - Let **h1** = **(x1 OR x2), h2 = (x1 AND x2)**
  - Then, **(x1 XOR x2) == h1 AND (NOT h2)**

One Solution to the XOR Problem

What if we use more than one layer?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

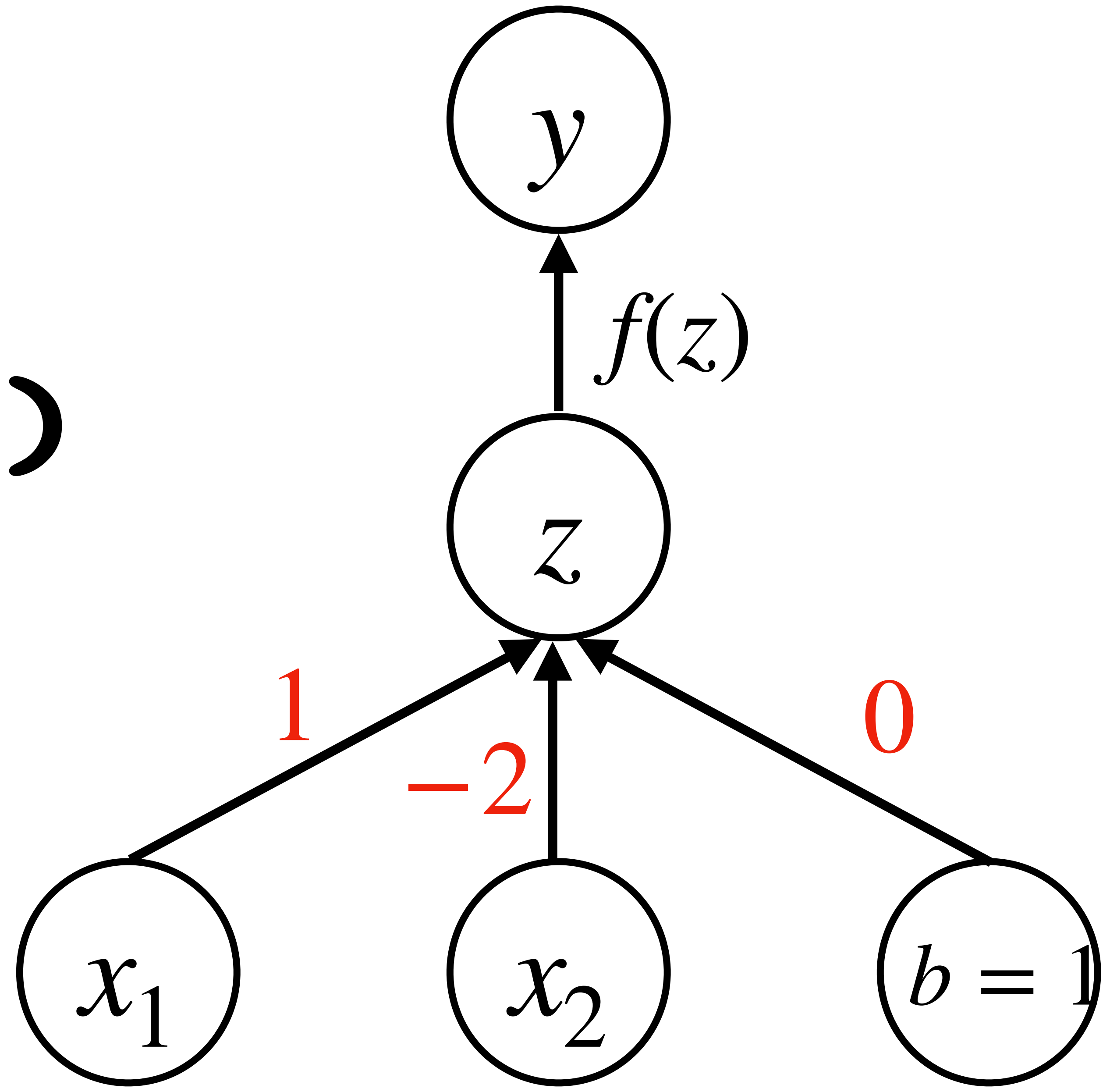
- Here is one way to think about it. By definition:
  - **XOR** = exclusive **OR**: **x1 OR x2, but not both**
  - Formally: **(x1 OR x2) AND (NOT (x1 AND x2))**
- Is it possible to do it compositionally?
  - Let **h1** = **(x1 OR x2)**, **h2** = **(x1 AND x2)**
  - Then, **(x1 XOR x2) == h1 AND (NOT h2)**
  - We do know how to express **AND**, and here is a simple tweak for **NOT**:

One Solution to the XOR Problem

What if we use more than one layer?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \le 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

x1 AND (NOT x2)



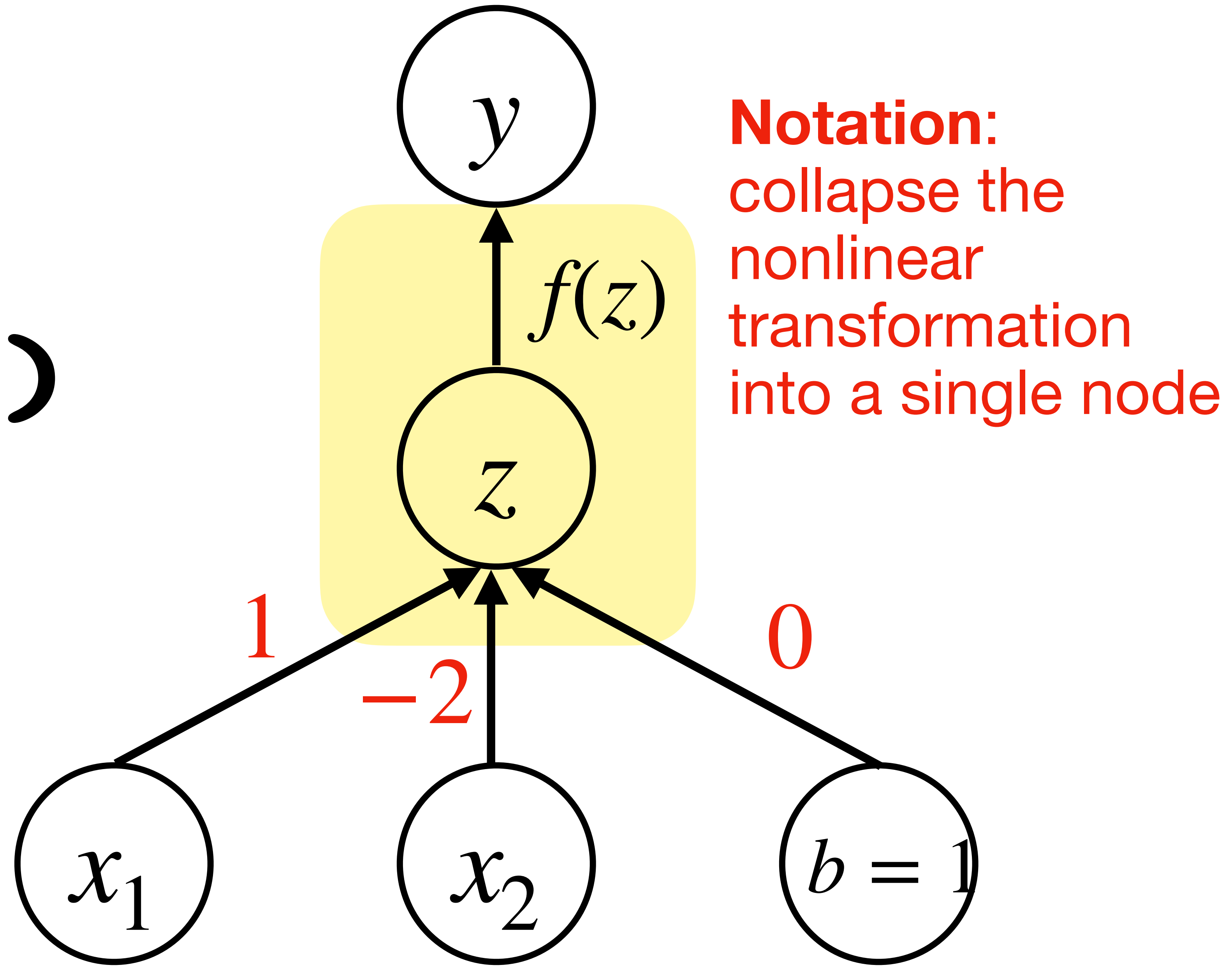
- Here is one way to think about it. By definition:
  - **XOR** = exclusive **OR**: **x1 OR x2, but not both**
  - Formally: **(x1 OR x2) AND (NOT (x1 AND x2))**
- Is it possible to do it compositionally?
  - Let **h1** = **(x1 OR x2)**, **h2** = **(x1 AND x2)**
  - Then, **(x1 XOR x2) == h1 AND (NOT h2)**
  - We do know how to express **AND**, and here is a simple tweak for **NOT**:

One Solution to the XOR Problem

What if we use more than one layer?

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \le 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

x1 AND (NOT x2)



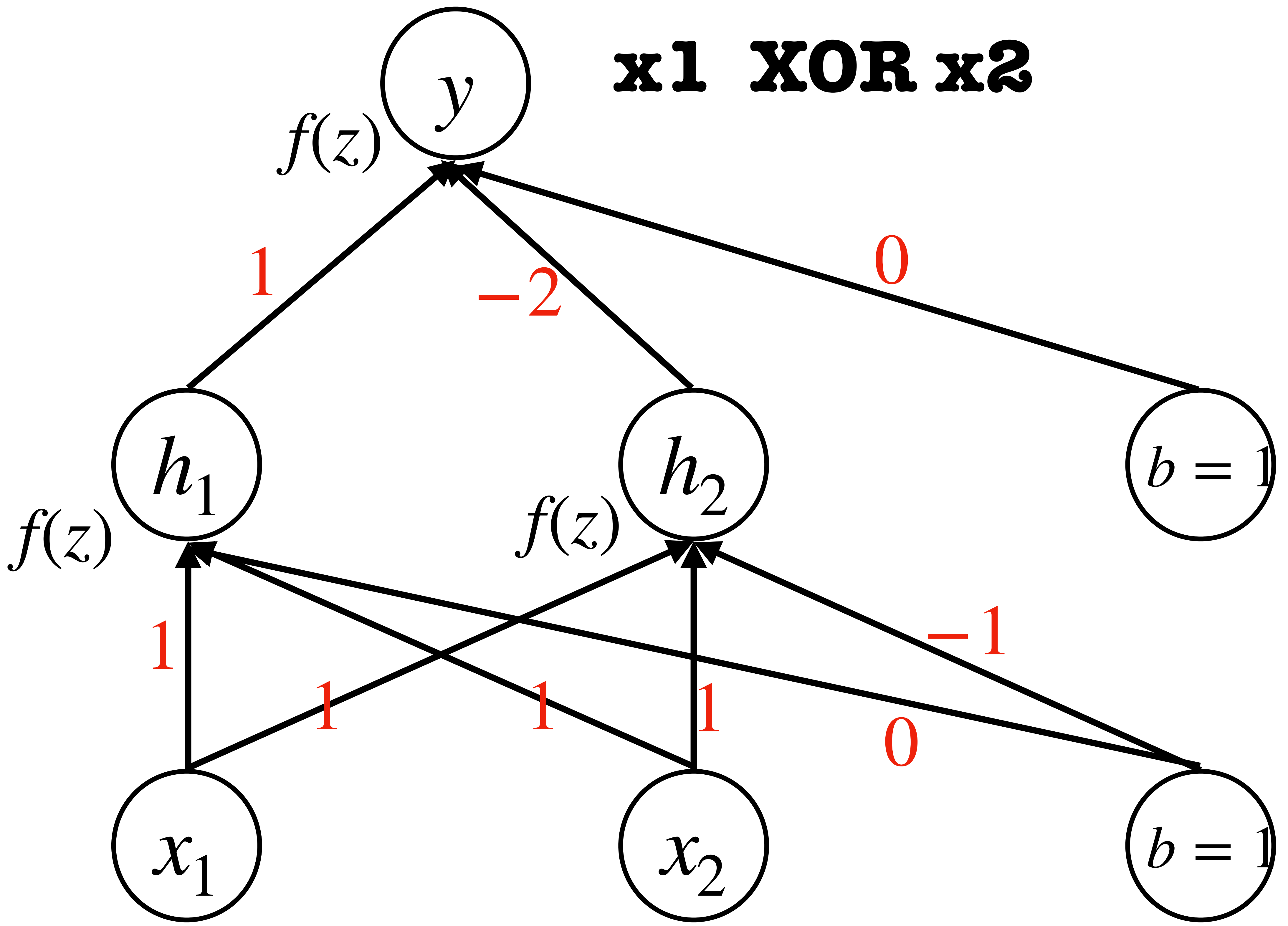
- Here is one way to think about it. By definition:
  - **XOR** = exclusive **OR**: **x1 OR x2, but not both**
  - Formally: **(x1 OR x2) AND (NOT (x1 AND x2))**
- Is it possible to do it compositionally?
  - Let **h1** = **(x1 OR x2)**, **h2** = **(x1 AND x2)**
  - Then, **(x1 XOR x2) == h1 AND (NOT h2)**
  - We do know how to express **AND**, and here is a simple tweak for **NOT**:

$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \le 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

One Solution to the XOR Problem

What if we use more than one layers?

- Here is one implementation:

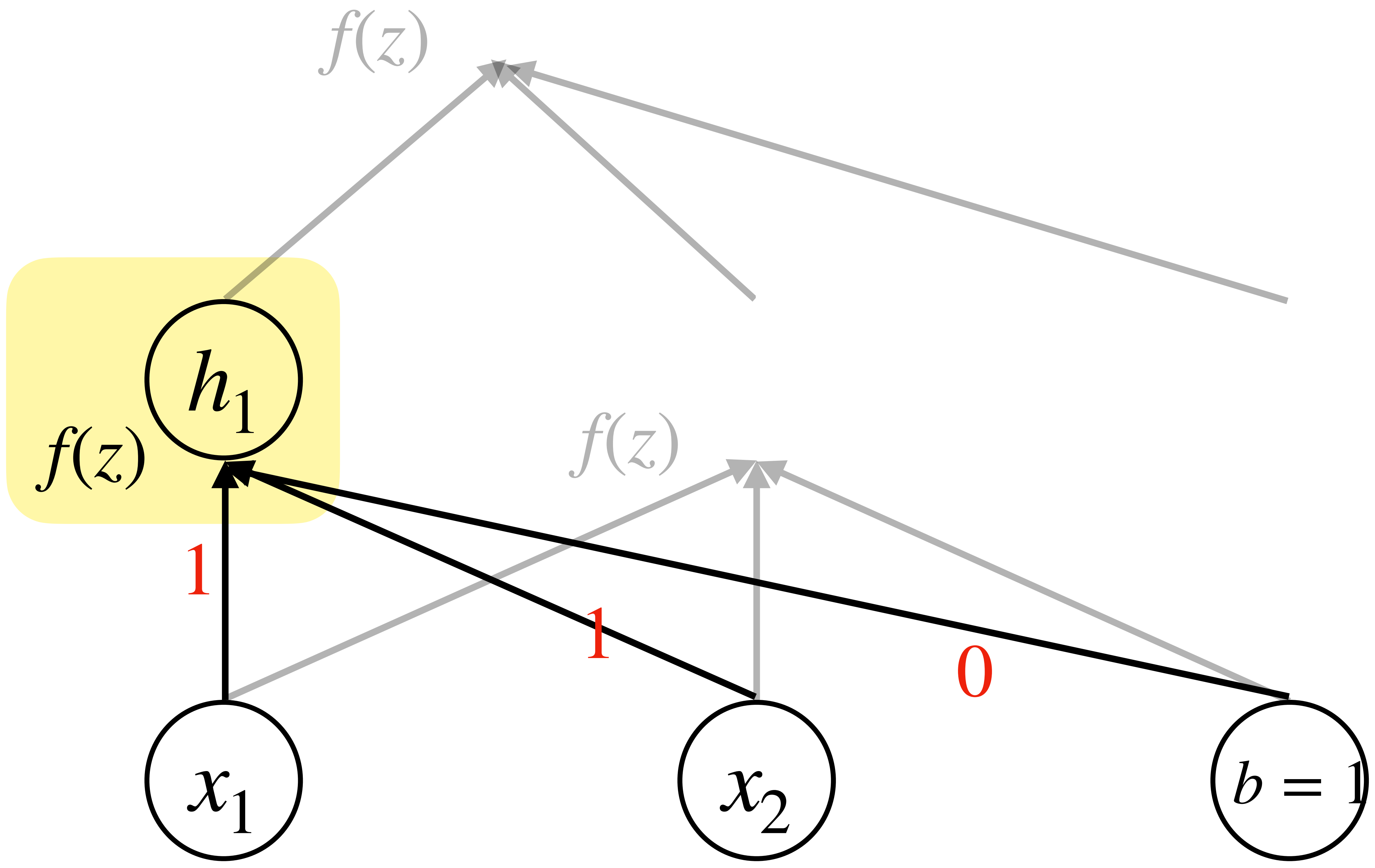


$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

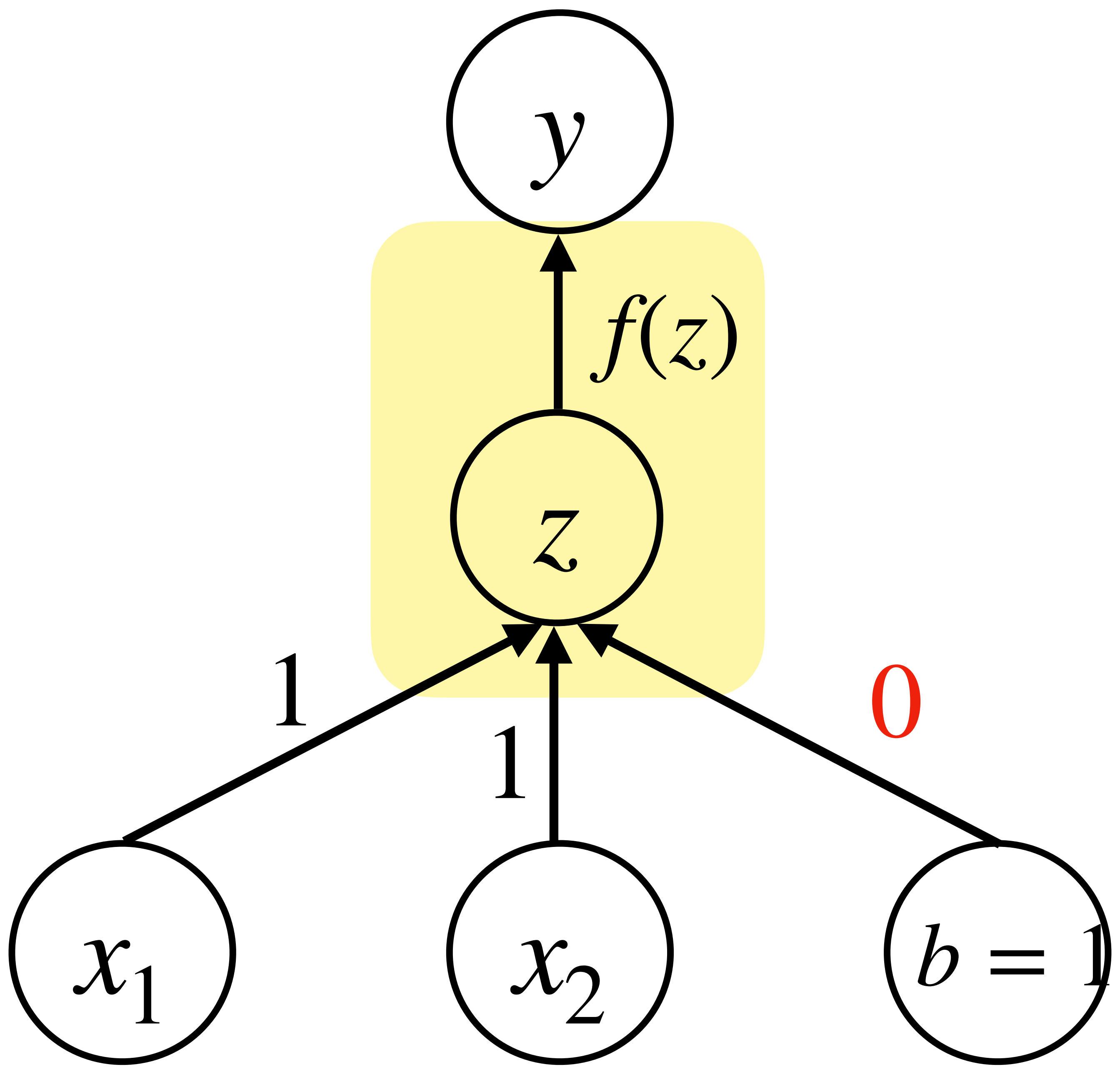
One Solution to the XOR Problem

What if we use more than one layers?

- Component 1: the **OR** gate



x1 OR x2

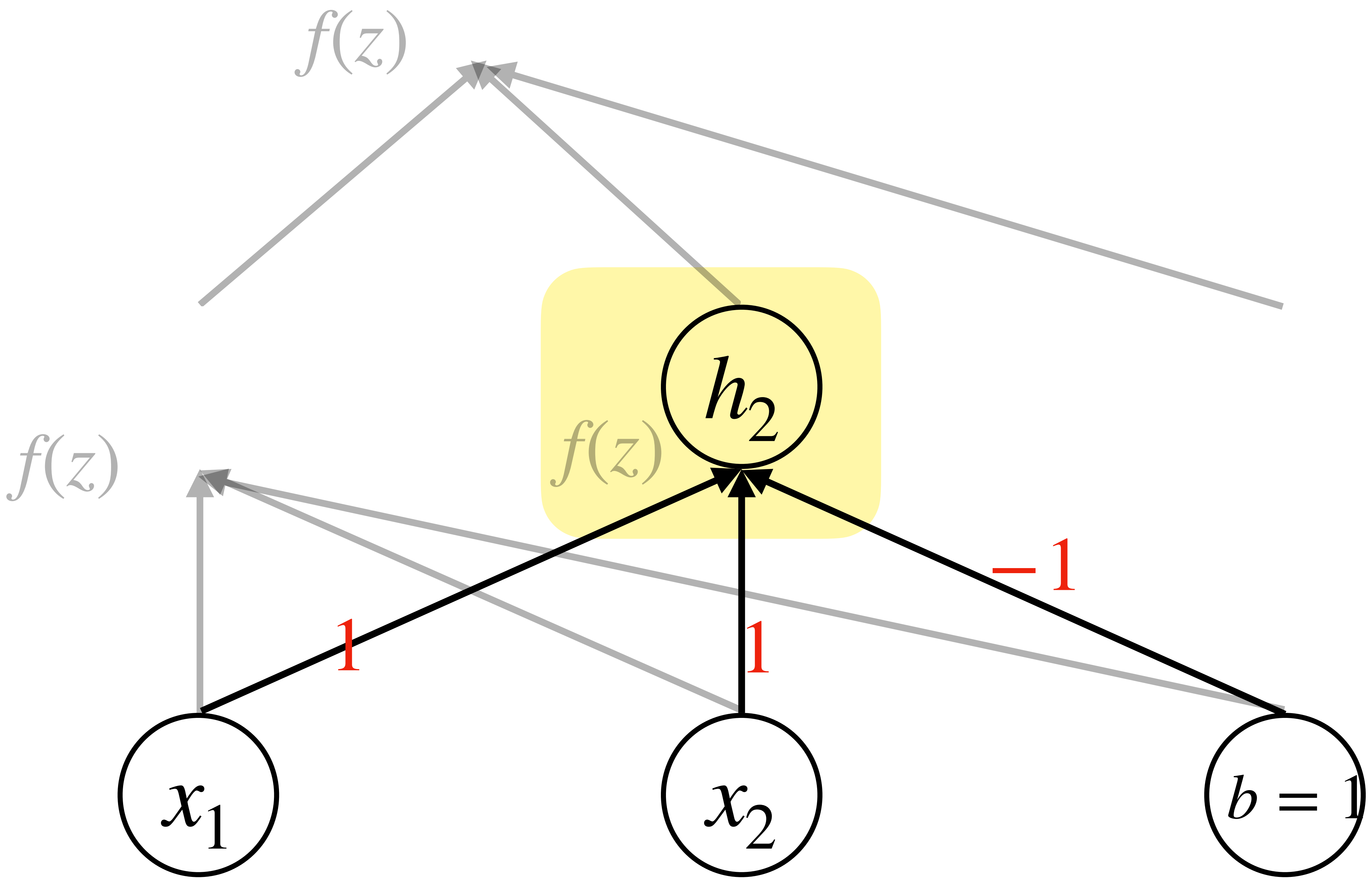


$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \le 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

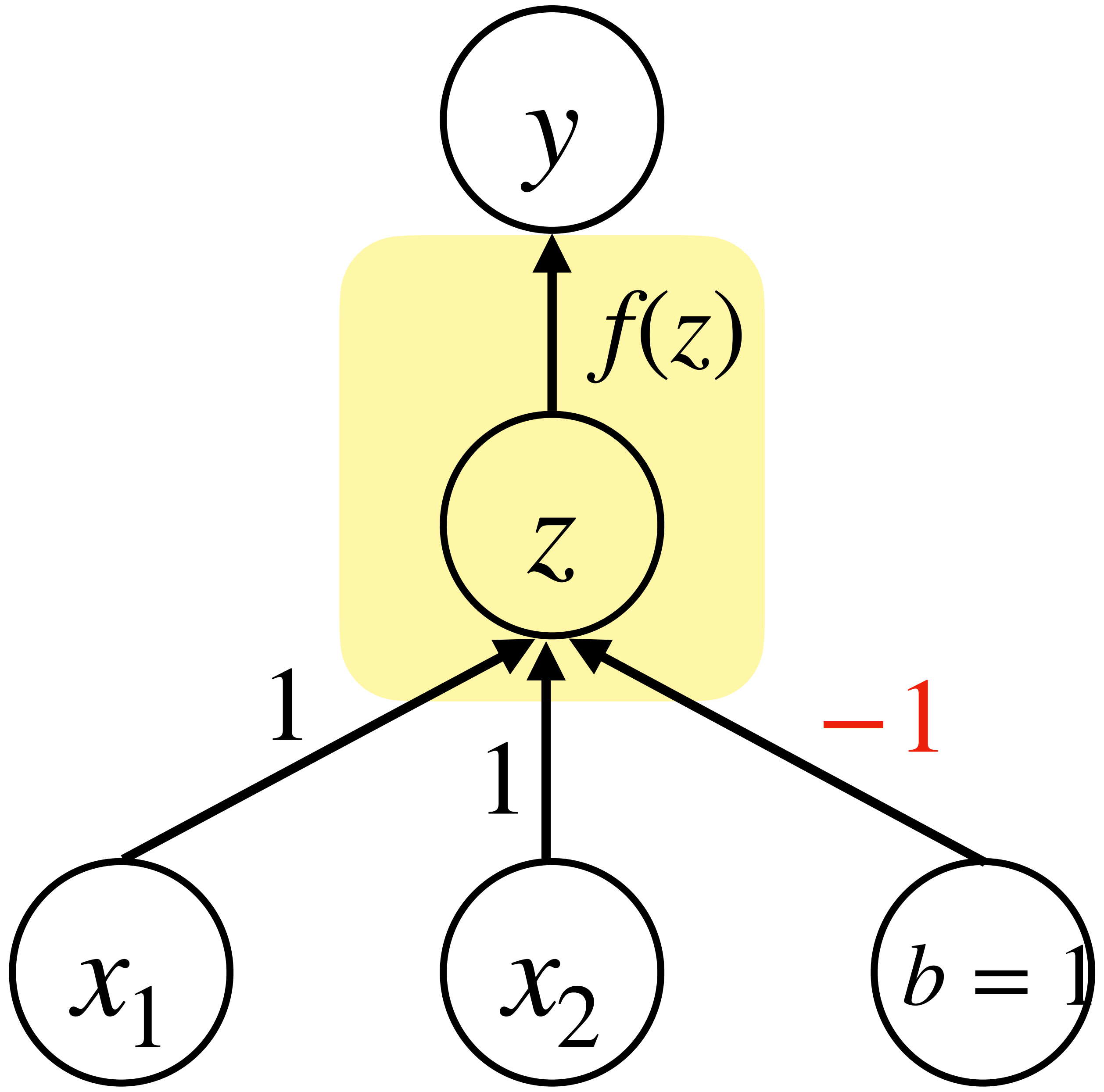
One Solution to the XOR Problem

What if we use more than one layers?

- Component 2: the **AND** gate



💡 **x1 AND x2**

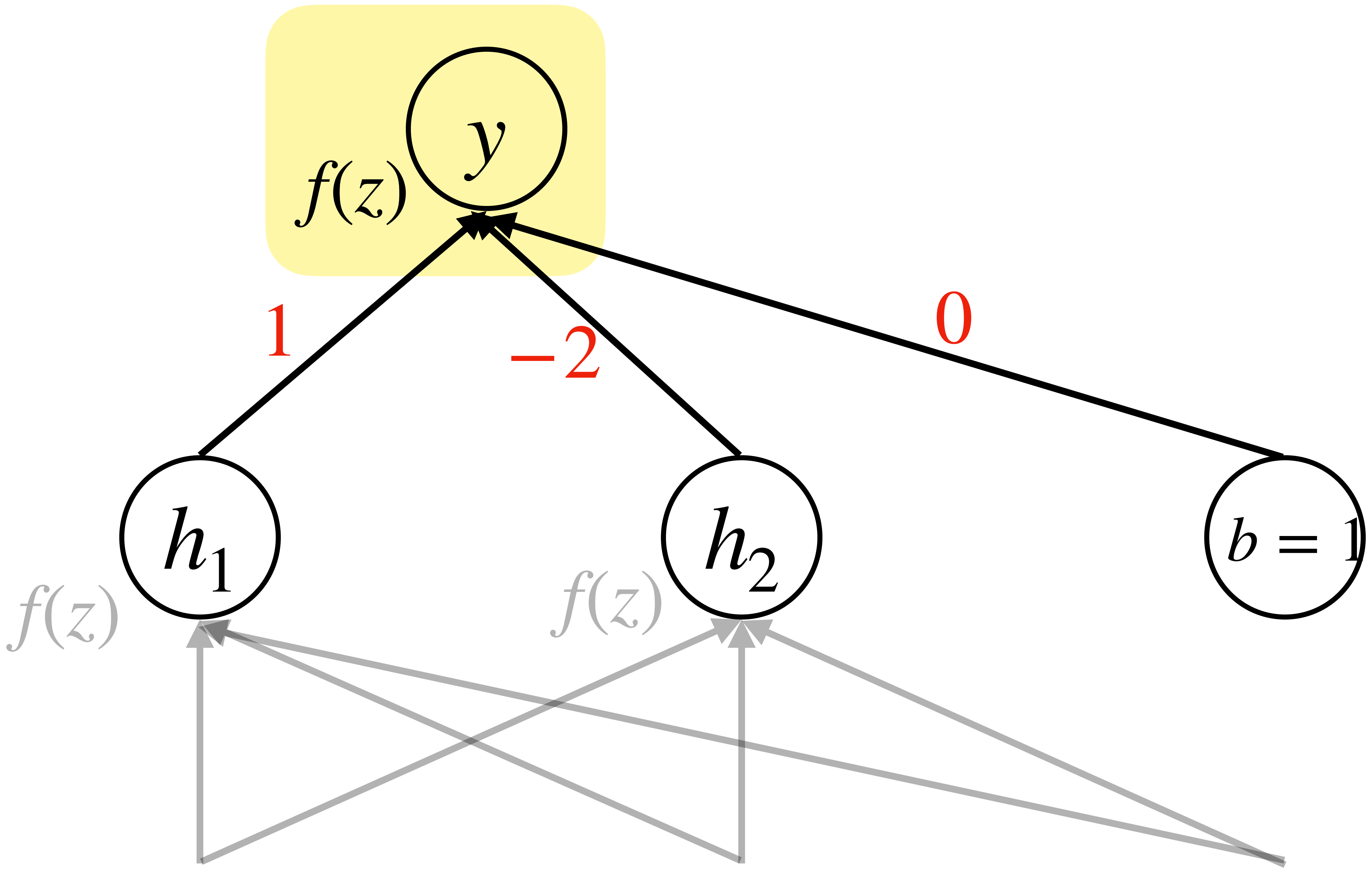


$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

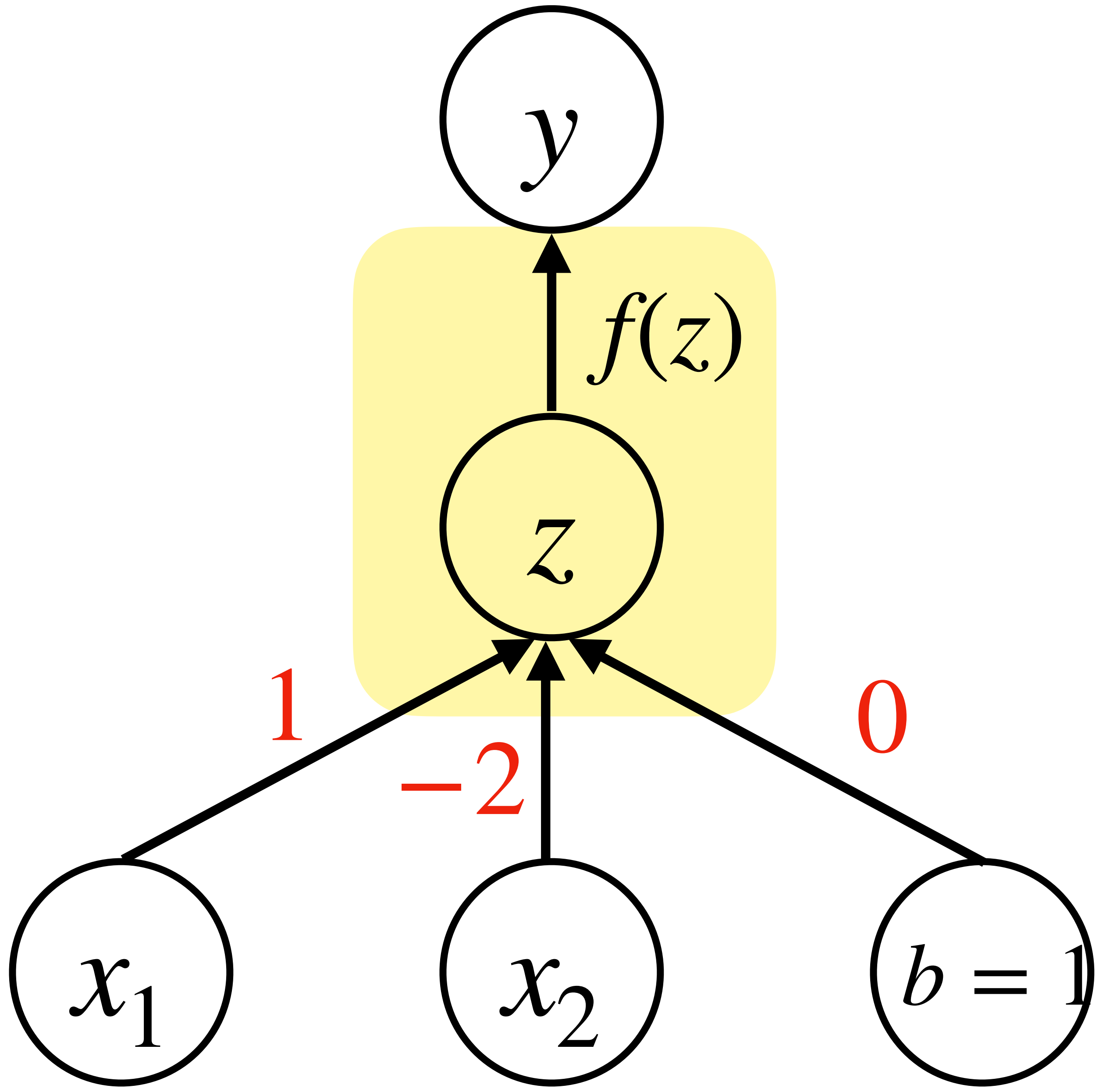
One Solution to the XOR Problem

What if we use more than one layers?

- Component 3: the **AND** gate with **NOT**

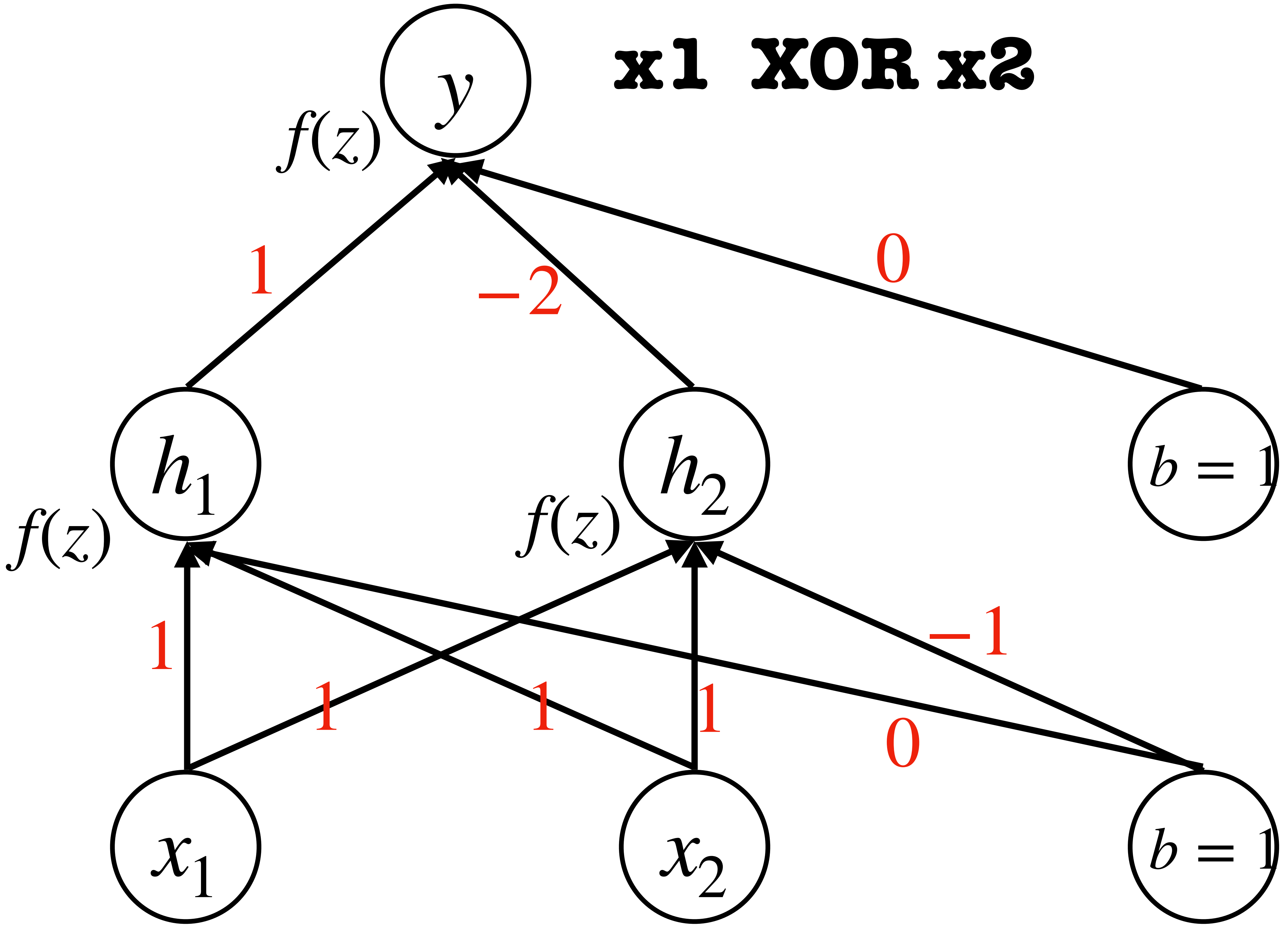


x1 AND (NOT x2)



One Solution to the XOR Problem

Let's verify!



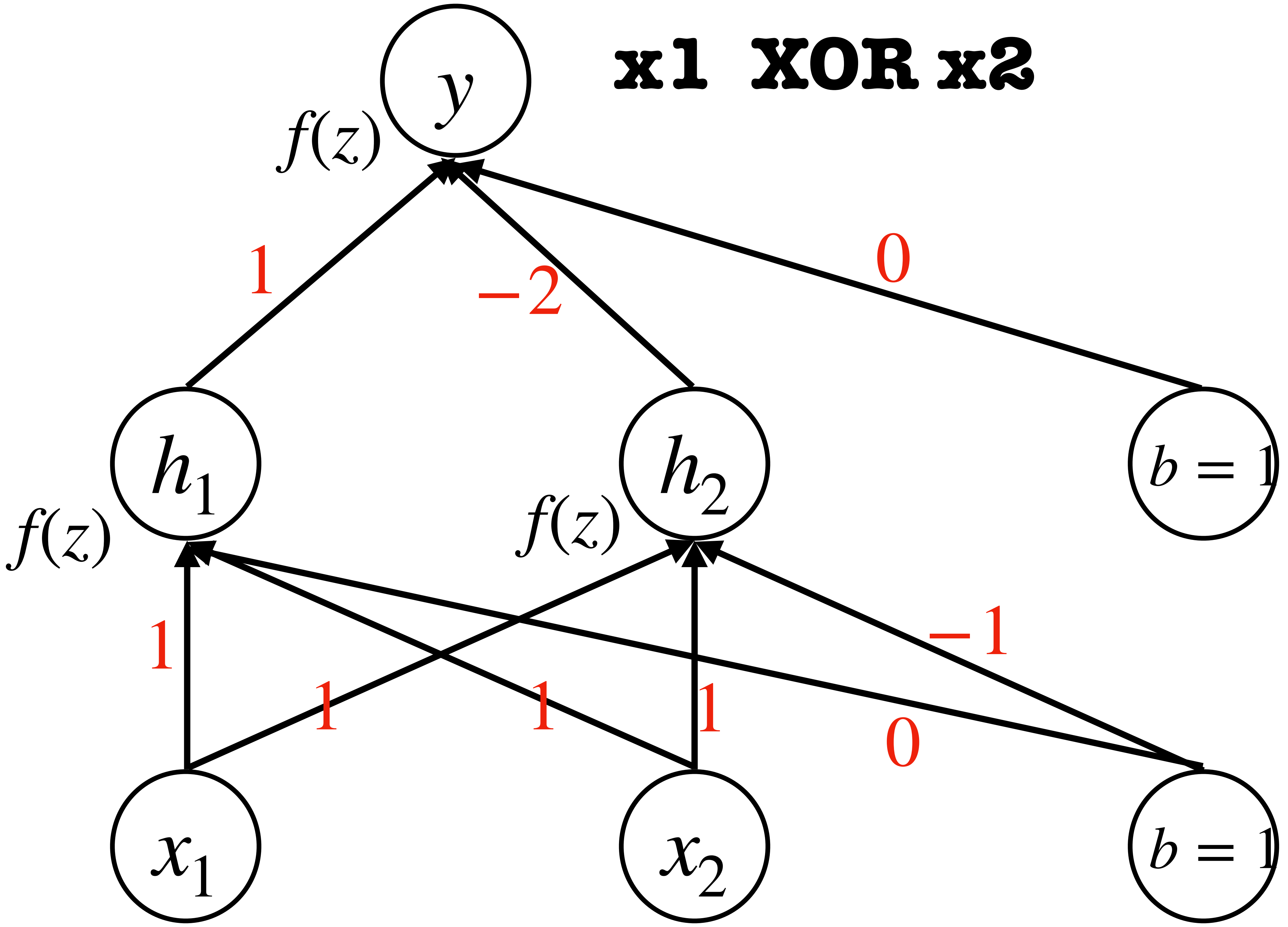
after transformation through f

x1	x2	$h1_0$	$h2_0$	h1	h2	y_0	y
0	0	0	-1	0	0	0	0
0	1	1	0	1	0	1	1
1	0	1	0	1	0	1	1
1	1	2	1	1	1	-1	0

Before transformation through f

One Solution to the XOR Problem

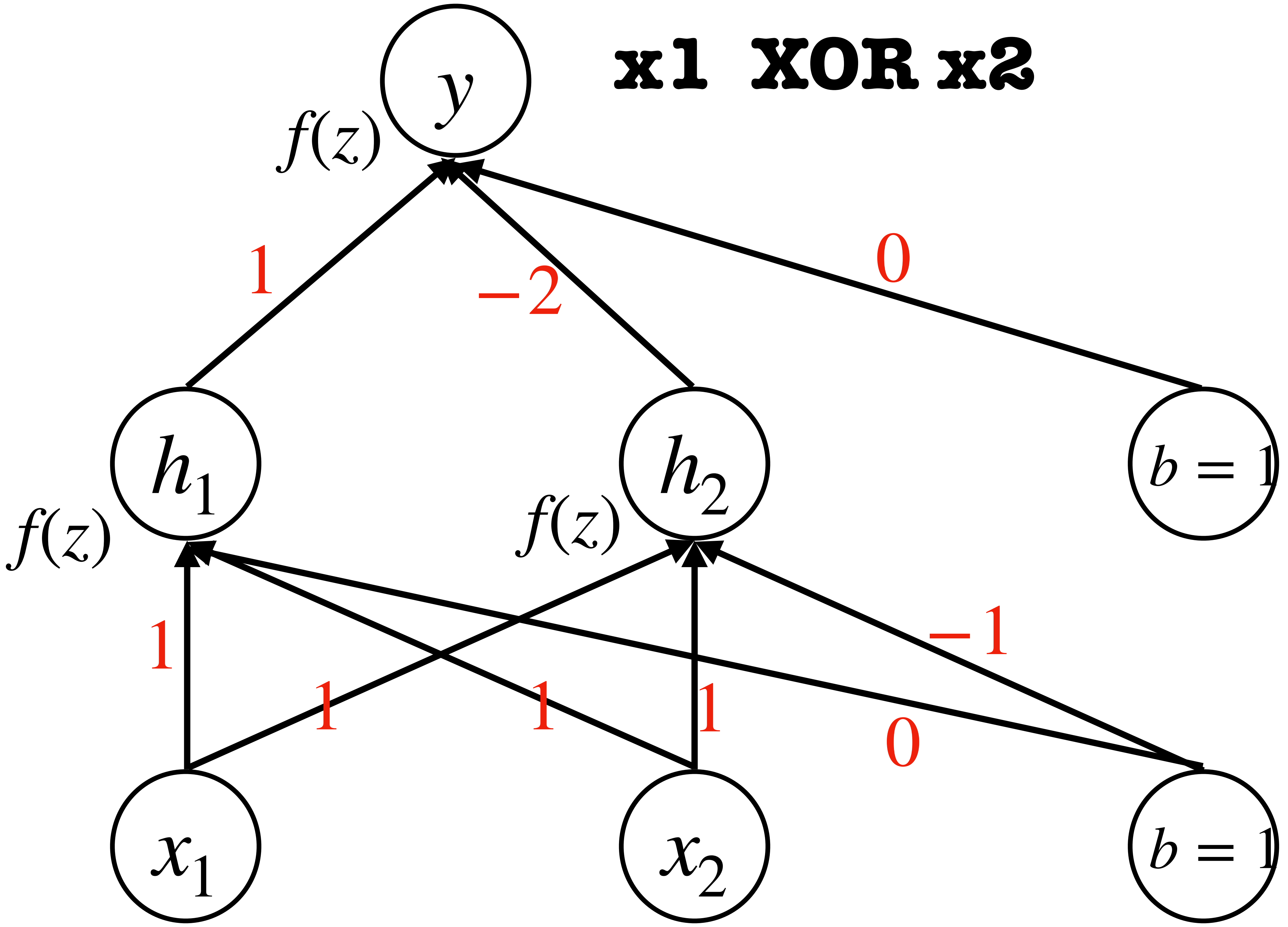
Let's verify!



x1	x2	$h1_0$	$h2_0$	h1	h2	y_0	y
0	0	0	-1	0	0	0	0
0	1	1	0	1	0	1	1
1	0	1	0	1	0	1	1
1	1	2	1	1	1	-1	0

One Solution to the XOR Problem

Let’s verify!

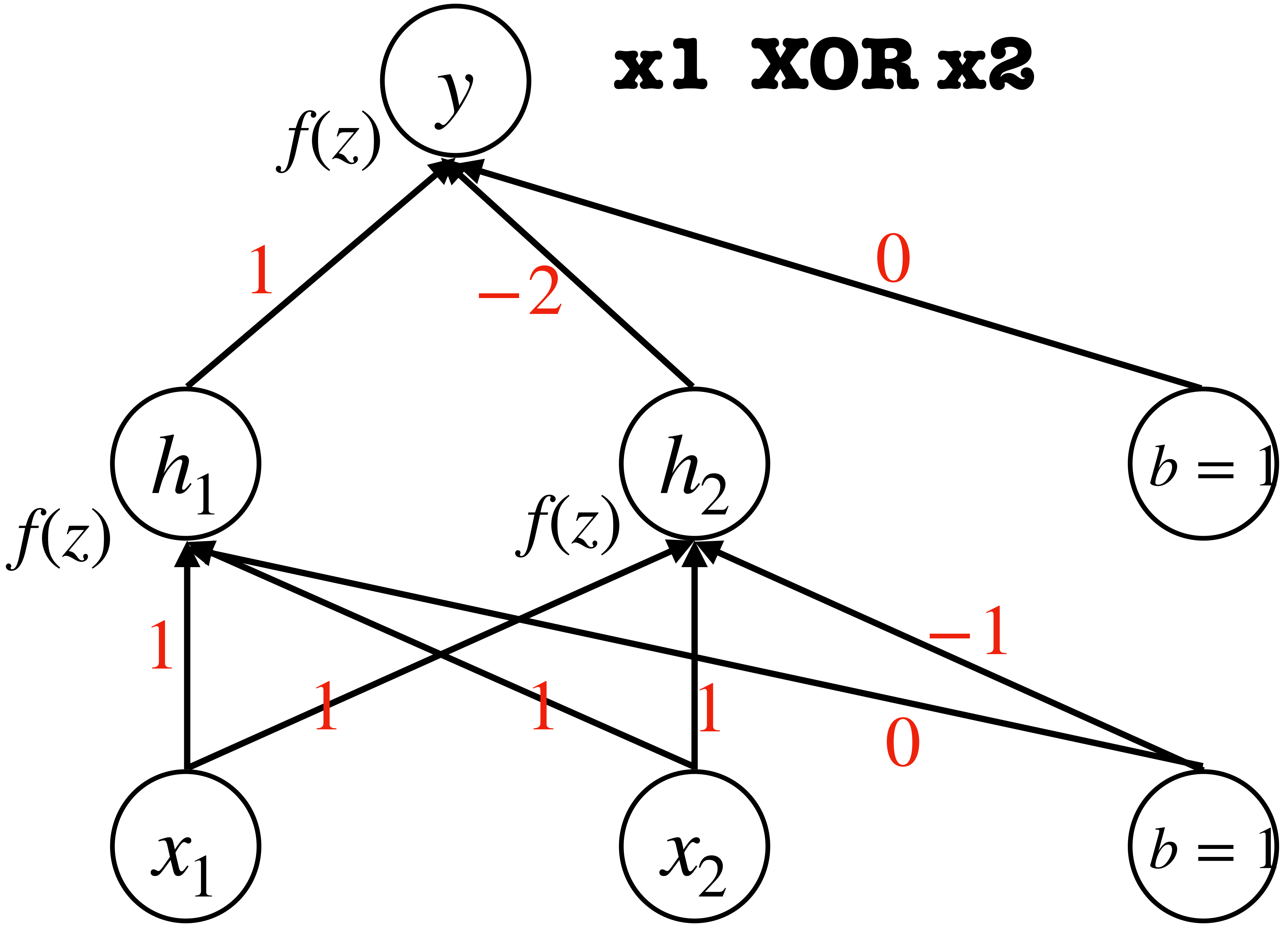


OR

x1	x2	$h1_0$	$h2_0$	h1	h2	y_0	y
0	0	0	-1	0	0	0	0
0	1	1	0	1	0	1	1
1	0	1	0	1	0	1	1
1	1	2	1	1	1	-1	0

One Solution to the XOR Problem

Let's verify!

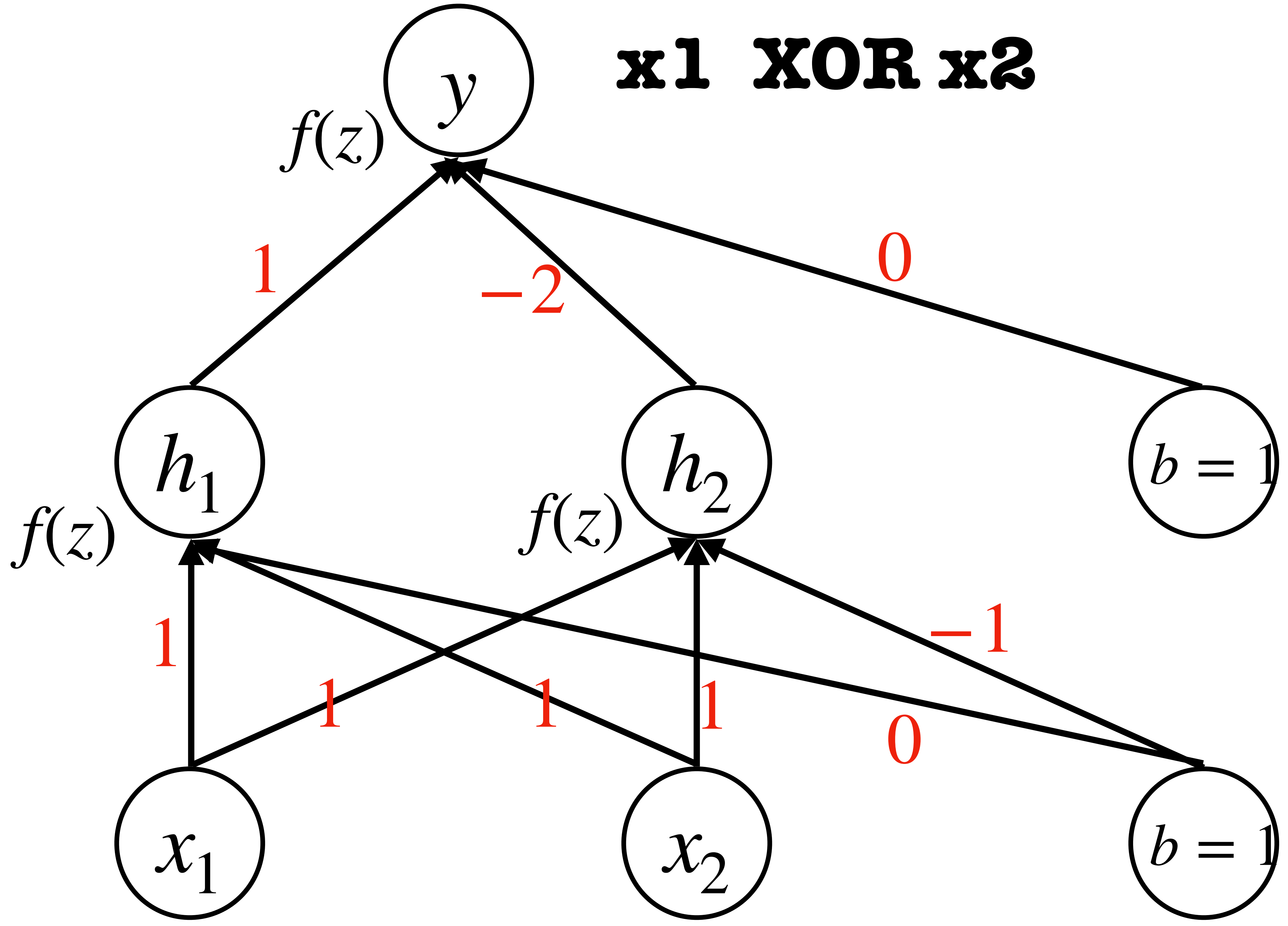


OR AND

x1	x2	$h1_0$	$h2_0$	h1	h2	y_0	y
0	0	0	-1	0	0	0	0
0	1	1	0	1	0	1	1
1	0	1	0	1	0	1	1
1	1	2	1	1	1	-1	0

One Solution to the XOR Problem

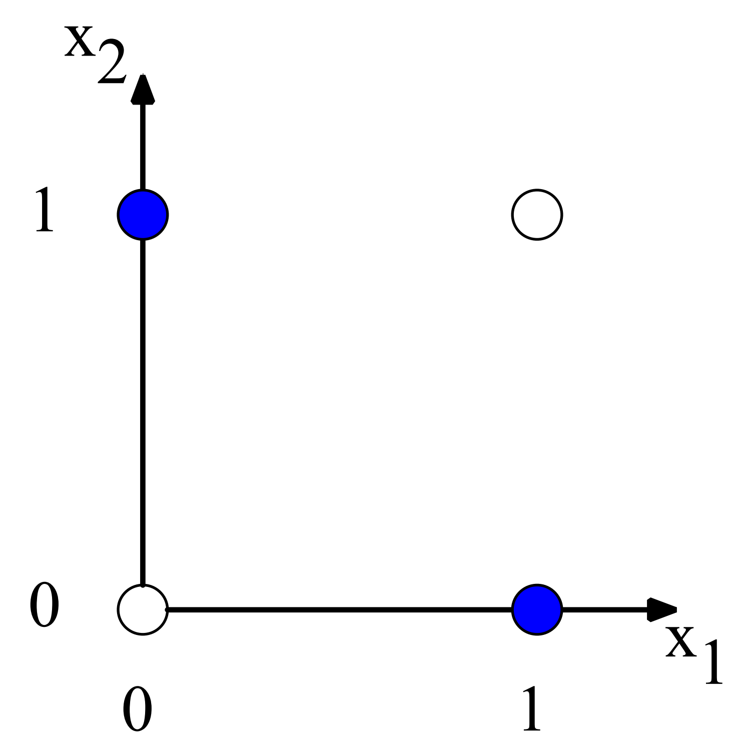
Let's verify!



				OR	AND		XOR
x1	**x2**	**$h1_0$**	**$h2_0$**	**h1**	**h2**	**y_0**	**y**
0	0	0	-1	0	0	0	0
0	1	1	0	1	0	1	1
1	0	1	0	1	0	1	1
1	1	2	1	1	1	-1	0

Why Adding a Layer Solves the Problem?

Key Idea = using nonlinear transformation at intermediate stages



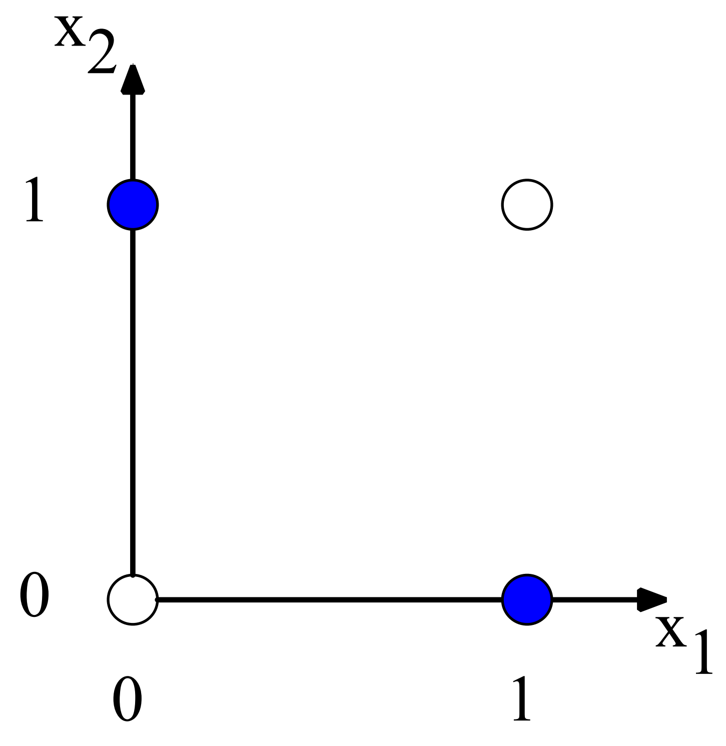
a) The original x space

Why Adding a Layer Solves the Problem?

Key Idea = using nonlinear transformation at intermediate stages

- For a Perceptron, the nonlinearly transformated result directly becomes the output;

➡ *There is no chance for the model to do further computation on it;*



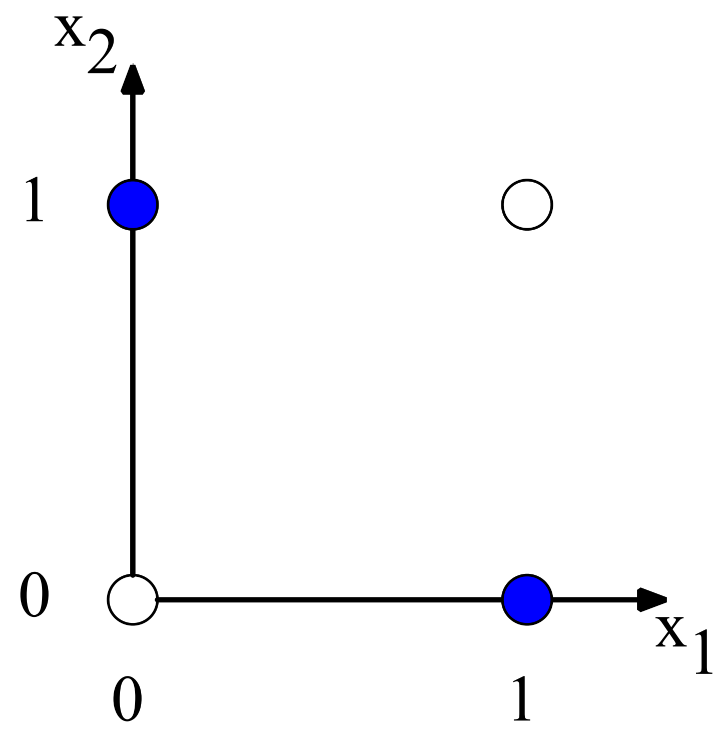
a) The original *x* space

Why Adding a Layer Solves the Problem?

Key Idea = using nonlinear transformation at intermediate stages

- For a Perceptron, the nonlinearly transformated result directly becomes the output;

  ➡ *There is no chance for the model to do further computation on it;*



a) The original *x* space

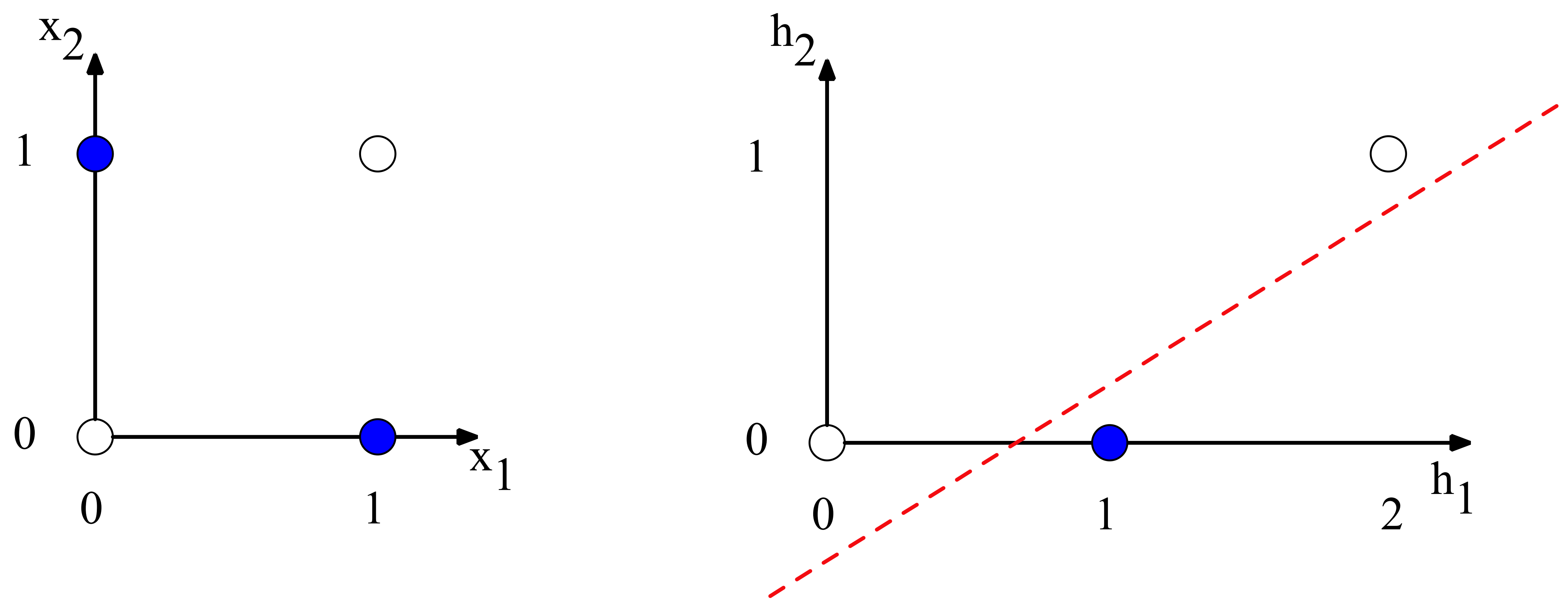
- For a two-layer neural network: it can **use the nonlinear transformation of the first layer to extract whatever useful features** (in our case: **AND** and **OR**);

Why Adding a Layer Solves the Problem?

Key Idea = using **nonlinear** transformation at **intermediate** stages

- For a Perceptron, the nonlinearly transformated result directly becomes the output;

  ➡ *There is no chance for the model to do further computation on it;*



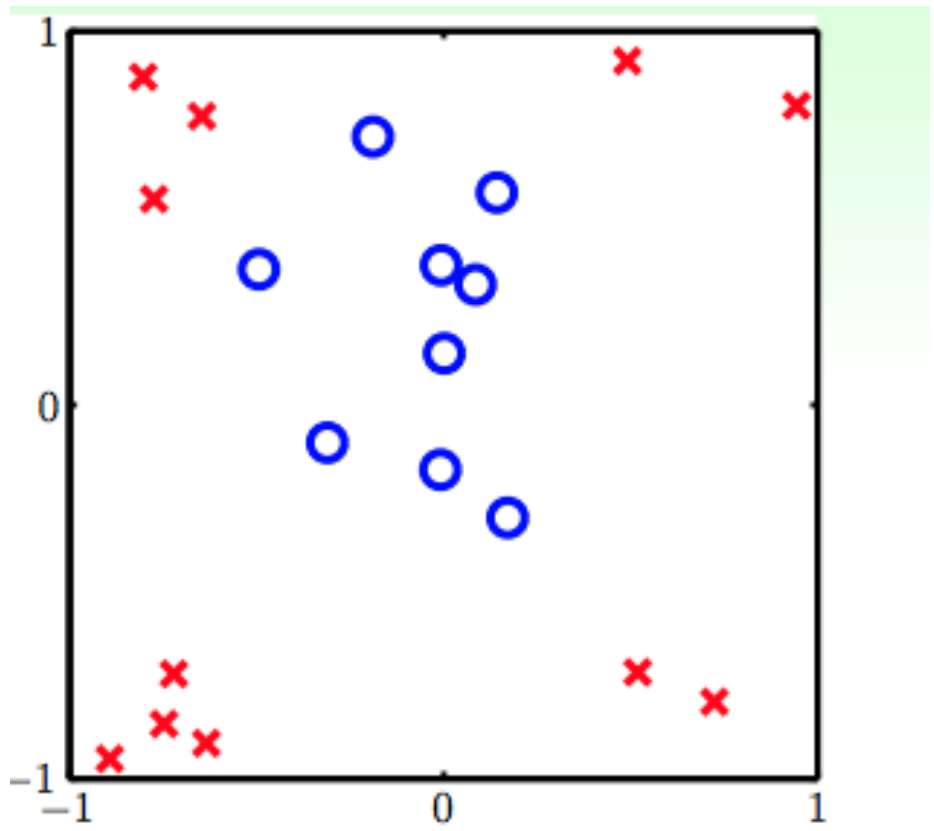
a) The original *x* space   b) The new (linearly separable) *h* space

- For a two-layer neural network: it can **use the nonlinear transformation of the first layer to extract whatever useful features** (in our case: **`AND`** and **`OR`**);

- And it can use the extracted hidden features to do more computation (further nonlinear transformation) — *it could be that those intermediate features are transformed to a linearly separable space!*

More on Nonlinearity

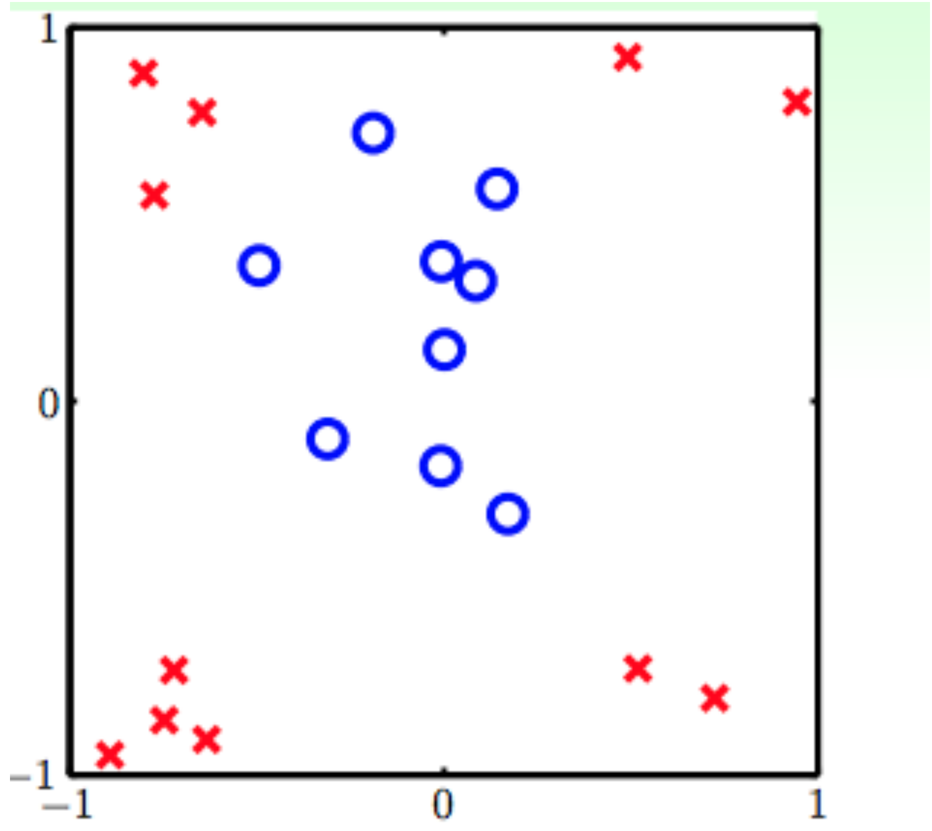
Nonlinear transformation are extremely powerful

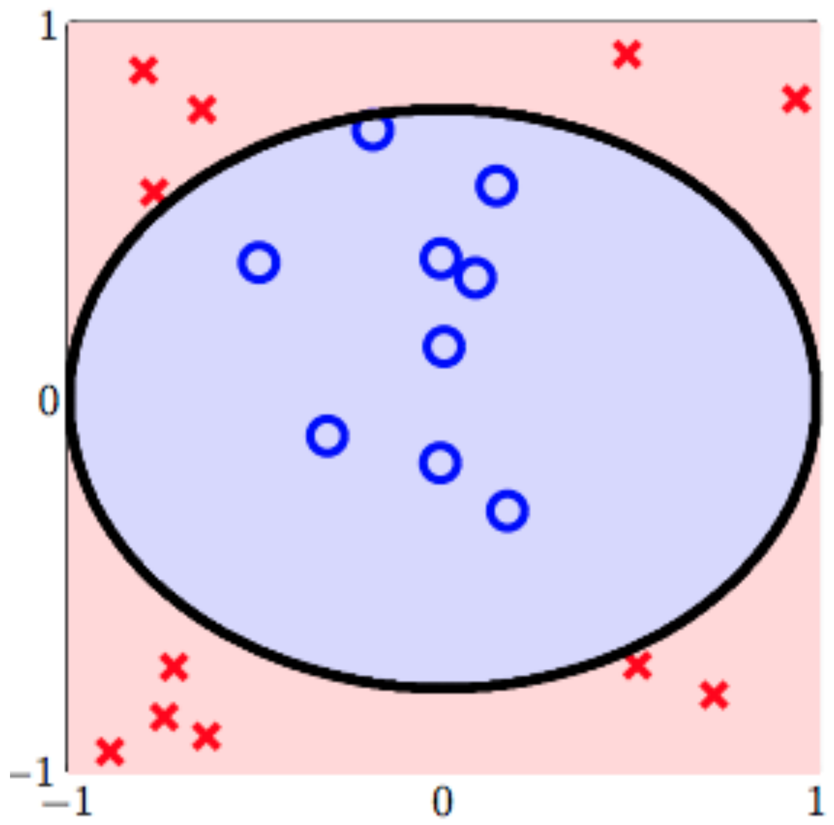


https://web.cs.ucla.edu/~chohsieh/teaching/CS260_Winter2019/lecture6.pdf

More on Nonlinearity

Nonlinear transformation are extremely powerful

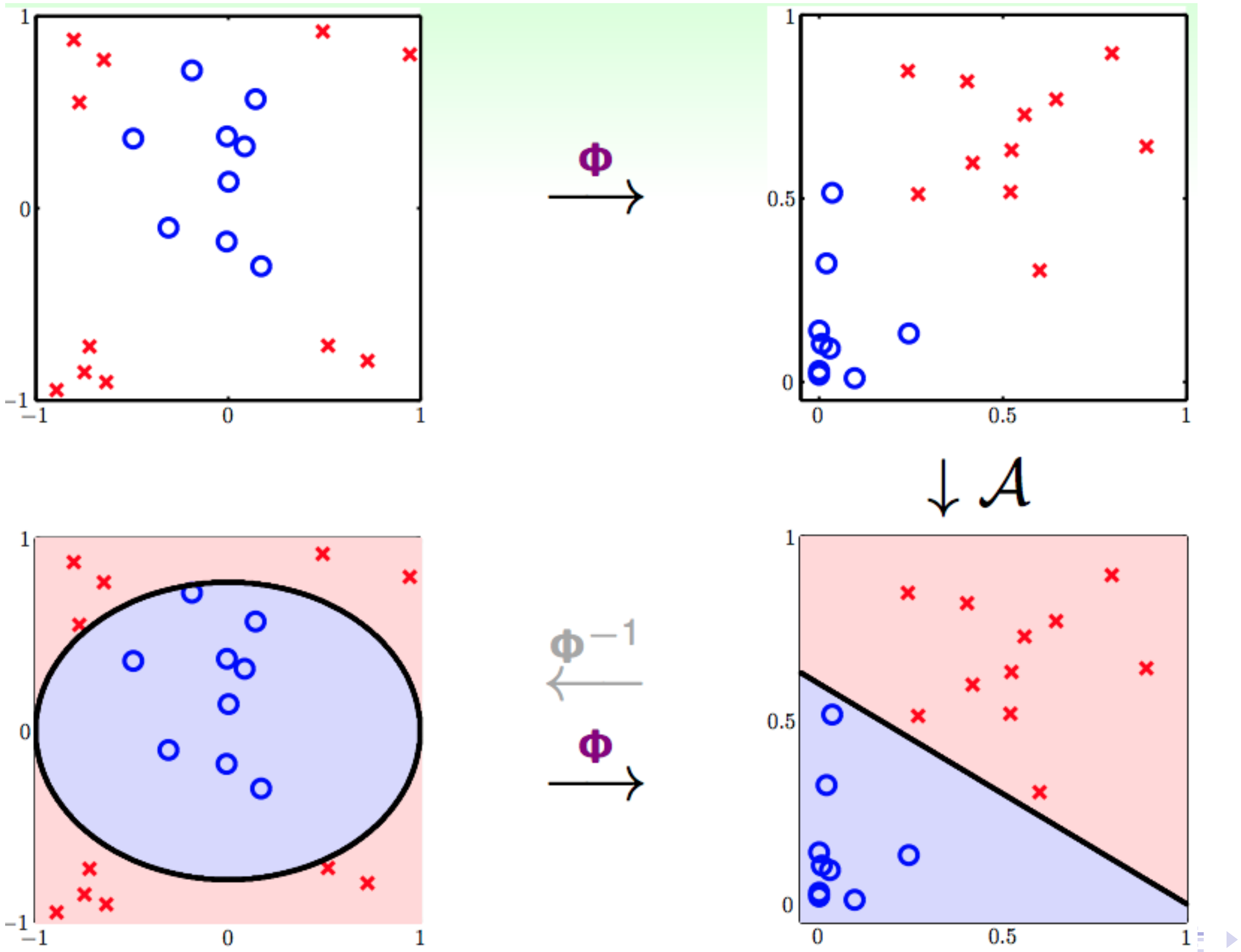




https://web.cs.ucla.edu/~chohsieh/teaching/CS260_Winter2019/lecture6.pdf

More on Nonlinearity

Nonlinear transformation are extremely powerful



e.g. Tranforming to the Polar coordinate to make an originally non-linearly-separable dataset separable!

https://web.cs.ucla.edu/~chohsieh/teaching/CS260_Winter2019/lecture6.pdf

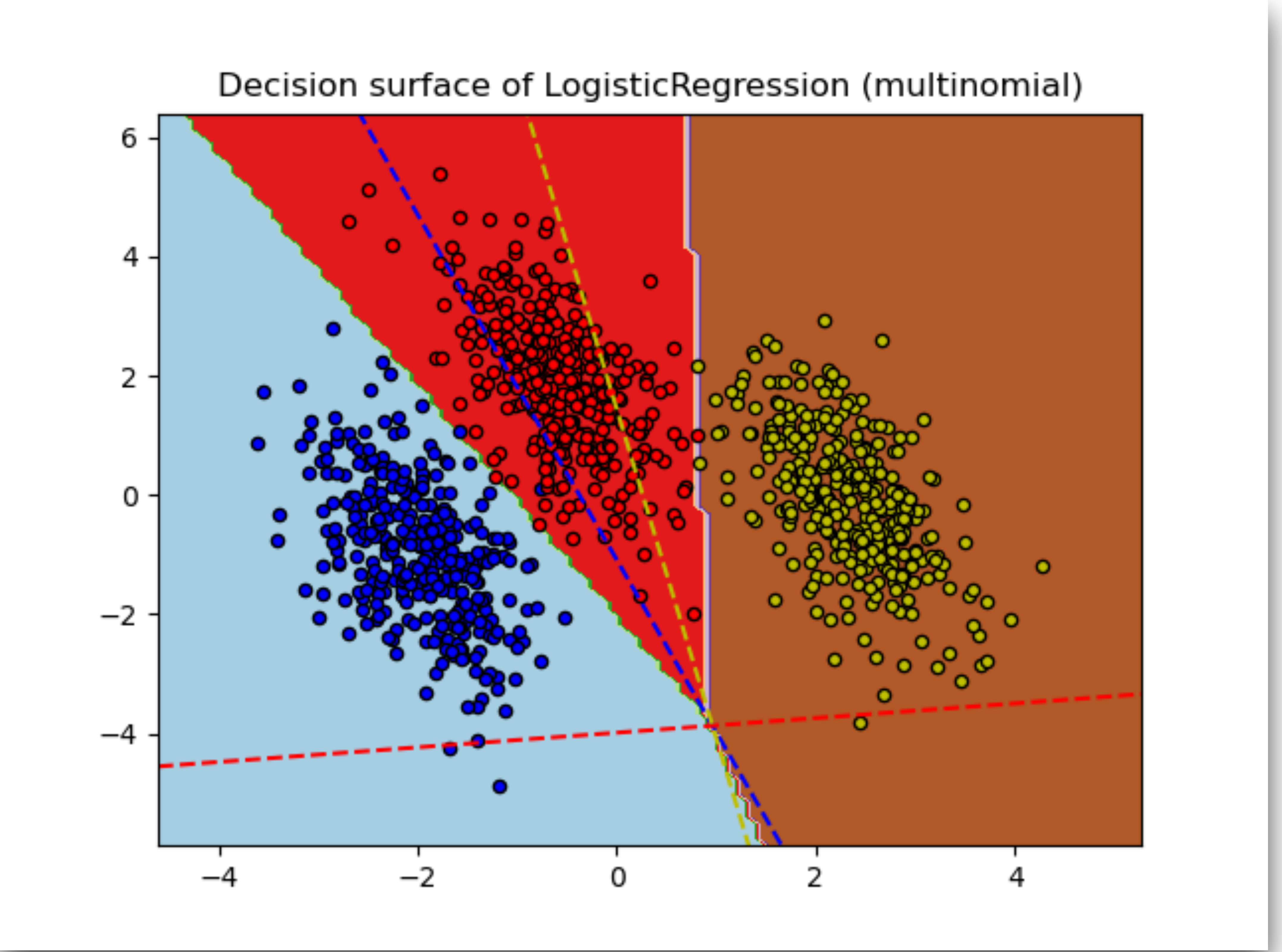
Generalizing the Architecture

From Logistic Regression to Multinomial Logistic Regression

Generalizing the Architecture

From Logistic Regression to Multinomial Logistic Regression

- Let's consider a simple generalization: imagine we are doing a **classification** task with **multiple classes / labels**. How to represent multiple outcomes?



Generalizing the Architecture

From Logistic Regression to Multinomial Logistic Regression

- Let's consider a simple generalization: imagine we are doing a **classification** task with **multiple classes / labels**. How to represent multiple outcomes?

- Two tweaks:

  - Have multiuple output nodes (one per class);

  - Switch the activation function to ***softmax***;

Generalizing the Architecture

From Logistic Regression to Multinomial Logistic Regression

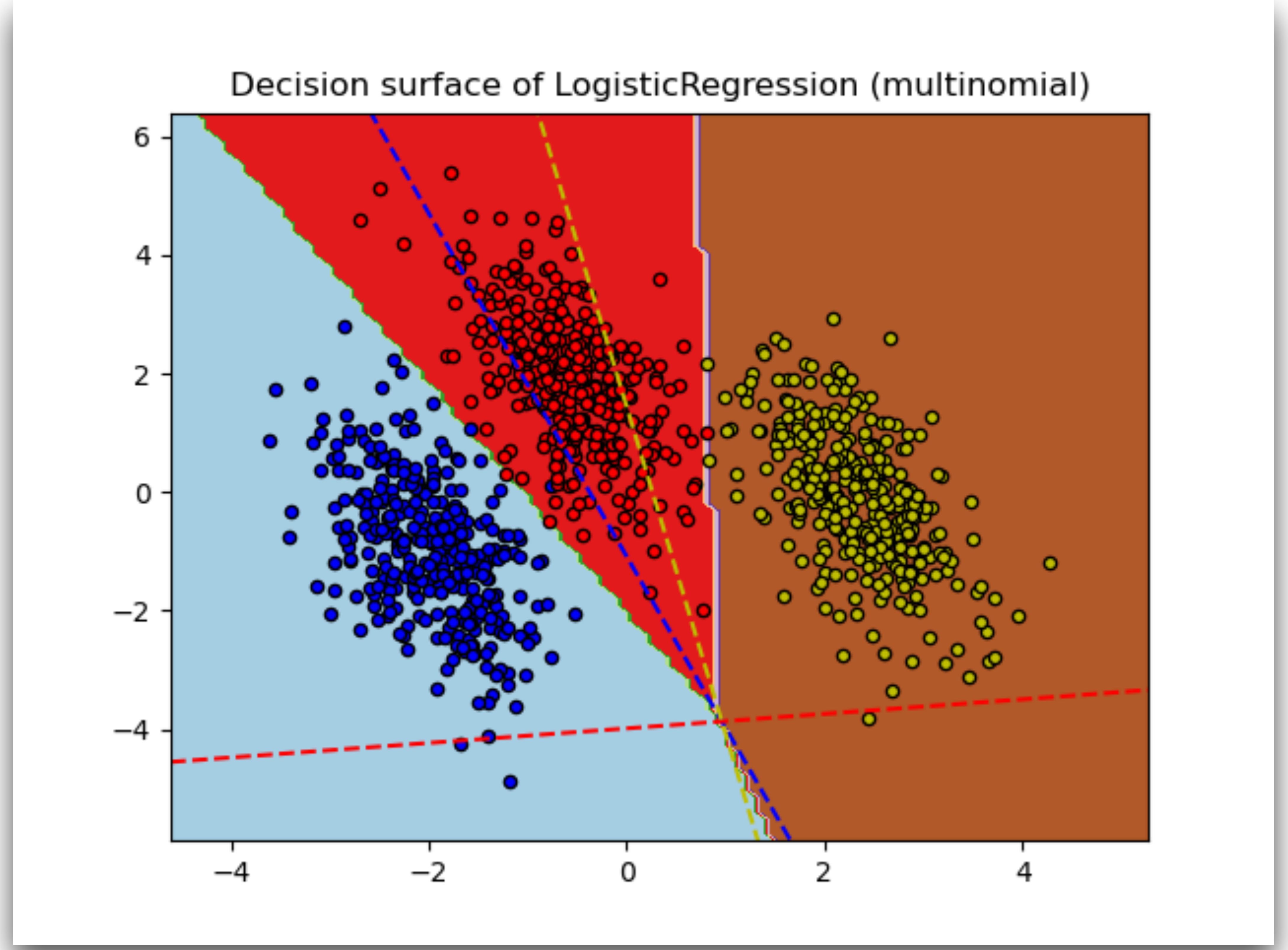
- Let's consider a simple generalization: imagine we are doing a **classification** task with **multiple classes / labels**. How to represent multiple outcomes?

- Two tweaks:

  - Have multiuple output nodes (one per class);

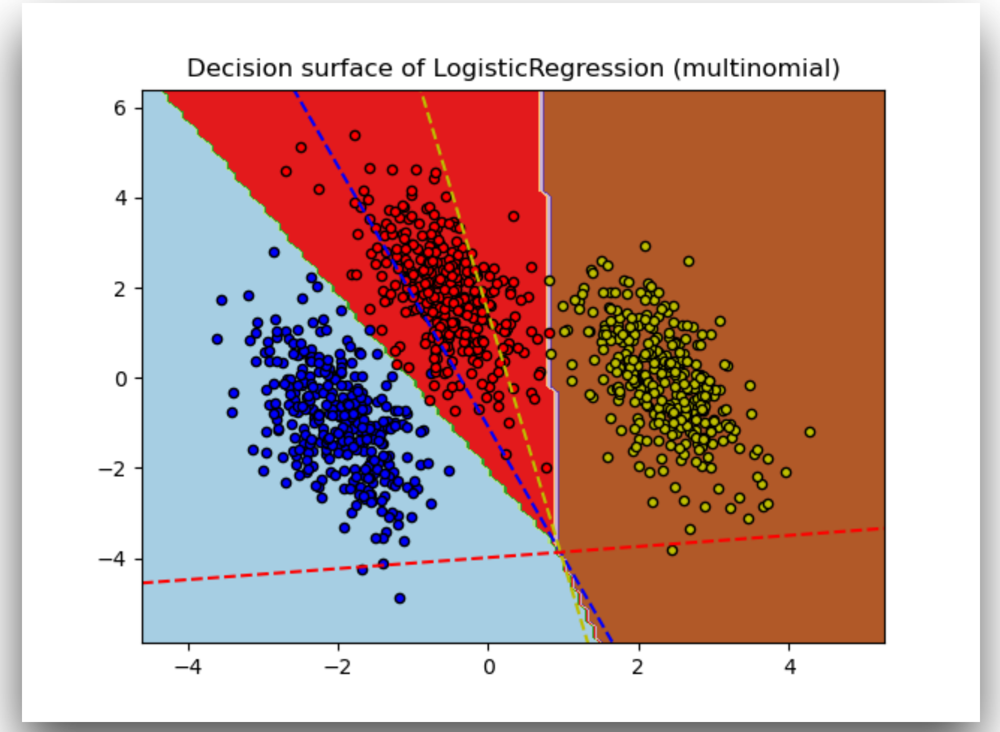
  - Switch the activation function to ***softmax***;

- This is also known as **multinomial logistic regression**.



Generalizing the Architecture

Multinomial Logistic Regression

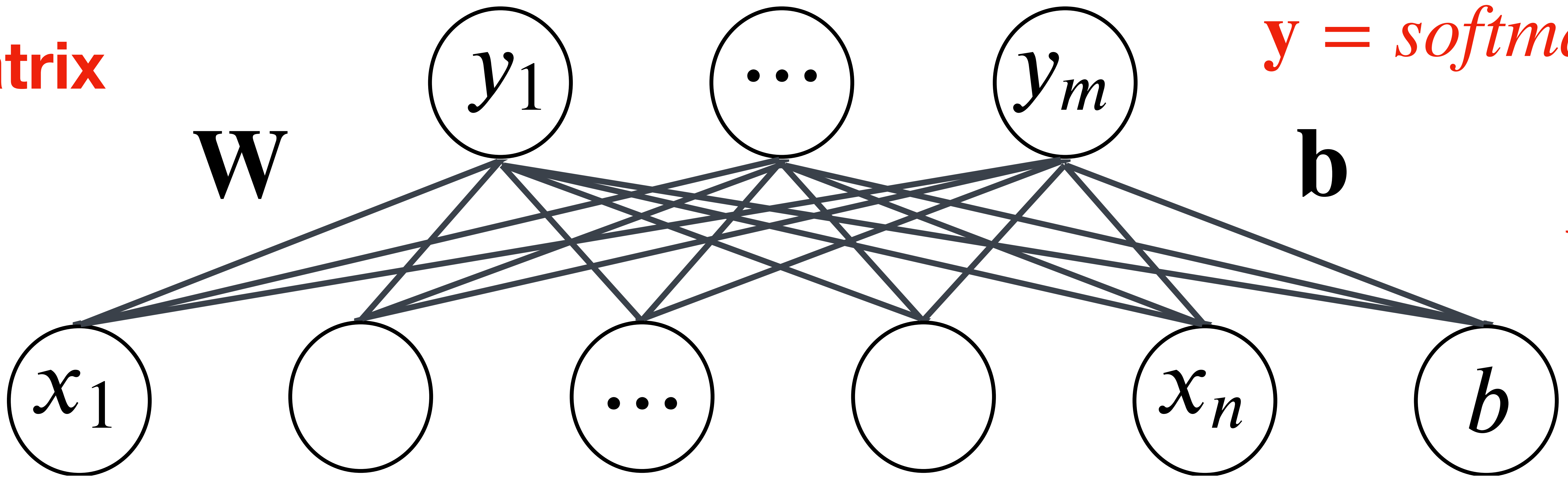


- Suppose we have n input features and m classes;
- One way to represent it is a 1-layer neural network with multiple output nodes (i.e., a layer of neurons)
- We don't count the input layer as a layer, so this network has 1 layer.

$\mathbf{y} = [y_1, \cdots, y_m]$ **is a vector:**

$\mathbf{y} = softmax(\mathbf{Wx} + \mathbf{b})$

W is a matrix



b is a vector

Input a list of scalars

Softmax: A generalization of Sigmoid

Getting a probability distribution for multi-class classification!

Softmax: A generalization of Sigmoid

Getting a probability distribution for multi-class classification!

- **Sigmoid** takes a real value and outputs a probability in range $[0, 1]$;

Softmax: A generalization of Sigmoid

Getting a probability distribution for multi-class classification!

- **Sigmoid** takes a real value and outputs a probability in range $[0, 1]$;
- **Softmax** takes a list of real values and outputs a probability distribution (a list of the same length as the input that sums to 1).

Softmax: A generalization of Sigmoid

Getting a probability distribution for multi-class classification!

- **Sigmoid** takes a real value and outputs a probability in range $[0, 1]$;
- **Softmax** takes a list of real values and outputs a probability distribution (a list of the same length as the input that sums to 1).
- Formally: for a vector z of dimensionality k:

$$softmax(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

Softmax: A generalization of Sigmoid

Getting a probability distribution for multi-class classification!

- **Sigmoid** takes a real value and outputs a probability in range $[0, 1]$;
- **Softmax** takes a list of real values and outputs a probability distribution (a list of the same length as the input that sums to 1).
- Formally: for a vector z of dimensionality k:

$$softmax(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

- An example:

$$\mathbf{z} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

$$softmax(\mathbf{z}) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

Softmax: A generalization of Sigmoid

Getting a probability distribution for multi-class classification!

- **Sigmoid** takes a real value and outputs a probability in range $[0, 1]$;
- **Softmax** takes a list of real values and outputs a probability distribution (a list of the same length as the input that sums to 1).
- Formally: for a vector z of dimensionality k:

$$softmax(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

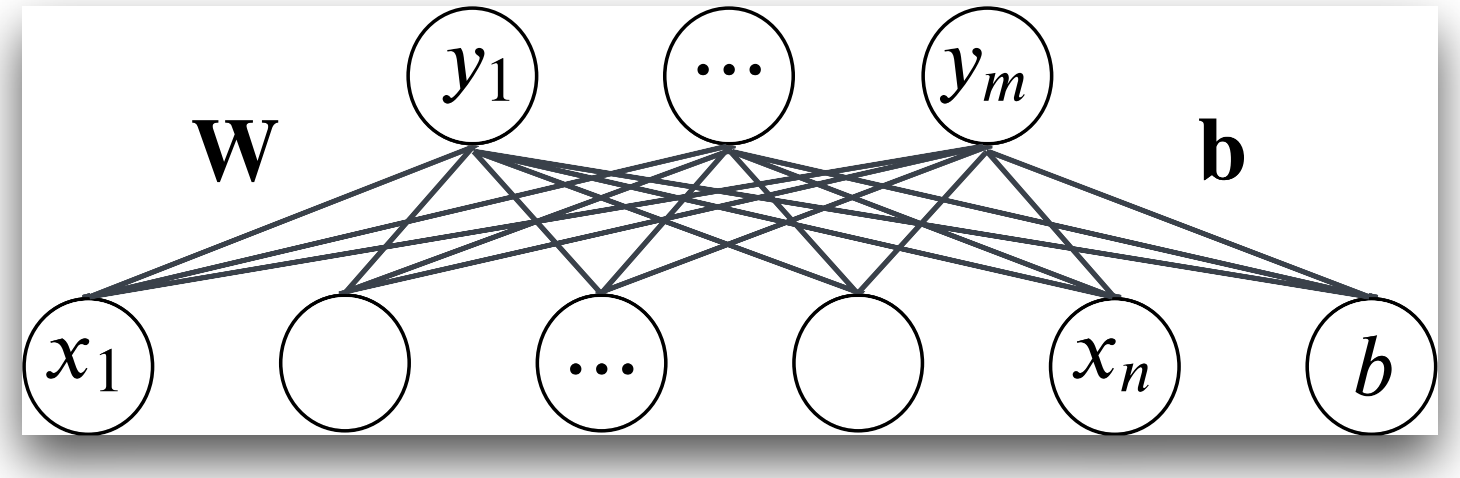
- An example:

$$\mathbf{z} = [0.6, 1.1, -1.5, 1.2, \boxed{3.2}, -1.1]$$

$$softmax(\mathbf{z}) = [0.055, 0.090, 0.006, 0.099, \boxed{0.74}, 0.010]$$

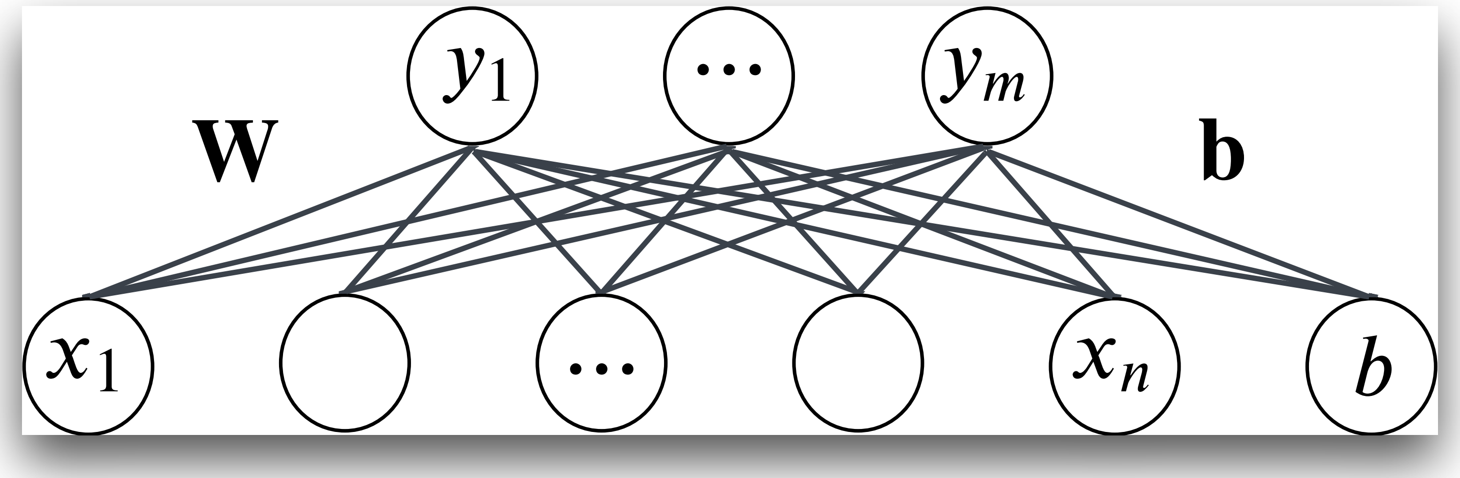
Generalizing the Notations

Representing a list of neurons



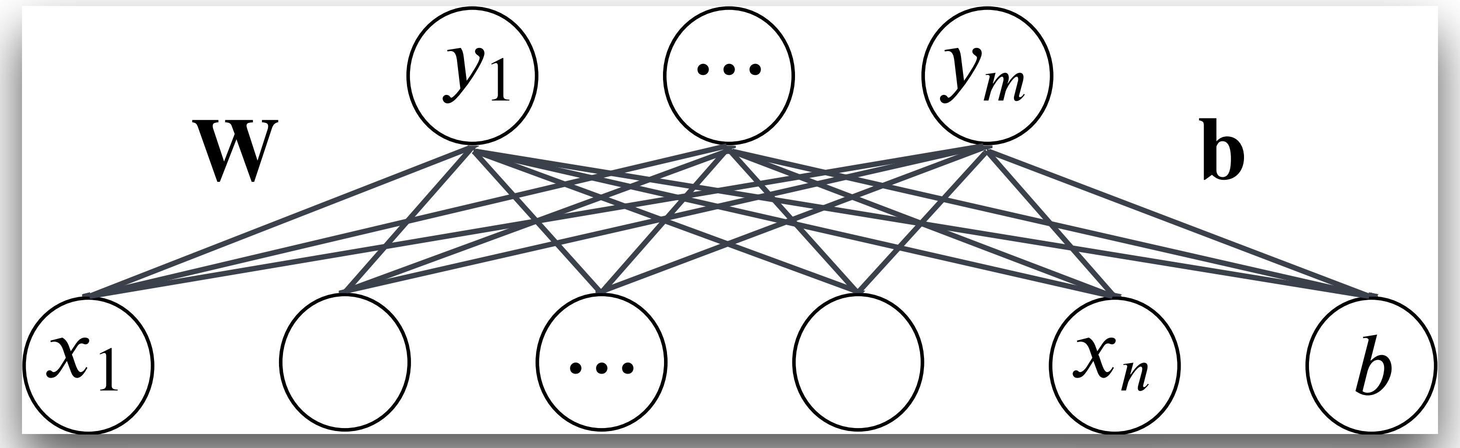
Generalizing the Notations

Representing a list of neurons



- Previously, we have only 1 neuron, so the weights are represented as a vector.

Generalizing the Notations

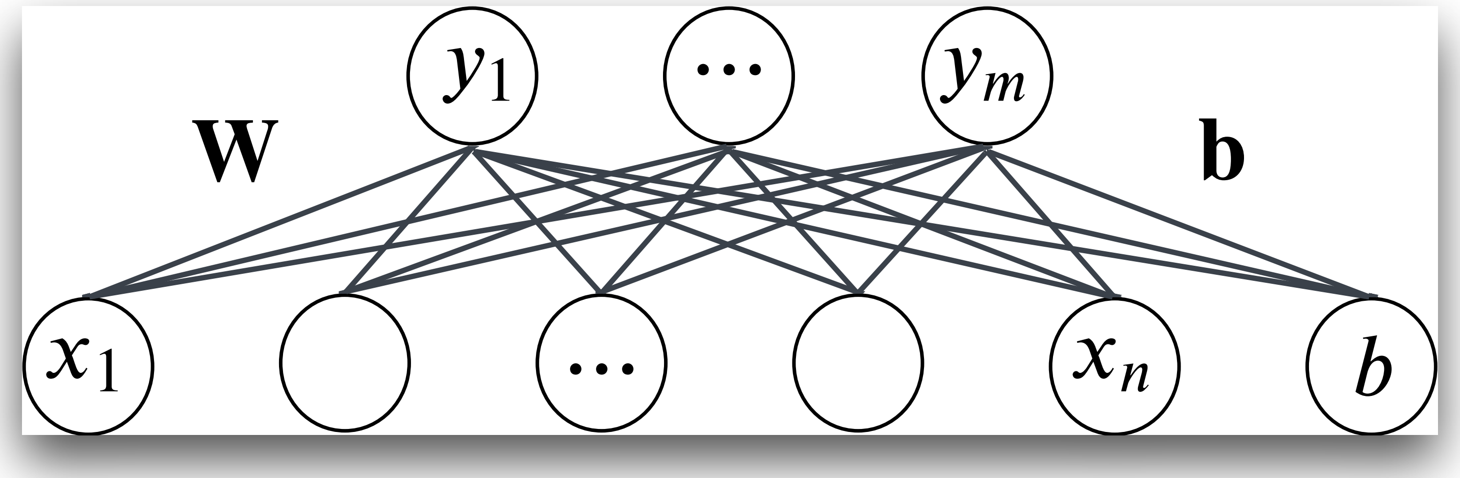


Representing a list of neurons

- Previously, we have only 1 neuron, so the weights are represented as a vector.
- Now, we have multiple neurons in the layer, and each neuron is connected to all the inputs, so we (naturally) need a matrix to represent the weights.

Generalizing the Notations

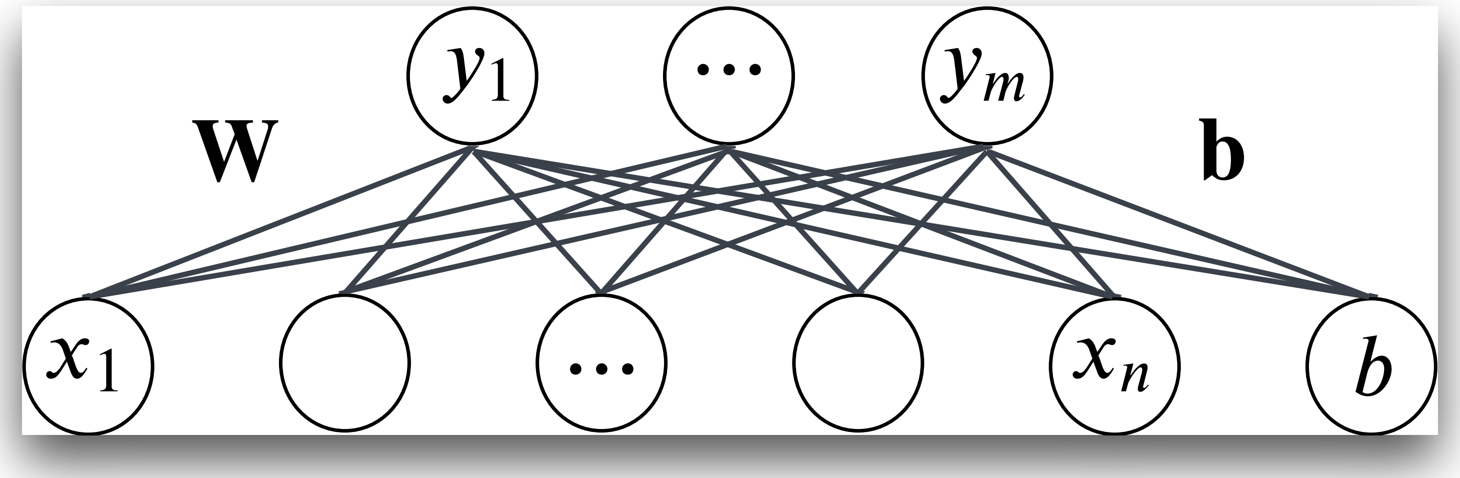
Representing a list of neurons



- Previously, we have only 1 neuron, so the weights are represented as a vector.
- Now, we have multiple neurons in the layer, and each neuron is connected to all the inputs, so we (naturally) need a matrix to represent the weights.
- Suppose we have 2 input features and 3 neurons:

Generalizing the Notations

Representing a list of neurons

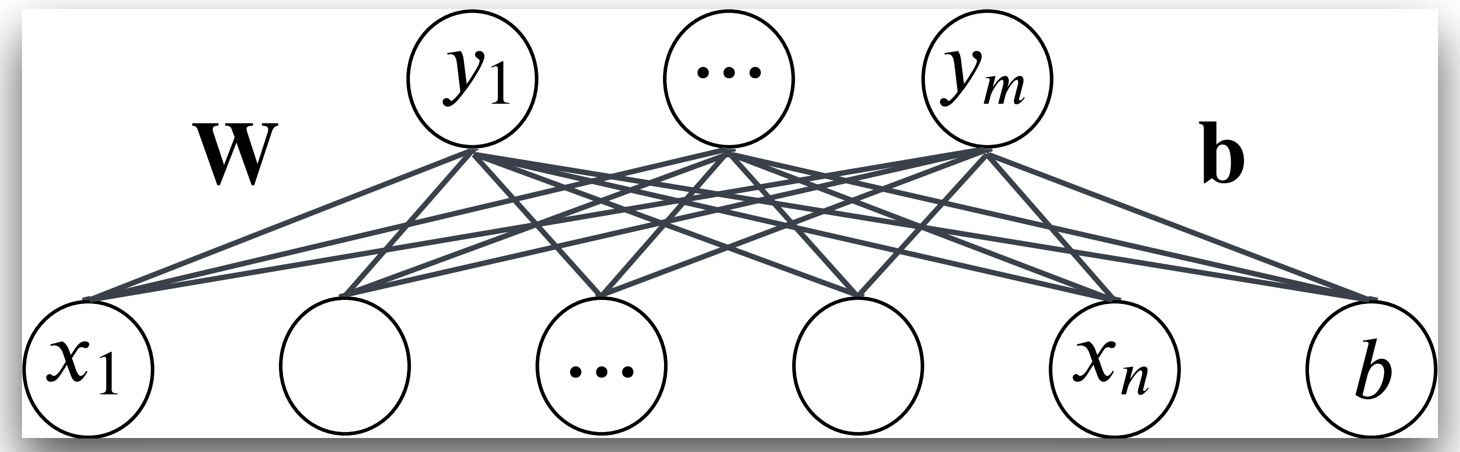


- Previously, we have only 1 neuron, so the weights are represented as a vector.
- Now, we have multiple neurons in the layer, and each neuron is connected to all the inputs, so we (naturally) need a matrix to represent the weights.
- Suppose we have 2 input features and 3 neurons:

$$\underbrace{\begin{bmatrix} 0.1 & 1.0 \\ 2.0 & -0.1 \\ -0.3 & 0.5 \end{bmatrix}}_{\mathbf{W}} \underbrace{\begin{bmatrix} 1.4 \\ -0.2 \end{bmatrix}}_{\mathbf{x}} = \begin{bmatrix} 1.4 \times 0.1 + (-0.2) \times 1.0 \\ 1.4 \times 2.0 + (-0.2) \times (-0.1) \\ 1.4 \times (-0.3) + (-0.2) \times 0.5 \end{bmatrix} = \underbrace{\begin{bmatrix} -0.06 \\ 2.82 \\ -0.52 \end{bmatrix}}_{\mathbf{Wx}}$$

Generalizing the Notations

Representing a list of neurons

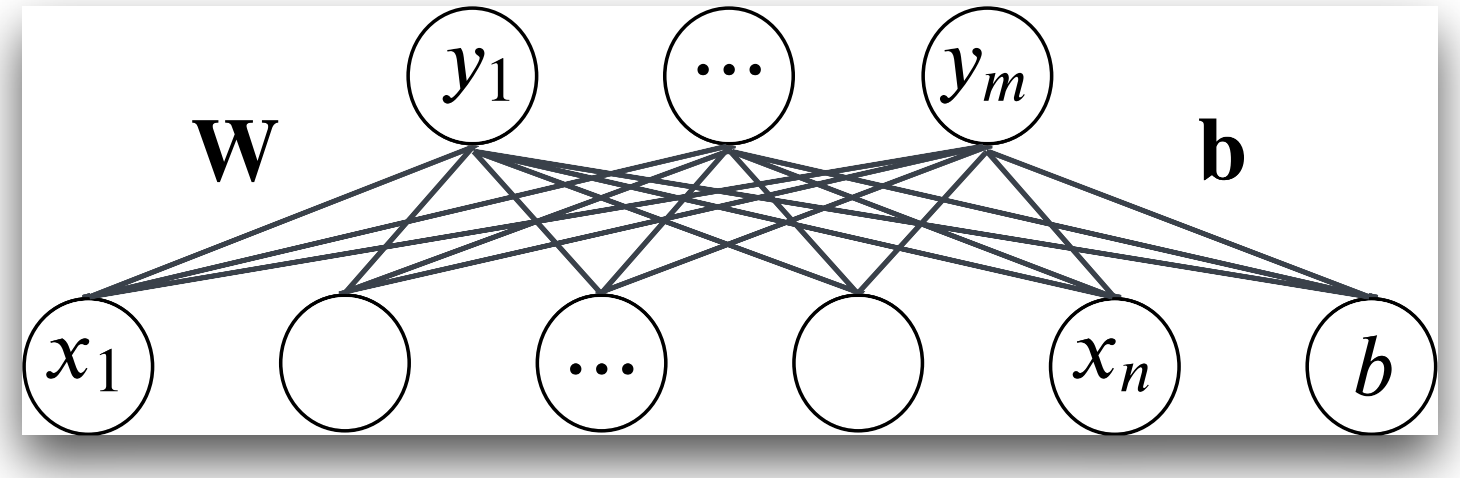


- Previously, we have only 1 neuron, so the weights are represented as a vector.
- Now, we have multiple neurons in the layer, and each neuron is connected to all the inputs, so we (naturally) need a matrix to represent the weights.
- Suppose we have 2 input features and 3 neurons:

Weight connecting 1st input to 1st neuron

$$\overset{\mathbf{W}}{\begin{bmatrix} \boxed{0.1} & 1.0 \\ 2.0 & -0.1 \\ -0.3 & 0.5 \end{bmatrix}} \overset{\mathbf{x}}{\begin{bmatrix} 1.4 \\ -0.2 \end{bmatrix}} = \begin{bmatrix} 1.4 \times 0.1 + (-0.2) \times 1.0 \\ 1.4 \times 2.0 + (-0.2) \times (-0.1) \\ 1.4 \times (-0.3) + (-0.2) \times 0.5 \end{bmatrix} = \overset{\mathbf{Wx}}{\begin{bmatrix} -0.06 \\ 2.82 \\ -0.52 \end{bmatrix}}$$

Generalizing the Notations



Representing a list of neurons

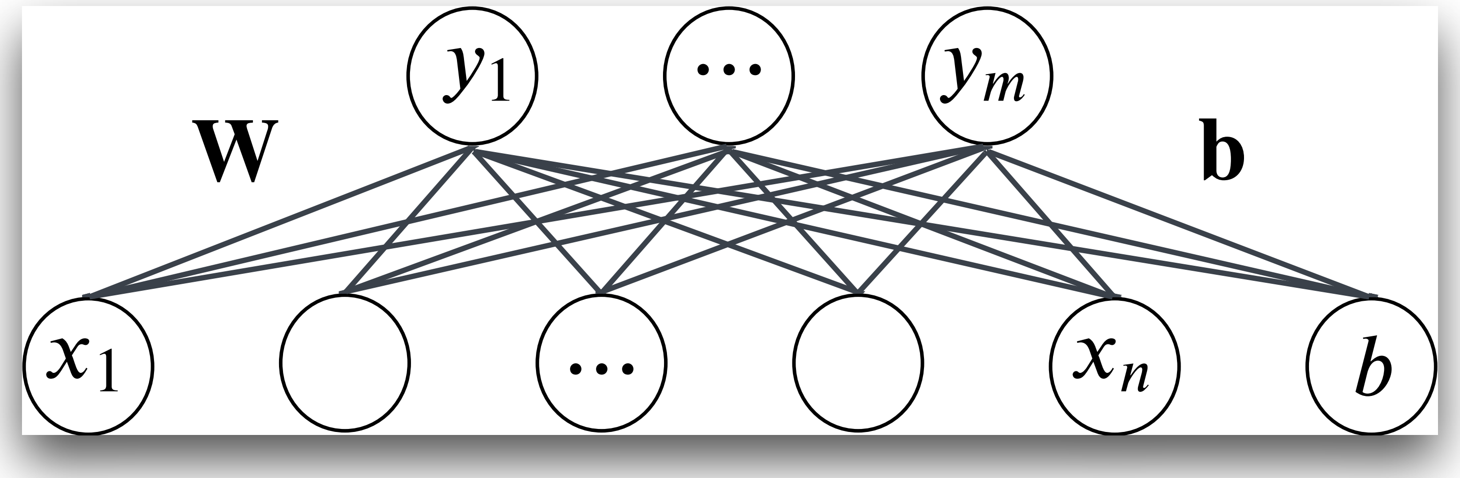
- Previously, we have only 1 neuron, so the weights are represented as a vector.
- Now, we have multiple neurons in the layer, and each neuron is connected to all the inputs, so we (naturally) need a matrix to represent the weights.
- Suppose we have 2 input features and 3 neurons:

Weight connecting 1st input to 1st neuron

$$\overset{\mathbf{W}}{\begin{bmatrix} \boxed{0.1} & 1.0 \\ 2.0 & -0.1 \\ -0.3 & 0.5 \end{bmatrix}} \overset{\mathbf{x}}{\begin{bmatrix} \boxed{1.4} \\ -0.2 \end{bmatrix}} = \begin{bmatrix} \boxed{1.4 \times 0.1} + (-0.2) \times 1.0 \\ 1.4 \times 2.0 + (-0.2) \times (-0.1) \\ 1.4 \times (-0.3) + (-0.2) \times 0.5 \end{bmatrix} = \overset{\mathbf{Wx}}{\begin{bmatrix} -0.06 \\ 2.82 \\ -0.52 \end{bmatrix}}$$

Generalizing the Notations

Representing a list of neurons



- Previously, we have only 1 neuron, so the weights are represented as a vector.
- Now, we have multiple neurons in the layer, and each neuron is connected to all the inputs, so we (naturally) need a matrix to represent the weights.
- Suppose we have 2 input features and 3 neurons:

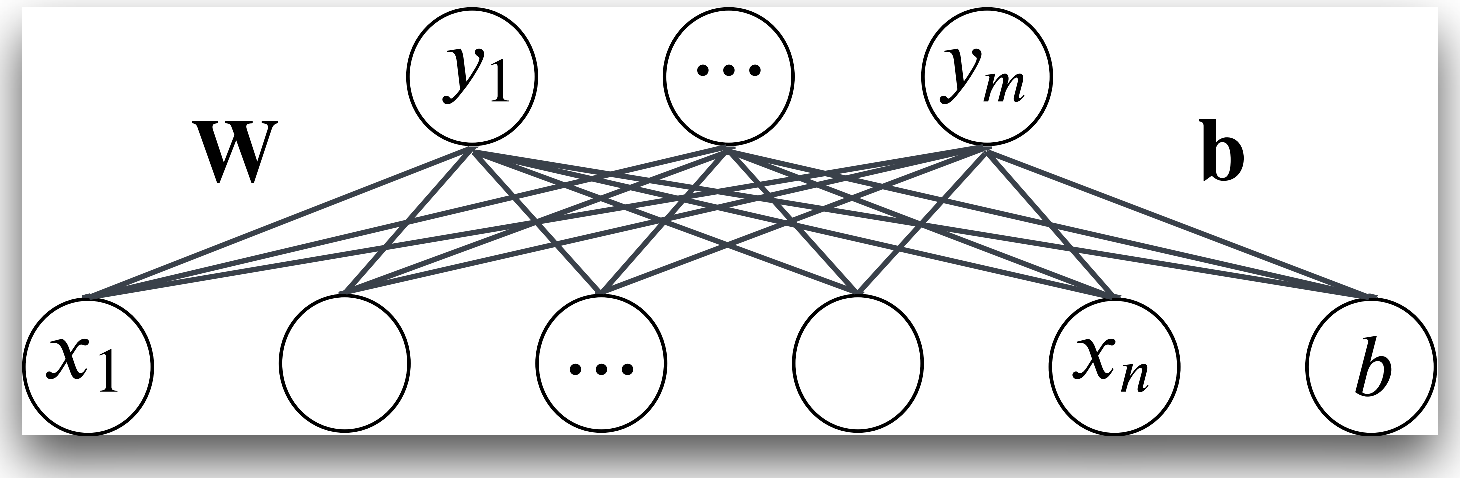
Weight connecting 1st input to 1st neuron

$$\overset{\mathbf{W}}{\begin{bmatrix} \boxed{0.1} & 1.0 \\ 2.0 & -0.1 \\ -0.3 & \boxed{0.5} \end{bmatrix}} \overset{\mathbf{x}}{\begin{bmatrix} \boxed{1.4} \\ -0.2 \end{bmatrix}} = \begin{bmatrix} \boxed{1.4 \times 0.1} + (-0.2) \times 1.0 \\ 1.4 \times 2.0 + (-0.2) \times (-0.1) \\ 1.4 \times (-0.3) + (-0.2) \times 0.5 \end{bmatrix} = \overset{\mathbf{Wx}}{\begin{bmatrix} -0.06 \\ 2.82 \\ -0.52 \end{bmatrix}}$$

Weight connecting 2nd input to 3rd neuron

Generalizing the Notations

Representing a list of neurons



- Previously, we have only 1 neuron, so the weights are represented as a vector.
- Now, we have multiple neurons in the layer, and each neuron is connected to all the inputs, so we (naturally) need a matrix to represent the weights.
- Suppose we have 2 input features and 3 neurons:

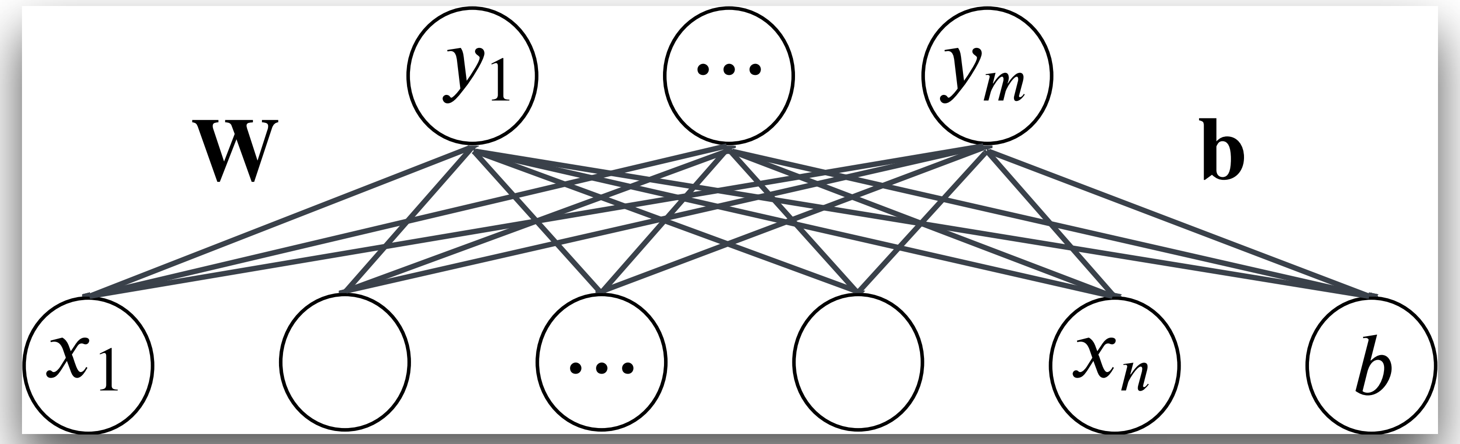
Weight connecting 1st input to 1st neuron

$$\overset{\mathbf{W}}{\begin{bmatrix} \boxed{0.1} & 1.0 \\ 2.0 & -0.1 \\ -0.3 & \boxed{0.5} \end{bmatrix}} \overset{\mathbf{x}}{\begin{bmatrix} \boxed{1.4} \\ \boxed{-0.2} \end{bmatrix}} = \begin{bmatrix} \boxed{1.4 \times 0.1} + (-0.2) \times 1.0 \\ 1.4 \times 2.0 + (-0.2) \times (-0.1) \\ 1.4 \times (-0.3) + \boxed{(-0.2) \times 0.5} \end{bmatrix} = \overset{\mathbf{Wx}}{\begin{bmatrix} -0.06 \\ 2.82 \\ -0.52 \end{bmatrix}}$$

Weight connecting 2nd input to 3rd neuron

Generalizing the Notations

Representing a list of neurons



- Previously, we have only 1 neuron, so the weights are represented as a vector.
- Now, we have multiple neurons in the layer, and each neuron is connected to all the inputs, so we (naturally) need a matrix to represent the weights.
- Suppose we have 2 input features and 3 neurons:

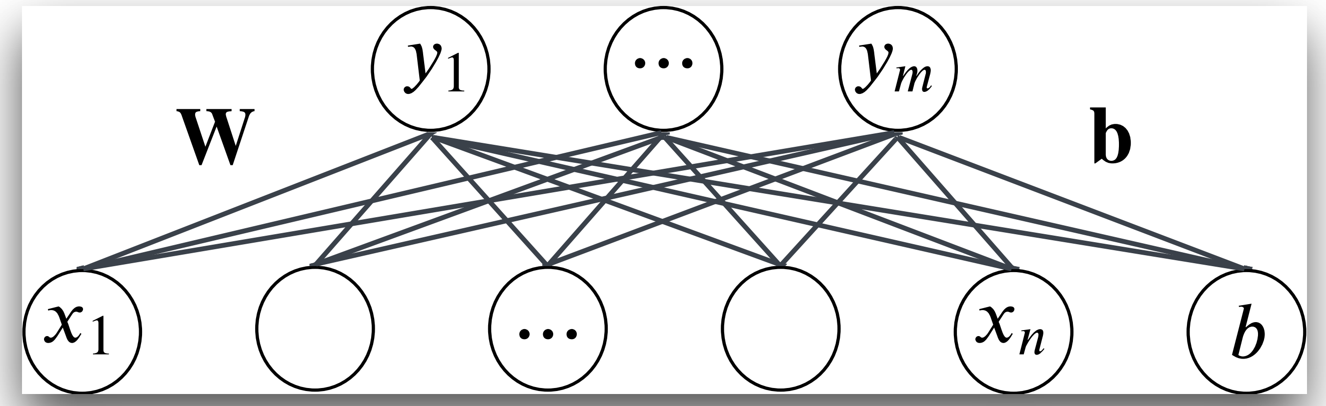
Weight connecting 1st input to 1st neuron

Weight connecting 2nd input to 3rd neuron

$$\underset{\mathbf{W}}{\begin{bmatrix} \boxed{0.1} & 1.0 \\ 2.0 & -0.1 \\ -0.3 & \boxed{0.5} \end{bmatrix}} \underset{\mathbf{x}}{\begin{bmatrix} \boxed{1.4} \\ \boxed{-0.2} \end{bmatrix}} = \begin{bmatrix} \boxed{1.4 \times 0.1} + (-0.2) \times 1.0 \\ 1.4 \times 2.0 + (-0.2) \times (-0.1) \\ 1.4 \times (-0.3) + \boxed{(-0.2) \times 0.5} \end{bmatrix} = \underset{\mathbf{Wx}}{\begin{bmatrix} -0.06 \\ 2.82 \\ -0.52 \end{bmatrix}}$$

In general: position $[i, j]$ in $\mathbf{W}$ is the weight connecting input j to neuron i.

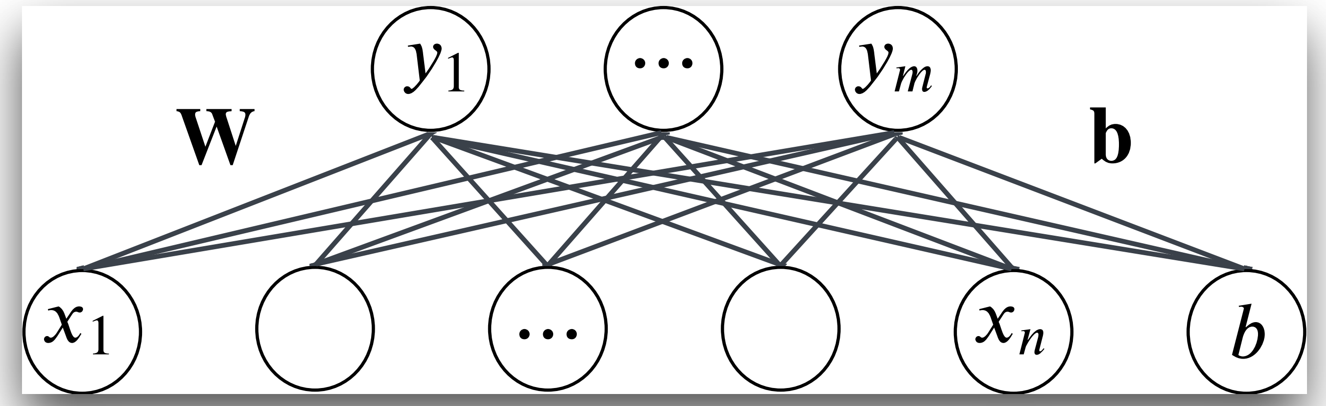
Generalizing the Notations, cont.



Handling the Bias Term

- As before, we incorporeate the bias vector $\mathbf{b}$ into $\mathbf{W}$ and $\mathbf{x}$ by adding a constant value 1 into the input vector, as follows:

Generalizing the Notations, cont.

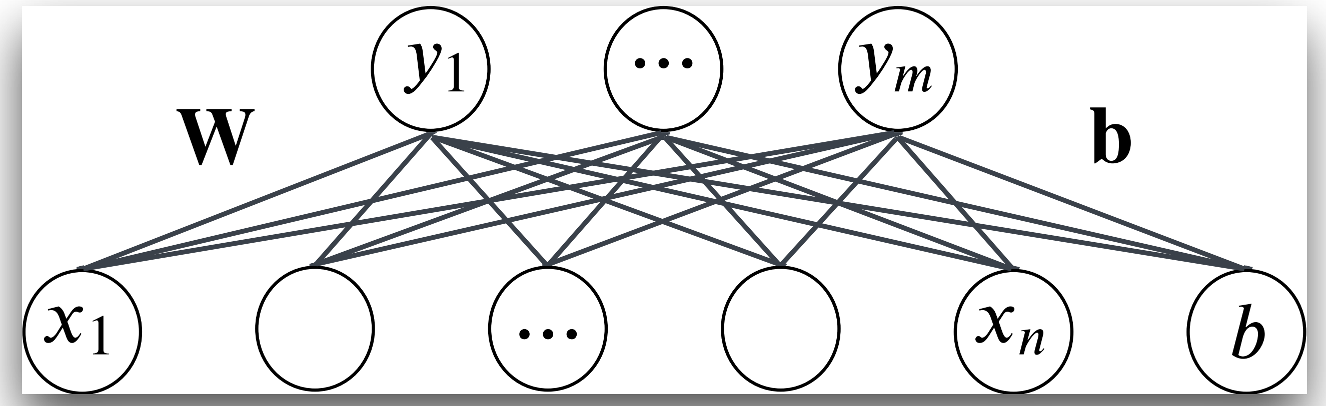


Handling the Bias Term

- As before, we incorporeate the bias vector $\mathbf{b}$ into $\mathbf{W}$ and $\mathbf{x}$ by adding a constant value 1 into the input vector, as follows:

$$\underbrace{\begin{bmatrix} 0.1 & 1.0 \\ 2.0 & -0.1 \\ -0.3 & 0.5 \end{bmatrix}}_{\mathbf{W}} \underbrace{\begin{bmatrix} 1.4 \\ -0.2 \end{bmatrix}}_{\mathbf{x}} + \underbrace{\begin{bmatrix} 0.2 \\ 1.0 \\ -0.3 \end{bmatrix}}_{\color{red}{\mathbf{b}}} = \underbrace{\begin{bmatrix} 0.14 \\ 3.82 \\ -0.82 \end{bmatrix}}_{\mathbf{Wx}+\mathbf{b}}$$

Generalizing the Notations, cont.

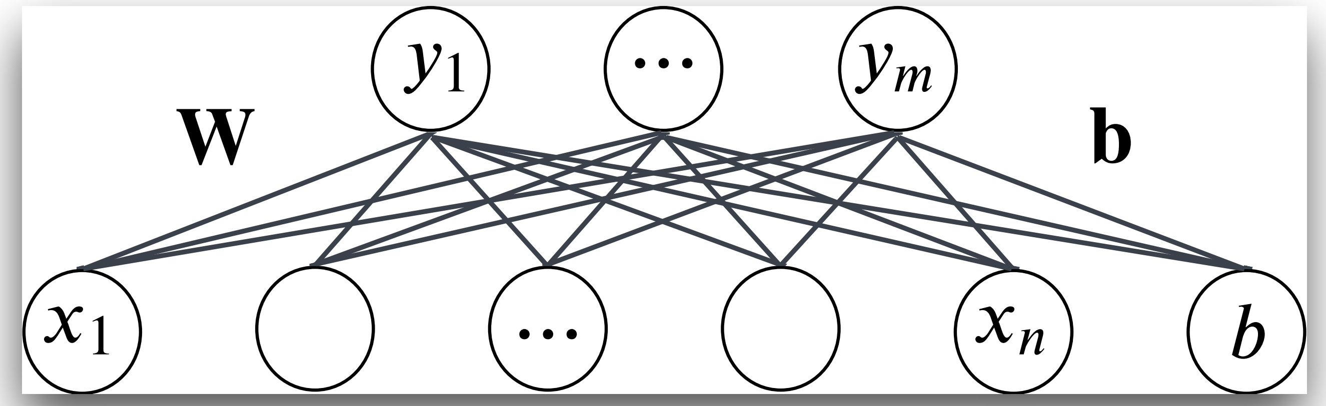


Handling the Bias Term

- As before, we incorporeate the bias vector $\mathbf{b}$ into $\mathbf{W}$ and $\mathbf{x}$ by adding a constant value 1 into the input vector, as follows:

$$\underbrace{\begin{bmatrix} 0.1 & 1.0 \\ 2.0 & -0.1 \\ -0.3 & 0.5 \end{bmatrix}}_{\mathbf{W}} \underbrace{\begin{bmatrix} 1.4 \\ -0.2 \end{bmatrix}}_{\mathbf{x}} + \underbrace{\begin{bmatrix} 0.2 \\ 1.0 \\ -0.3 \end{bmatrix}}_{\color{red}{\mathbf{b}}} = \underbrace{\begin{bmatrix} 0.14 \\ 3.82 \\ -0.82 \end{bmatrix}}_{\mathbf{Wx}+\mathbf{b}} \Rightarrow \underbrace{\begin{bmatrix} 0.1 & 1.0 & 0.2 \\ 2.0 & -0.1 & 1.0 \\ -0.3 & 0.5 & -0.3 \end{bmatrix}}_{\mathbf{W}} \underbrace{\begin{bmatrix} 1.4 \\ -0.2 \\ 1.0 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} 0.14 \\ 3.82 \\ -0.82 \end{bmatrix}}_{\mathbf{Wx}+\mathbf{b}}$$

Generalizing the Notations, cont.

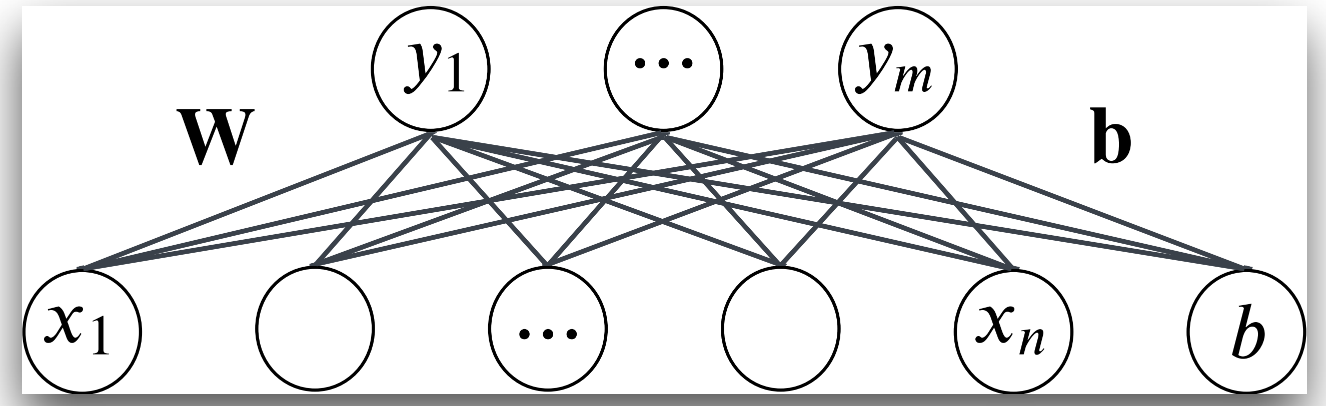


Handling the Bias Term

- As before, we incorporeate the bias vector $\mathbf{b}$ into $\mathbf{W}$ and $\mathbf{x}$ by adding a constant value 1 into the input vector, as follows:

$$\underbrace{\begin{bmatrix} 0.1 & 1.0 \\ 2.0 & -0.1 \\ -0.3 & 0.5 \end{bmatrix}}_{\mathbf{W}} \underbrace{\begin{bmatrix} 1.4 \\ -0.2 \end{bmatrix}}_{\mathbf{x}} + \underbrace{\begin{bmatrix} 0.2 \\ 1.0 \\ -0.3 \end{bmatrix}}_{\color{red}{\mathbf{b}}} = \underbrace{\begin{bmatrix} 0.14 \\ 3.82 \\ -0.82 \end{bmatrix}}_{\mathbf{Wx}+\mathbf{b}} \Rightarrow \left[\underbrace{\begin{matrix} 0.1 & 1.0 \\ 2.0 & -0.1 \\ -0.3 & 0.5 \end{matrix}}_{\mathbf{W}} \; \underbrace{\boxed{\begin{matrix} 0.2 \\ 1.0 \\ -0.3 \end{matrix}}}_{\color{red}{\mathbf{b}}}\right] \underbrace{\begin{bmatrix} 1.4 \\ -0.2 \\ 1.0 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} 0.14 \\ 3.82 \\ -0.82 \end{bmatrix}}_{\mathbf{Wx}+\mathbf{b}}$$

Generalizing the Notations, cont.



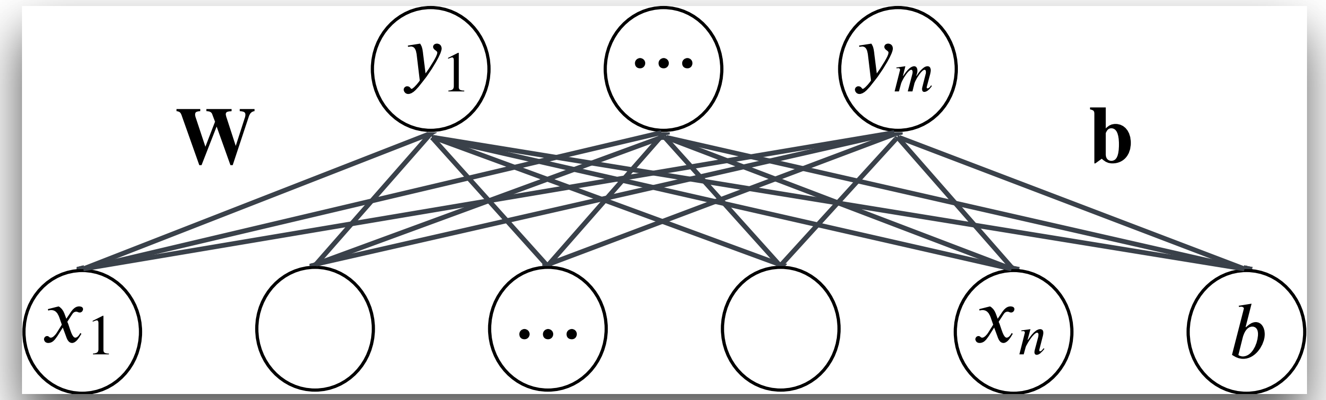
Handling the Bias Term

- As before, we incorporeate the bias vector $\mathbf{b}$ into $\mathbf{W}$ and $\mathbf{x}$ by adding a constant value 1 into the input vector, as follows:

$$\underbrace{\begin{bmatrix} 0.1 & 1.0 \\ 2.0 & -0.1 \\ -0.3 & 0.5 \end{bmatrix}}_{\mathbf{W}} \underbrace{\begin{bmatrix} 1.4 \\ -0.2 \end{bmatrix}}_{\mathbf{x}} + \underbrace{\begin{bmatrix} 0.2 \\ 1.0 \\ -0.3 \end{bmatrix}}_{\color{red}{\mathbf{b}}} = \underbrace{\begin{bmatrix} 0.14 \\ 3.82 \\ -0.82 \end{bmatrix}}_{\mathbf{Wx}+\mathbf{b}} \Rightarrow \underbrace{\begin{bmatrix} 0.1 & 1.0 & \boxed{0.2} \\ 2.0 & -0.1 & \boxed{1.0} \\ -0.3 & 0.5 & \boxed{-0.3} \end{bmatrix}}_{\mathbf{W}\ \ \color{red}{\mathbf{b}}} \underbrace{\begin{bmatrix} 1.4 \\ -0.2 \\ \boxed{1.0} \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} 0.14 \\ 3.82 \\ -0.82 \end{bmatrix}}_{\mathbf{Wx}+\mathbf{b}}$$

Generalizing the Notations, cont.

Handling the Bias Term



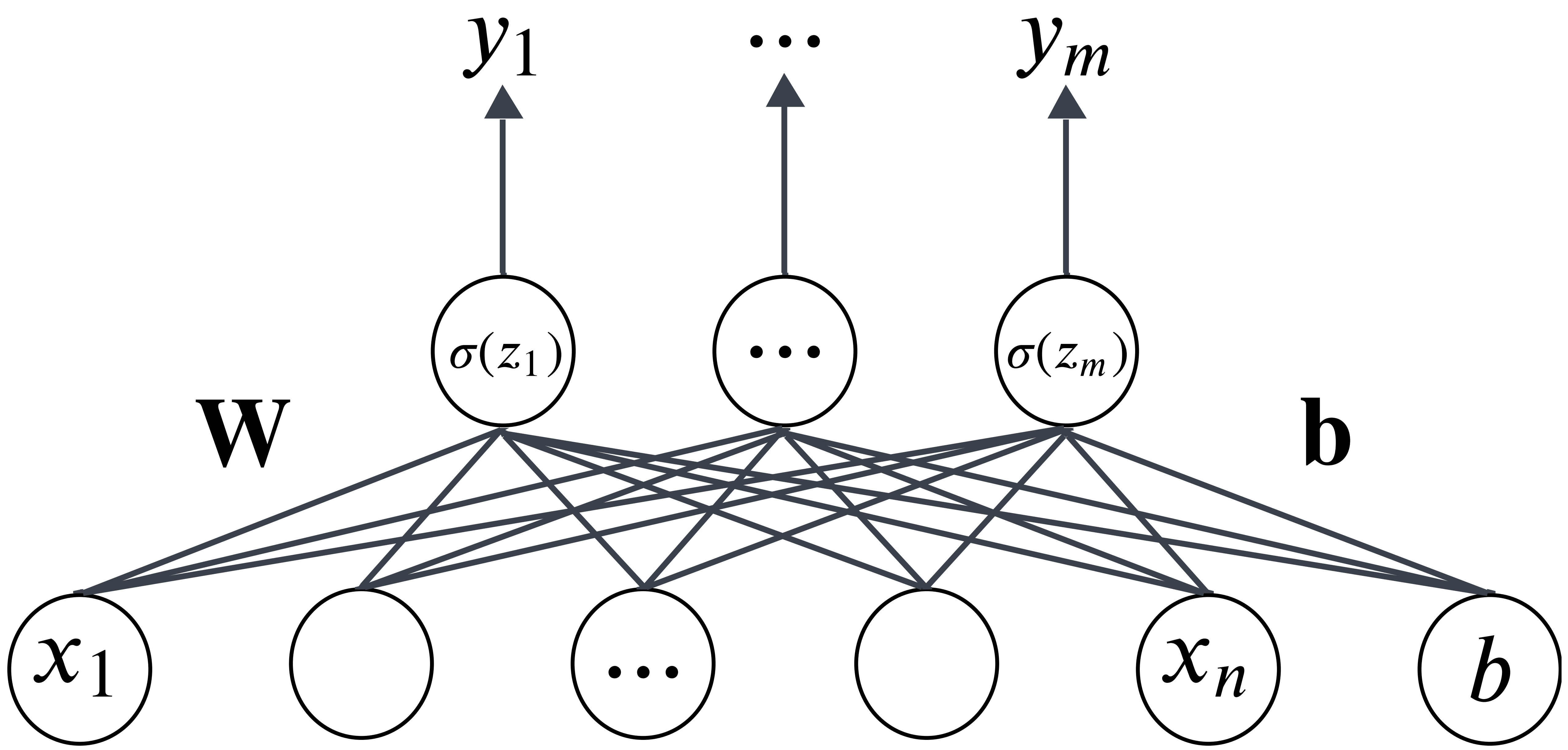
- As before, we incorporeate the bias vector $\mathbf{b}$ into $\mathbf{W}$ and $\mathbf{x}$ by adding a constant value 1 into the input vector, as follows:

$$\underset{\mathbf{W}}{\begin{bmatrix} 0.1 & 1.0 \\ 2.0 & -0.1 \\ -0.3 & 0.5 \end{bmatrix}} \underset{\mathbf{x}}{\begin{bmatrix} 1.4 \\ -0.2 \end{bmatrix}} + \underset{\mathbf{b}}{\begin{bmatrix} 0.2 \\ 1.0 \\ -0.3 \end{bmatrix}} = \underset{\mathbf{Wx}+\mathbf{b}}{\begin{bmatrix} 0.14 \\ 3.82 \\ -0.82 \end{bmatrix}} \Rightarrow \underset{\mathbf{W}\qquad\qquad\mathbf{b}}{\begin{bmatrix} 0.1 & 1.0 & 0.2 \\ 2.0 & -0.1 & 1.0 \\ -0.3 & 0.5 & -0.3 \end{bmatrix}} \underset{\mathbf{x}}{\begin{bmatrix} 1.4 \\ -0.2 \\ 1.0 \end{bmatrix}} = \underset{\mathbf{Wx}+\mathbf{b}}{\begin{bmatrix} 0.14 \\ 3.82 \\ -0.82 \end{bmatrix}}$$

Add $\mathbf{b}$ as another column of $\mathbf{W}$, and stick a 1 to the input vector $\mathbf{x}$.

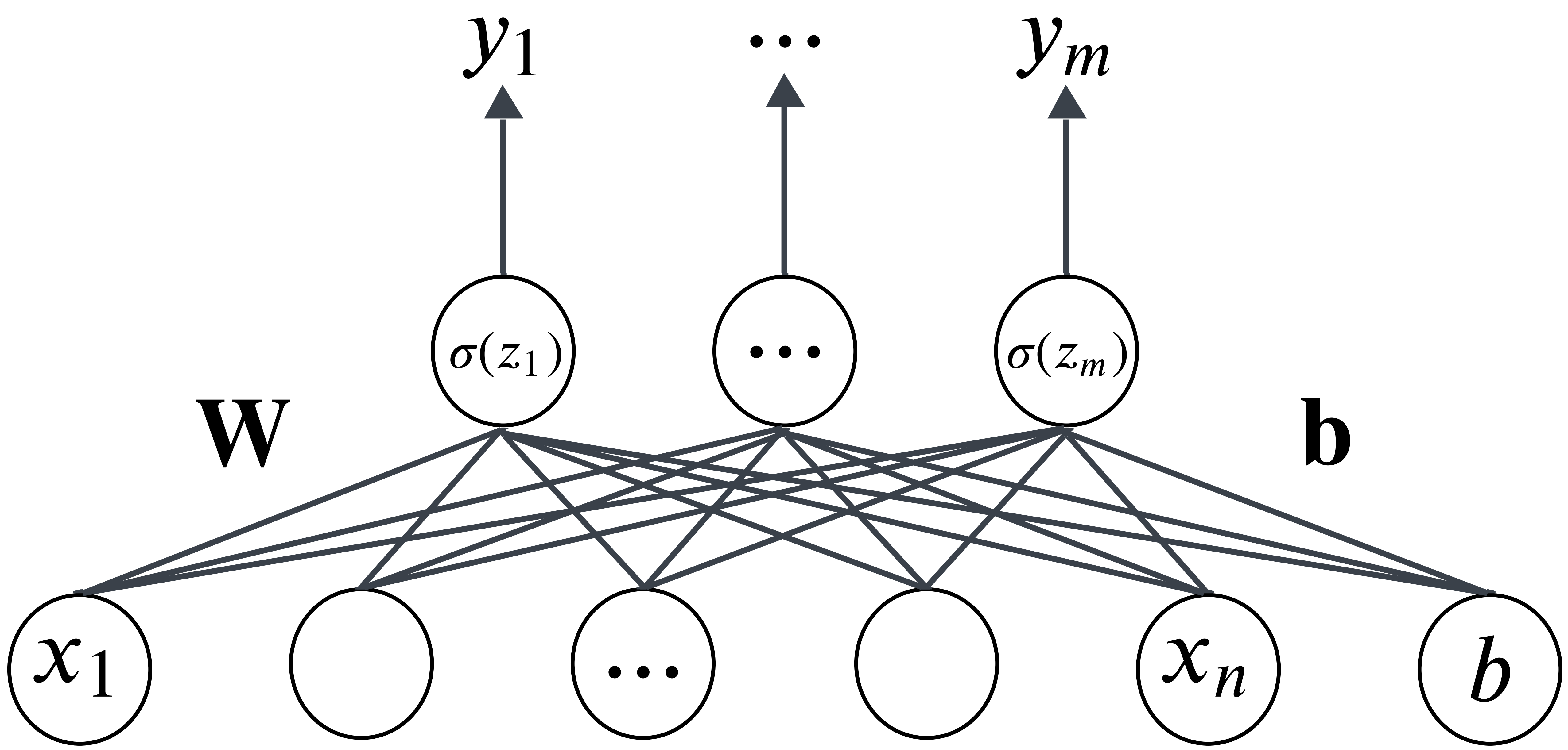
Generalizing the Notations, cont.

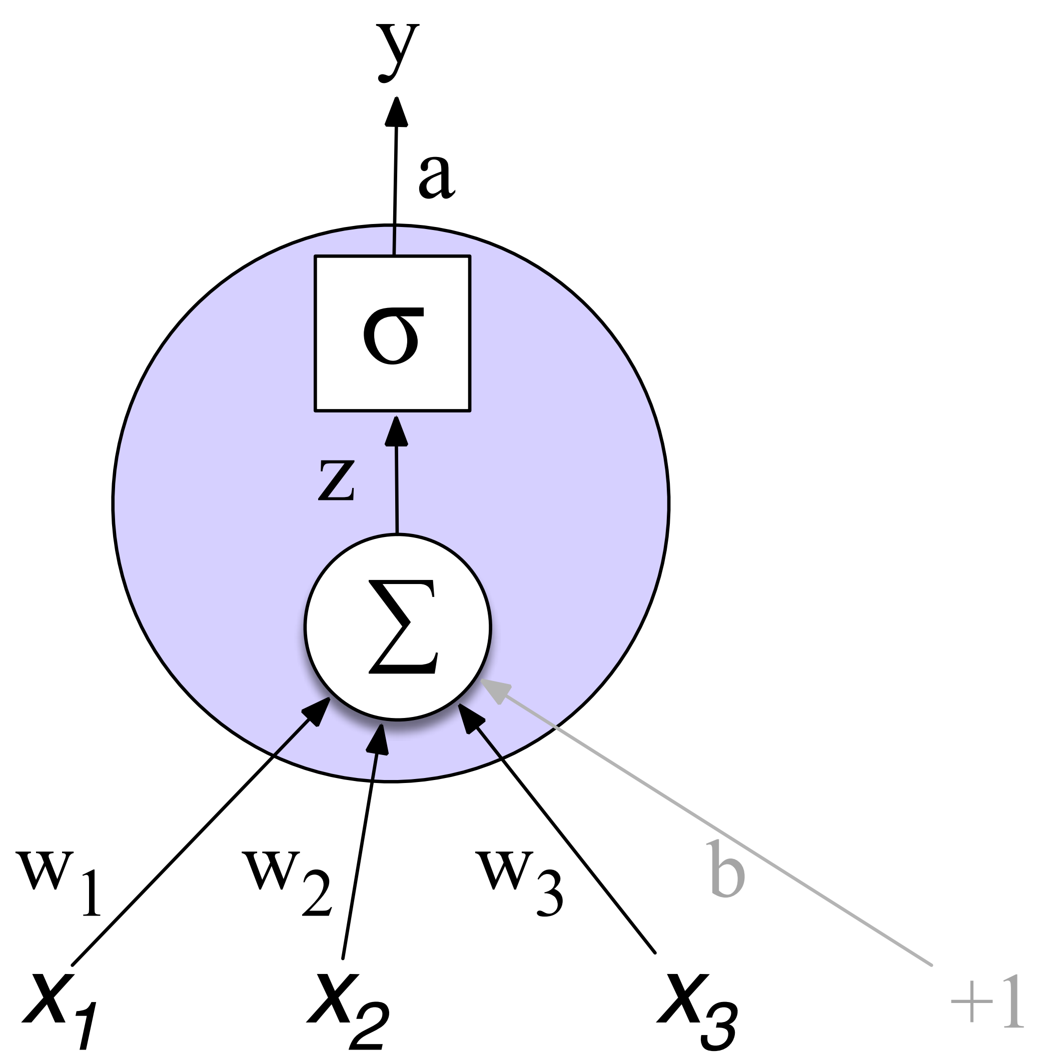
Folding the nonlinear transformation into the neurons



Generalizing the Notations, cont.

Folding the nonlinear transformation into the neurons



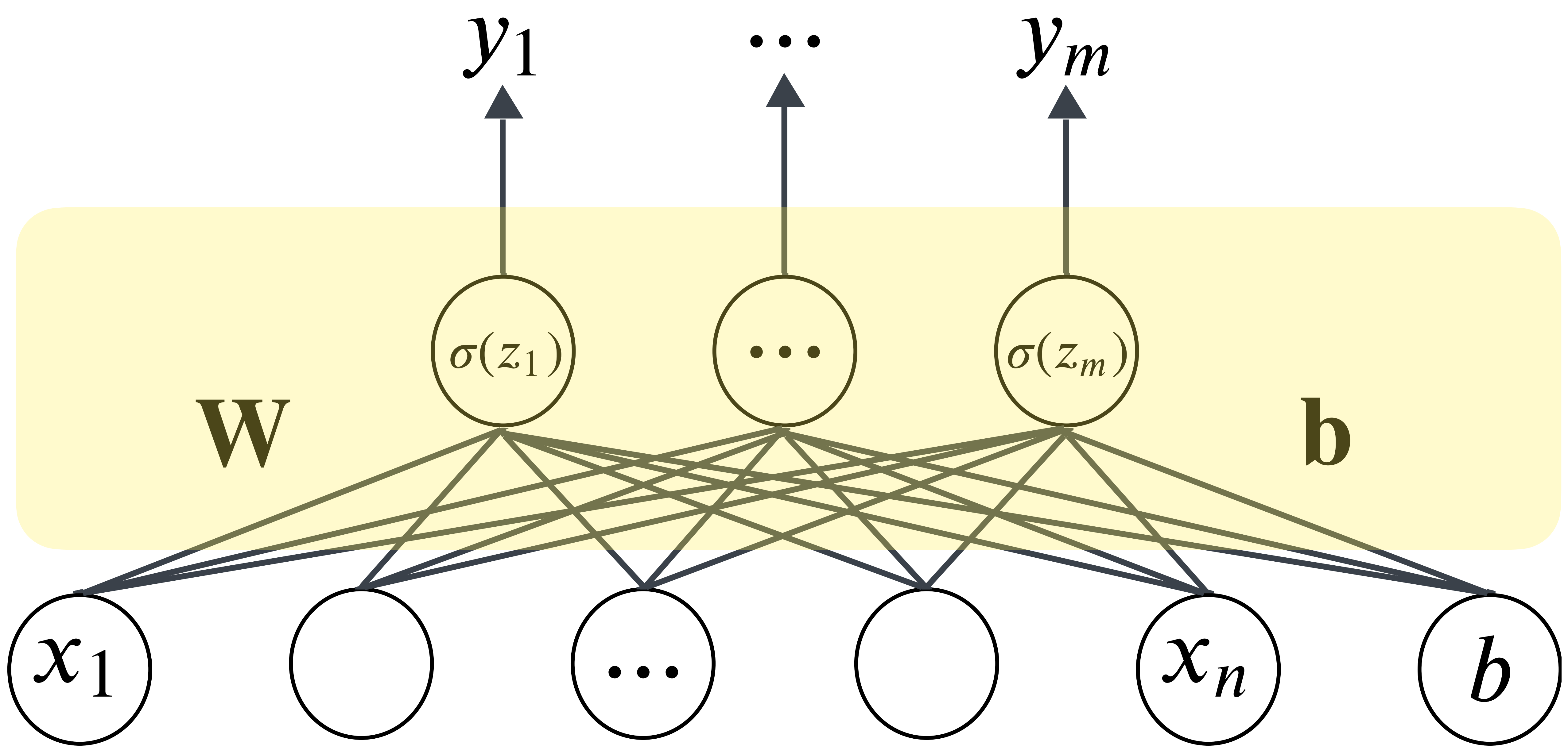


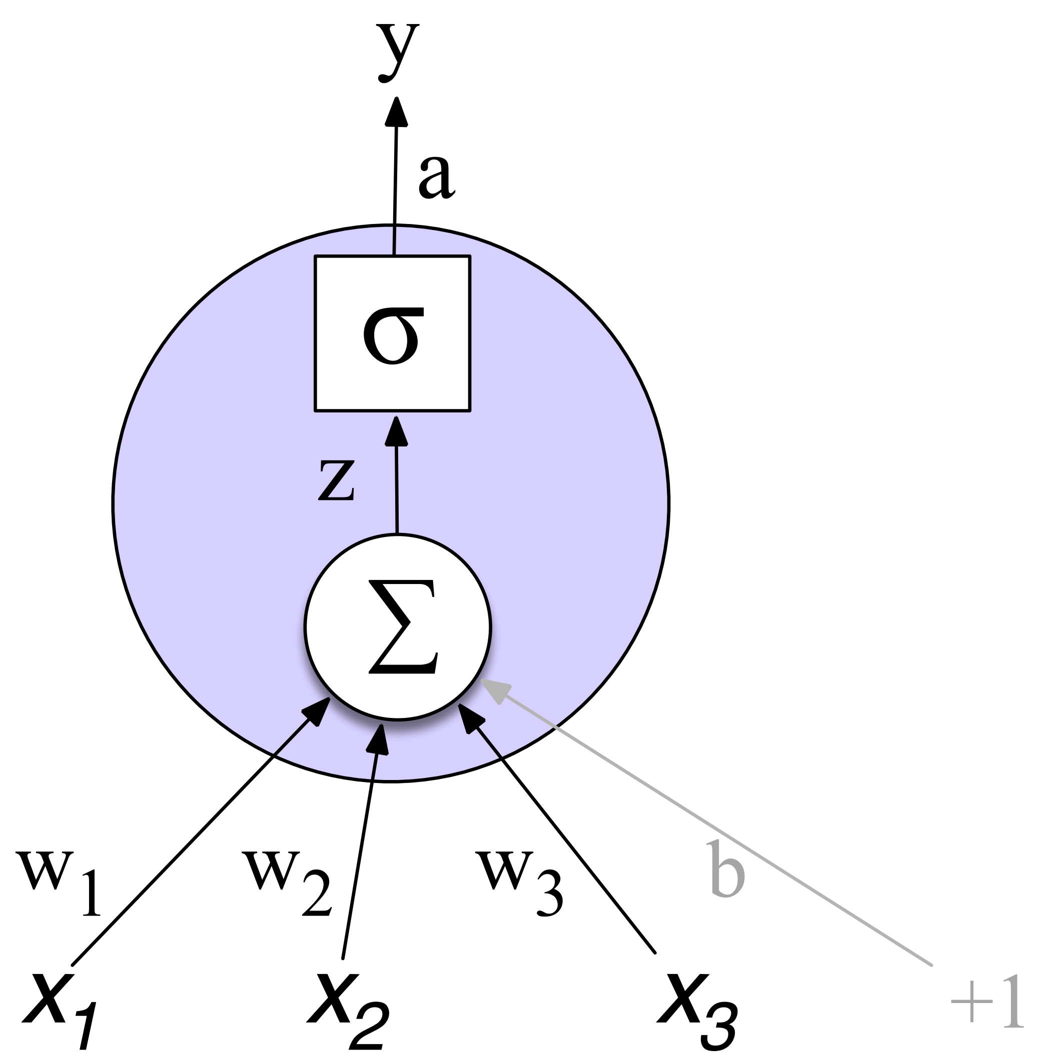
- A neuron in a neural network computes the linearly transformed weighted sum: $f(\mathbf{W}\mathbf{x} + \mathbf{b})$, where f can be softmax, sigmoid, ReLU, tanh, etc.

Generalizing the Notations, cont.

Folding the nonlinear transformation into the neurons

Layer 1

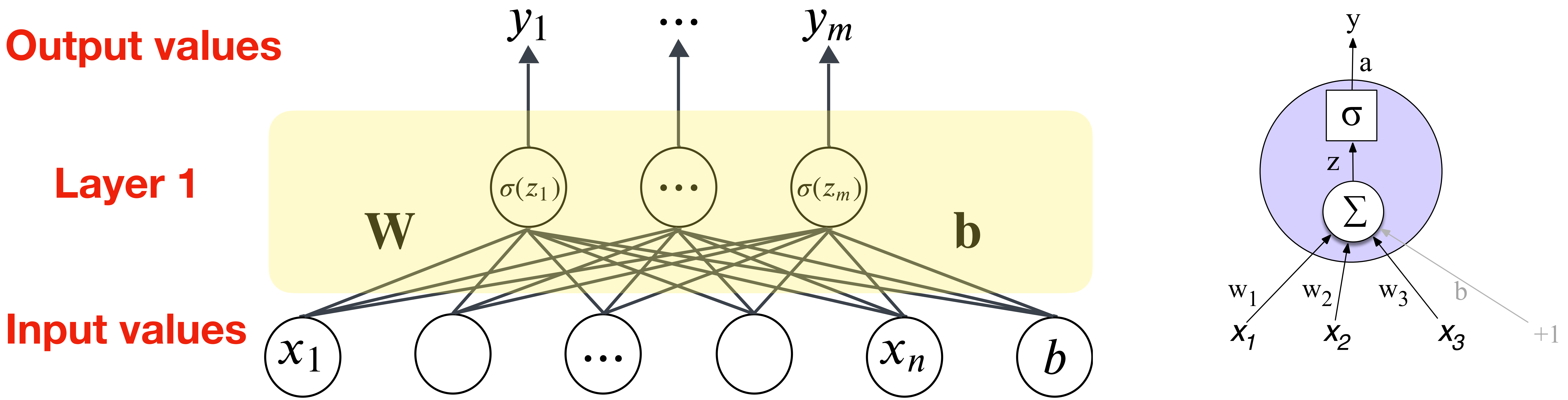




- A neuron in a neural network computes the linearly transformed weighted sum: $f(\mathbf{Wx} + \mathbf{b})$, where f can be softmax, sigmoid, ReLU, tanh, etc.
- A layer is a list of neurons, defined by $\mathbf{W}$, $\mathbf{b}$, and the nonlinear transformation.

Generalizing the Notations, cont.

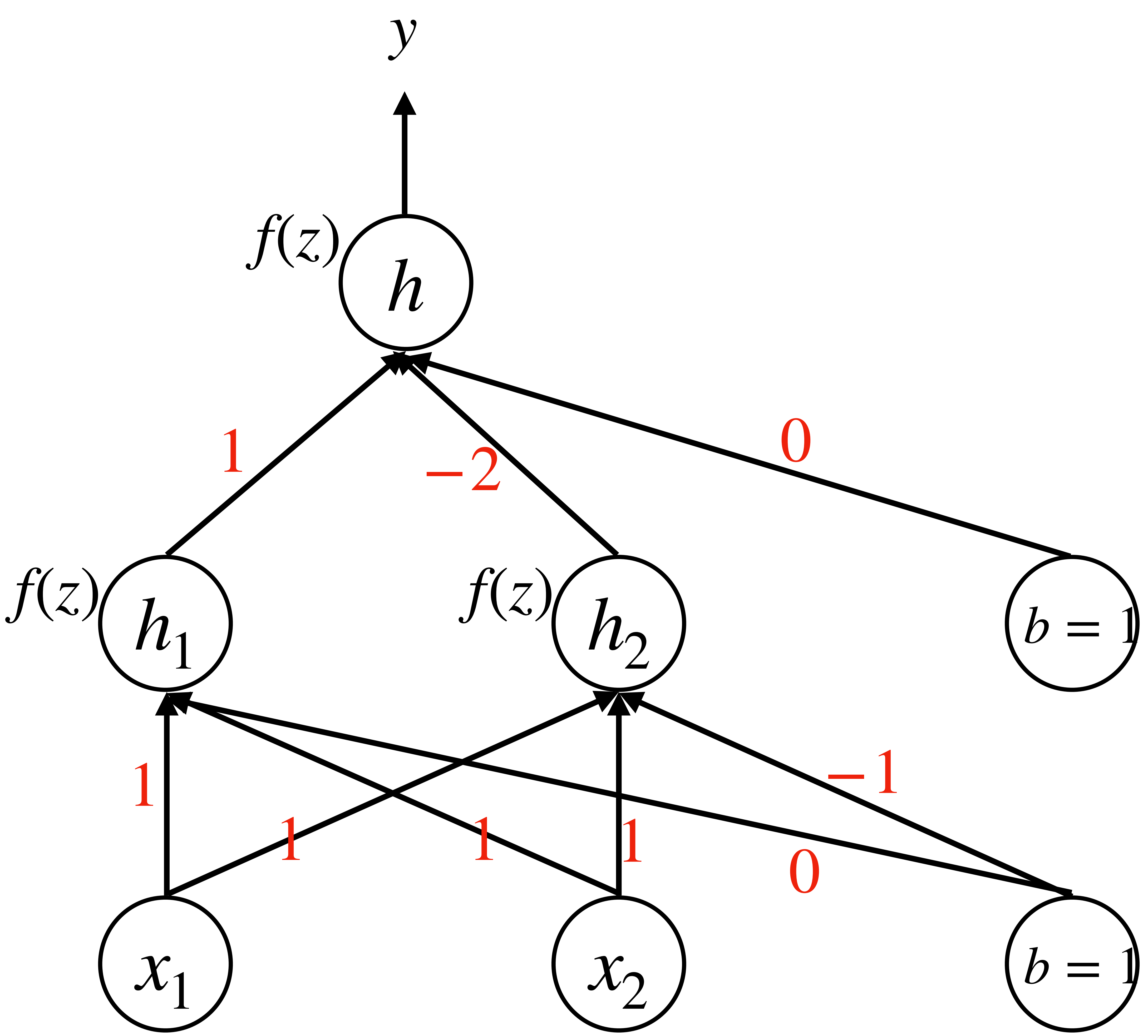
Folding the nonlinear transformation into the neurons



- A neuron in a neural network computes the linearly transformed weighted sum: $f(\mathbf{Wx} + \mathbf{b})$, where f can be softmax, sigmoid, ReLU, tanh, etc.
- A layer is a list of neurons, defined by $\mathbf{W}$, $\mathbf{b}$, and the nonlinear transformation.
- Input nodes ($\mathbf{x}$) don’t count as a layer; and output values also don’t.

Generalizing the Notations, cont.

Our XOR network is a two-layer neural network:

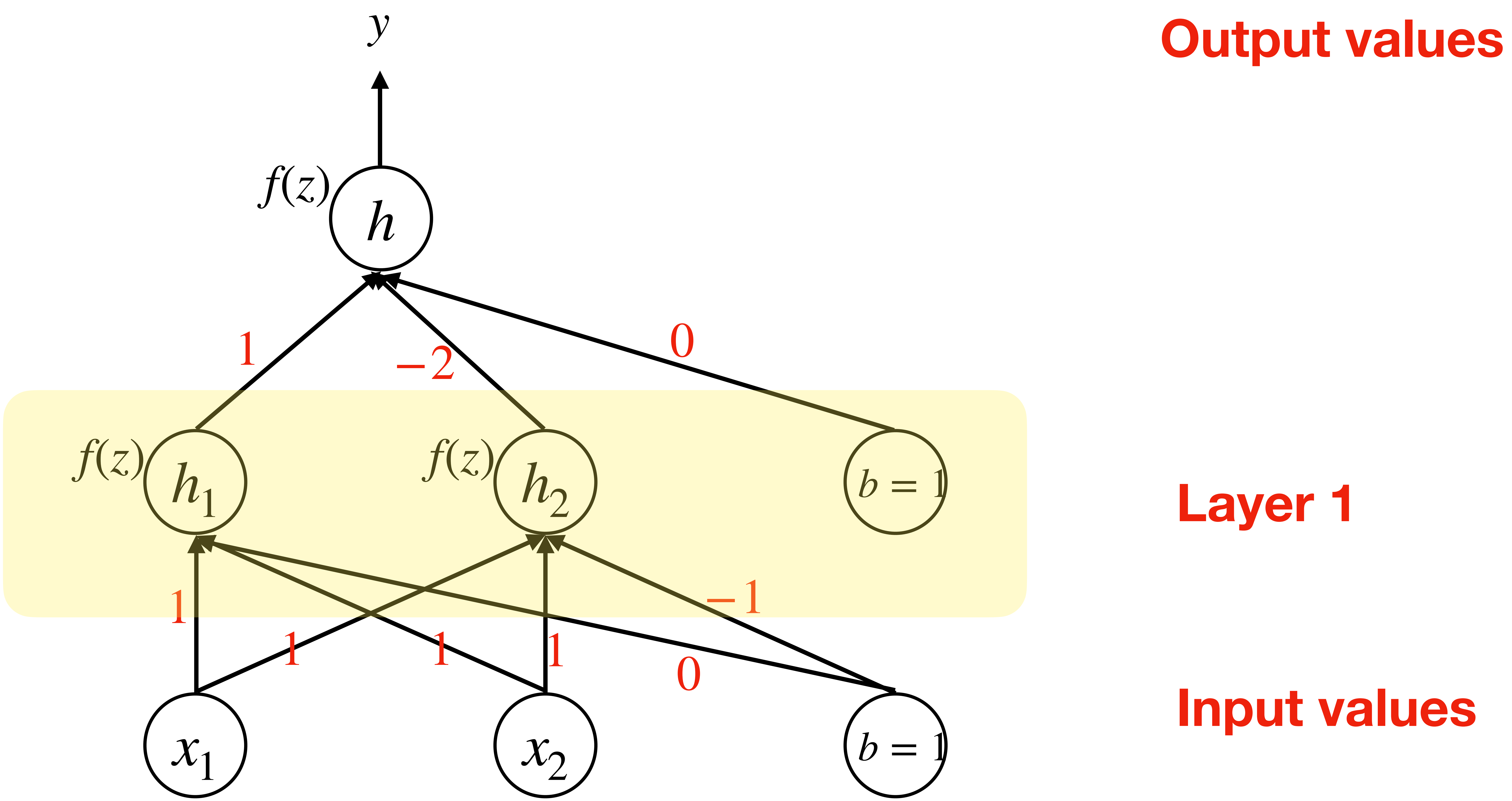


Output values

Input values

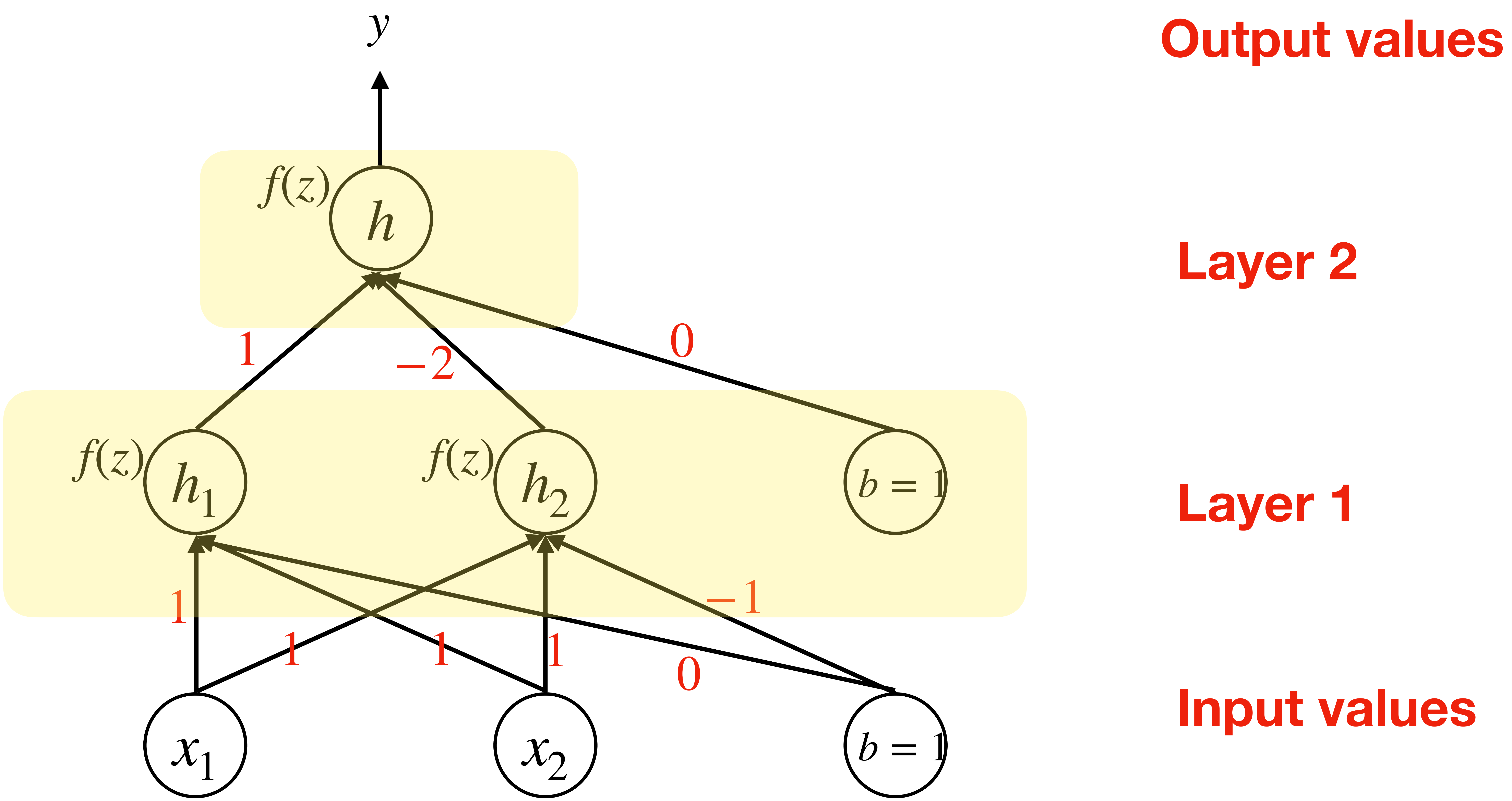
Generalizing the Notations, cont.

Our XOR network is a two-layer neural network:



Generalizing the Notations, cont.

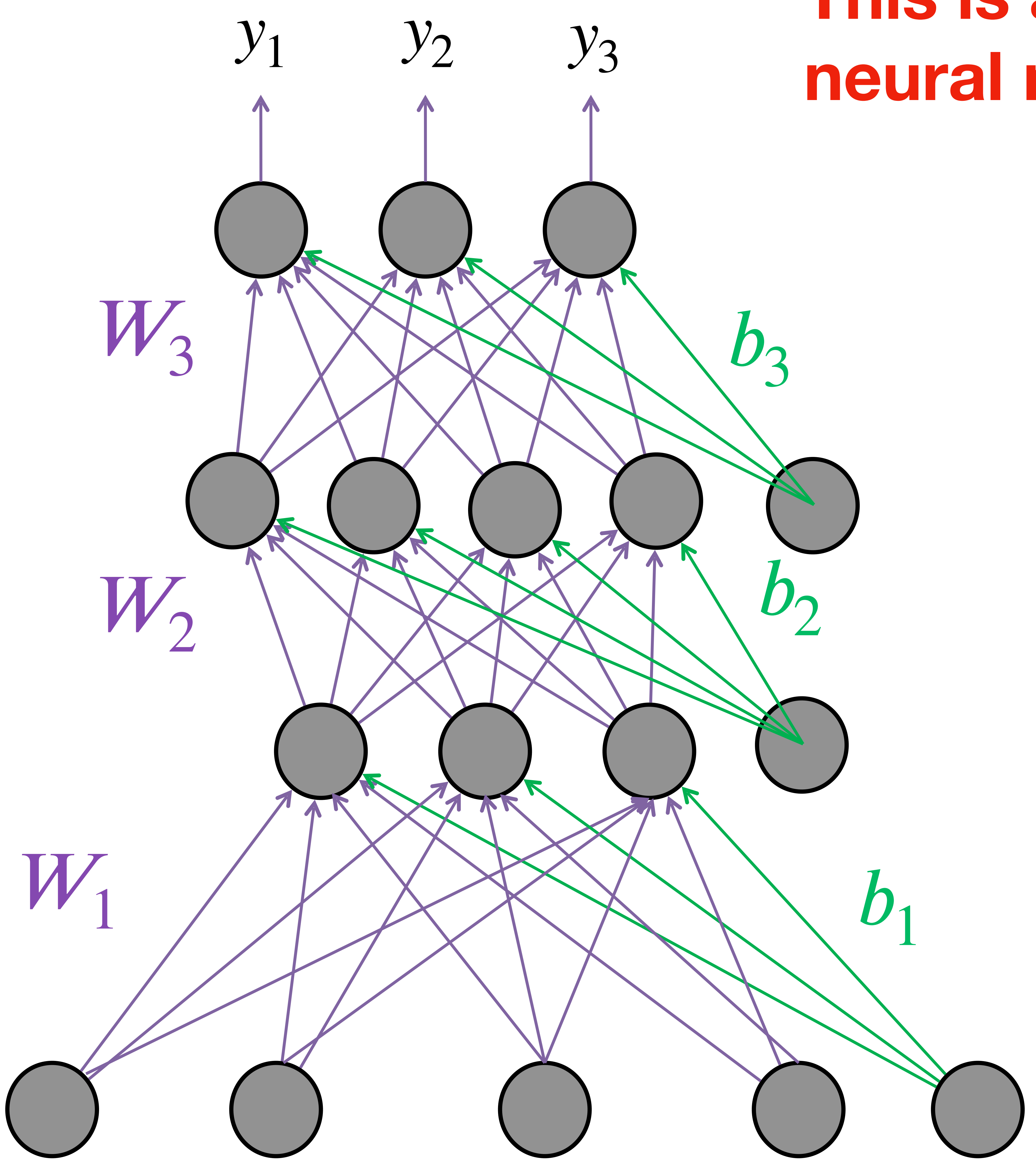
Our XOR network is a two-layer neural network:



Generalizing the Notations, cont.

Multi-layer Neural Networks

This is a 3-layer neural network

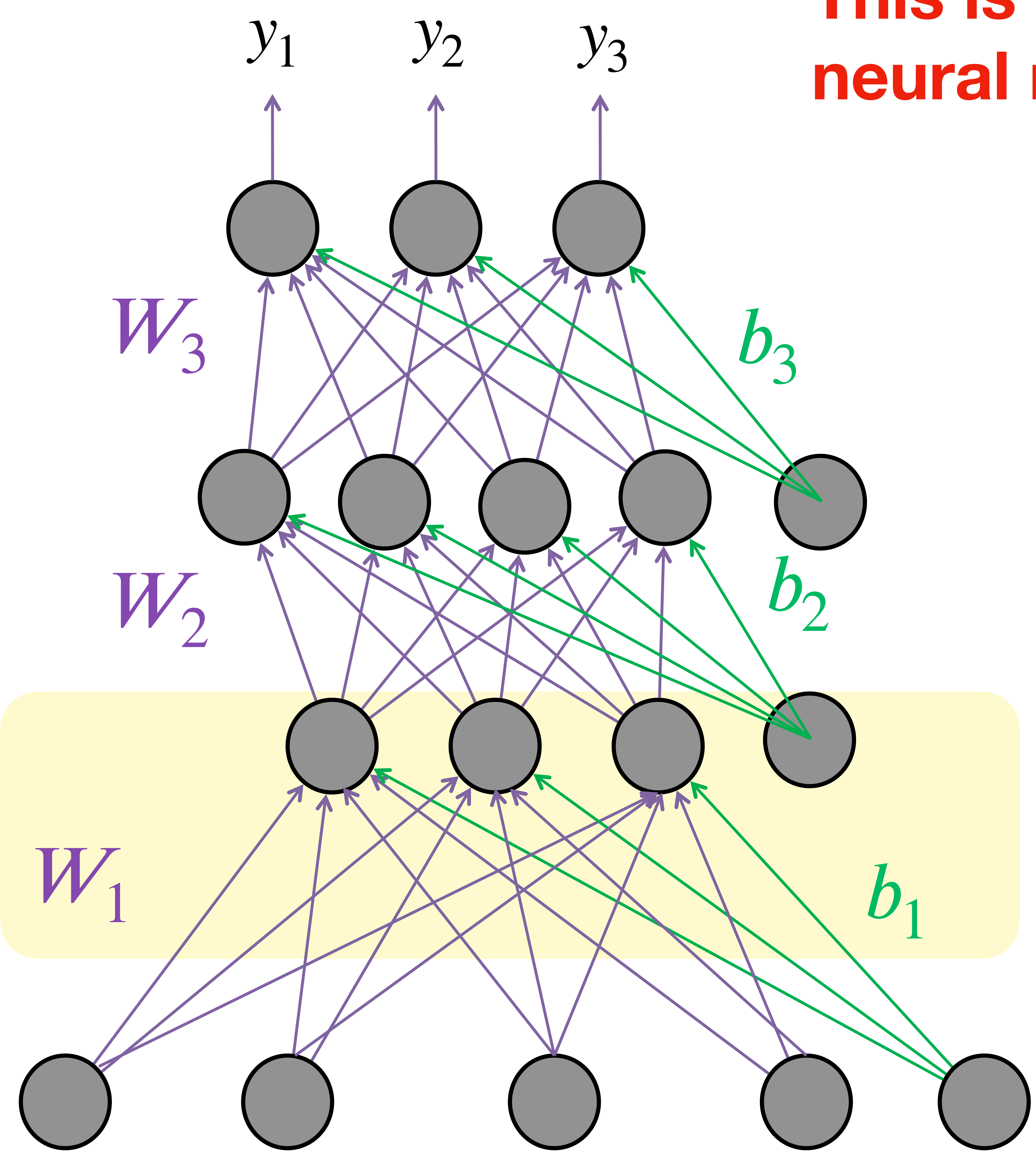


Input layer $\mathbf{x}$ (vector)

Generalizing the Notations, cont.

Multi-layer Neural Networks

This is a 3-layer neural network



Hidden state $\mathbf{h_1}$ (vector)

$$\mathbf{h_1} = ReLU(\mathbf{W_1 x} + \mathbf{b_1})$$

Input layer $\mathbf{x}$ (vector)

Generalizing the Notations, cont.

Multi-layer Neural Networks

This is a 3-layer neural network

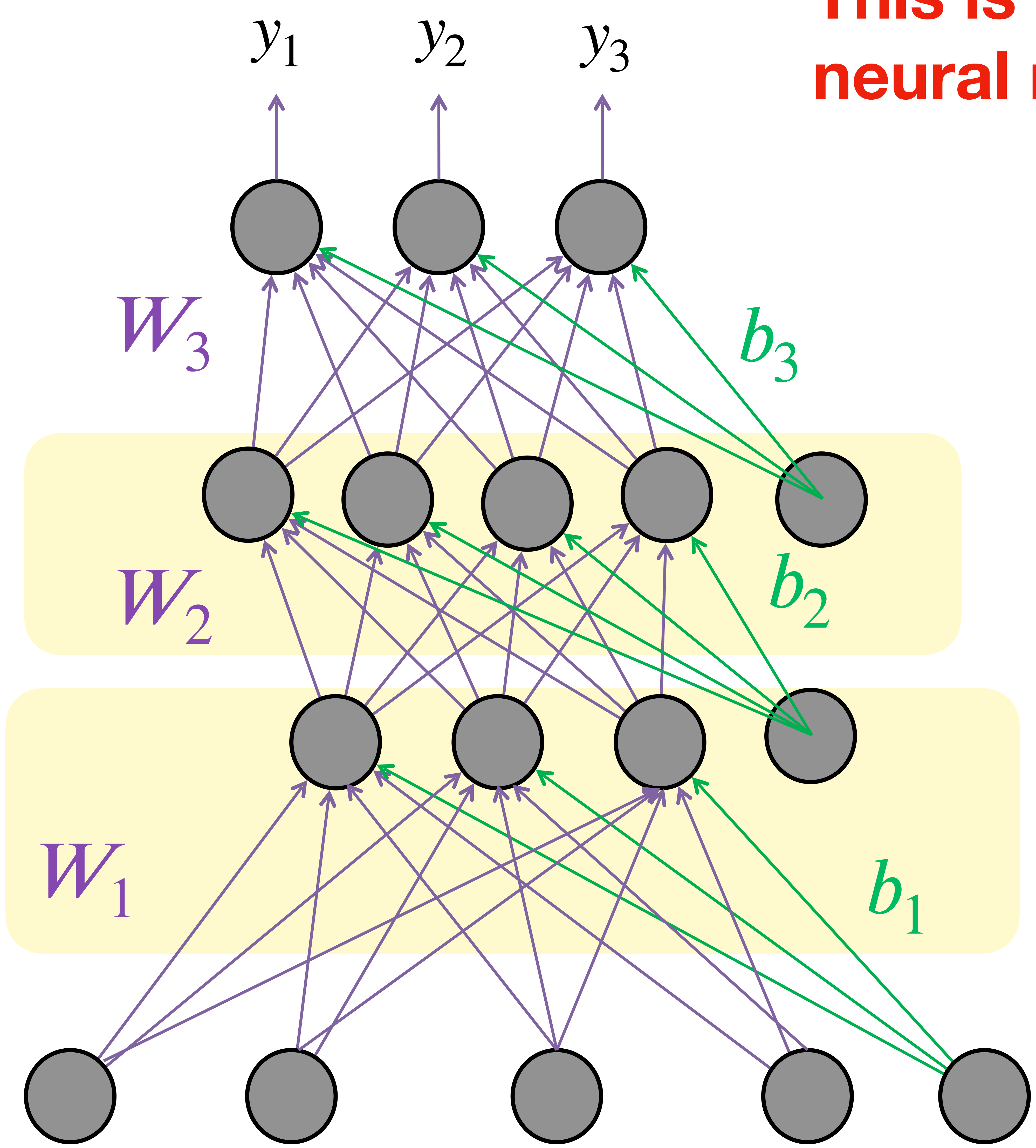
Hidden state $\mathbf{h_2}$ (vector)

$$\mathbf{h_2} = ReLU(\mathbf{W_2 h_1} + \mathbf{b_2})$$

Hidden state $\mathbf{h_1}$ (vector)

$$\mathbf{h_1} = ReLU(\mathbf{W_1 x} + \mathbf{b_1})$$

Input layer $\mathbf{x}$ (vector)



Generalizing the Notations, cont.

Multi-layer Neural Networks

This is a 3-layer neural network

$$\mathbf{y} = softmax(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$$

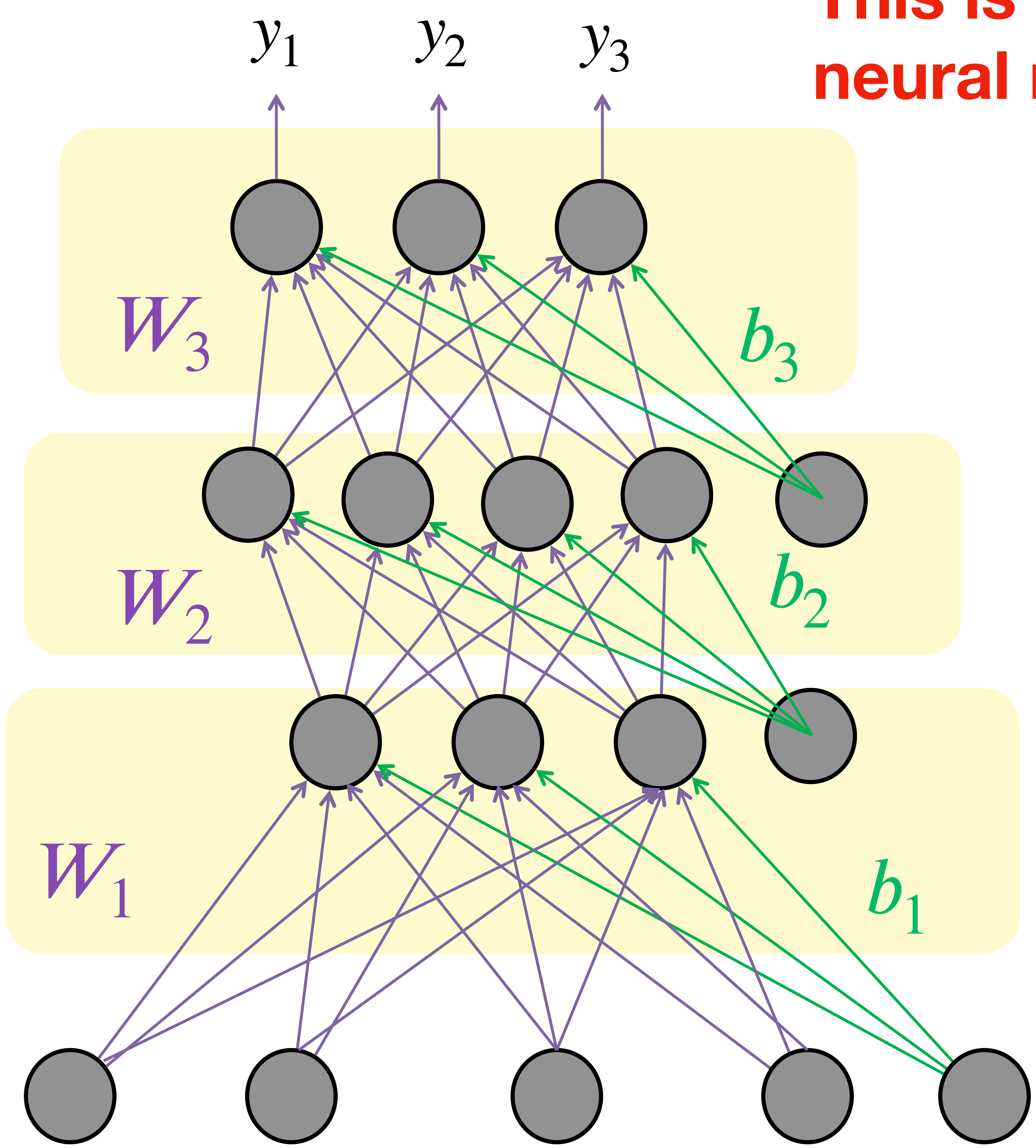
Hidden state $\mathbf{h}_2$ (vector)

$$\mathbf{h}_2 = ReLU(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$$

Hidden state $\mathbf{h}_1$ (vector)

$$\mathbf{h}_1 = ReLU(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

Input layer $\mathbf{x}$ (vector)



Generalizing the Notations, cont.

Multi-layer Neural Networks

Output $\mathbf{y}$ (vector)

$$\mathbf{y} = softmax(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$$

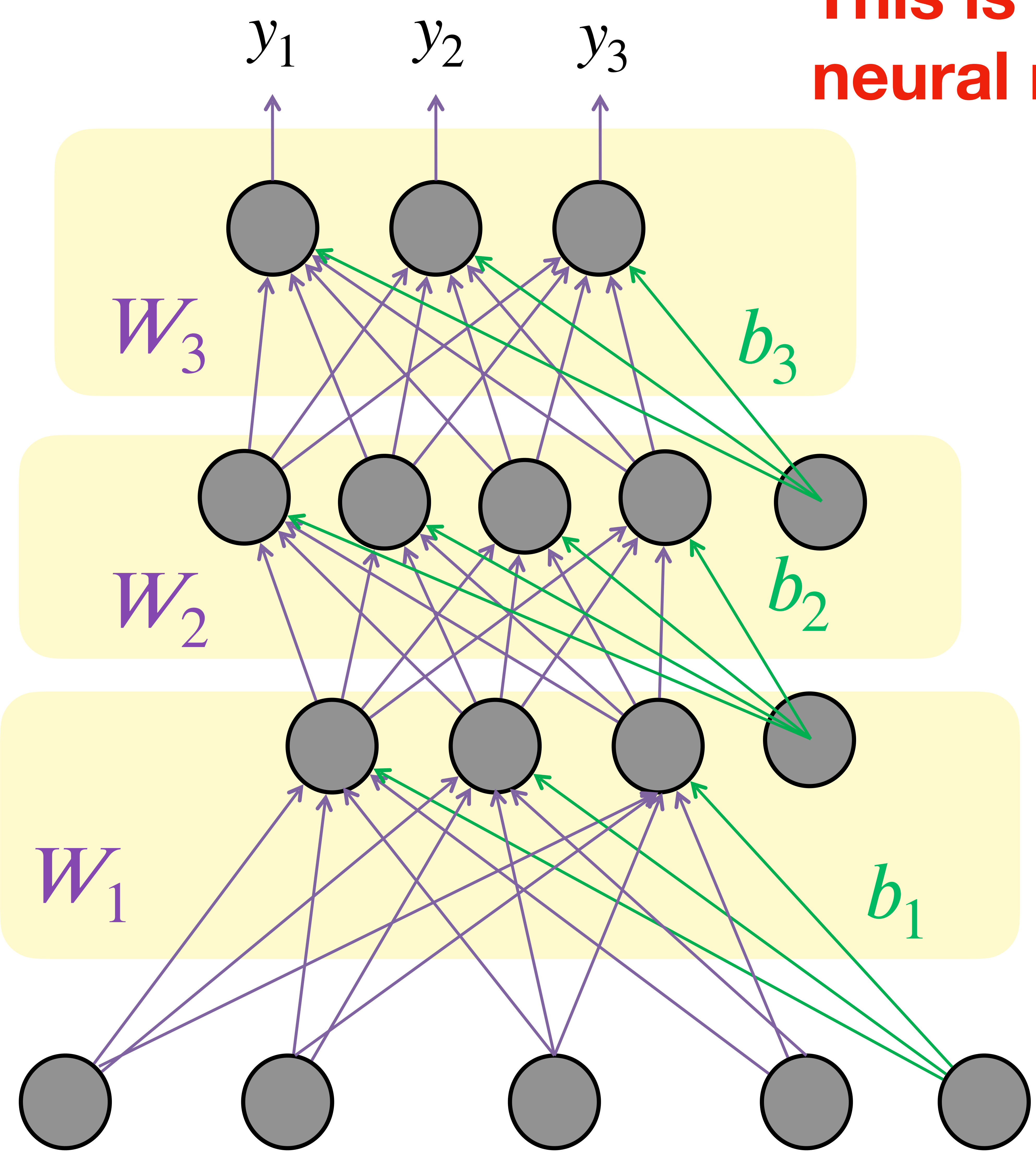
Hidden state $\mathbf{h}_2$ (vector)

$$\mathbf{h}_2 = ReLU(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$$

Hidden state $\mathbf{h}_1$ (vector)

$$\mathbf{h}_1 = ReLU(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

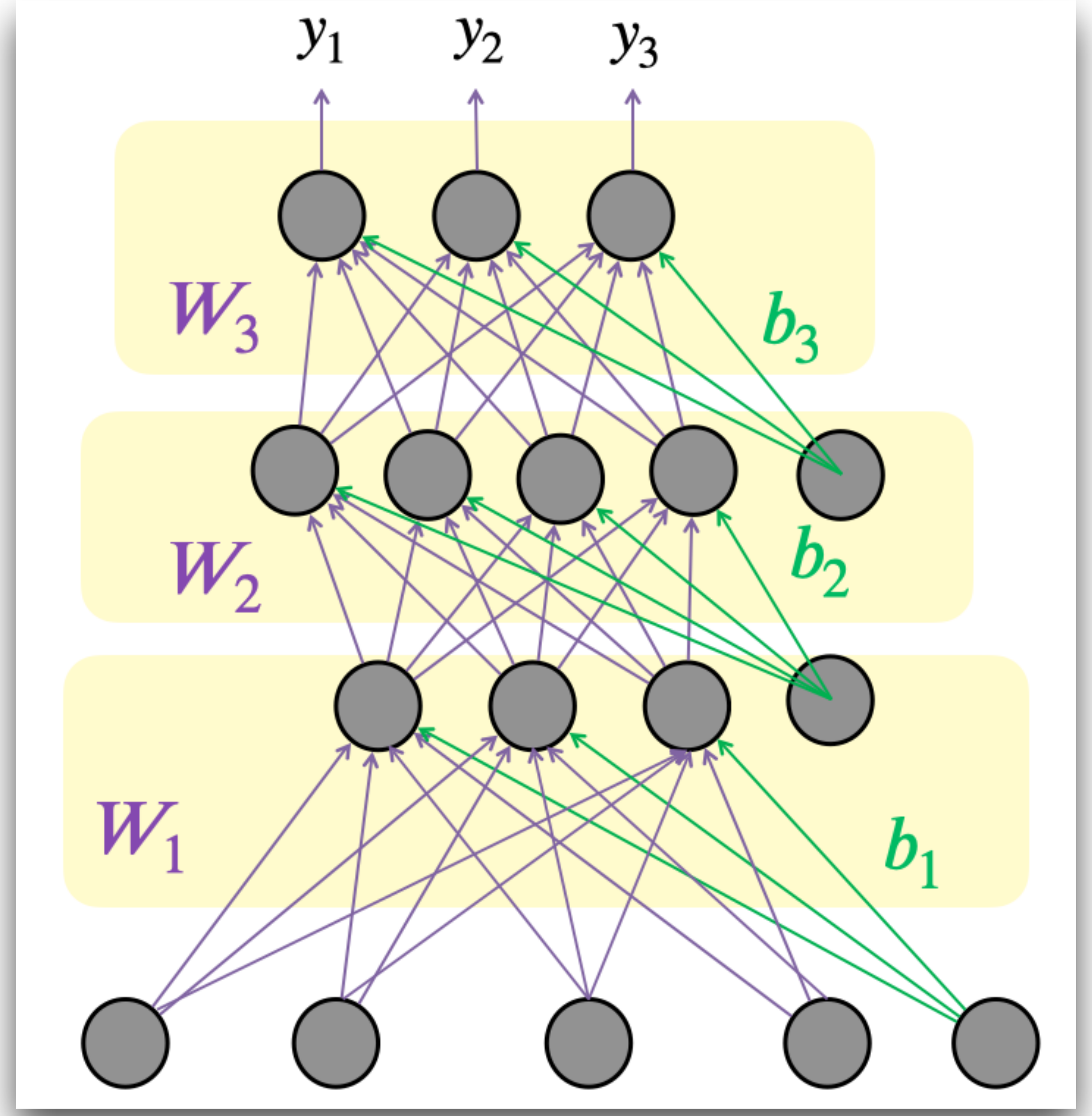
Input layer $\mathbf{x}$ (vector)



This is a 3-layer neural network

Terminology for this type of Neural Networks

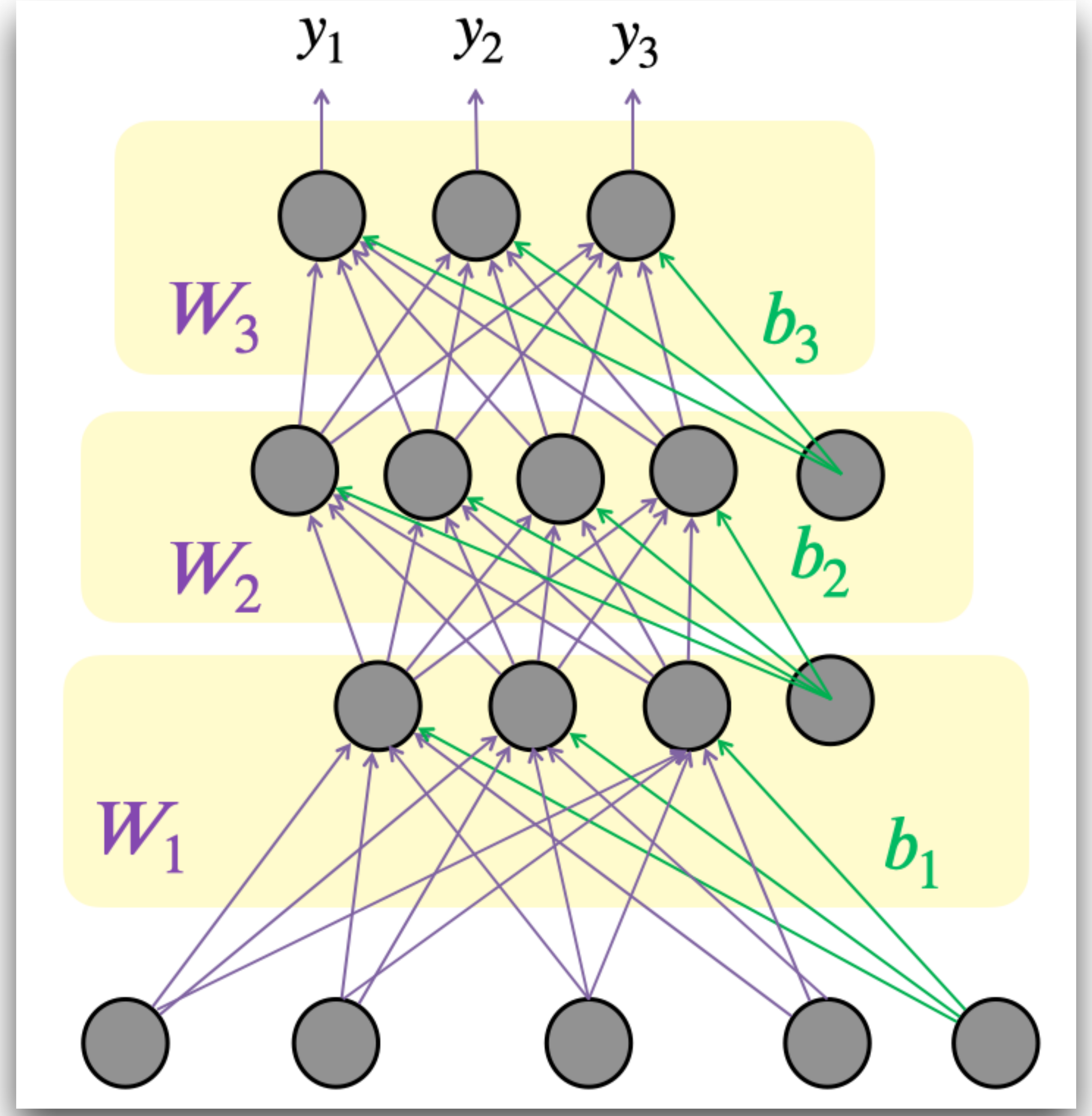
Different names for the same architecture



Terminology for this type of Neural Networks

Different names for the same architecture

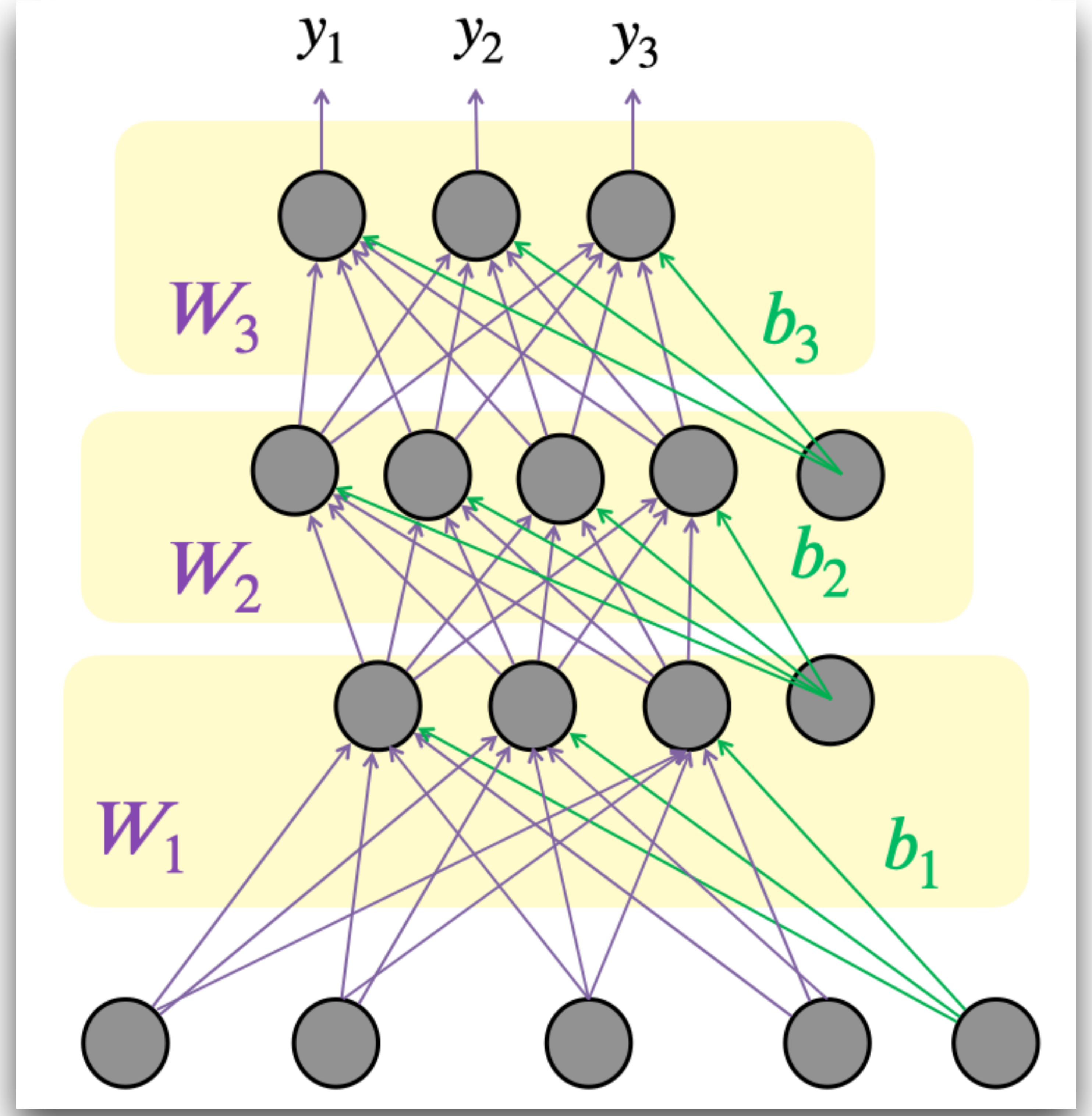
- **Feedforward Neural Network (FFNN)**
  - ✳ Because layers are feeding foward to each other — the output of layer l becomes the input of layer $l+1$;



Terminology for this type of Neural Networks

Different names for the same architecture

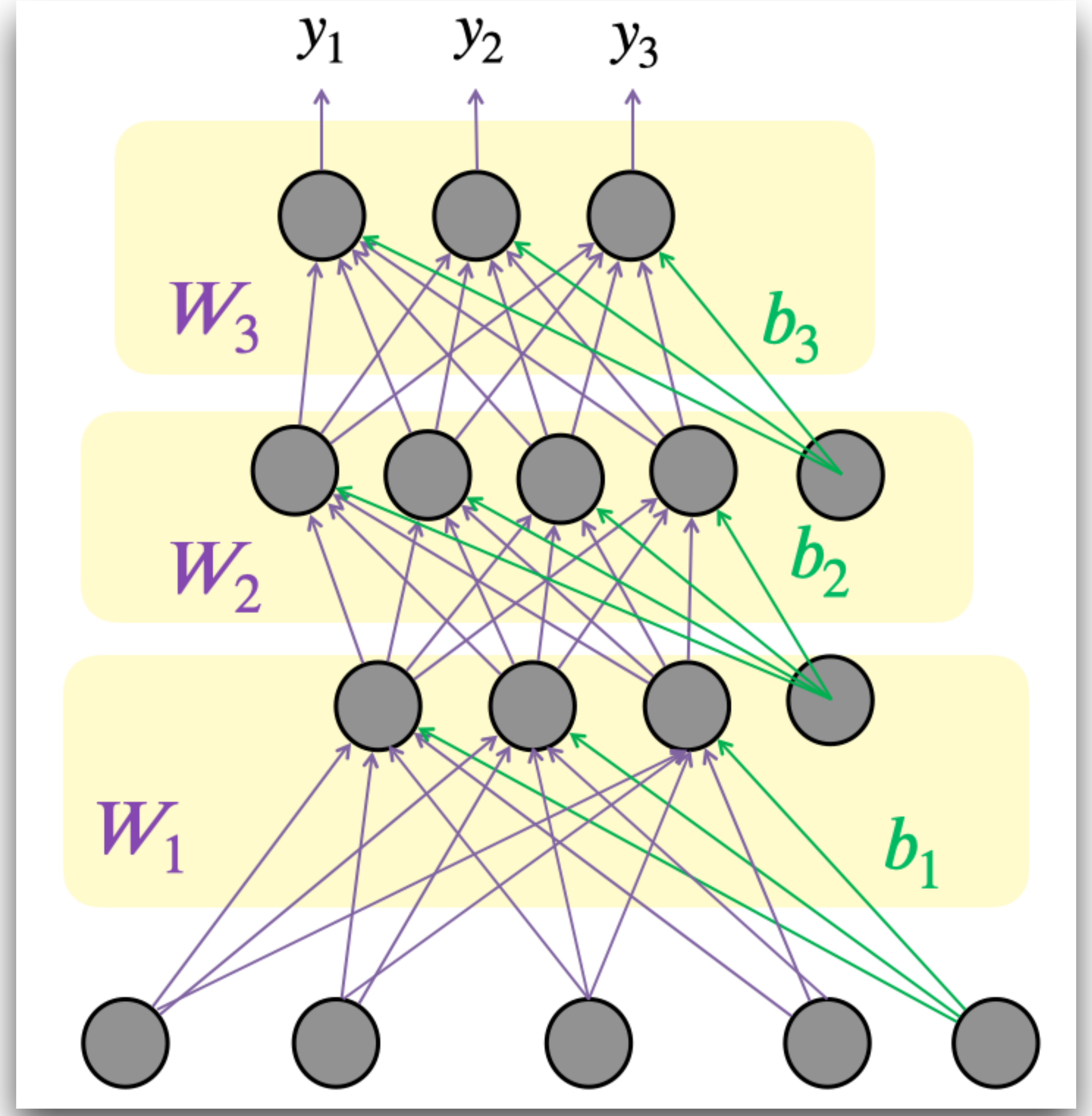
- **Feedforward Neural Network (FFNN)**
  - ✳ Because layers are feeding foward to each other — the output of layer l becomes the input of layer $l+1$;
- **Fully-connected Neural Network (FCNN)**
  - ✳ From layer l to layer $l+1$, all nodes are connected to all other nodes;



Terminology for this type of Neural Networks

Different names for the same architecture

- **Feedforward Neural Network (FFNN)**
  - ✳ Because layers are feeding foward to each other — the output of layer l becomes the input of layer $l+1$;
- **Fully-connected Neural Network (FCNN)**
  - ✳ From layer l to layer $l+1$, all nodes are connected to all other nodes;
- **Multi-layer Perceptron (MLP)**
  - ✳ This generalizes a single-neuron Perceptron into a multi-neuron, multi-layer network



Terminology for this type of Neural Networks

More generally, at the field level

Terminology for this type of Neural Networks

More generally, at the field level

- **Connectionist Models**
  - ✳ Named after the school of thought “connectionism” (vs. “symbolism);

Terminology for this type of Neural Networks

More generally, at the field level

- **Connectionist Models**
  - ✳ Named after the school of thought “connectionism” (vs. “symbolism);
- **Parallel Distributed Processing (PDP)**
  - ✳ Each input is multiplied by a weight, and such computations are independent from each other, thus can be done in parallel / simultaneously.

Terminology for this type of Neural Networks

More generally, at the field level

- **Connectionist Models**
  - ✳ Named after the school of thought “connectionism” (vs. “symbolism);
- **Parallel Distributed Processing (PDP)**
  - ✳ Each input is multiplied by a weight, and such computations are independent from each other, thus can be done in parallel / simultaneously.
- **Deep Learning**
  - ✳ Recent branding for neural networks with many layers;

Expressivity of Neural Networks

A significant upgrade

Expressivity of Neural Networks

A significant upgrade

- 1-layer Perceptrons are very limited;

Expressivity of Neural Networks

A significant upgrade

- 1-layer Perceptrons are very limited;
- But having one more layer makes it much more powerful!

Expressivity of Neural Networks

A significant upgrade

- 1-layer Perceptrons are very limited;
- But having one more layer makes it much more powerful!
- In fact, **a 2-layer perceptron can *approximate* any function to an arbitrary degree of precision!**
    - This is an “in principle” claim;
    - Assumes arbitrarily large layers (i.e., infinite neurons);

Multilayer Feedforward Networks are Universal Approximators

KURT HORNIK

Technische Universität Wien

MAXWELL STINCHCOMBE AND HALBERT WHITE

University of California, San Diego

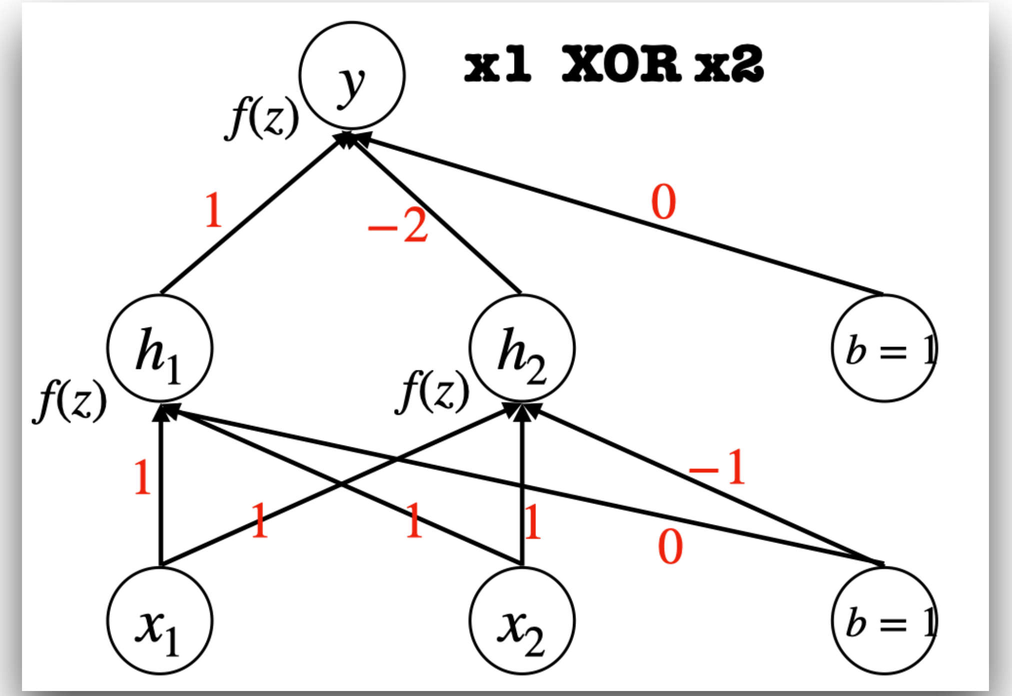
(*Received* 16 *September* 1988; *revised and accepted* 9 *March* 1989)

Interium Summary 3

From individual neurons to networks

Interium Summary 3

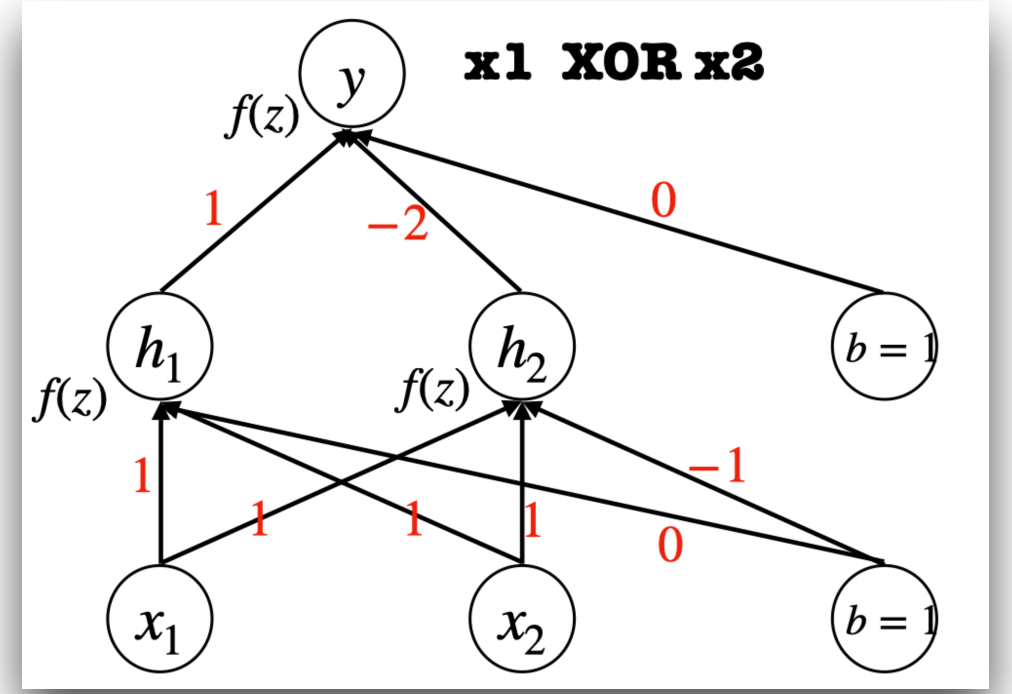
From individual neurons to networks

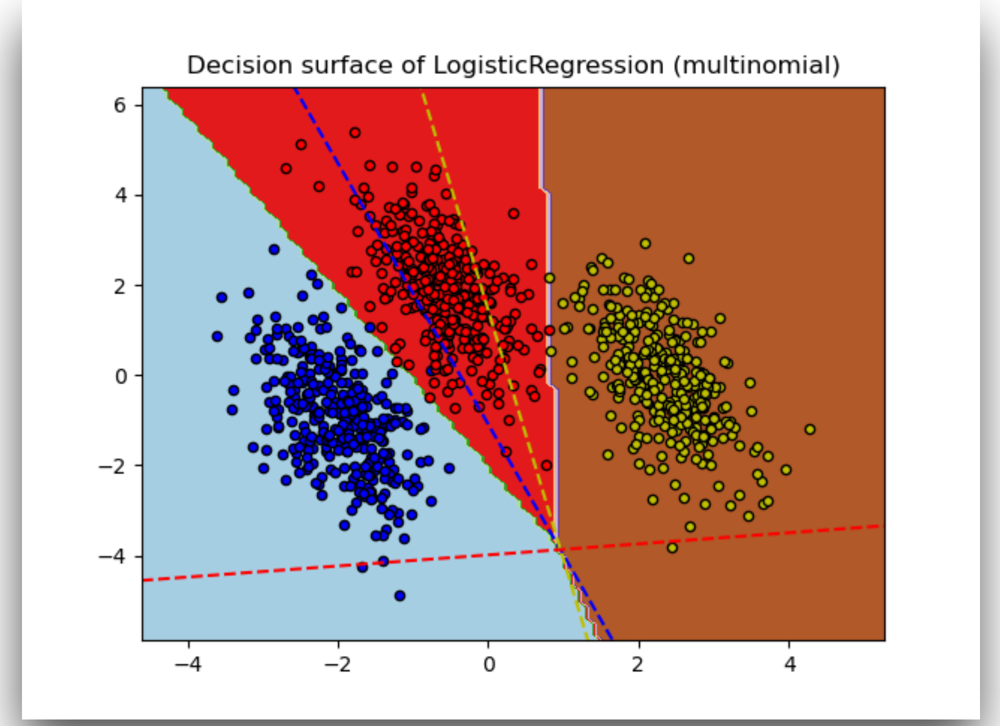


- Multi-layer perceptron / Neural networks can handle XOR.

Interium Summary 3

From individual neurons to networks

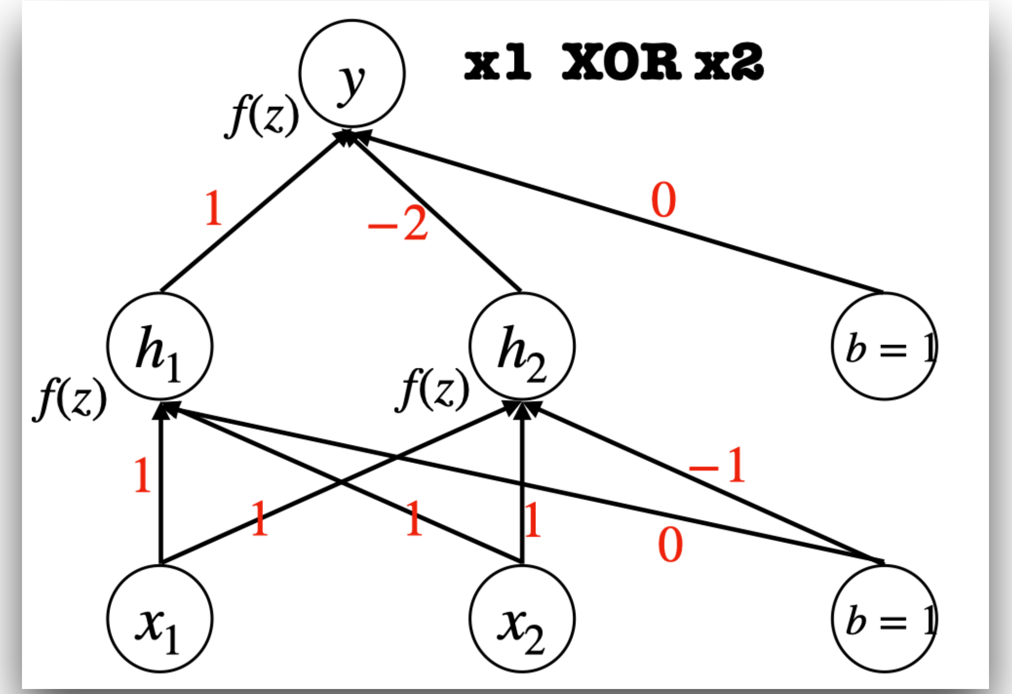


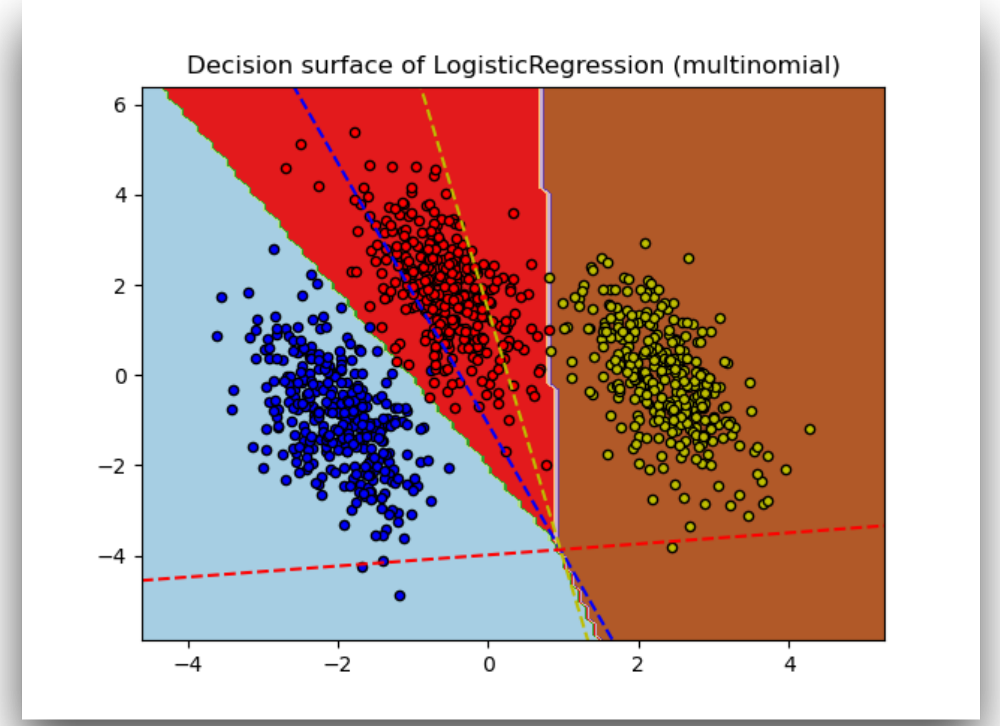


- Multi-layer perceptron / Neural networks can handle XOR.
- Multinomial logistic regressoin is a specific version of neural networks;

Interium Summary 3

From individual neurons to networks

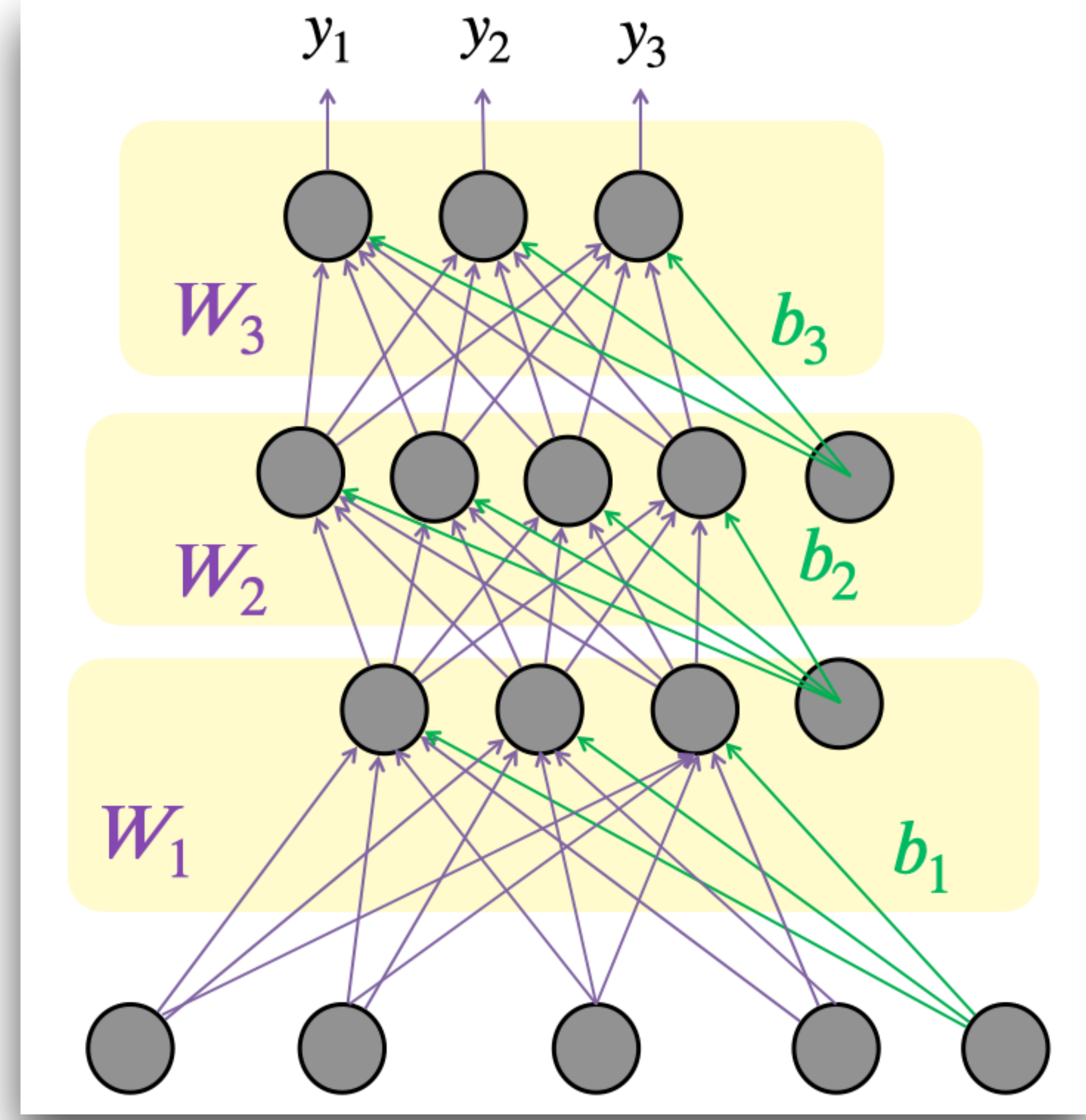




- Multi-layer perceptron / Neural networks can handle XOR.

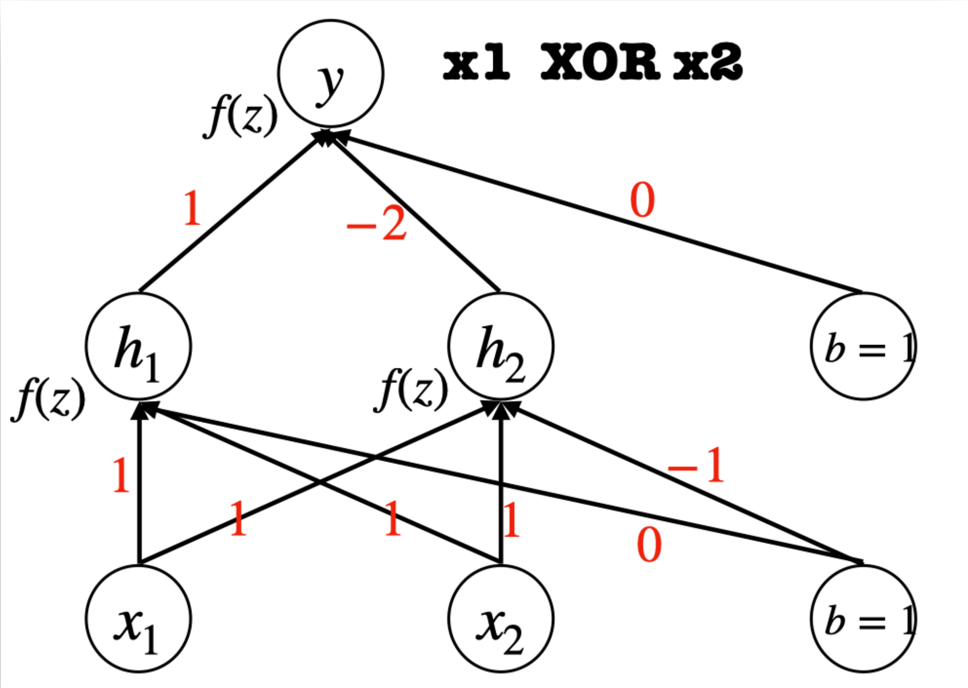
- Multinomial logistic regressoin is a specific version of neural networks;

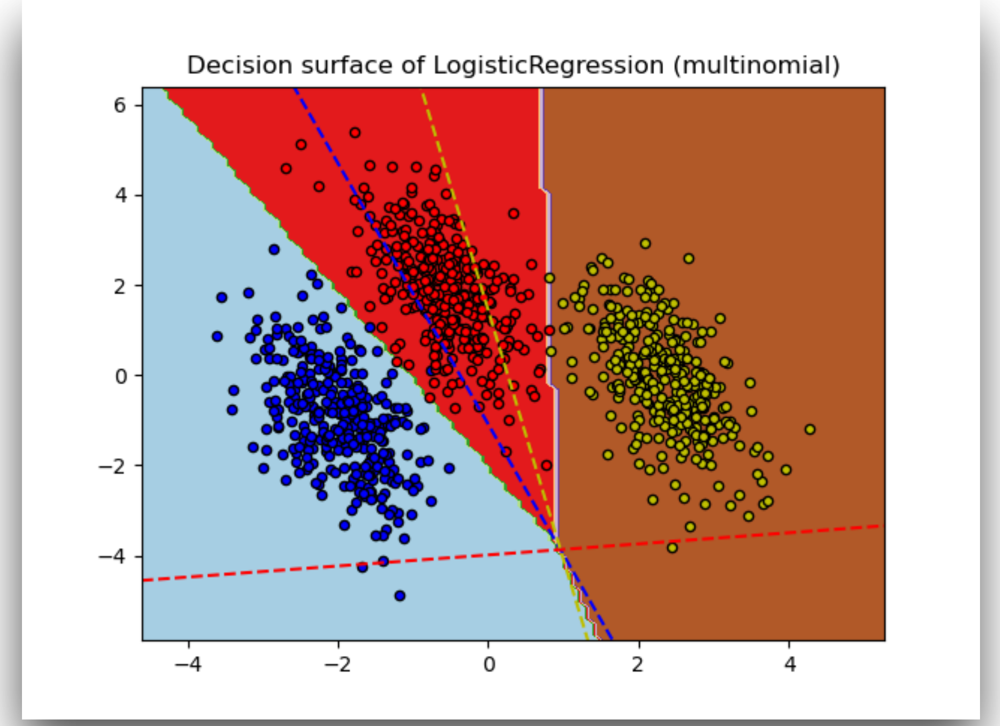
- We use matrix and vector notations to represent neural networks of arbitrary depths;



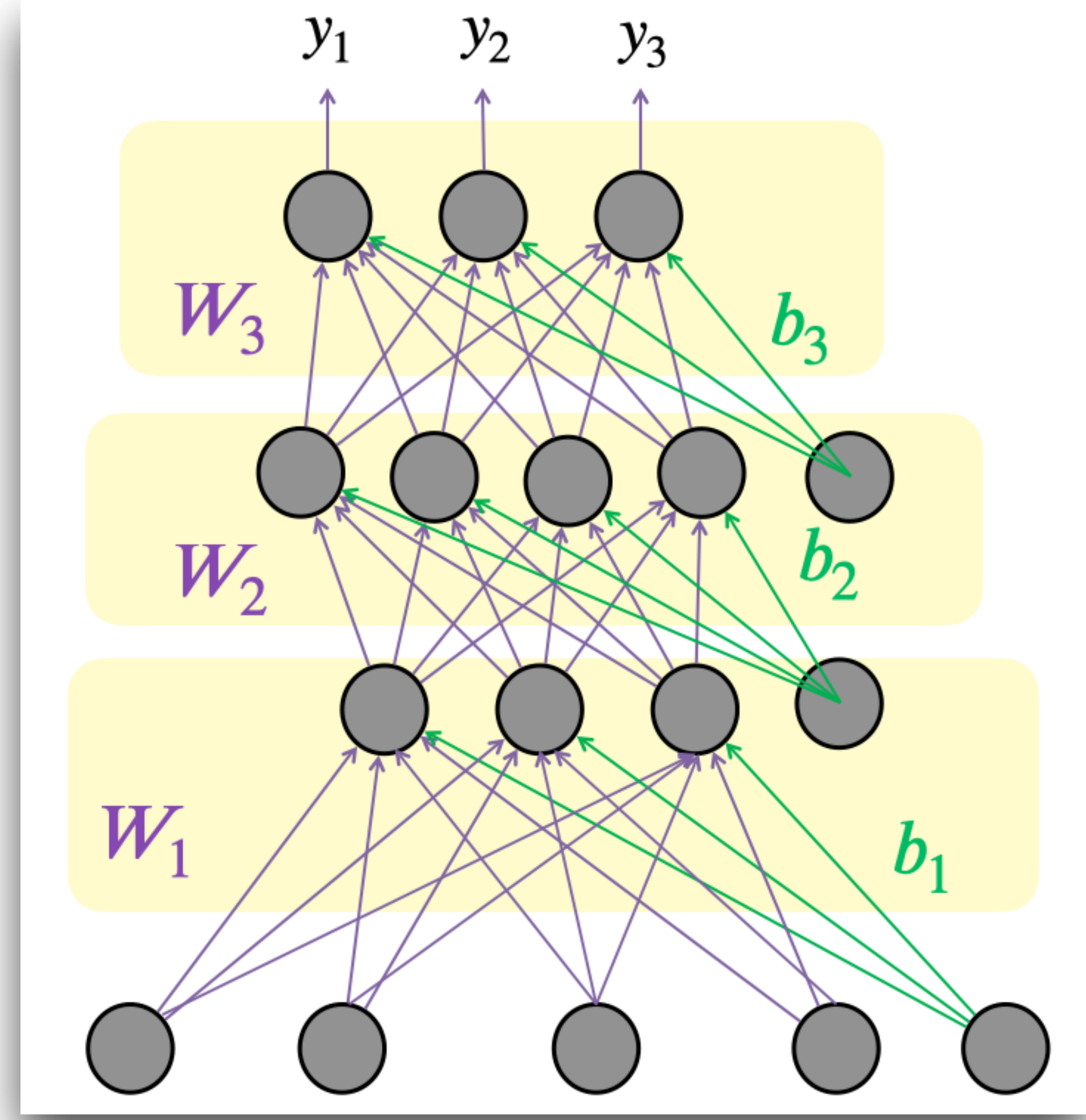
Interium Summary 3

From individual neurons to networks





- Multi-layer perceptron / Neural networks can handle XOR.
- Multinomial logistic regressoin is a specific version of neural networks;
- We use matrix and vector notations to represent neural networks of arbitrary depths;
- Neural networks that have more than 1 layers is ARBITRARILY EXPRESSIVE in principle: they can approximate any function given arbitrarily many neurons;




Multilayer Feedforward Networks are Universal Approximators

Kur' Hornik

Technische Universität Wien

Back Propagation & Gradient Descent

1974 (first proposed) & 1986 (polularized)

Revisit the Perceptron Learning Rule

Error-based Learning as the secret sauce

Revisit the Perceptron Learning Rule

Error-based Learning as the secret sauce

- Remember the perceptron learning rule: $\mathbf{w} := \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$

Revisit the Perceptron Learning Rule

Error-based Learning as the secret sauce

- Remember the perceptron learning rule: $\mathbf{w} := \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$
  - Roughly: update each weight proportional to the error (i.e., the loss between predicted $\hat{y}$ and true label y).

Revisit the Perceptron Learning Rule

Error-based Learning as the secret sauce

- Remember the perceptron learning rule: $\mathbf{w} := \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$
  - Roughly: update each weight proportional to the error (i.e., the loss between predicted $\hat{y}$ and true label y).
- How to do it for a neural network? — How to compute the error across multiple layers?

Revisit the Perceptron Learning Rule

Error-based Learning as the secret sauce

- Remember the perceptron learning rule: $\mathbf{w} := \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$
  - Roughly: update each weight proportional to the error (i.e., the loss between predicted $\hat{y}$ and true label y).
- How to do it for a neural network? — How to compute the error across multiple layers?
  - Assume we have some loss function that compute the difference between two vectors: $L(\hat{y} - y)$

Revisit the Perceptron Learning Rule

Error-based Learning as the secret sauce

- Remember the perceptron learning rule: $\mathbf{w} := \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$
    - Roughly: update each weight proportional to the error (i.e., the loss between predicted $\hat{y}$ and true label y).
- How to do it for a neural network? — How to compute the error across multiple layers?
    - Assume we have some loss function that compute the difference between two vectors: $L(\hat{y} - y)$
    - Goal = to derive **gradient descent** with the help of **backpropagation**!

Revisit the Perceptron Learning Rule

Error-based Learning as the secret sauce

- Remember the perceptron learning rule: $\mathbf{w} := \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$
    - Roughly: update each weight proportional to the error (i.e., the loss between predicted $\hat{y}$ and true label y).
- How to do it for a neural network? — How to compute the error across multiple layers?
    - Assume we have some loss function that compute the difference between two vectors: $L(\hat{y} - y)$
    - Goal = to derive **gradient descent** with the help of **backpropagation**!

Update each weight: $w' = w - \eta \frac{d}{dw} L(f(x; w), y)$, where $L(f(x; w), y) = L(\hat{y} - y)$

Computation Graphs

Representing the procedure of the computation

Computation Graphs

Representing the procedure of the computation

- For training, we need the **derivative of the loss** with respect to **each weight in every layer** of the network;
  - But the loss is computed only at the very end of the network….
  - How do we know how much a weight at Layer 1 contribute the final loss?

Computation Graphs

Representing the procedure of the computation

- For training, we need the **derivative of the loss** with respect to **each weight in every layer** of the network;
    - But the loss is computed only at the very end of the network….
    - How do we know how much a weight at Layer 1 contribute the final loss?
- **Solution** = represent the computation of the entire neural network with a (very big) *computation graph*!
    - A computation graph represents the entire process of computing a function (which can be a very complex composition of multiple functions) — representing the dependencies between any two steps of the computation.

Computation Graphs

[Exercise] Try to construct a computation graph for a simple equation

- Say, here is a dummy loss function with 3 inputs: $\underline{L(a,b,c) = c(a+2b)}$
- In a computation graph: a **node** represents a value, and an **edge** represents a function / arithmetic operation / computation.
- Try to construct a computation graph for L!

Computation Graphs

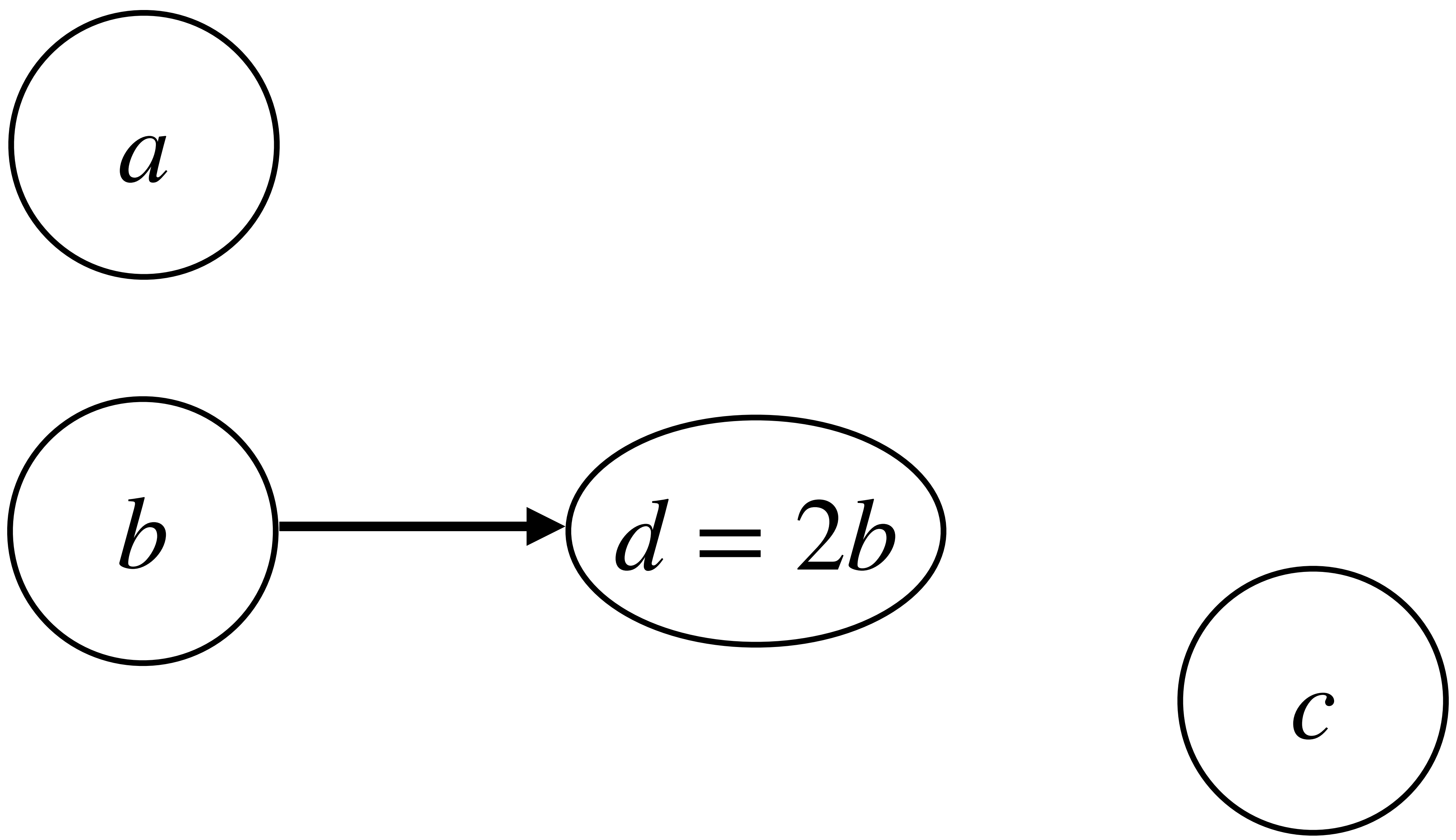
[Exercise] Try to construct a computation graph for a simple equation

- Say, here is a dummy loss function with 3 inputs: $\underline{L(a, b, c) = c(a + 2b)}$
- In a computation graph: a **node** represents a value, and an **edge** represents a function / arithmetic operation / computation.
- Try to construct a computation graph for L!

Computation Graphs

[Exercise] Try to construct a computation graph for a simple equation

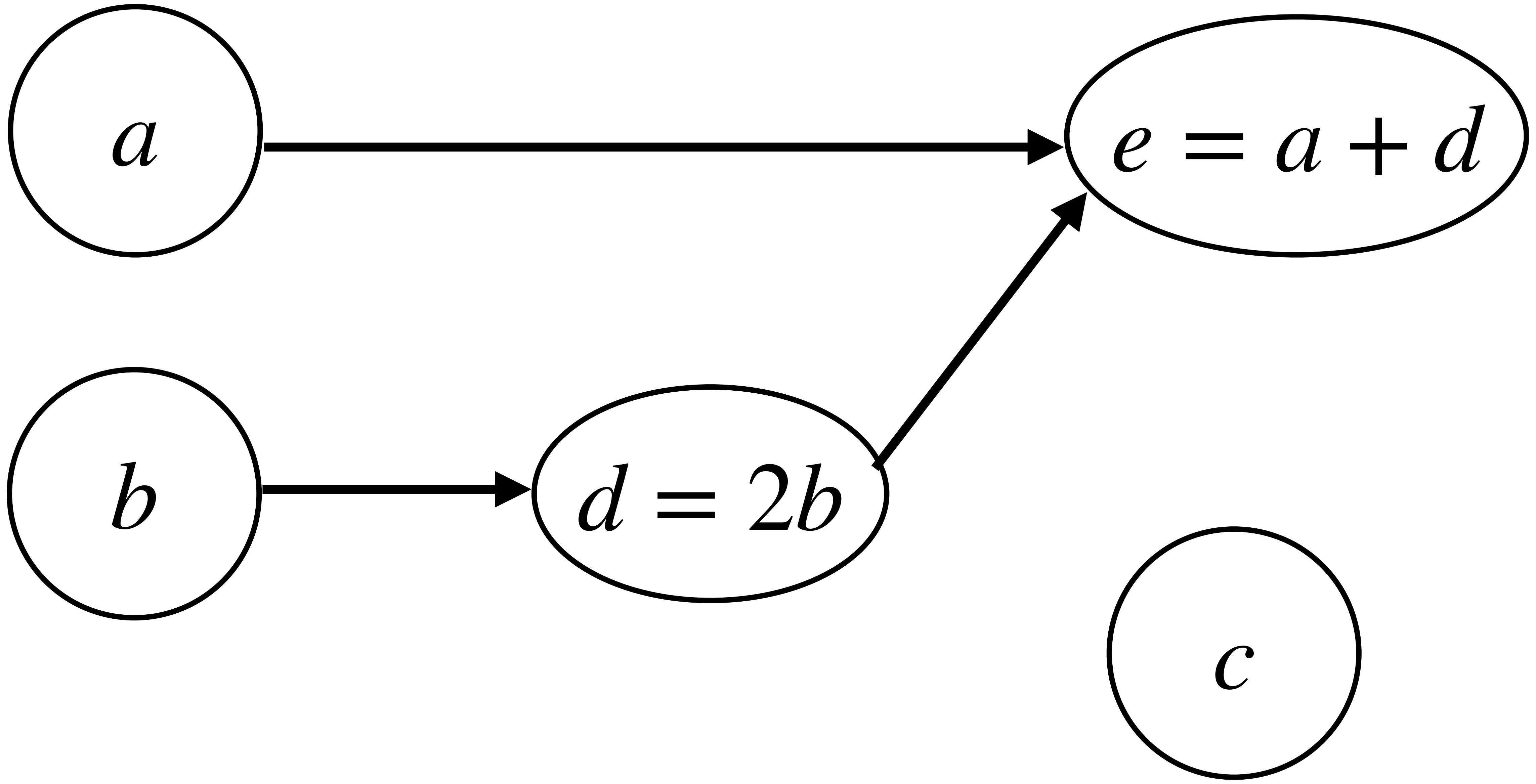
- Say, here is a dummy loss function with 3 inputs: $\underline{L(a, b, c) = c(a + 2b)}$
- In a computation graph: a **node** represents a value, and an **edge** represents a function / arithmetic operation / computation.
- Try to construct a computation graph for L!



Computation Graphs

[Exercise] Try to construct a computation graph for a simple equation

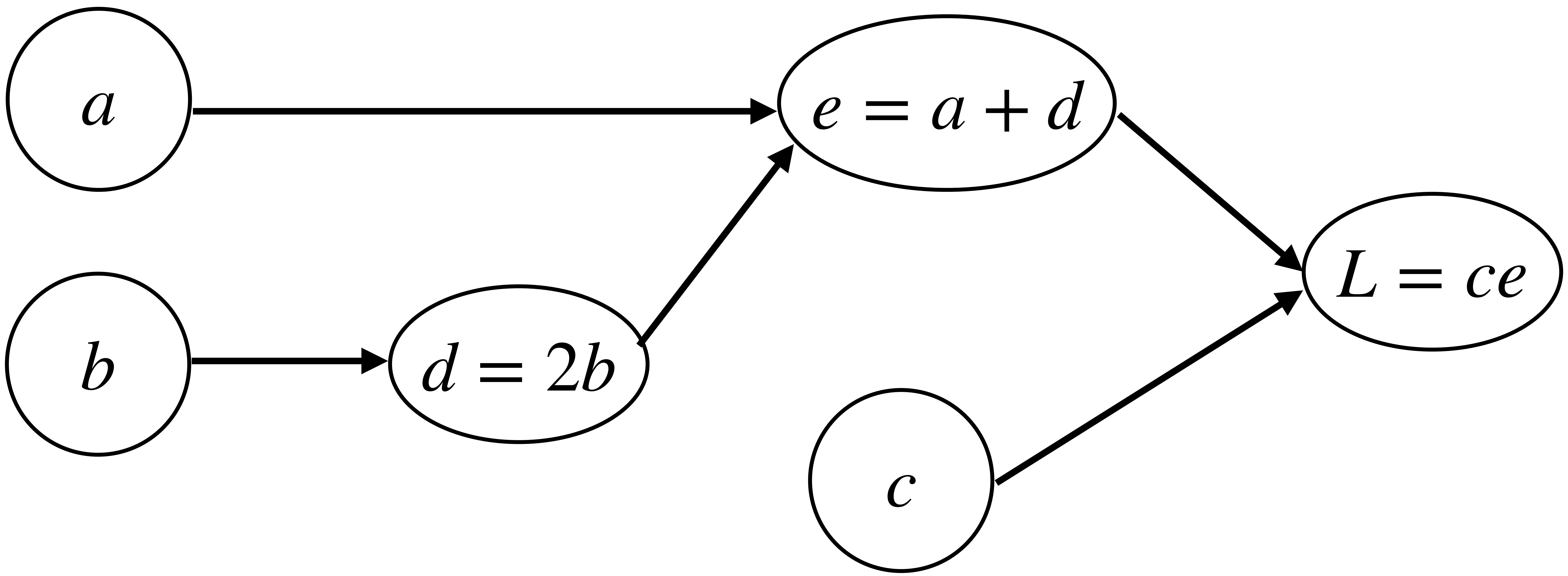
- Say, here is a dummy loss function with 3 inputs: $\underline{L(a, b, c) = c(a + 2b)}$
- In a computation graph: a **node** represents a value, and an **edge** represents a function / arithmetic operation / computation.
- Try to construct a computation graph for L!



Computation Graphs

[Exercise] Try to construct a computation graph for a simple equation

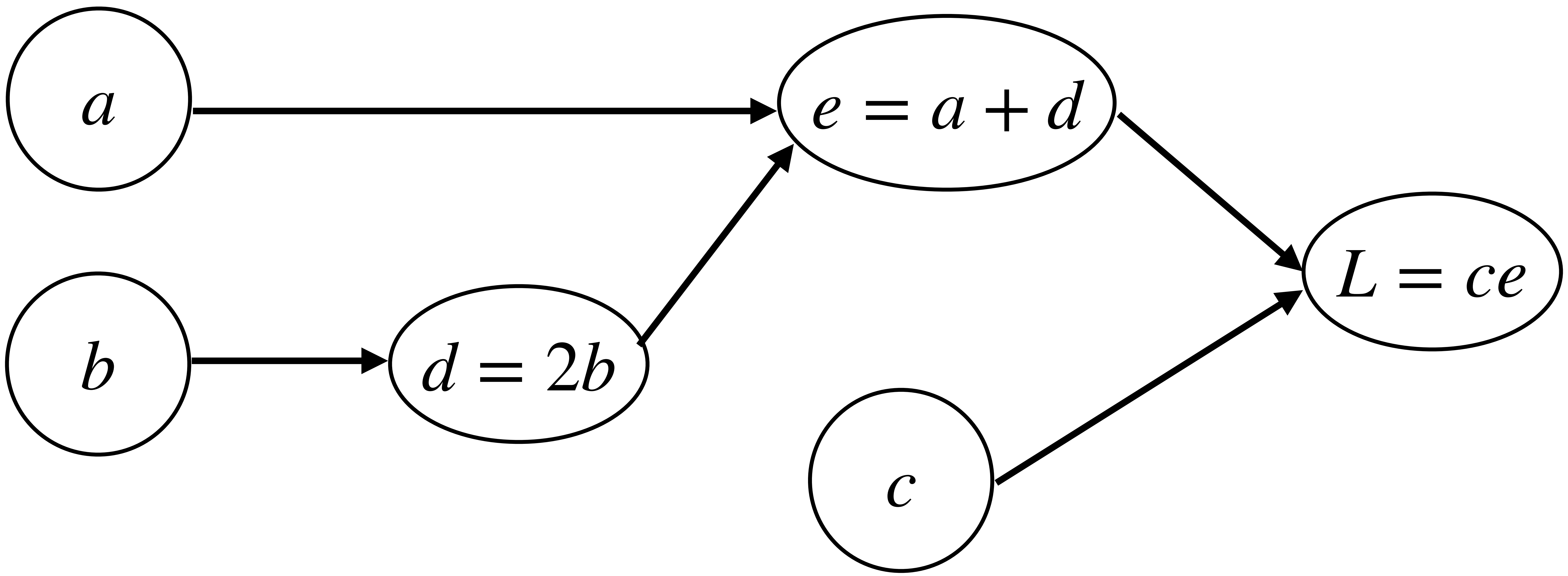
- Say, here is a dummy loss function with 3 inputs: $\underline{L(a,b,c) = c(a+2b)}$
- In a computation graph: a **node** represents a value, and an **edge** represents a function / arithmetic operation / computation.
- Try to construct a computation graph for L!



Computation Graphs

[Exercise] Try to construct a computation graph for a simple equation

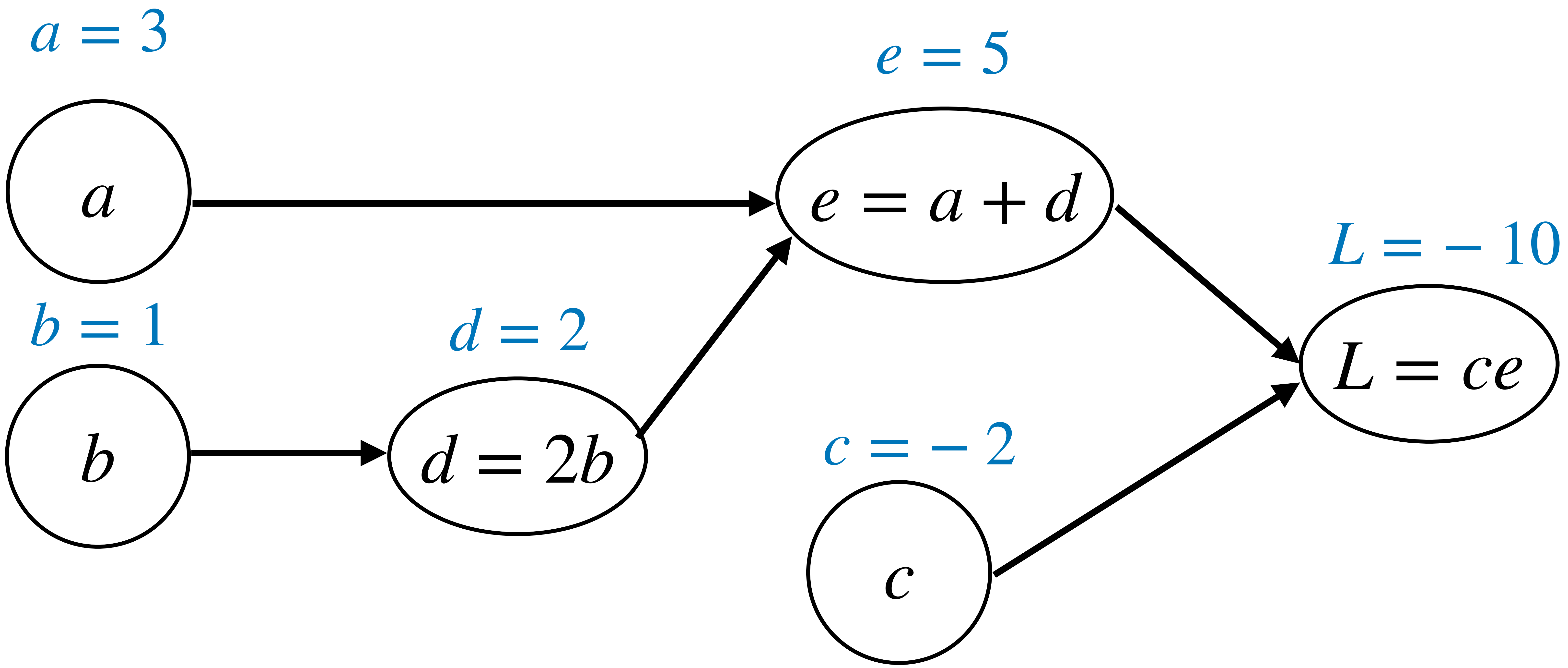
- $L(a, b, c) = c(a + 2b)$. Now let $a = 3, b = 1, c = -2$



Computation Graphs

[Exercise] Try to construct a computation graph for a simple equation

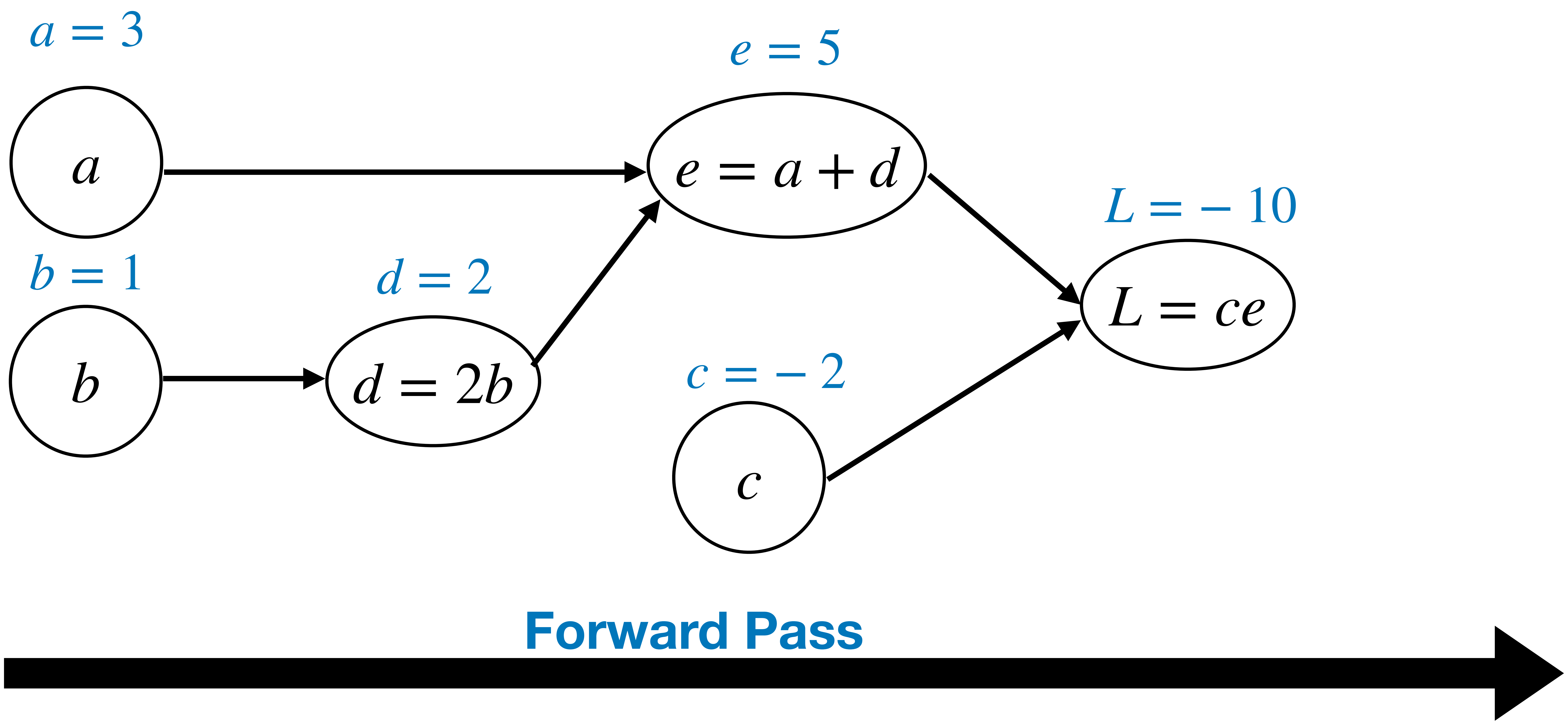
- $\underline{L(a, b, c) = c(a + 2b)}$. Now let $a = 3, b = 1, c = -2$



Computation Graphs

[Exercise] Try to construct a computation graph for a simple equation

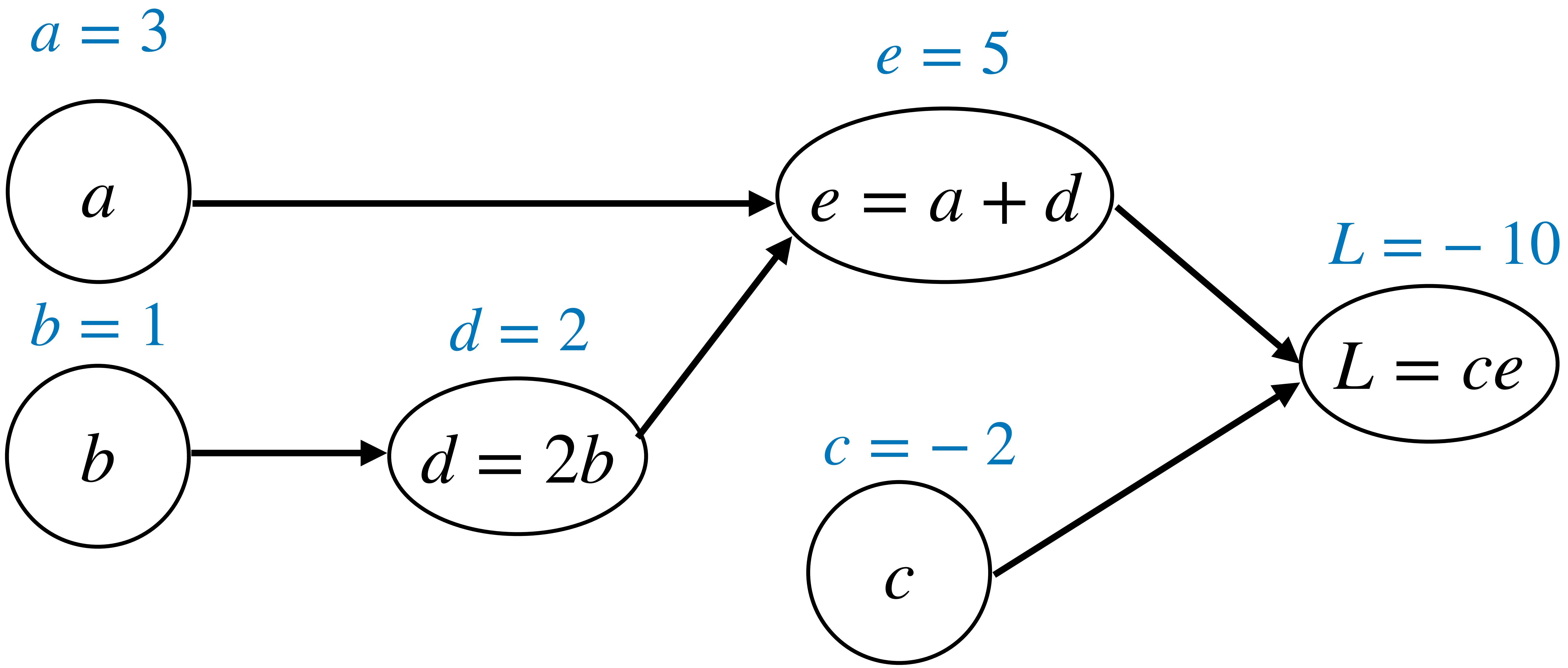
- $\underline{L(a, b, c) = c(a + 2b)}$. Now let $a = 3, b = 1, c = -2$



Computation Graphs

[Exercise] Try to construct a computation graph for a simple equation

- $\underline{L(a,b,c) = c(a+2b)}$. Now let $a = 3, b = 1, c = -2$

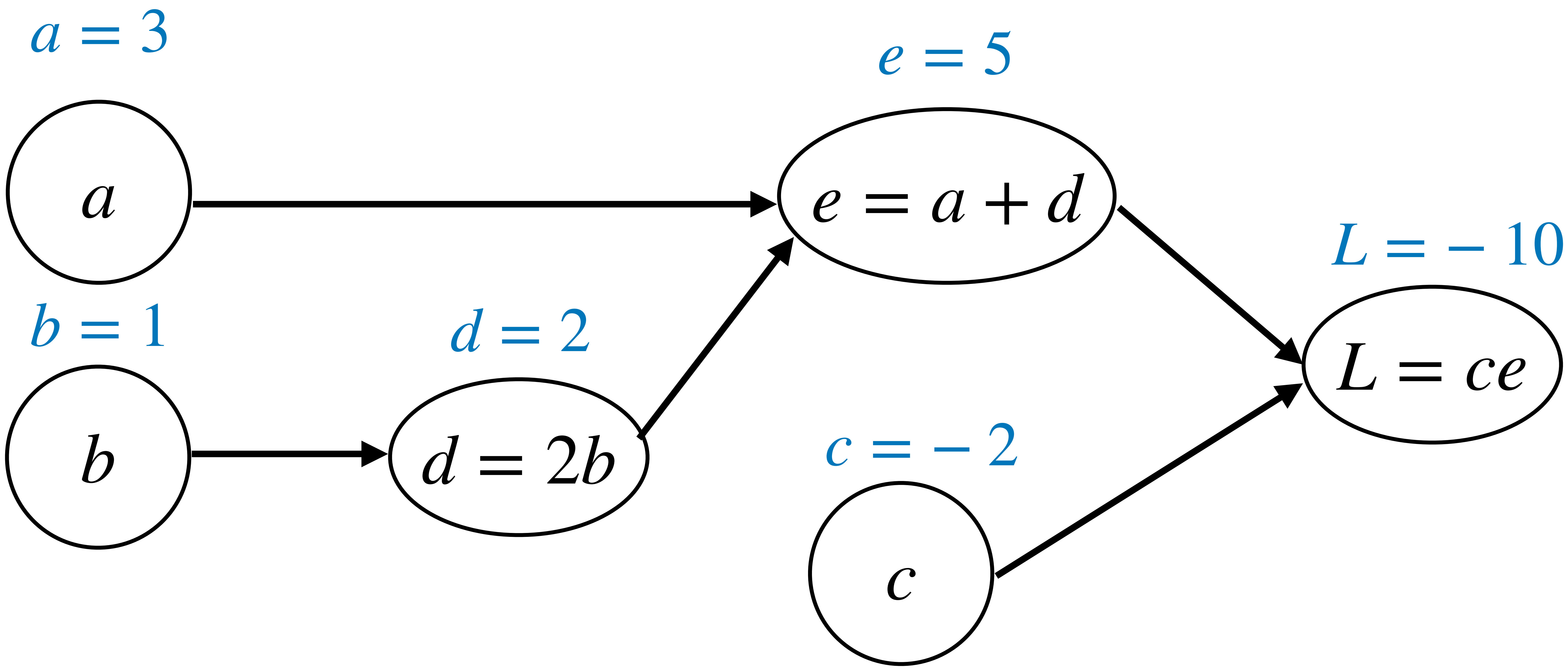


Remember:
We want to know how much does a, b, c each contribute to the final loss L

Forward Pass

Backpropagation

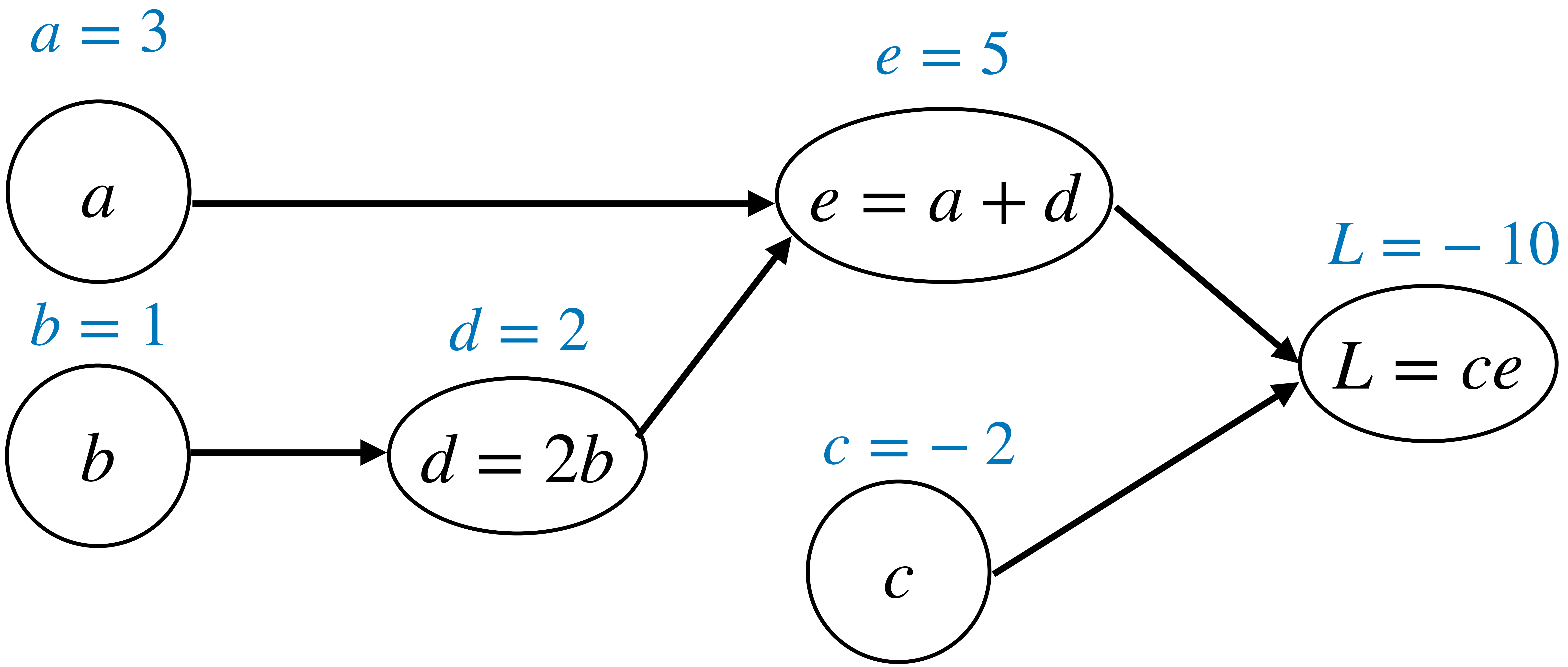
Responsibility attribution through gradient



Backpropagation

Responsibility attribution through gradient

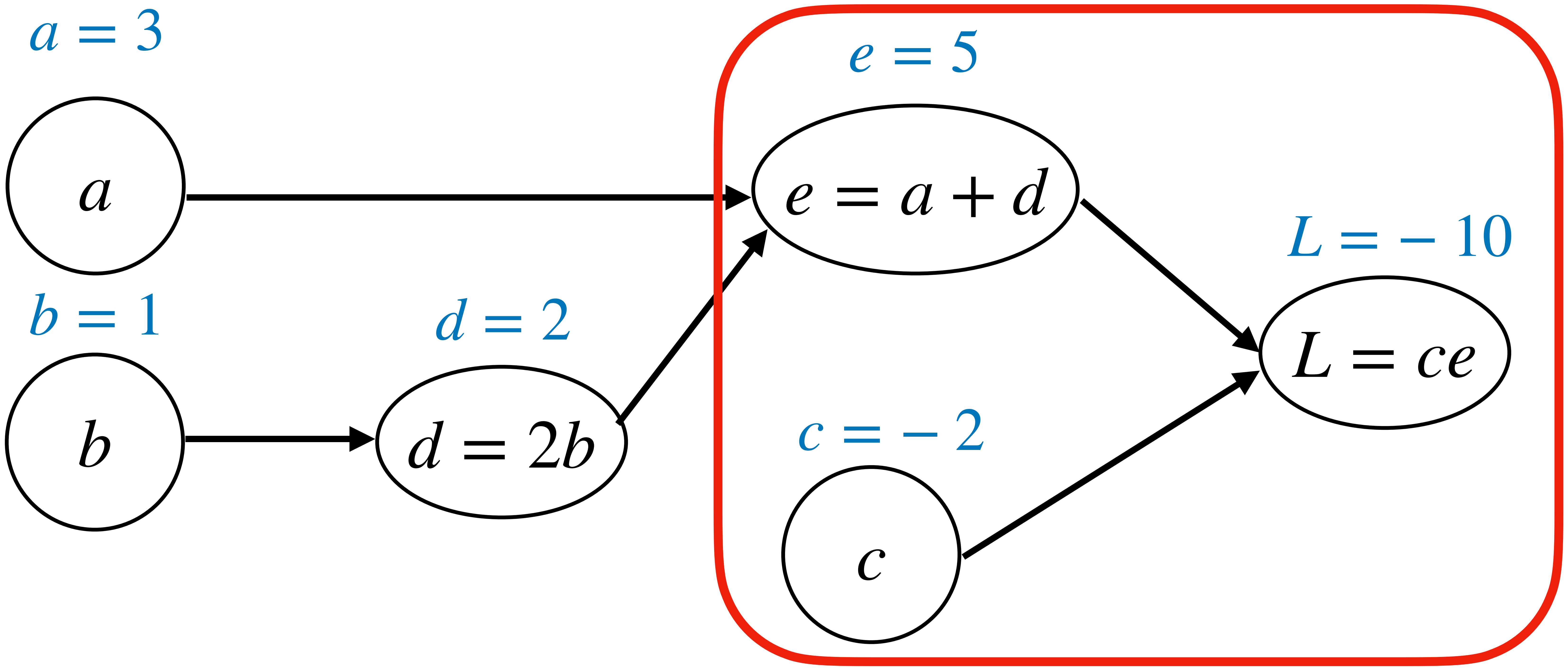
- $\underline{L(a,b,c) = c(a+2b)}$. Now let $a = 3, b = 1, c = -2$;
- We don't know immediately how much a contribute to L…
- But we do know how much c and e contributes (since they are the final step)!



Backpropagation

Responsibility attribution through gradient

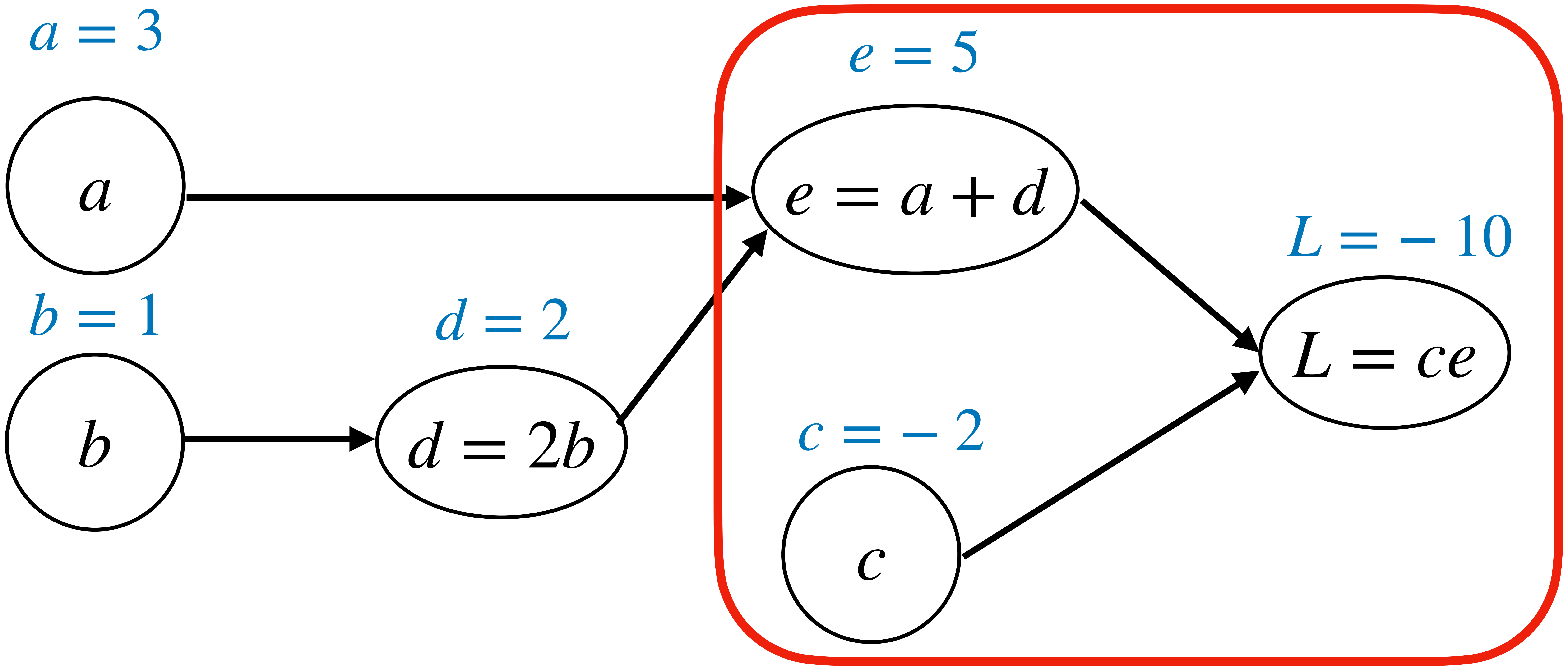
- $\underline{L(a, b, c) = c(a + 2b)}$. Now let $a = 3, b = 1, c = -2$;
- We don't know immediately how much a contribute to L…
- But we do know how much c and e contributes (since they are the final step)!



Backpropagation

Responsibility attribution through gradient

- $\underline{L(a, b, c) = c(a + 2b)}$. Now let $a = 3, b = 1, c = -2$;
- We don't know immediately how much a contribute to L…
- But we do know how much c and e contributes (since they are the final step)!

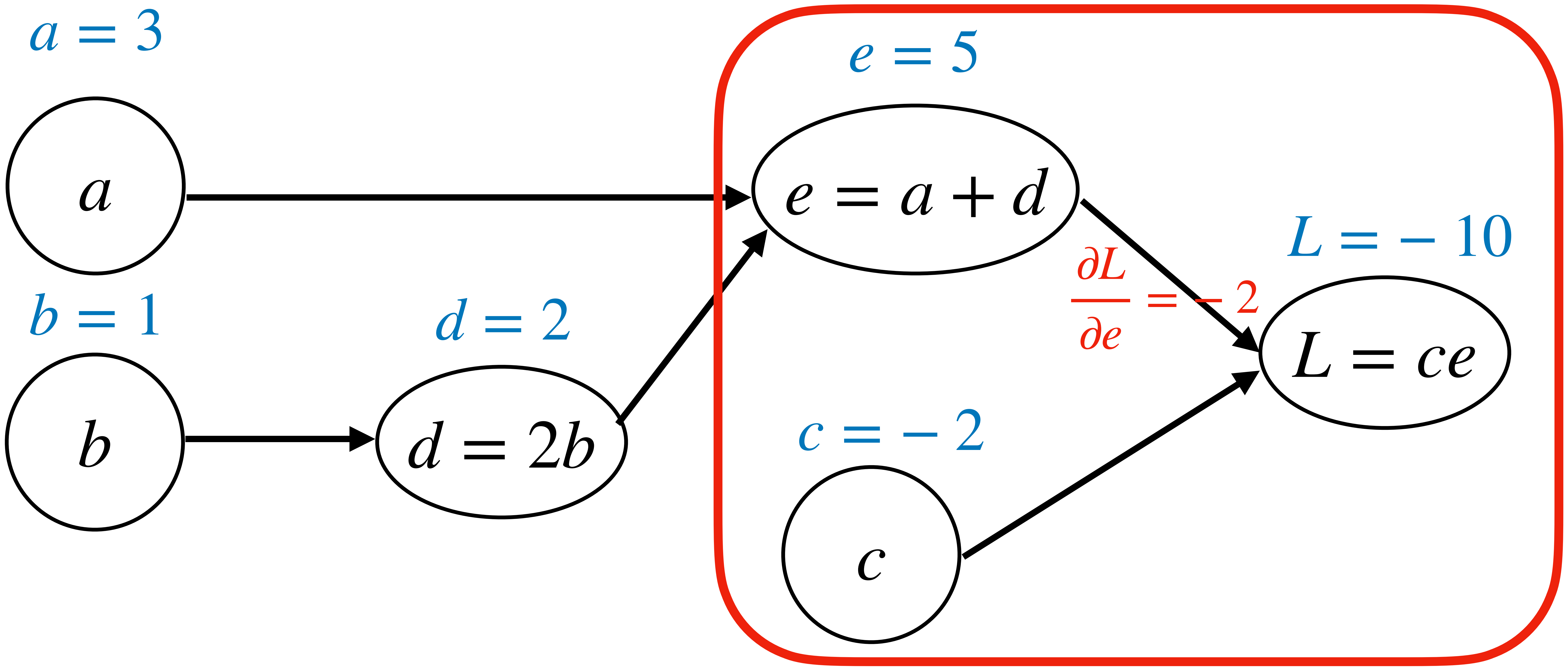


$L = ce$

Backpropagation

Responsibility attribution through gradient

- $L(a, b, c) = c(a + 2b)$. Now let $a = 3, b = 1, c = -2$;
- We don't know immediately how much a contribute to L…
- But we do know how much c and e contributes (since they are the final step)!



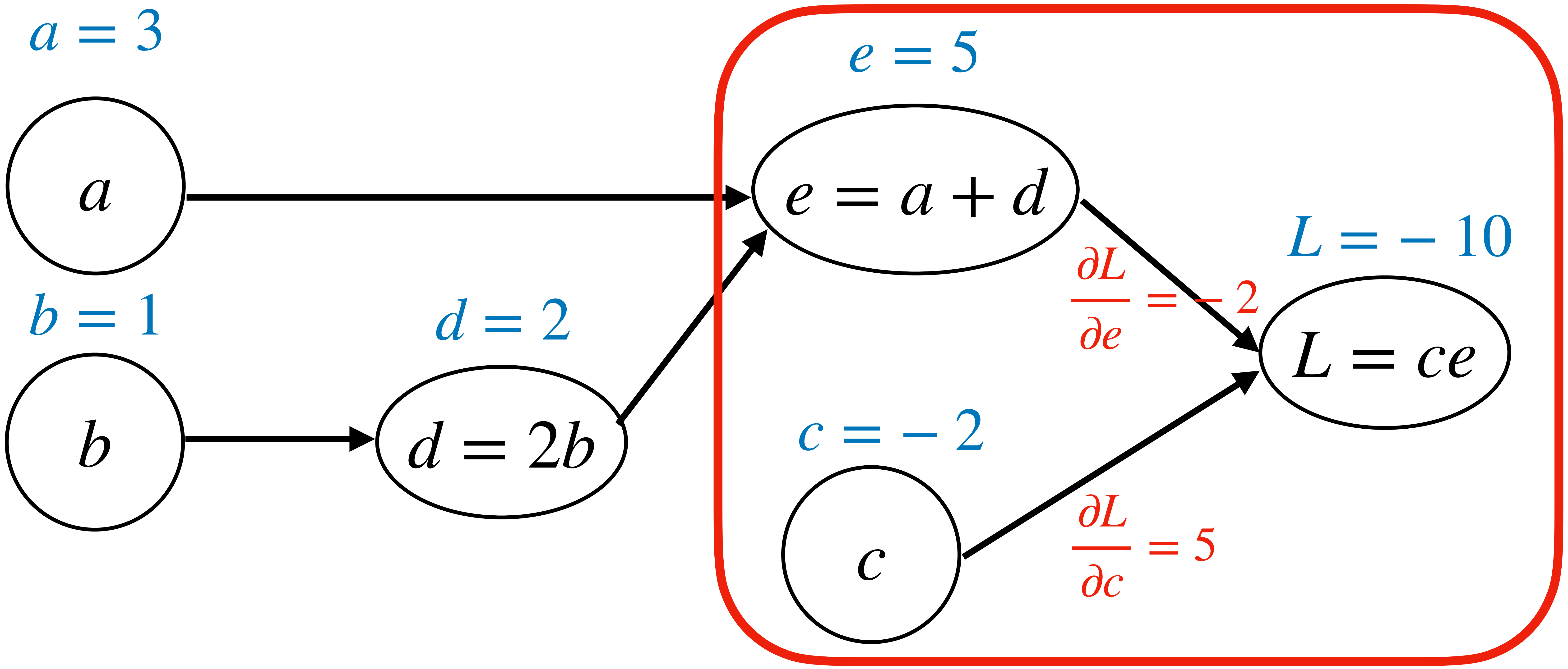
$L = ce$

$\Rightarrow \frac{\partial L}{\partial e} = c = -2$

Backpropagation

Responsibility attribution through gradient

- $L(a, b, c) = c(a + 2b)$. Now let $a = 3, b = 1, c = -2$;
- We don't know immediately how much a contribute to L…
- But we do know how much c and e contributes (since they are the final step)!



$$L = ce$$

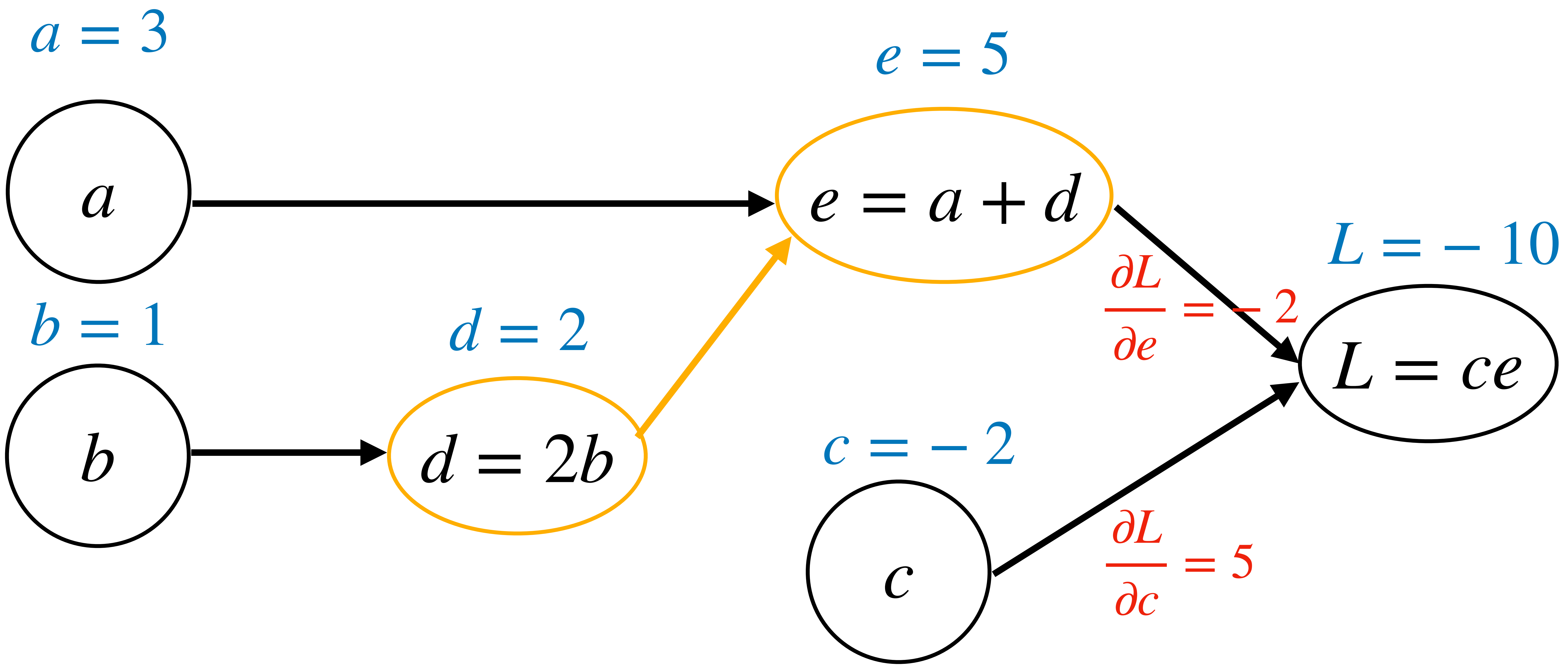
$$\Rightarrow \frac{\partial L}{\partial e} = c = -2$$

$$\Rightarrow \frac{\partial L}{\partial c} = e = 5$$

Backpropagation

Responsibility attribution through gradient

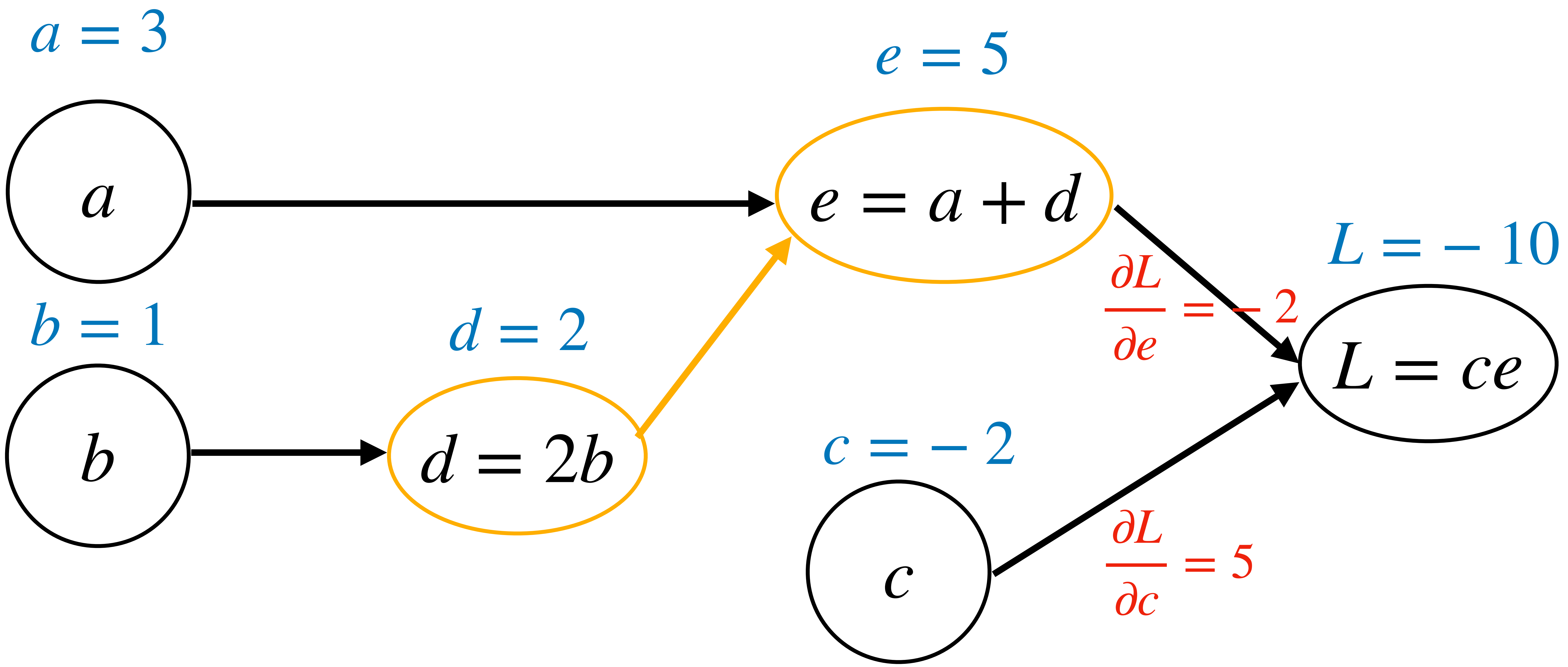
- $\underline{L(a, b, c) = c(a + 2b)}$. Now let $a = 3, b = 1, c = -2$;
- We don't know immediately how much a contribute to L…
- We use e's contribute to compute d's contribution.



Backpropagation

Responsibility attribution through gradient

- $\underline{L(a,b,c) = c(a+2b)}$. Now let $a = 3, b = 1, c = -2$;
- We don't know immediately how much a contribute to L…
- We use e's contribute to compute d's contribution.

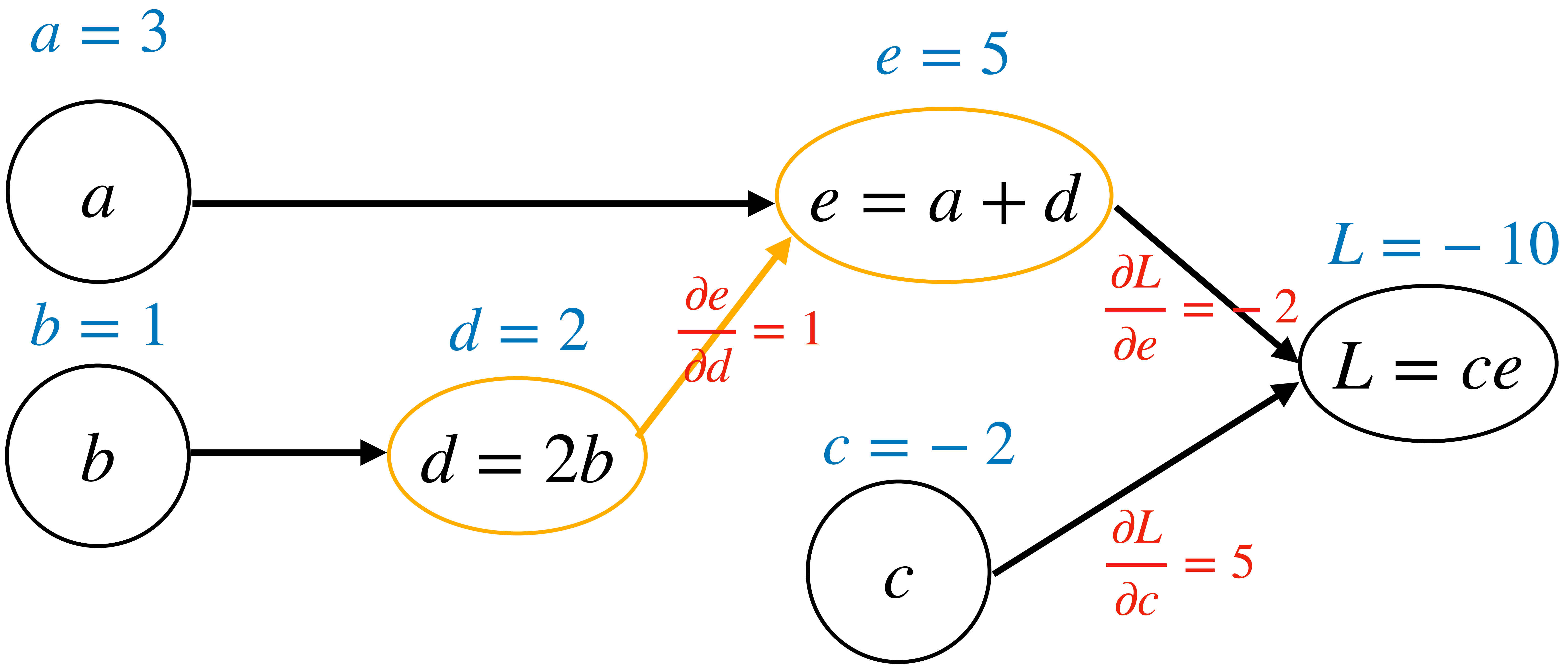


$e = a + d$

Backpropagation

Responsibility attribution through gradient

- $L(a, b, c) = c(a + 2b)$. Now let $a = 3, b = 1, c = -2$;
- We don't know immediately how much a contribute to L…
- We use e's contribute to compute d's contribution.



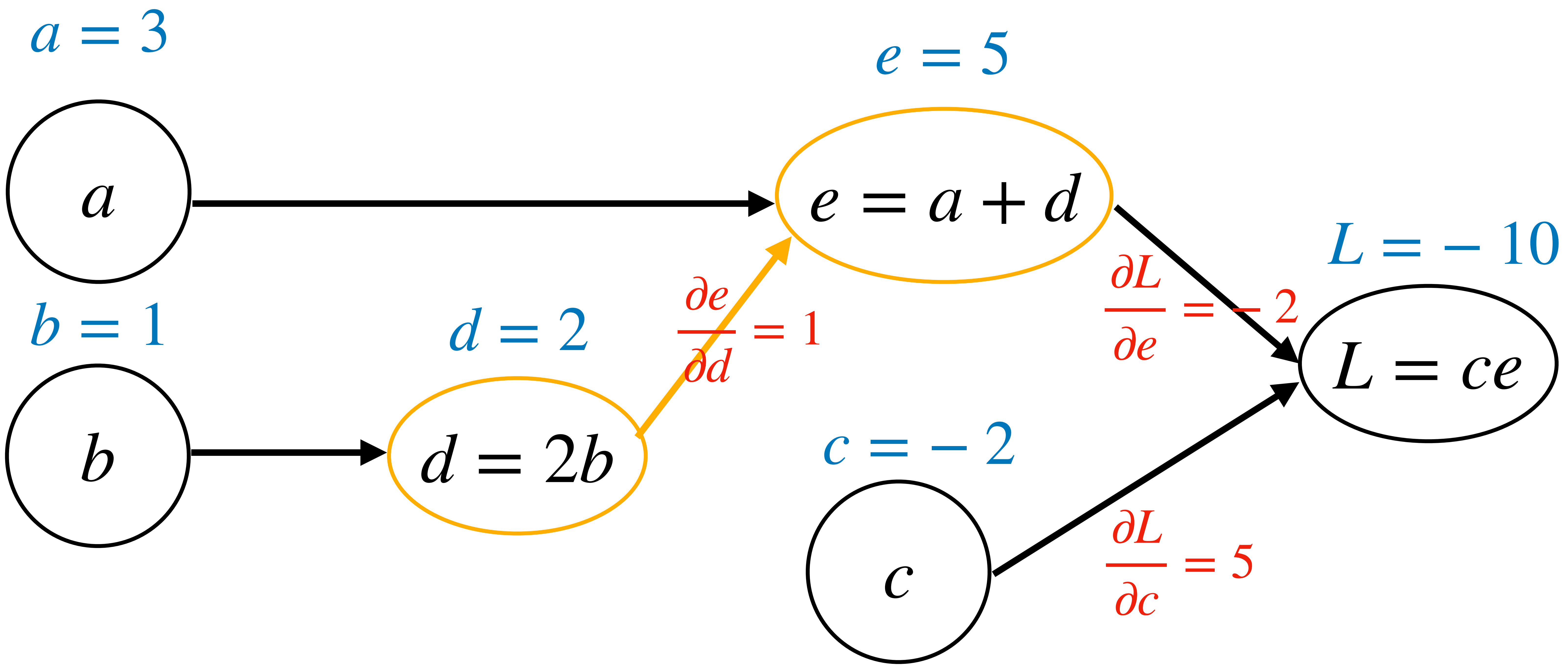
$$e = a + d$$

$$\Rightarrow \frac{\partial e}{\partial d} = 1$$

Backpropagation

Responsibility attribution through gradient

- $\underline{L(a,b,c) = c(a+2b)}$. Now let $a=3, b=1, c=-2$;
- We don't know immediately how much a contribute to L…
- We use e's contribute to compute d's contribution.



The Chain Rule

$$f(x) = u(v(w(x)))$$

$$\frac{\partial f}{\partial x} = \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial w} \cdot \frac{\partial w}{\partial x}$$

$$e = a + d$$

$$\Rightarrow \frac{\partial e}{\partial d} = 1$$

Backpropagation

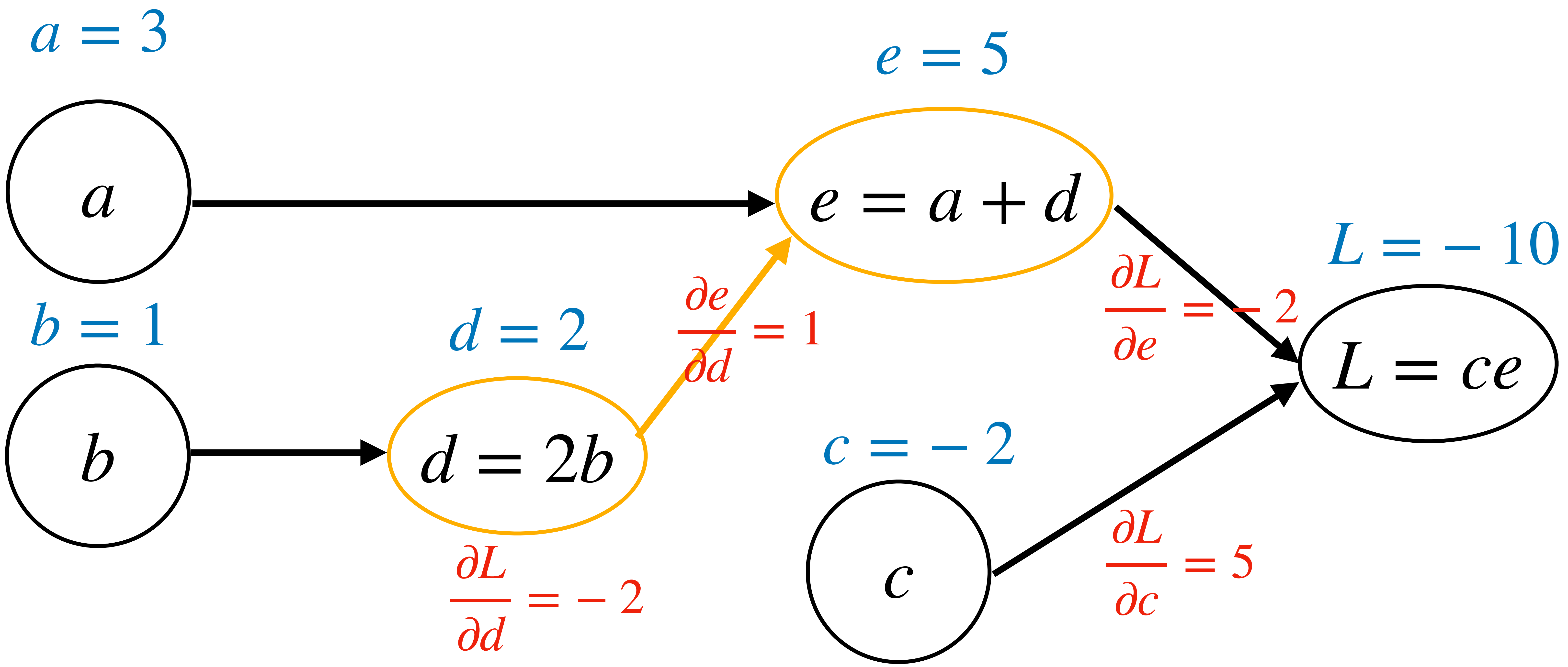
Responsibility attribution through gradient

- $\underline{L(a,b,c) = c(a+2b)}$. Now let $a=3, b=1, c=-2$;
- We don't know immediately how much a contribute to L…
- We use e's contribute to compute d's contribution.

The Chain Rule

$f(x) = u(v(w(x)))$

$$\frac{\partial f}{\partial x} = \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial w} \cdot \frac{\partial w}{\partial x}$$



$$e = a + d$$

$$\Rightarrow \frac{\partial e}{\partial d} = 1$$

$$\Rightarrow \frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}$$

$$= 1 \times -2$$

$$= -2$$

Backpropagation

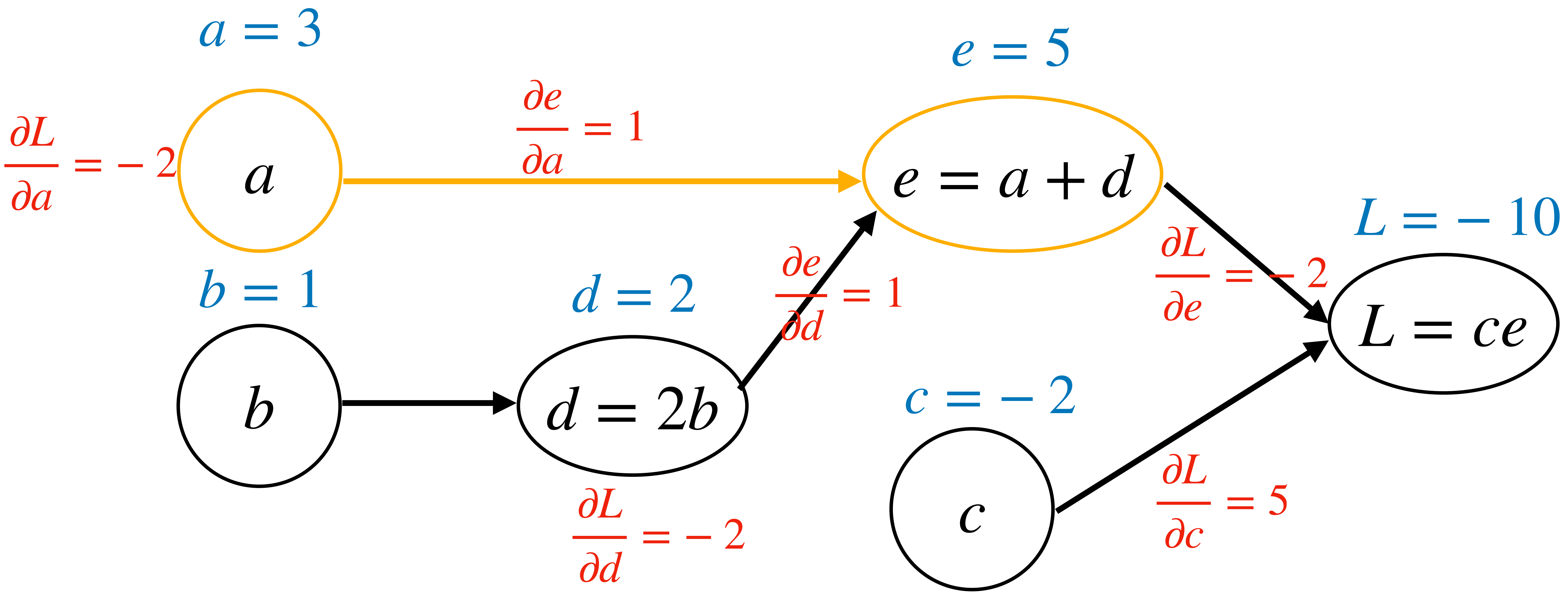
Responsibility attribution through gradient

The Chain Rule

$$f(x) = u(v(w(x)))$$

$$\frac{\partial f}{\partial x} = \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial w} \cdot \frac{\partial w}{\partial x}$$

- $\underline{L(a, b, c) = c(a + 2b)}$. Now let $a = 3, b = 1, c = -2$;
- We don't know immediately how much a contribute to L…
- We continue to compute the contribution of a and b, respectively.



$$e = a + d$$

$$\Rightarrow \frac{\partial e}{\partial a} = 1$$

$$\Rightarrow \frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$= 1 \times -2$$

$$= -2$$

Backpropagation

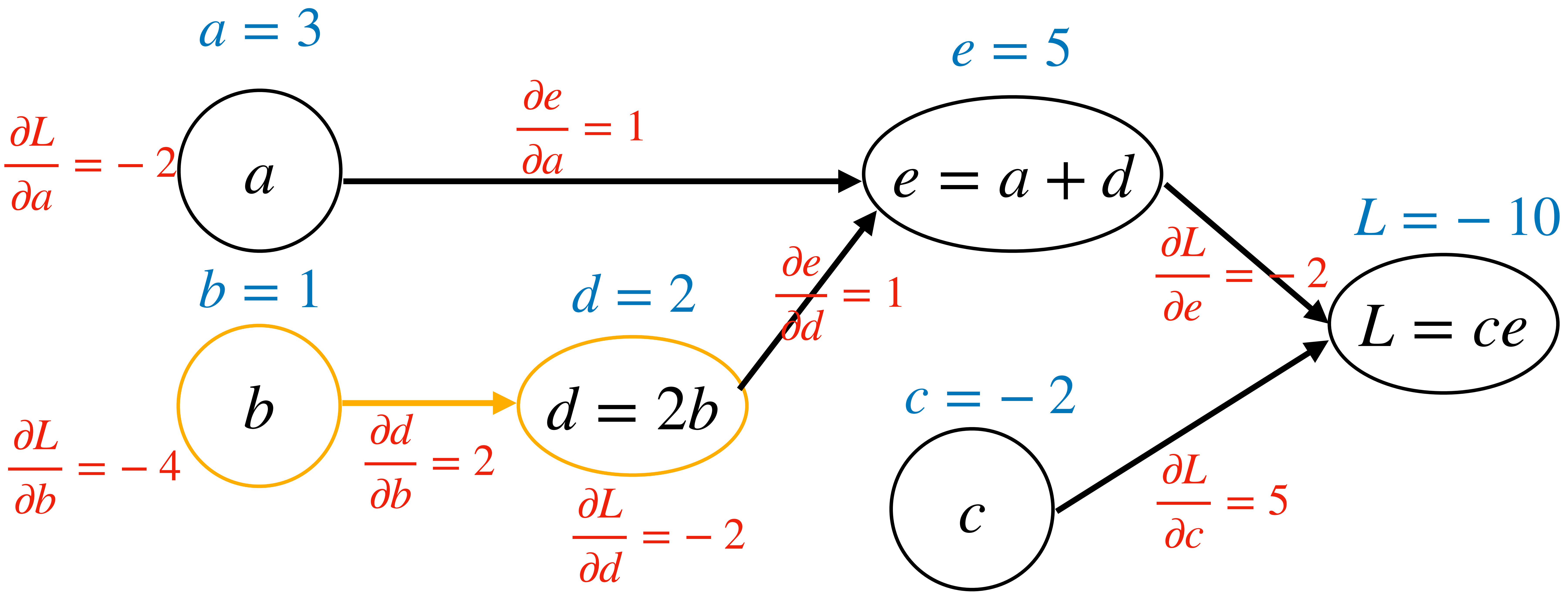
Responsibility attribution through gradient

- $\underline{L(a,b,c) = c(a+2b)}$. Now let $a=3, b=1, c=-2$;
- We don't know immediately how much a contribute to L…
- We continue to compute the contribution of a and b, respectively.

The Chain Rule

$$f(x) = u(v(w(x)))$$

$$\frac{\partial f}{\partial x} = \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial w} \cdot \frac{\partial w}{\partial x}$$



$$d = 2b$$

$$\Rightarrow \frac{\partial d}{\partial b} = 2$$

$$\Rightarrow \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

$$= -2 \times 1 \times 2$$

$$= -4$$

Backpropagation

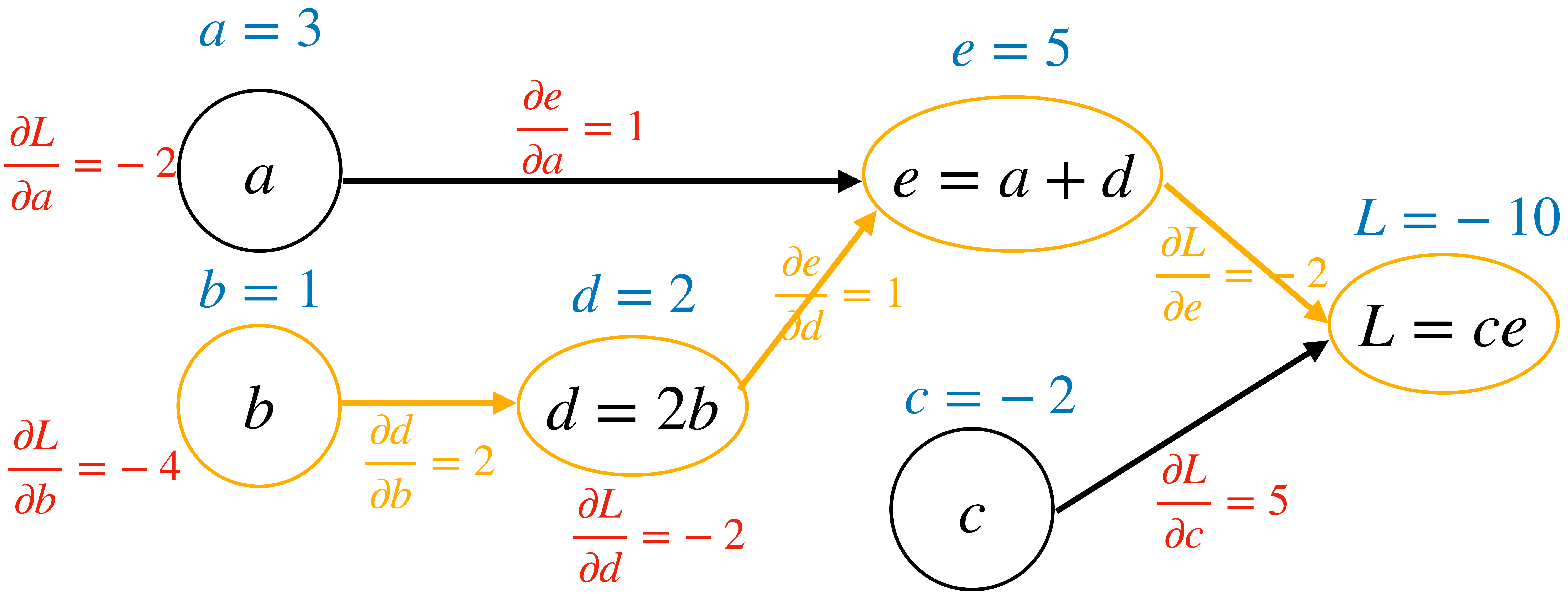
Responsibility attribution through gradient

The Chain Rule

$f(x) = u(v(w(x)))$

$$\frac{\partial f}{\partial x} = \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial w} \cdot \frac{\partial w}{\partial x}$$

- $\underline{L(a,b,c) = c(a+2b)}$. Now let $a = 3, b = 1, c = -2$;
  - **Derivative on an edge**: *local* dependency;
  - **Derivative on node**: *long* dependency all the way from the final node (L)



$d = 2b$

$$\Rightarrow \frac{\partial d}{\partial b} = 2$$

$$\Rightarrow \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

$$= -2 \times 1 \times 2$$

$$= -4$$

Backpropagation

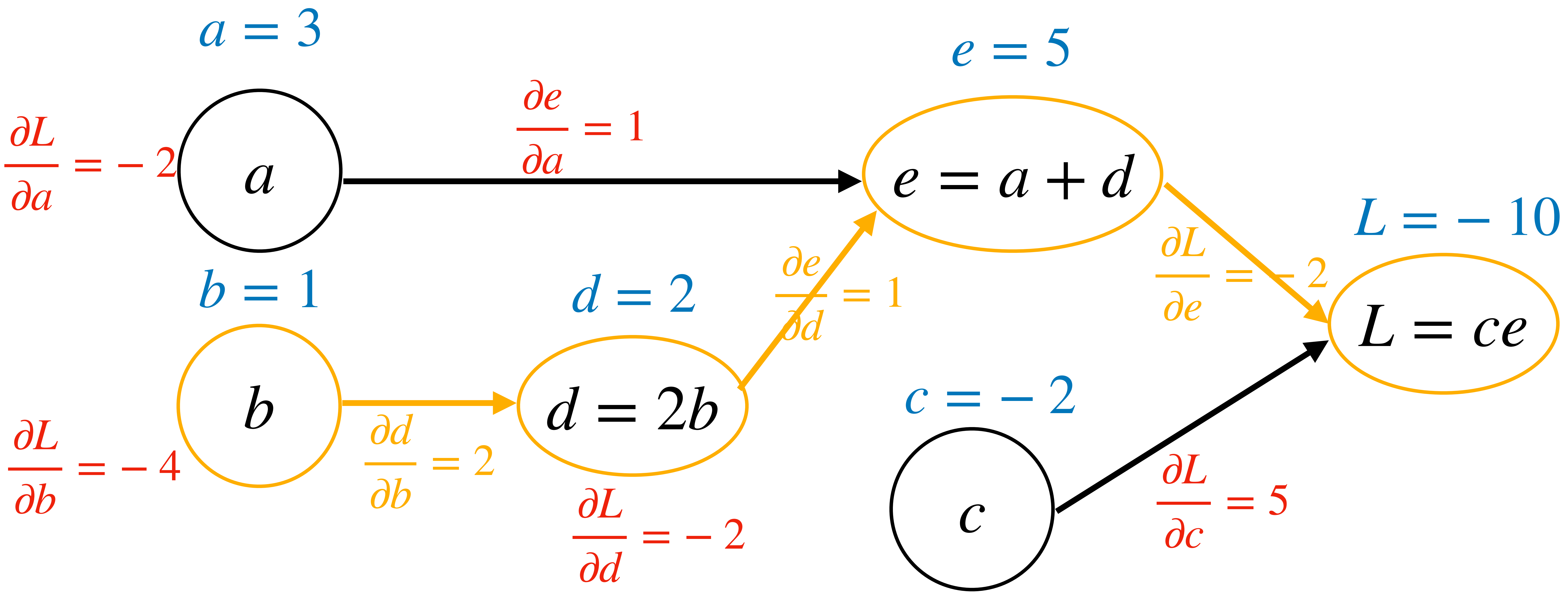
Responsibility attribution through gradient

- $\underline{L(a, b, c) = c(a + 2b)}$. Now let $a = 3, b = 1, c = -2$;
  - **Derivative on an edge**: *local* dependency;
  - **Derivative on node**: *long* dependency all the way from the final node (L)

The Chain Rule

$$f(x) = u(v(w(x)))$$

$$\frac{\partial f}{\partial x} = \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial w} \cdot \frac{\partial w}{\partial x}$$



$d = 2b$

Edge: local

$$\Rightarrow \frac{\partial d}{\partial b} = 2$$

$$\Rightarrow \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e} \frac{\partial e}{\partial d} \frac{\partial d}{\partial b}$$

$$= -2 \times 1 \times 2$$

$$= -4$$

Backpropagation

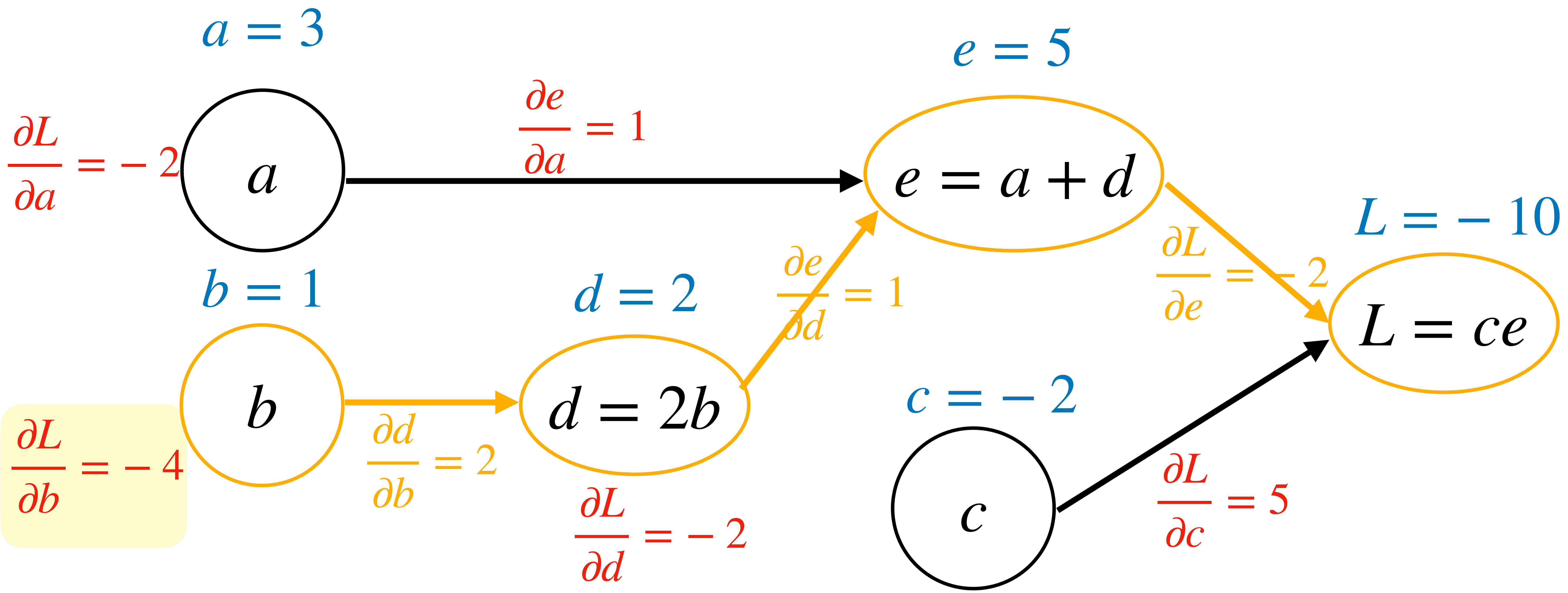
The Chain Rule

$f(x) = u(v(w(x)))$

$$\frac{\partial f}{\partial x} = \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial w} \cdot \frac{\partial w}{\partial x}$$

Responsibility attribution through gradient

- $\underline{L(a,b,c) = c(a+2b)}$. Now let $a = 3, b = 1, c = -2$;
  - **Derivative on an edge**: *local* dependency;
  - **Derivative on node**: *long* dependency all the way from the final node (L)



$d = 2b$

$\Rightarrow \frac{\partial d}{\partial b} = 2$ **Edge: local**

$$\Rightarrow \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

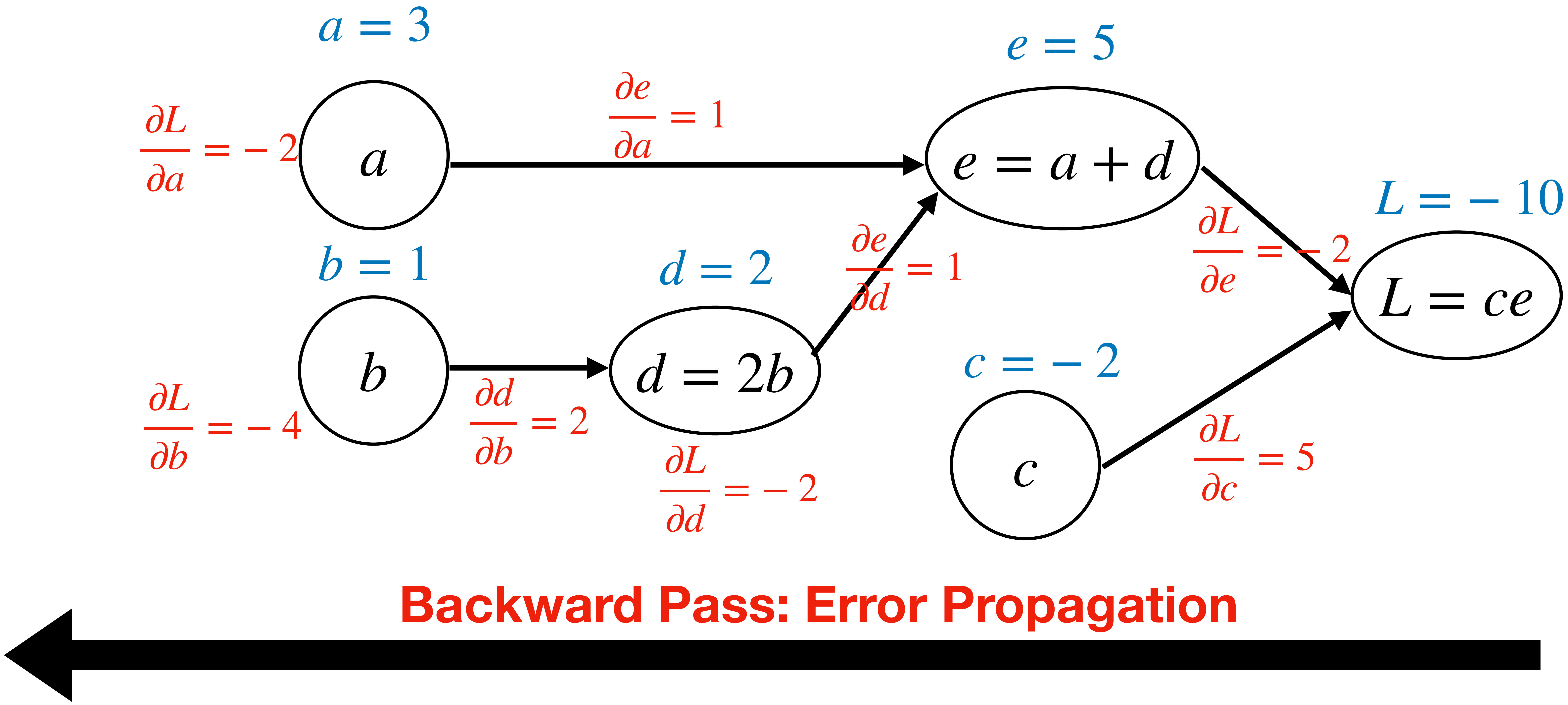
Node: long $= -2 \times 1 \times 2$

$= -4$

Backpropagation → Gradient Descent

Learning/Updating the weights to minimize loss

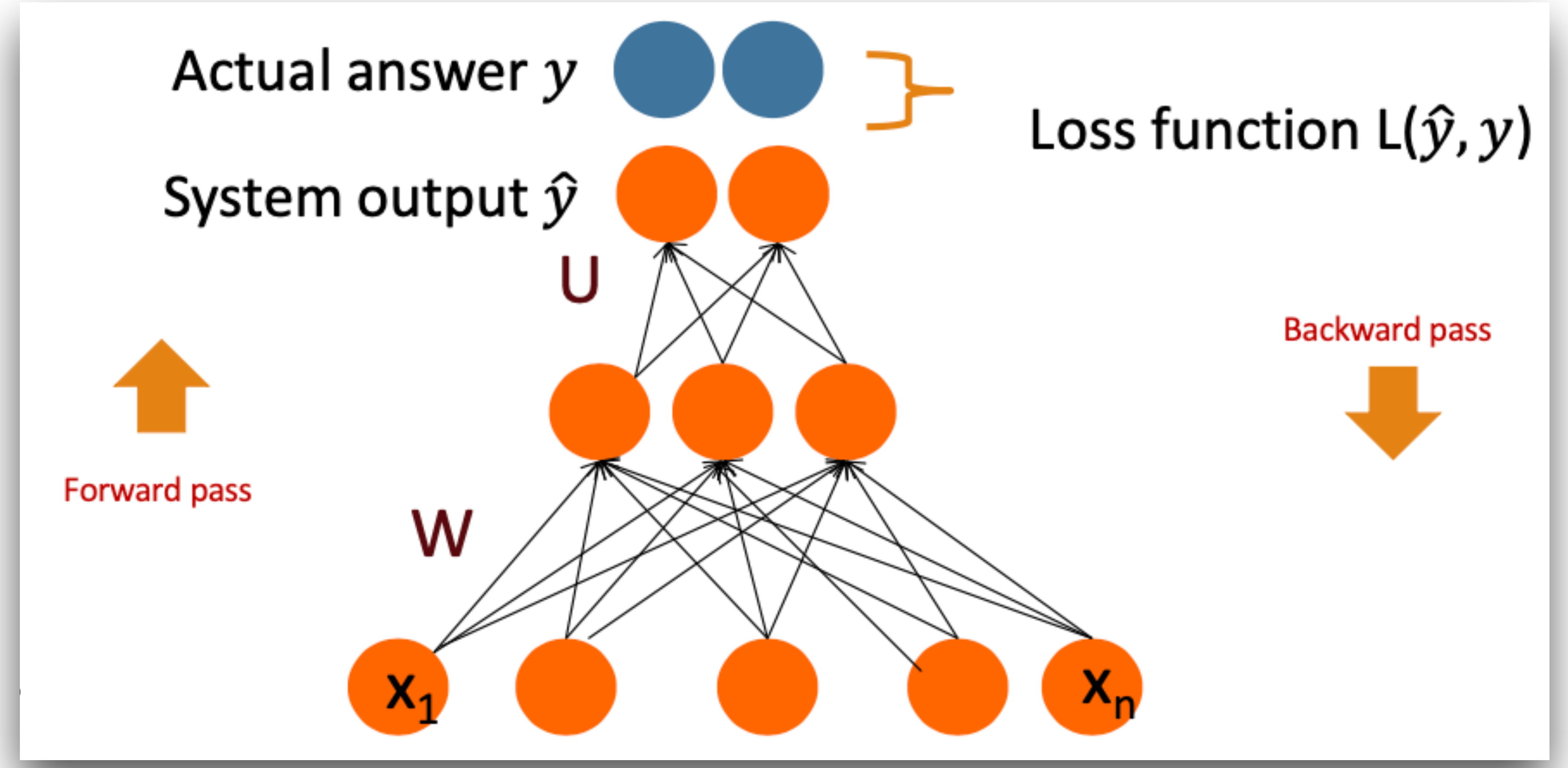
Update each weight: $w' = w - \eta \frac{d}{dw} L(f(x;w), y)$, where $L(f(x;w), y) = L(\hat{y} - y)$



Gradient Descent in Neural Networks

For every training tuple (x, y)

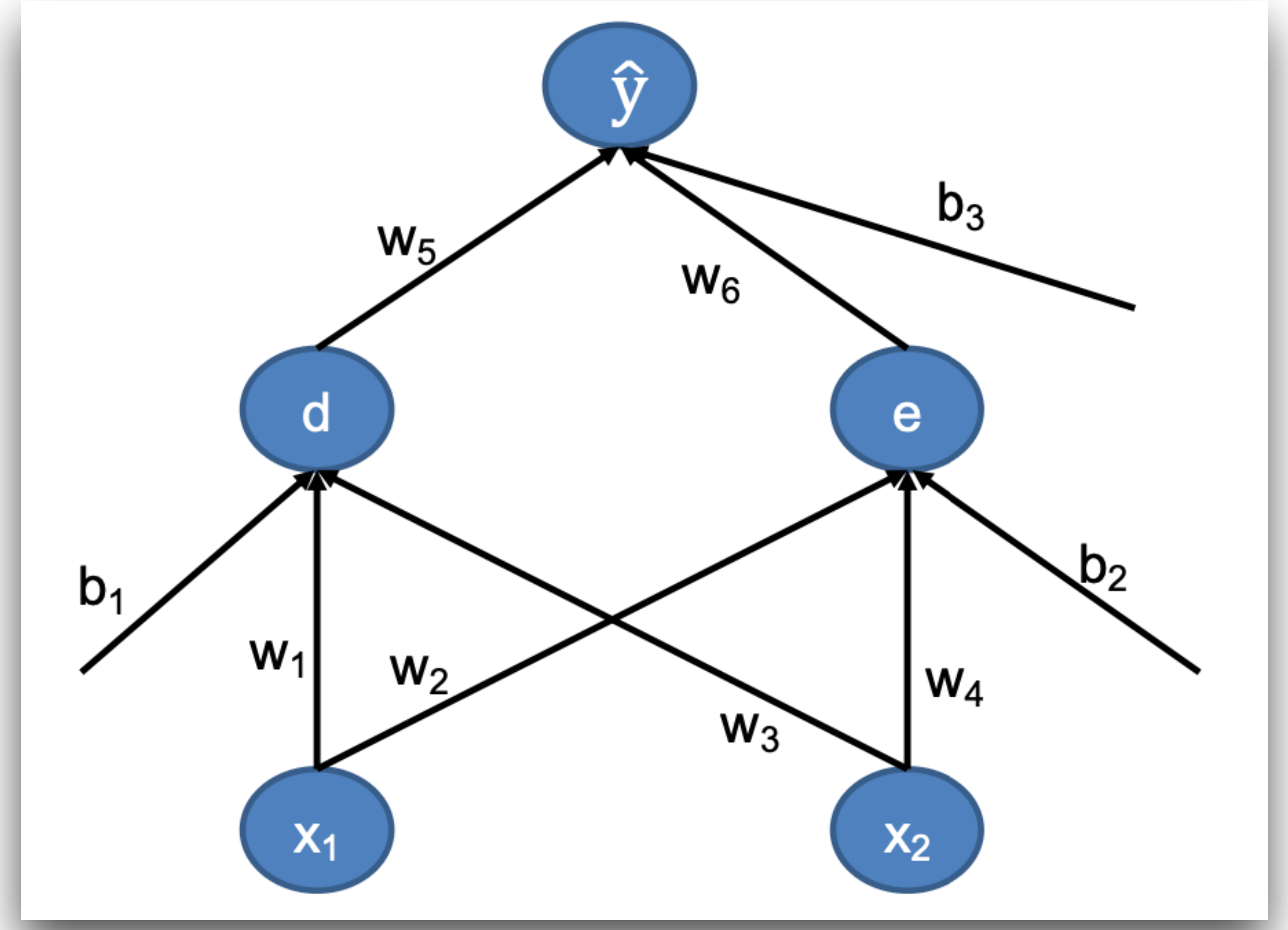
- Run *forward* computation to find our estimate $\hat{y}$
- Run *backward* computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer
      - Update the weight
  - For every hidden node
    - Assess how much blame it deserves for the current answer
    - For every weight w from input layer to the hidden layer
      - Update the weight



The computation quickly blows up

Verify this when you are free!

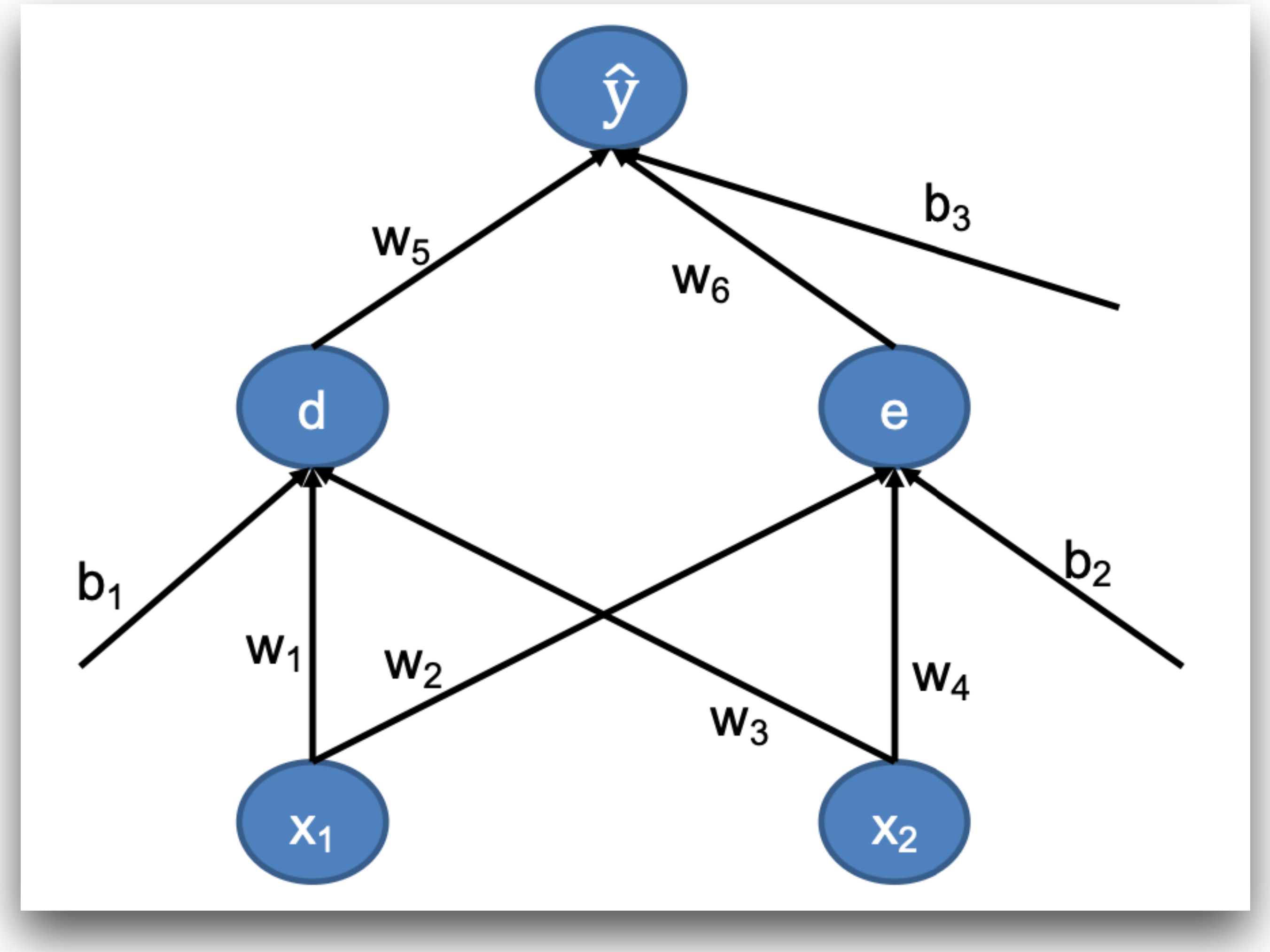
For a simple two-layer neural network:



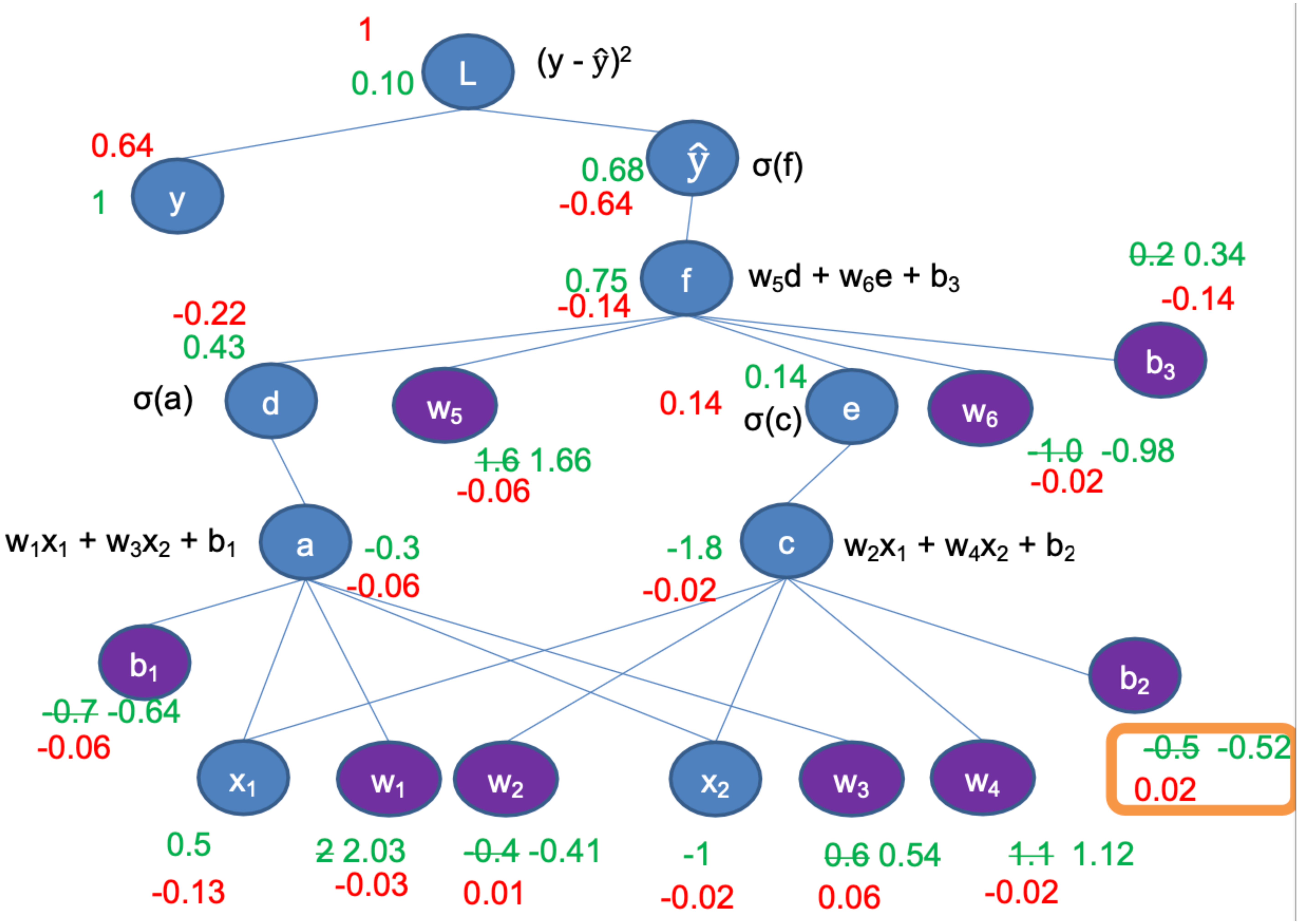
The computation quickly blows up

Verify this when you are free!

For a simple two-layer neural network:



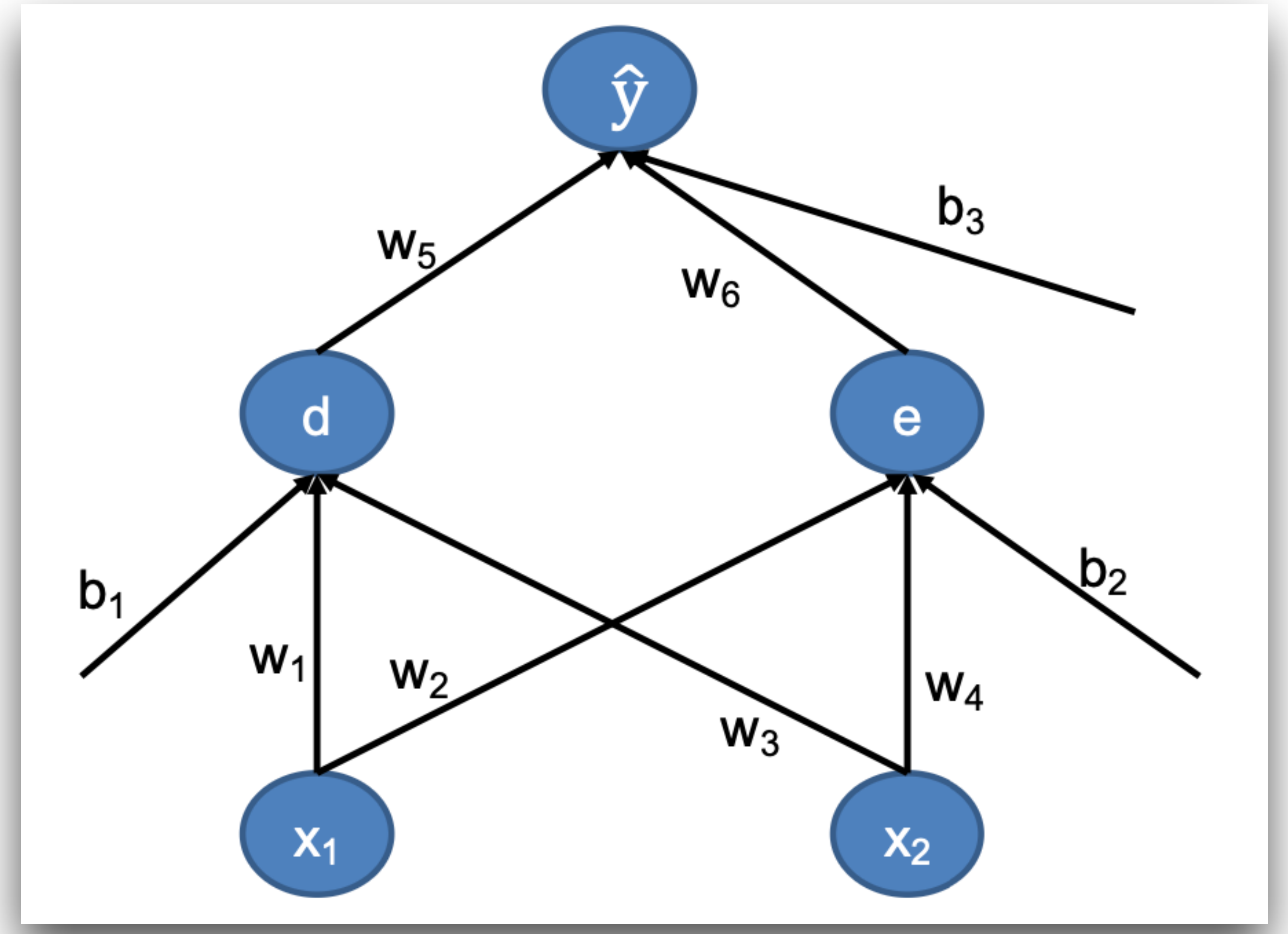
One round of gradient descent:



The computation quickly blows up

Verify this when you are free!

For a simple two-layer neural network:



One round of gradient descent:

Example: Binary logistic regression

- $L(\hat{y}, y) = \ln(1 - \frac{1}{1+ e^{-(w_1x_1+w_2x_2+b)}})$ — Repeat from previous slide
- $L(\hat{y}, y) = \ln(\frac{1+ e^{-(w_1x_1+w_2x_2+b)}}{1+ e^{-(w_1x_1+w_2x_2+b)}} - \frac{1}{1+ e^{-(w_1x_1+w_2x_2+b)}})$ — Algebra
- $L(\hat{y}, y) = \ln(\frac{e^{-(w_1x_1+w_2x_2+b)}}{1+ e^{-(w_1x_1+w_2x_2+b)}})$ — Algebra
- $L(\hat{y}, y) = \ln\left(e^{-(w_1x_1+w_2x_2+b)}\right) - \ln(1 + e^{-(w_1x_1+w_2x_2+b)})$ — Algebra
- $L(\hat{y}, y) = -(w_1x_1 + w_2x_2 + b) - \ln(1 + e^{-(w_1x_1+w_2x_2+b)})$ — Algebra

- $\frac{\partial}{\partial w_1} L(\hat{y}, y) = -x_1 - \frac{1}{1+ e^{-(w_1x_1+w_2x_2+b)}} e^{-(w_1x_1+w_2x_2+b)}(-x_1)$ — Take partial derivative
- $\frac{\partial}{\partial w_1} L(\hat{y}, y) = x_1(-1 + \frac{e^{-(w_1x_1+w_2x_2+b)}}{1+ e^{-(w_1x_1+w_2x_2+b)}})$ — Algebra
- $\frac{\partial}{\partial w_1} L(\hat{y}, y) = x_1(\frac{-1}{1+ e^{-(w_1x_1+w_2x_2+b)}})$ — Algebra

Notes on Loss Functions

Bonus on math

- If you wonder why perceptron learning uses $+$ while neural network uses $-$ to update their weights….!

Perceptron: $\mathbf{w} = \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$; Neural Network: $w' = w - \eta \frac{d}{dw} L(f(x; w), y)$

- **Short answer: they are actually the same thing!** Here is a derivation:

✳ For loss function $L = \frac{1}{2}(y - \hat{y})^2$ or $L = -\left[y \log(\hat{y}) + (1 - y)\log(1 - \hat{y}) \right]$:

✳ The gradient with respect to weight w_i turns out to be: $\frac{\partial L}{\partial w_i} = -(y - \hat{y})\, x_i$

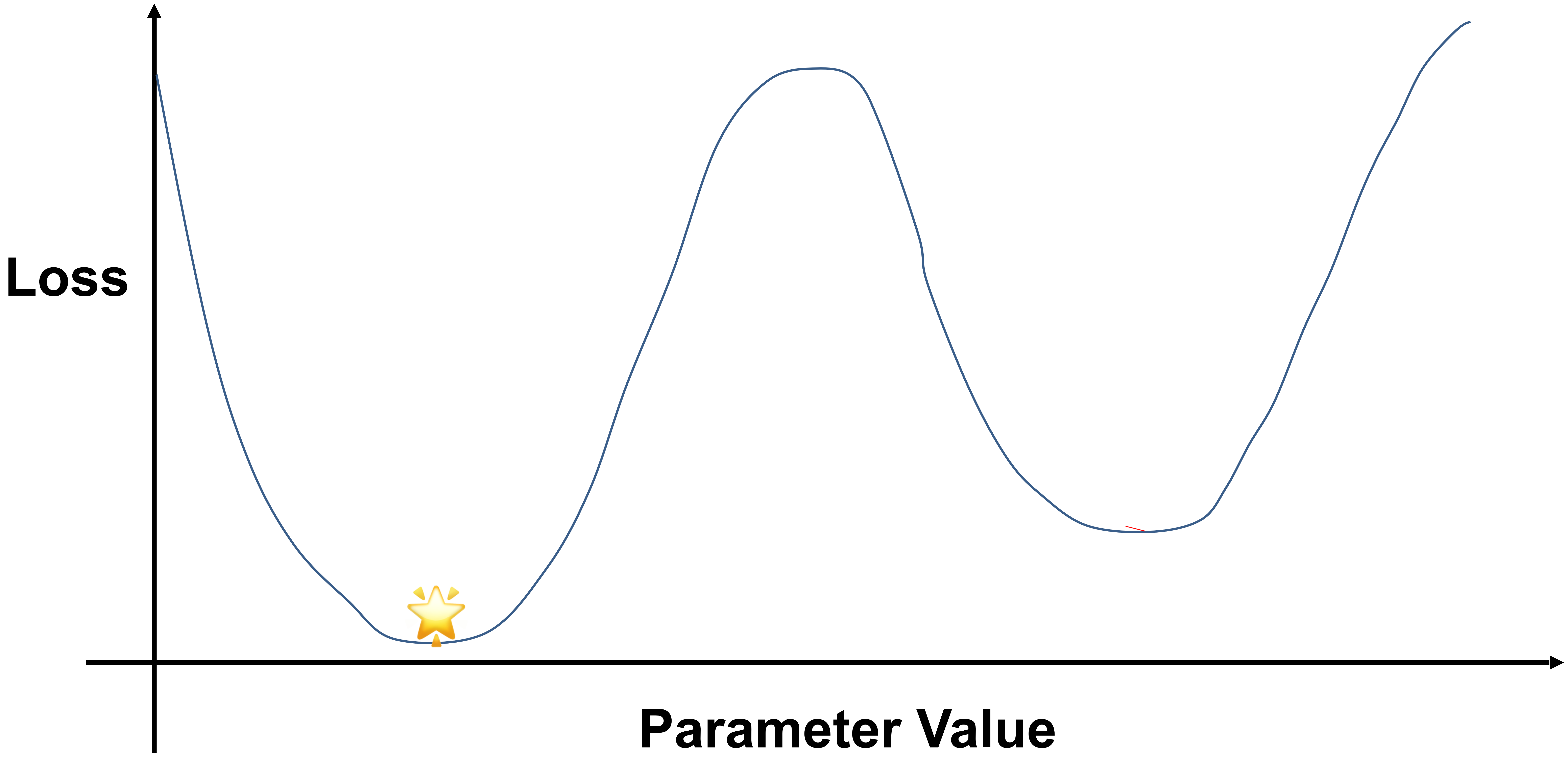
✳ If we plug this into the gradient descent formula:

$$\mathbf{w} := \mathbf{w} - \eta \frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \eta \left(-(y - \hat{y})\, \mathbf{x} \right) = \mathbf{w} + \eta(y - \hat{y})\, \mathbf{x}$$

✳ The two minus signs cancels, giving the same direction as perception update.

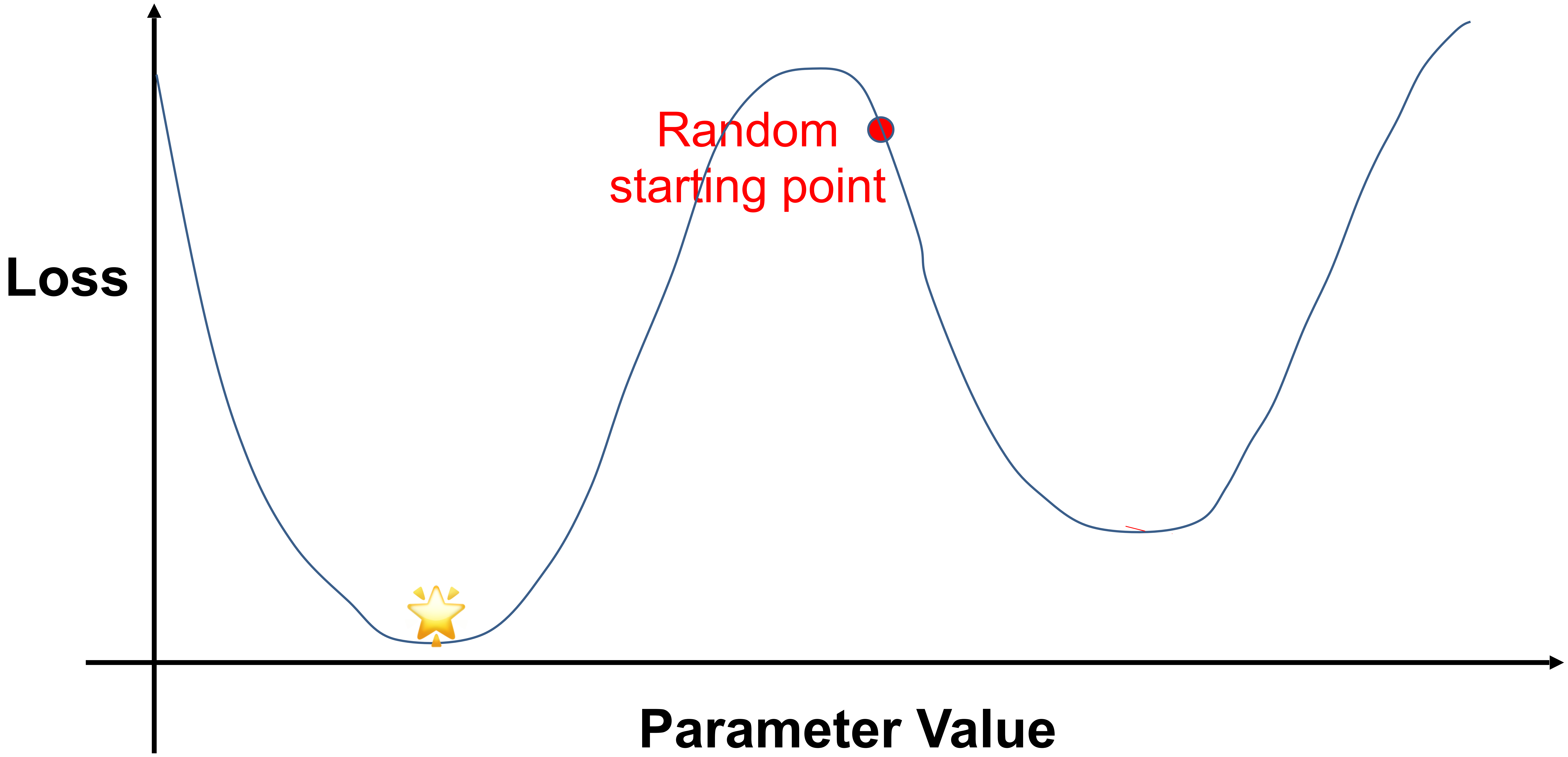
Gradient Descent: Intuitions in 1D Space

Learning to minimize loss (as an optimization problem)



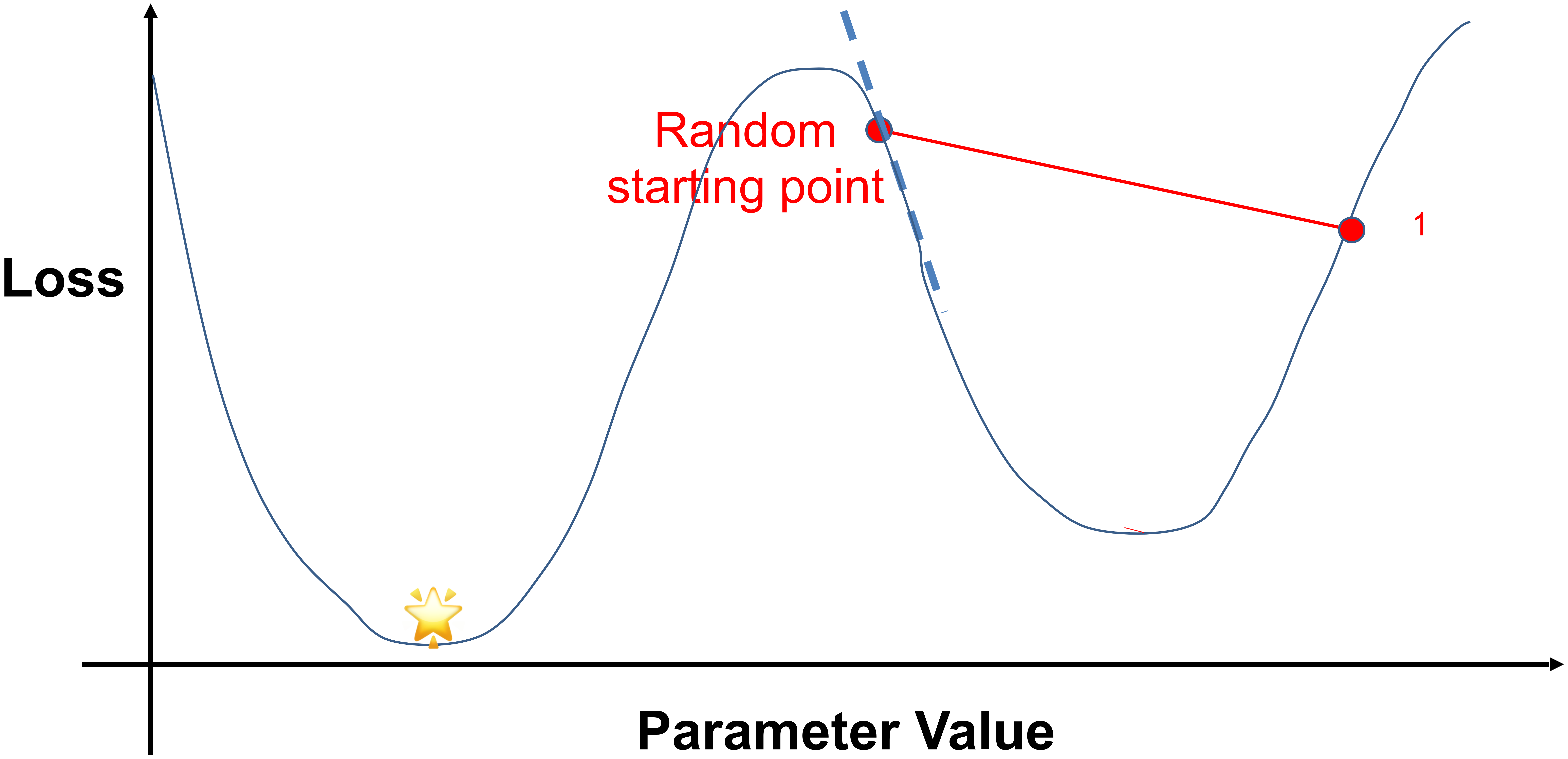
Gradient Descent: Intuitions in 1D Space

Learning to minimize loss (as an optimization problem)



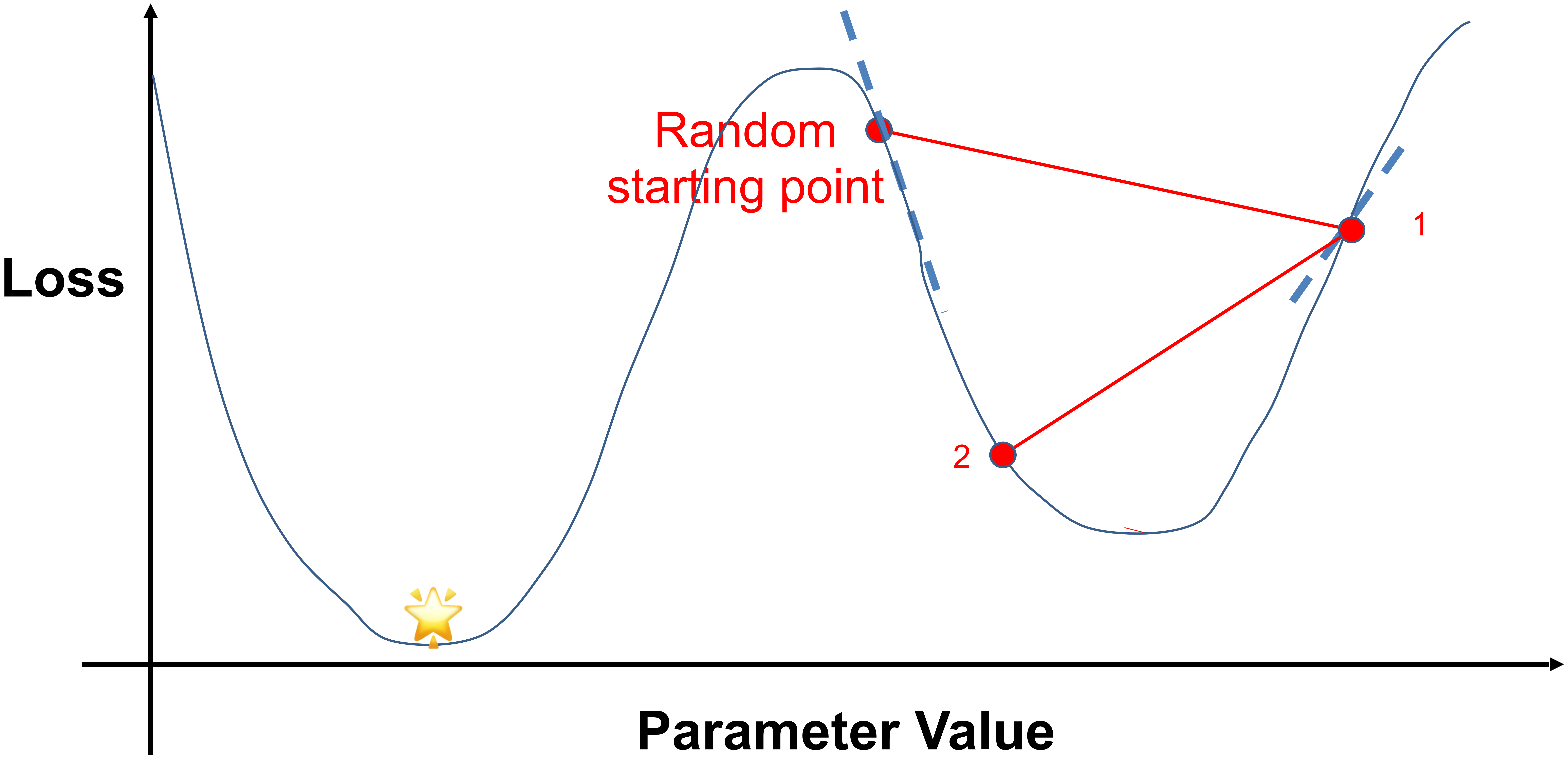
Gradient Descent: Intuitions in 1D Space

Learning to minimize loss (as an optimization problem)



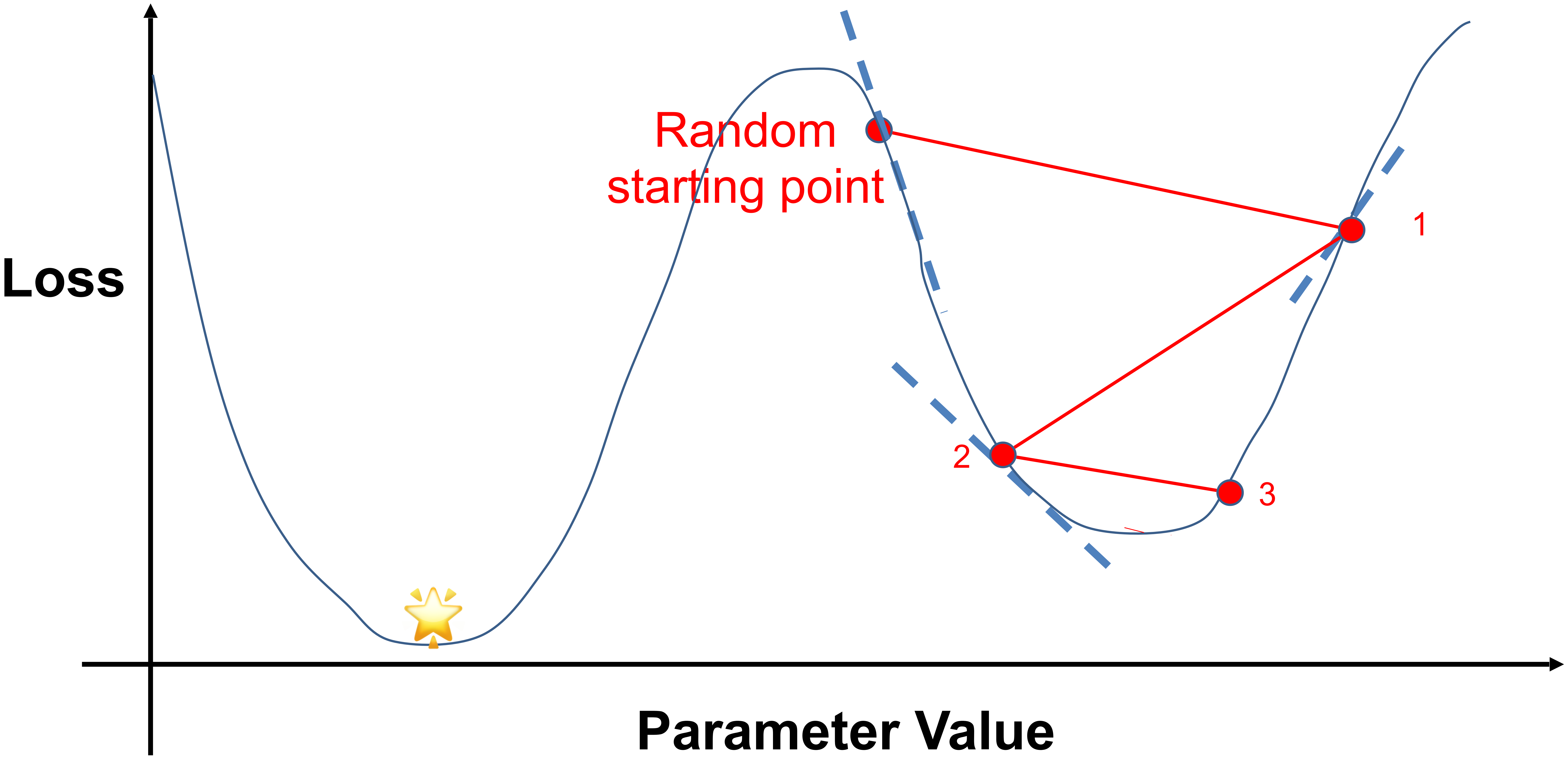
Gradient Descent: Intuitions in 1D Space

Learning to minimize loss (as an optimization problem)



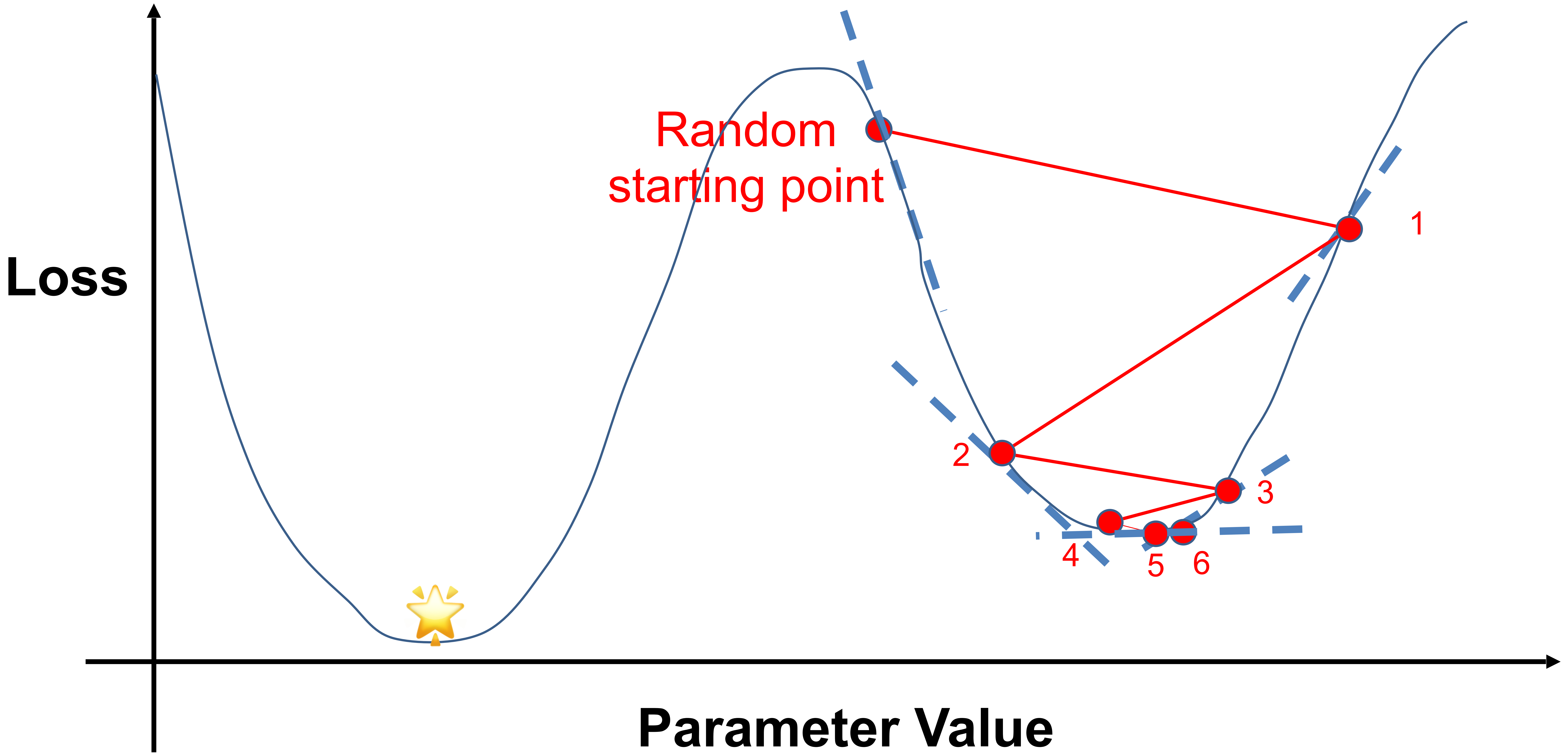
Gradient Descent: Intuitions in 1D Space

Learning to minimize loss (as an optimization problem)



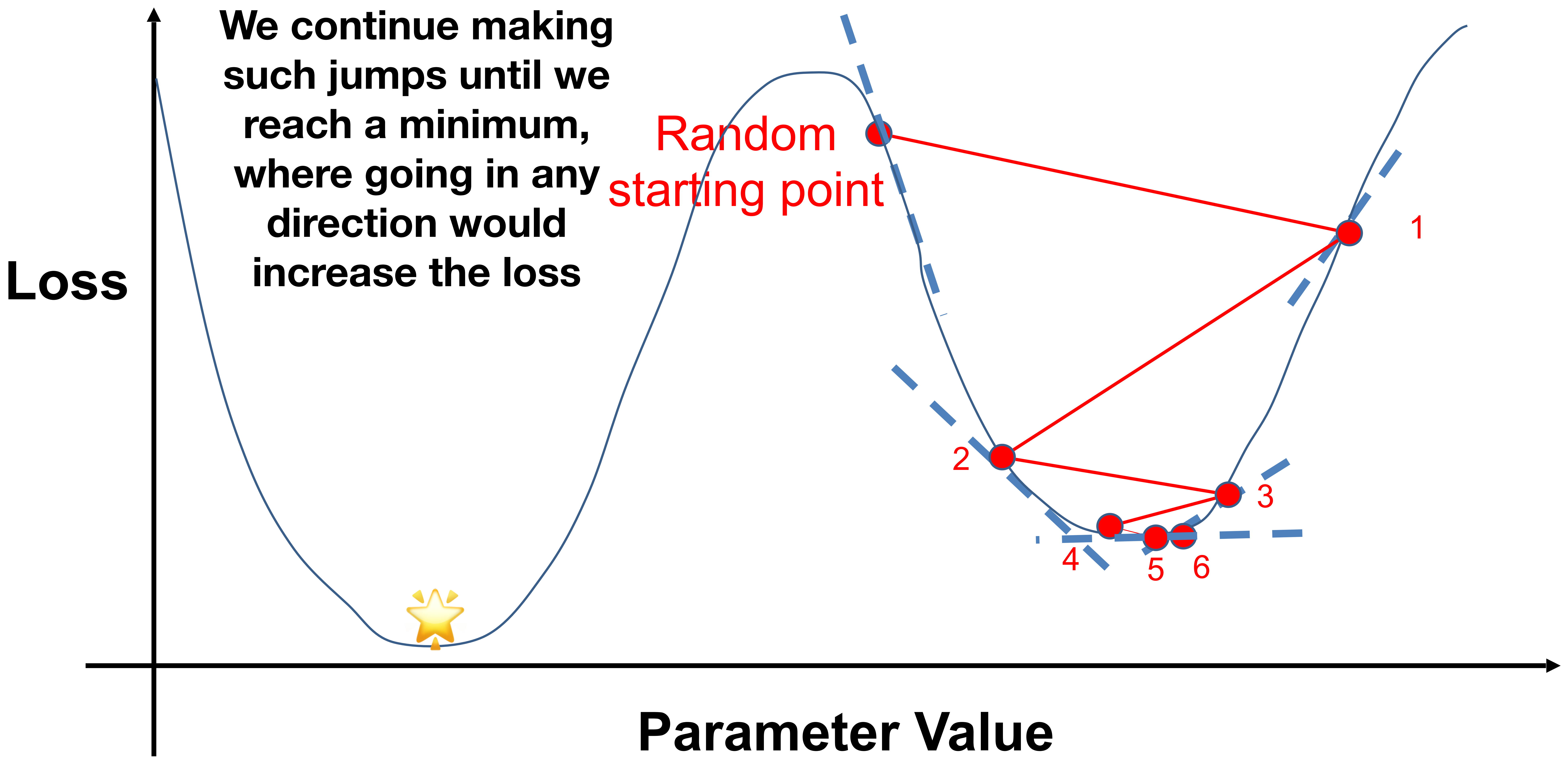
Gradient Descent: Intuitions in 1D Space

Learning to minimize loss (as an optimization problem)



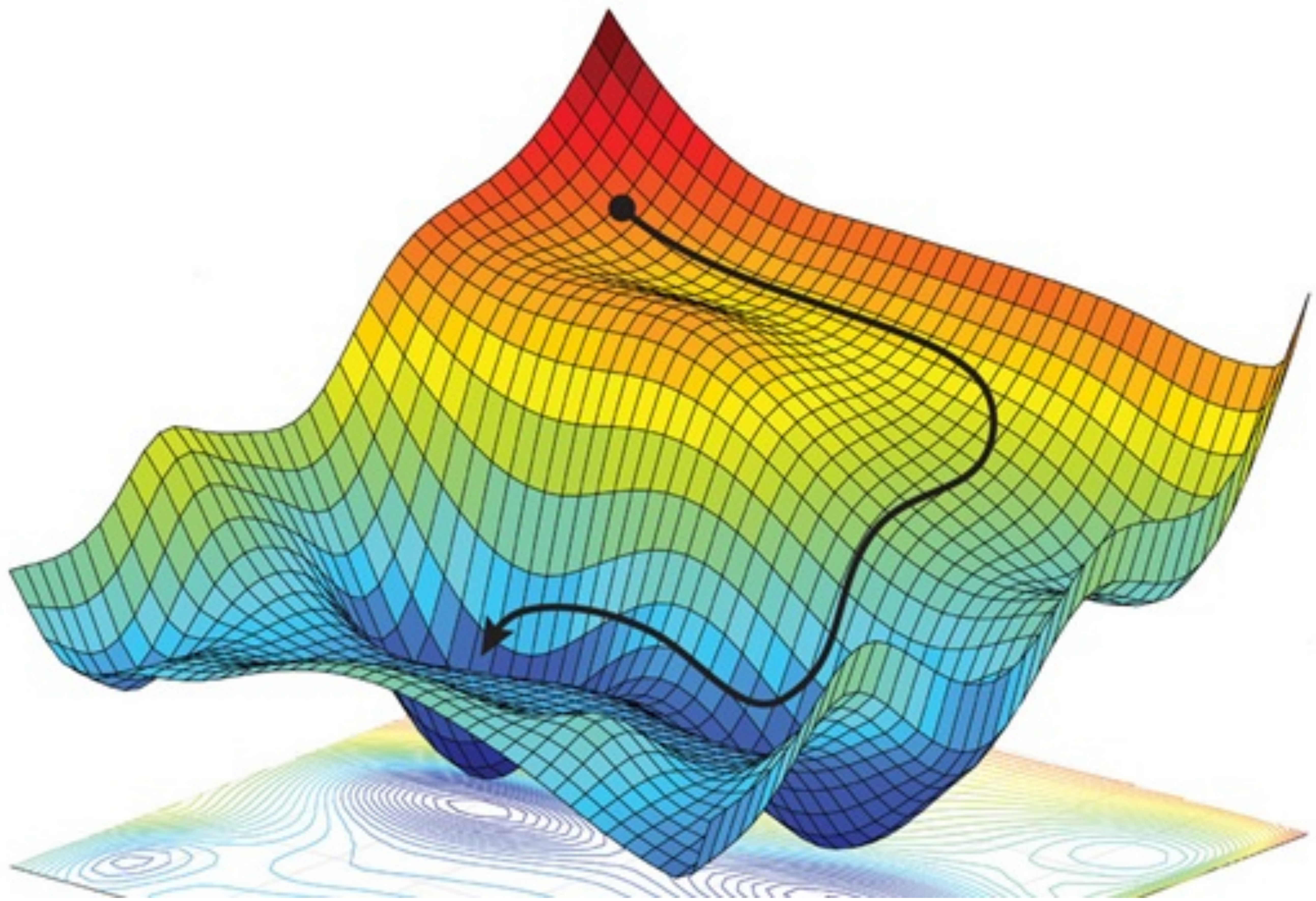
Gradient Descent: Intuitions in 1D Space

Learning to minimize loss (as an optimization problem)



Gradient Descent: Intuitions in High Dimensions

Learning to minimize loss (as an optimization problem)



Interium Summary 4

Backpropagation and Gradient Descent

Interium Summary 4

Backpropagation and Gradient Descent

- **Backpropagation** (a method for efficiently computing the gradients) tells us *what the gradients are* for each weight.
  - Computing dependencies across layers through chain rule in computation graphs;

Interium Summary 4

Backpropagation and Gradient Descent

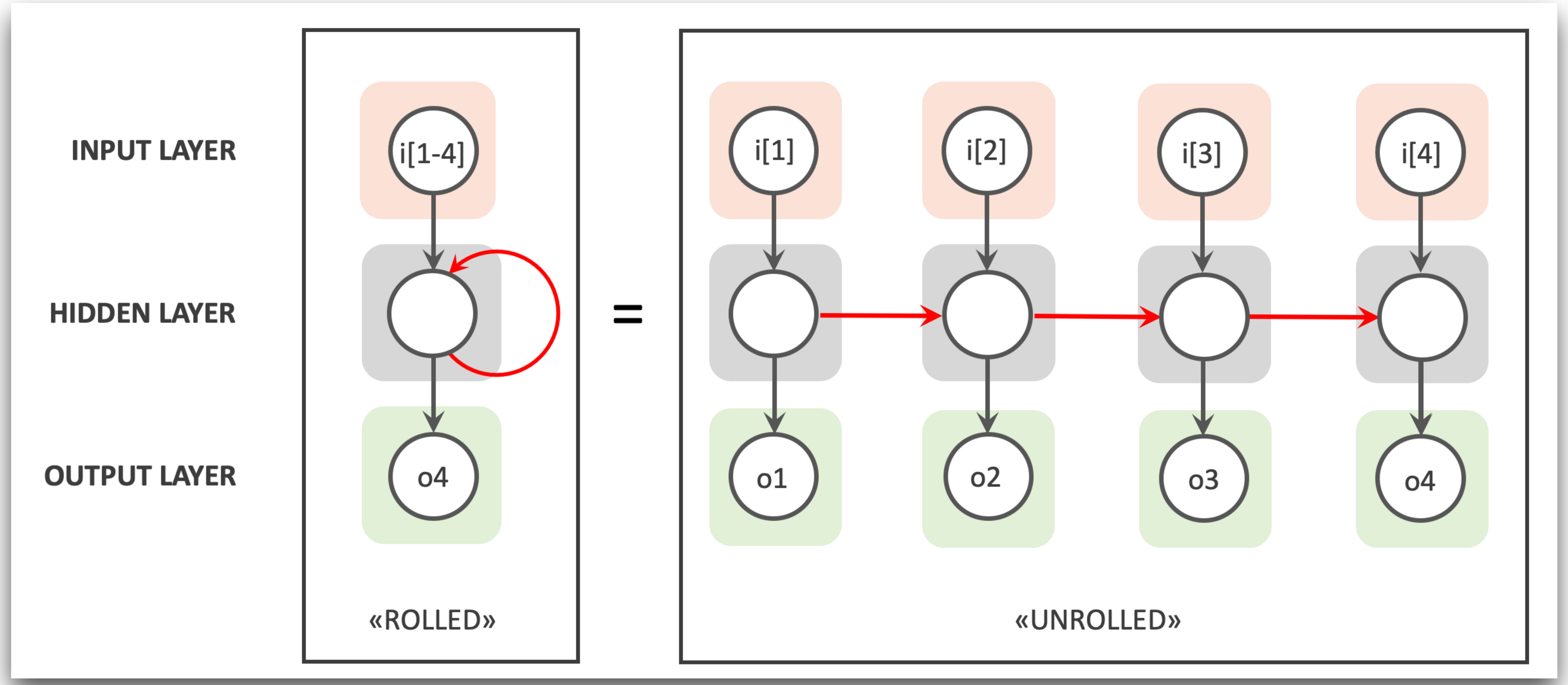
- **Backpropagation** (a method for efficiently computing the gradients) tells us *what the gradients are* for each weight.
  - Computing dependencies across layers through chain rule in computation graphs;
- **Gradient descent** (an optimization algorithm) tells us *how to update* the weight.
  - Iteratively optimizing to minimize the loss for each training example.
  - A forward pass (generating prediction) followed by a backward pass (computing the gradient and updating the weights).

Development of Neural Network Architectures

1990s - present

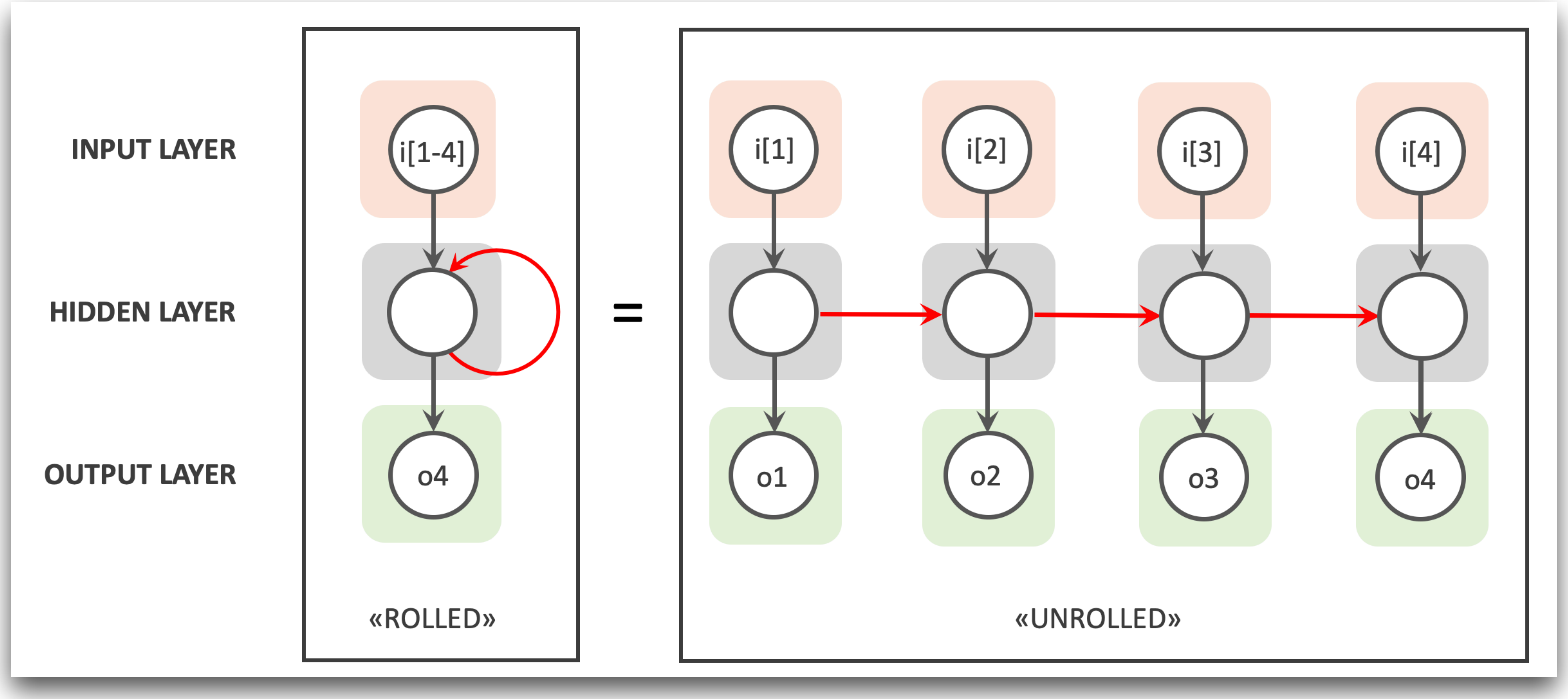
Recurrent Neural Network (RNN)

Introducing the notion of *Time*



Recurrent Neural Network (RNN)

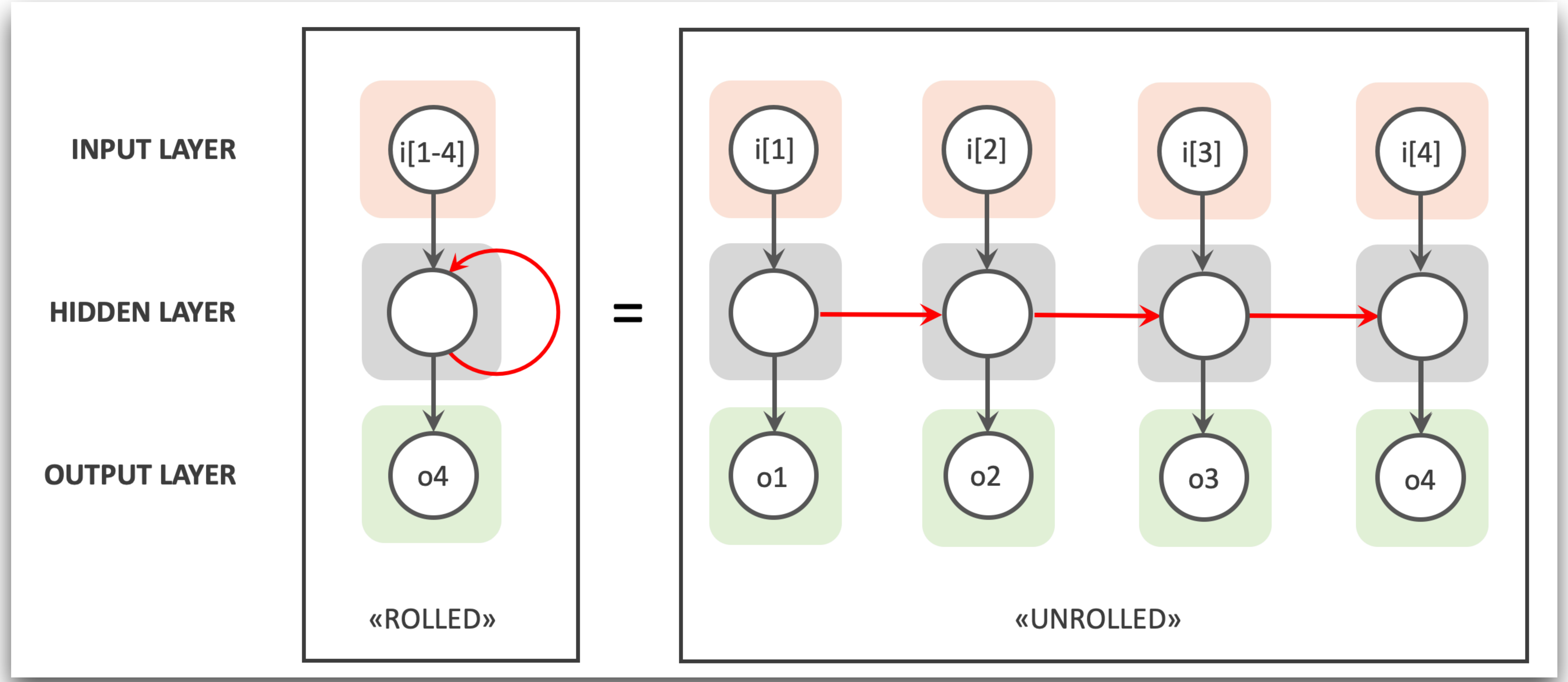
Introducing the notion of *Time*



- **Motivation:** Standard feed-forward networks couldn't handle **sequential or time-dependent data**, since they treat all inputs as independent.

Recurrent Neural Network (RNN)

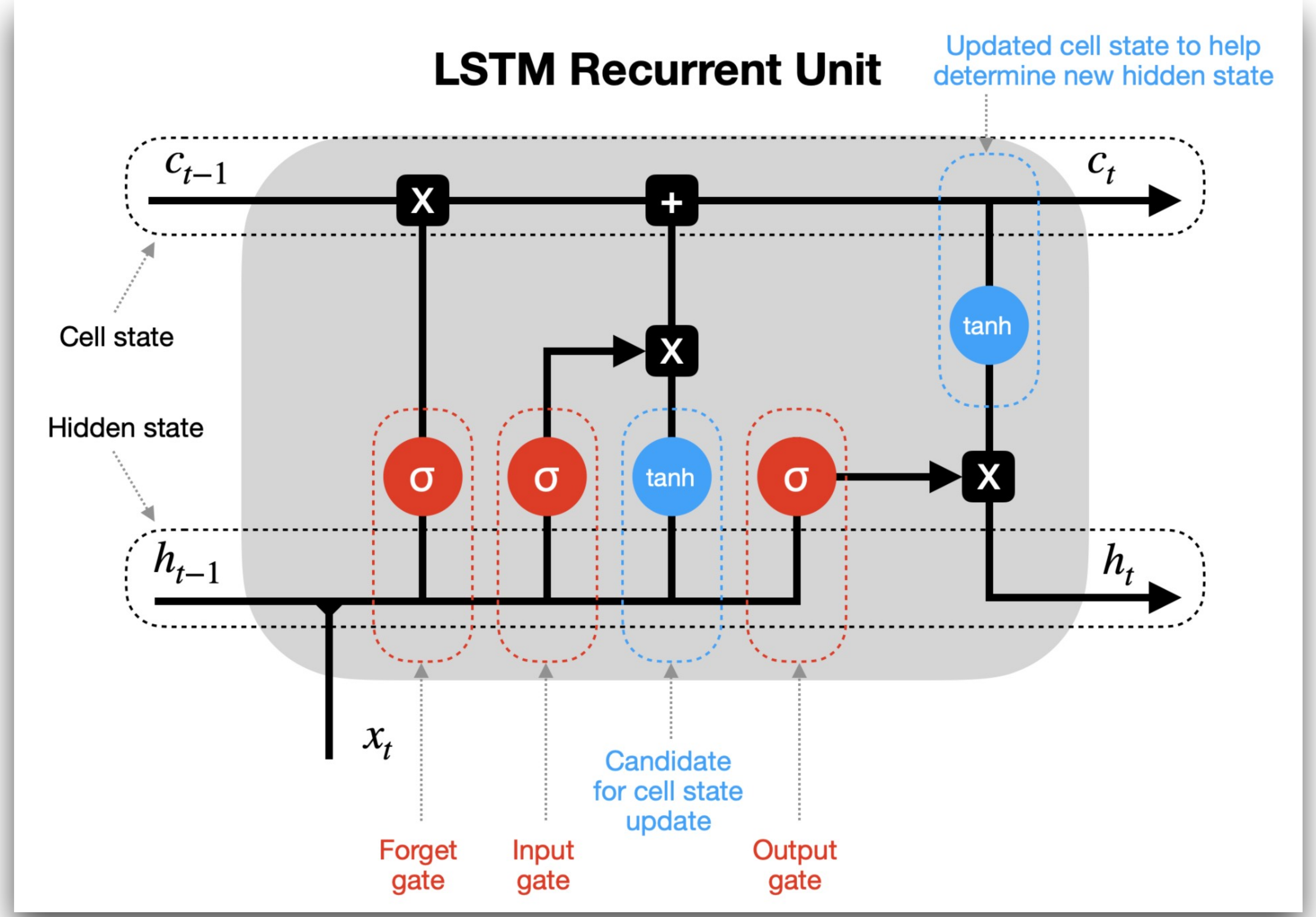
Introducing the notion of *Time*



- **Motivation:** Standard feed-forward networks couldn't handle **sequential or time-dependent data**, since they treat all inputs as independent.
- **Usefulness:** RNNs introduce ***recurrent connections* that let information persist across time steps**, enabling modeling of language, speech, and temporal patterns.

Long Short-Term Memory (LSTM)

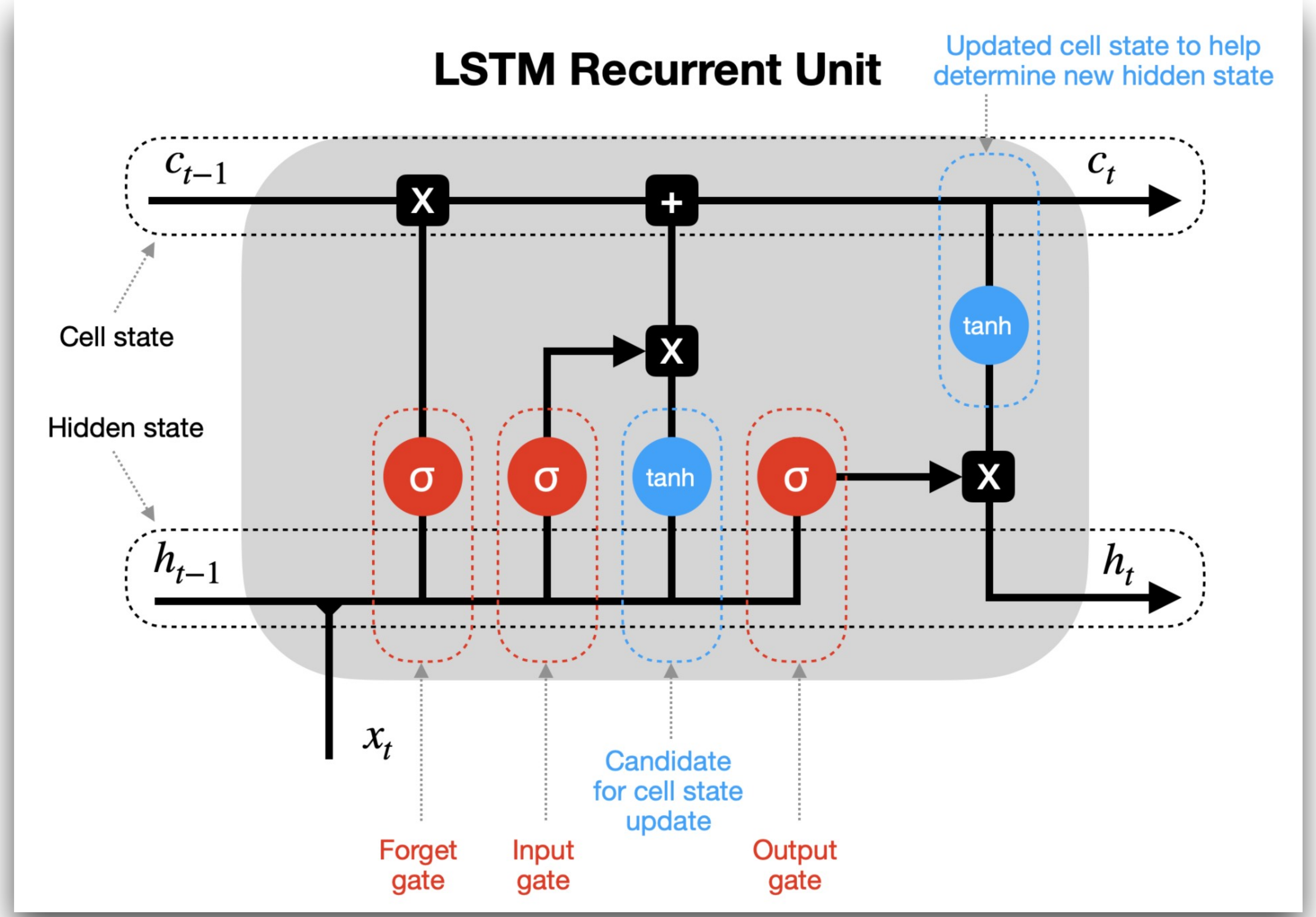
Handling long-distance dependencies through managing memory



Long Short-Term Memory (LSTM)

Handling long-distance dependencies through managing memory

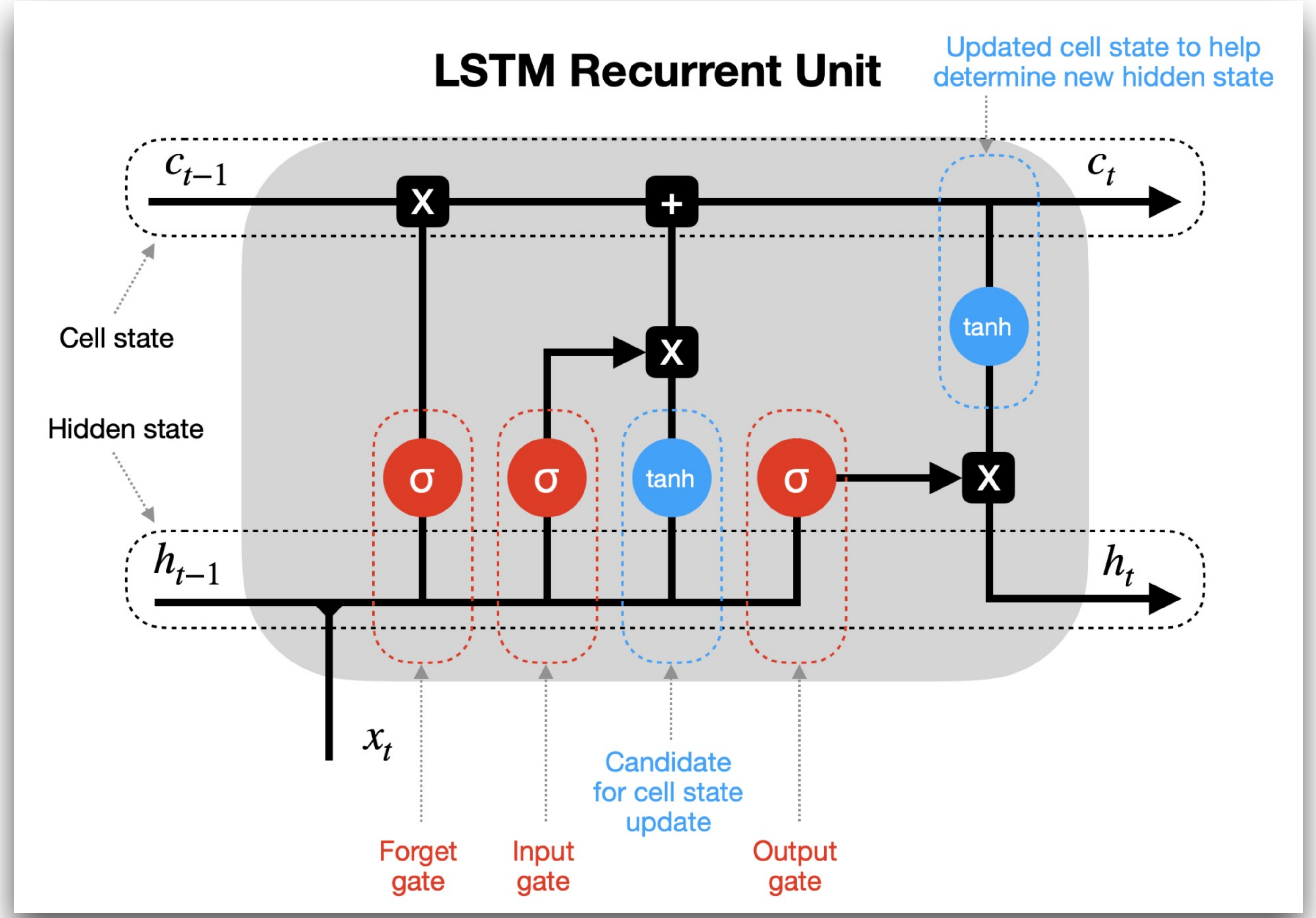
- **Motivation:** Vanilla RNNs struggled with *long-term dependencies* because of vanishing/exploding gradients during training.



Long Short-Term Memory (LSTM)

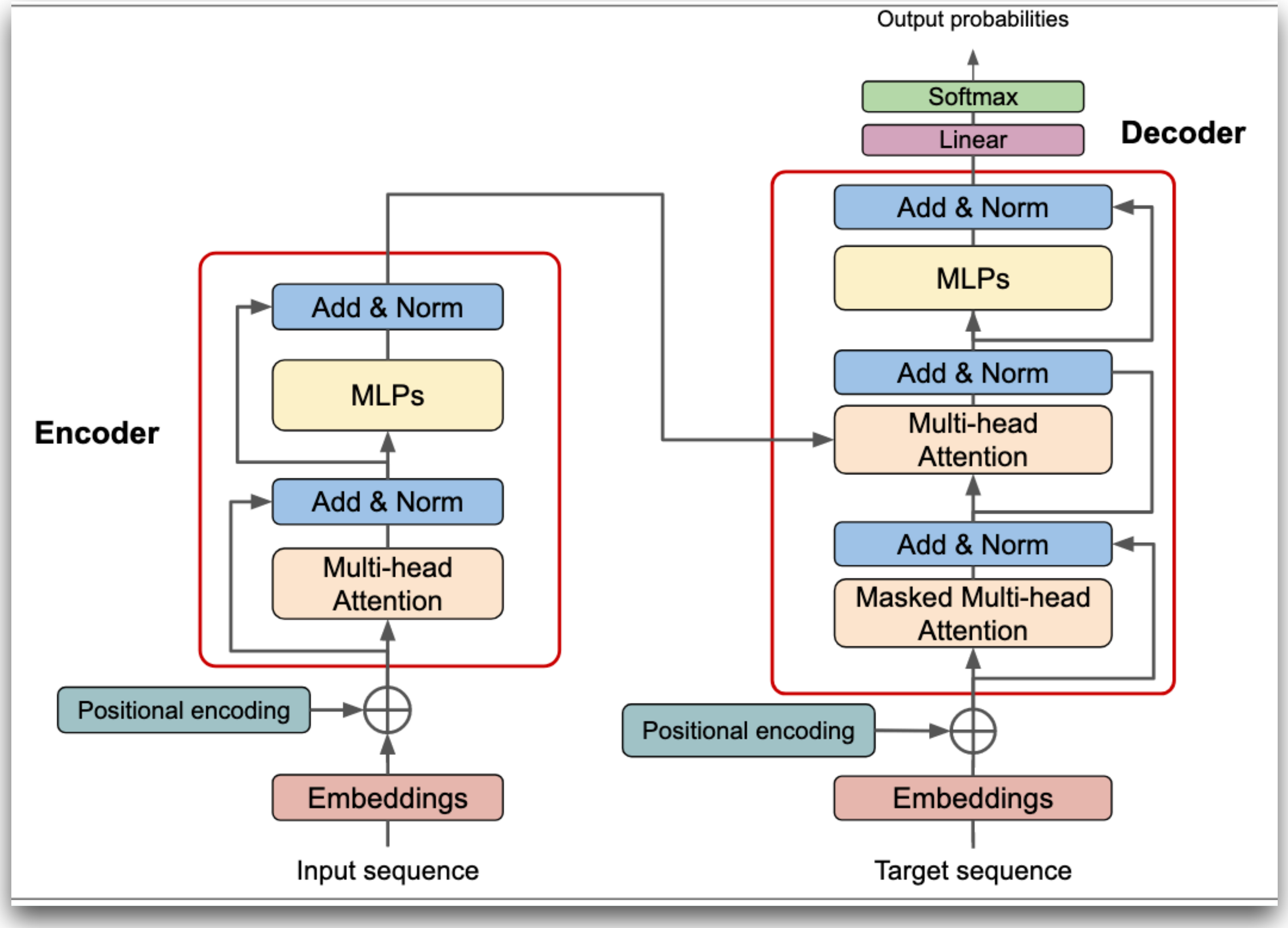
Handling long-distance dependencies through managing memory

- **Motivation:** Vanilla RNNs struggled with *long-term dependencies* because of vanishing/exploding gradients during training.
- **Usefulness:** LSTMs use *gated cells* to *selectively remember or forget information*, allowing **stable learning over long sequences**—powering early breakthroughs in speech recognition, translation, and text generation.



Transformers

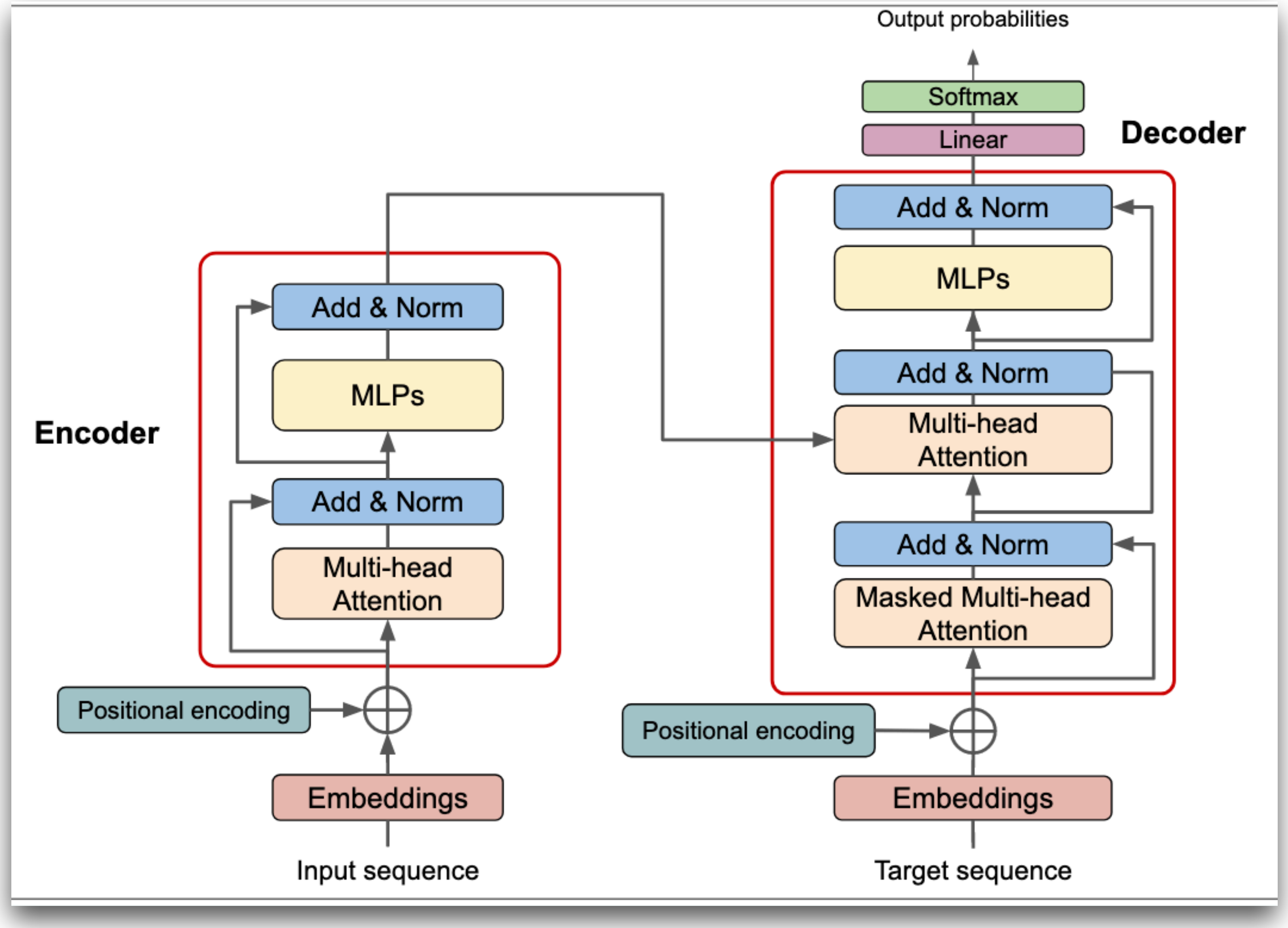
Global information access



Transformers

Global information access

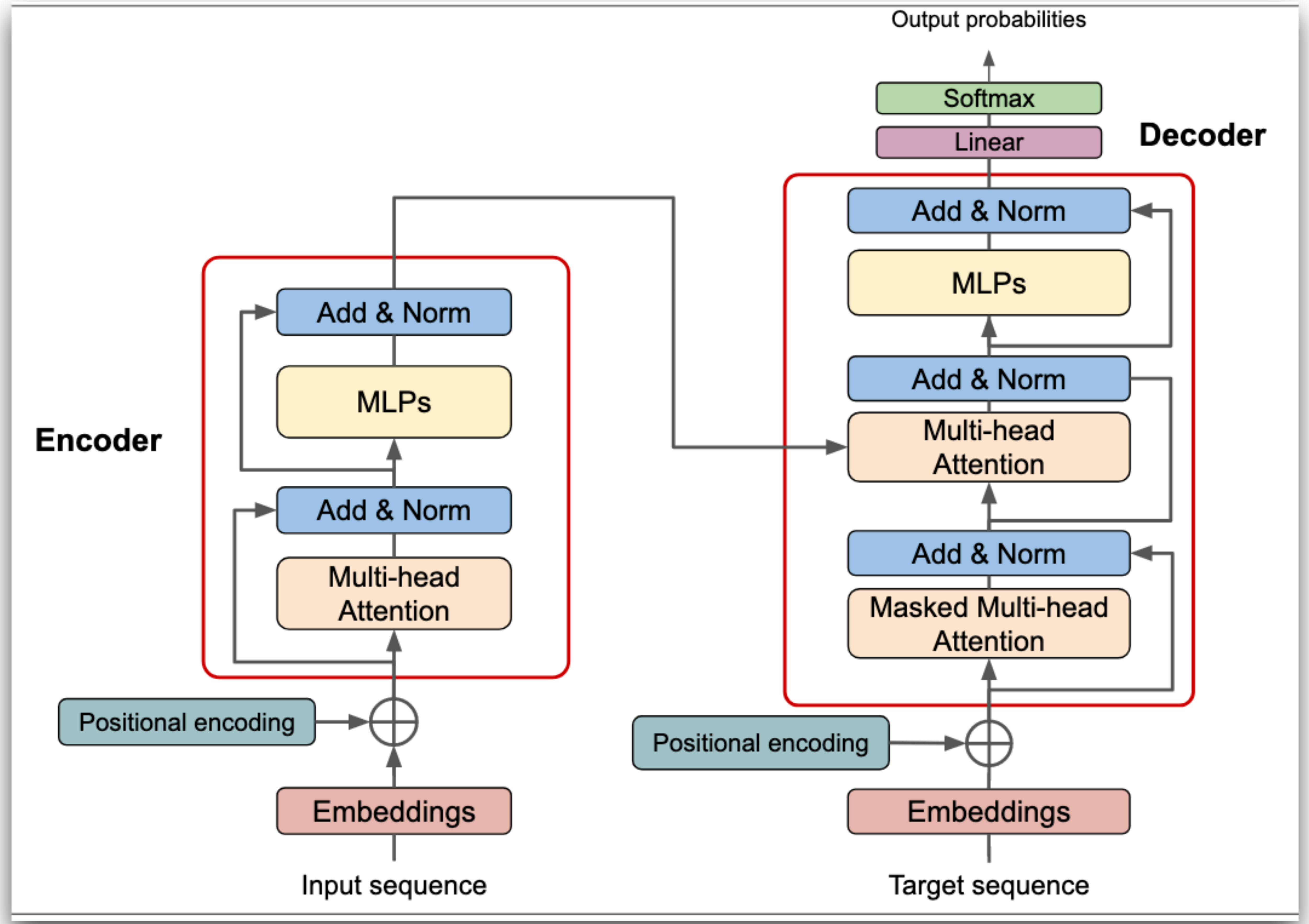
- **Motivation:** Even LSTMs process sequences step-by-step, limiting *parallelization* and *global context* access.



Transformers

Global information access

- **Motivation:** Even LSTMs process sequences step-by-step, limiting *parallelization* and *global context* access.
- **Usefulness:** Transformers replace recurrence with ***self-attention***, letting the model **directly relate every token to every other**—making large-scale training efficient and forming the foundation of modern large language models.



Next Time (after fall break): Tom McCoy presenting on LLMs

Thank you for listening!

Slides are partially adapted from Bob Frank (left) and Tom McCoy (right)

And their slides are adpated from Jurafsky and Martin: https://web.stanford.edu/~jurafsky/slp3/