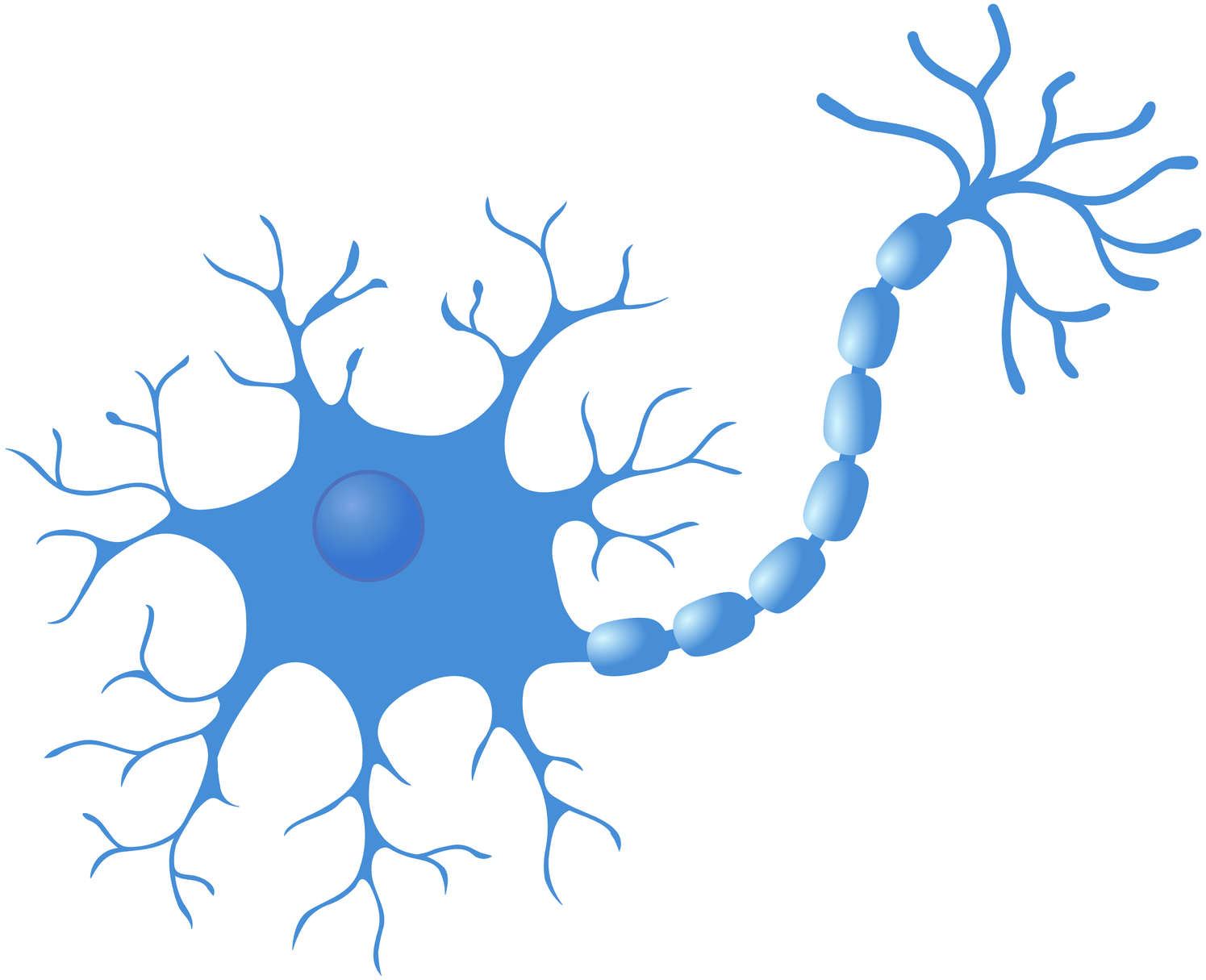
CMPU 366 · Natural Language Processing

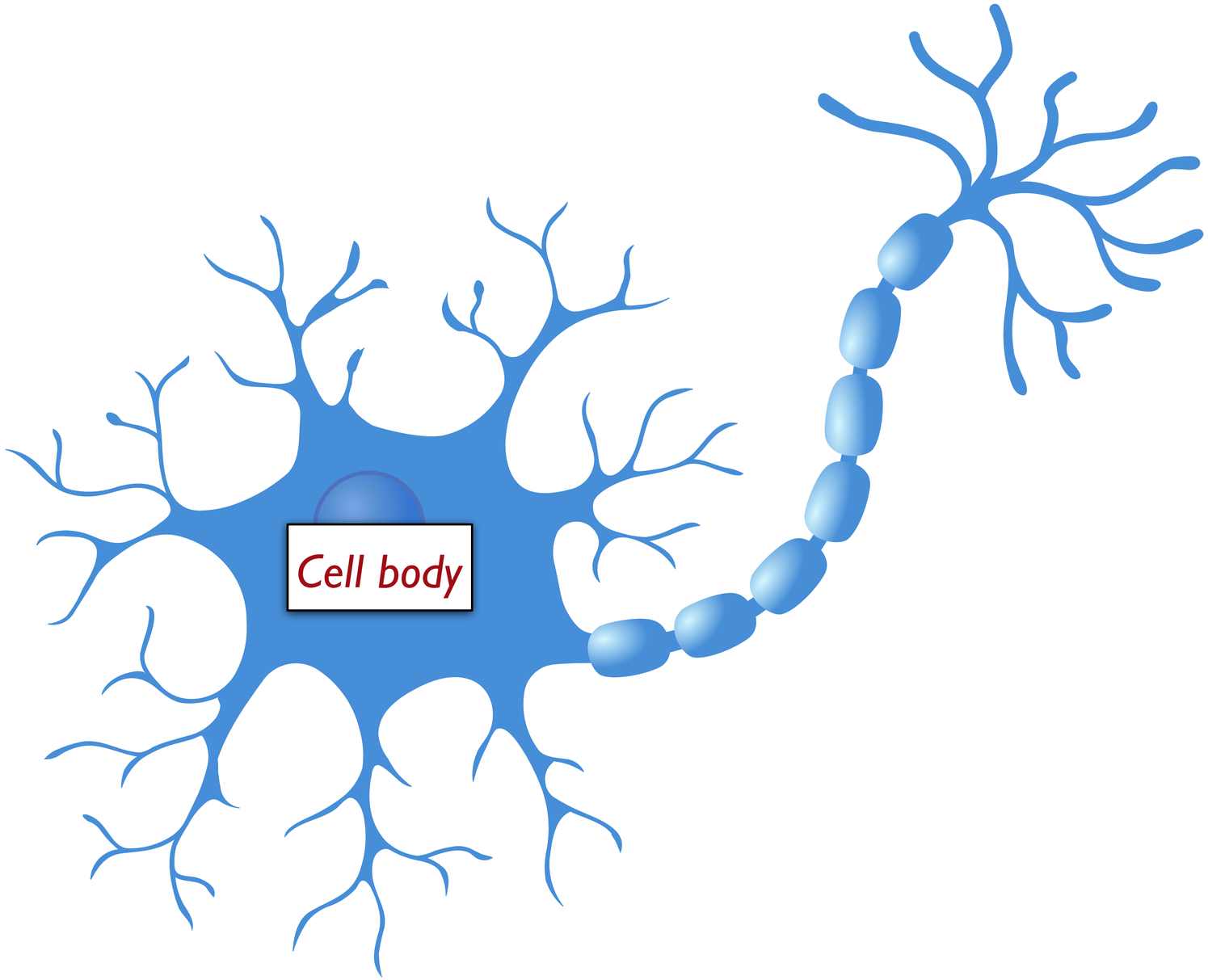
Neural Networks

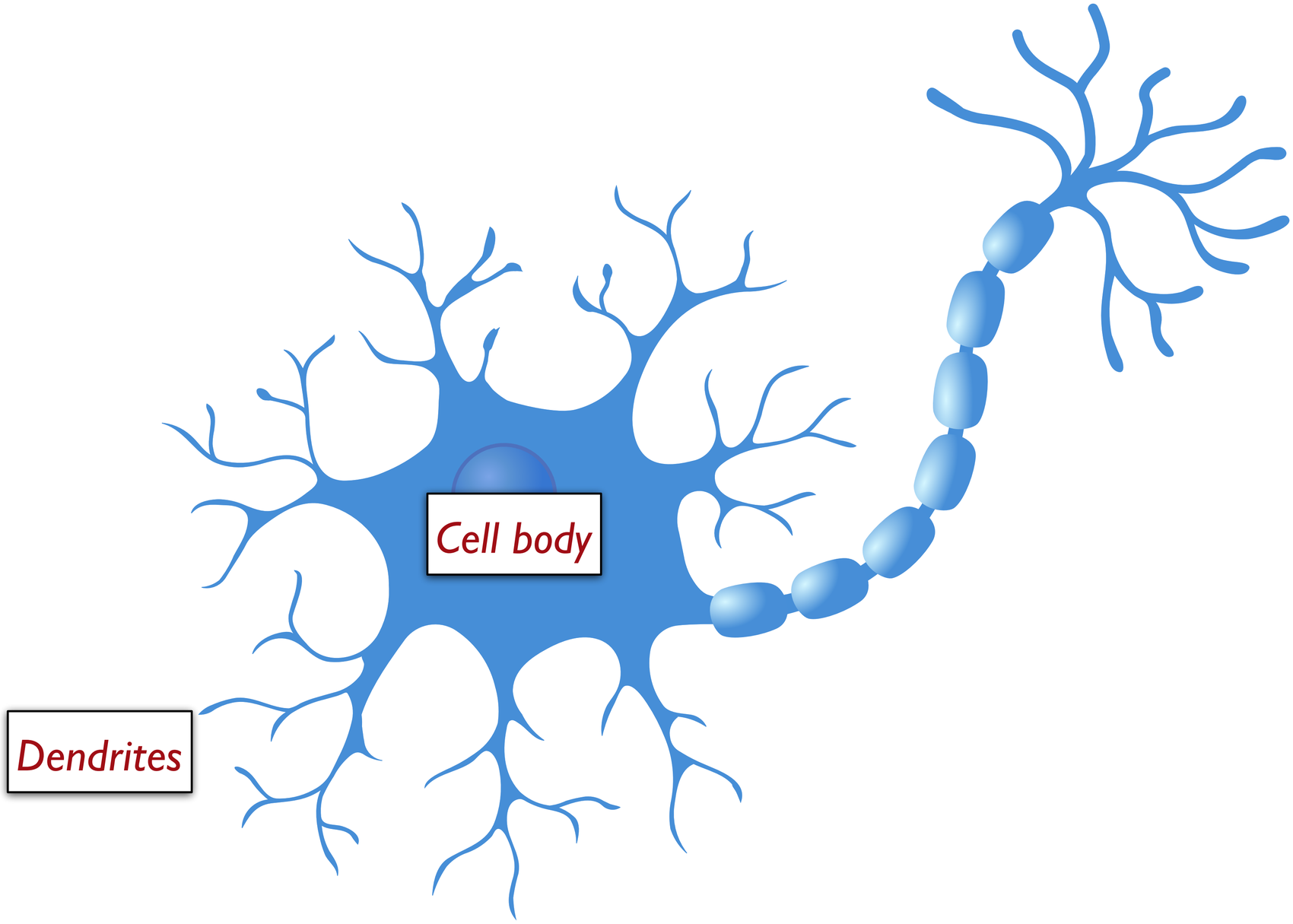
6 October 2025

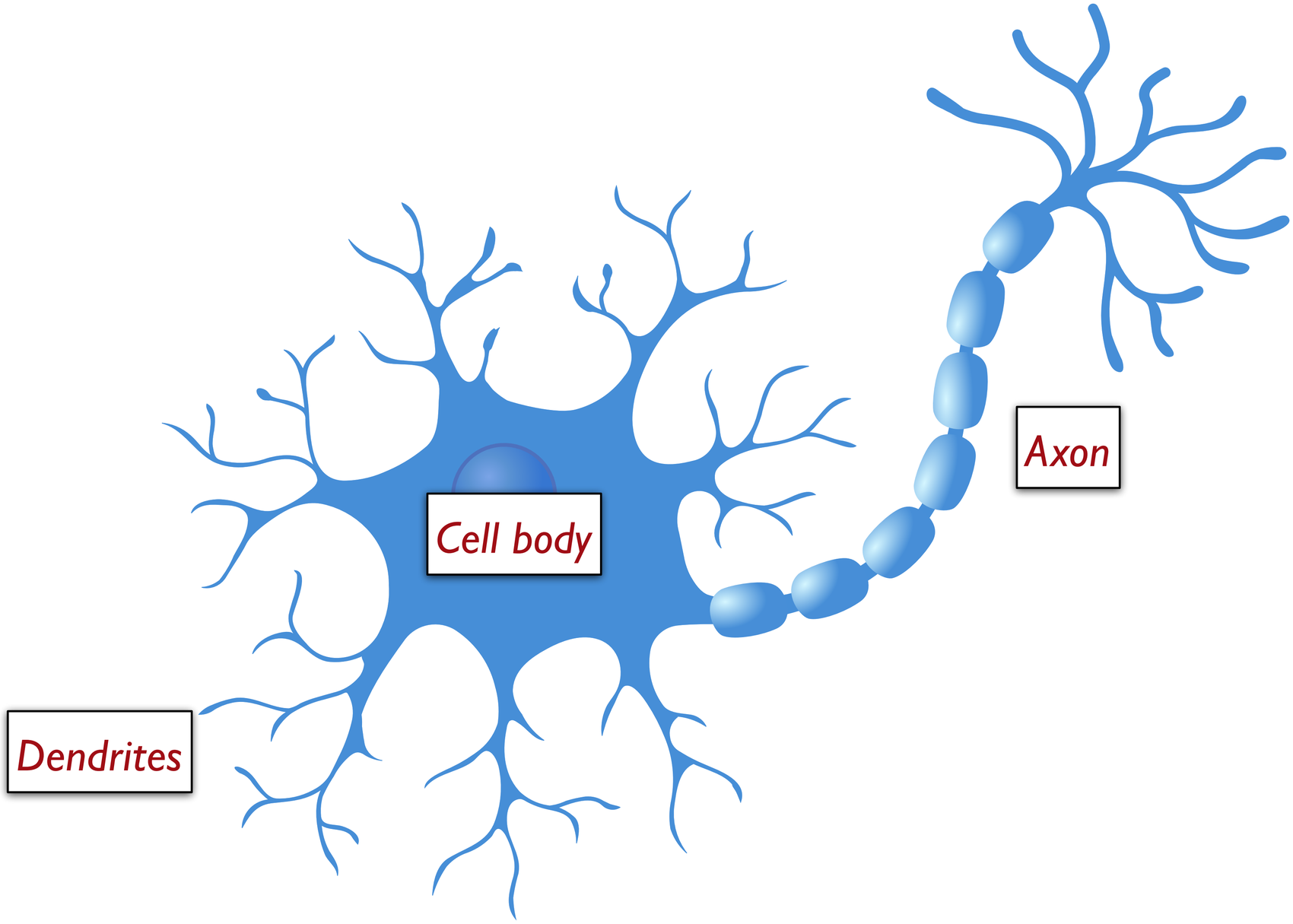
Artificial *neural networks* are an essential computational tool for language processing, and *deep learning* neural network architectures have been at the core of NLP research in recent years.

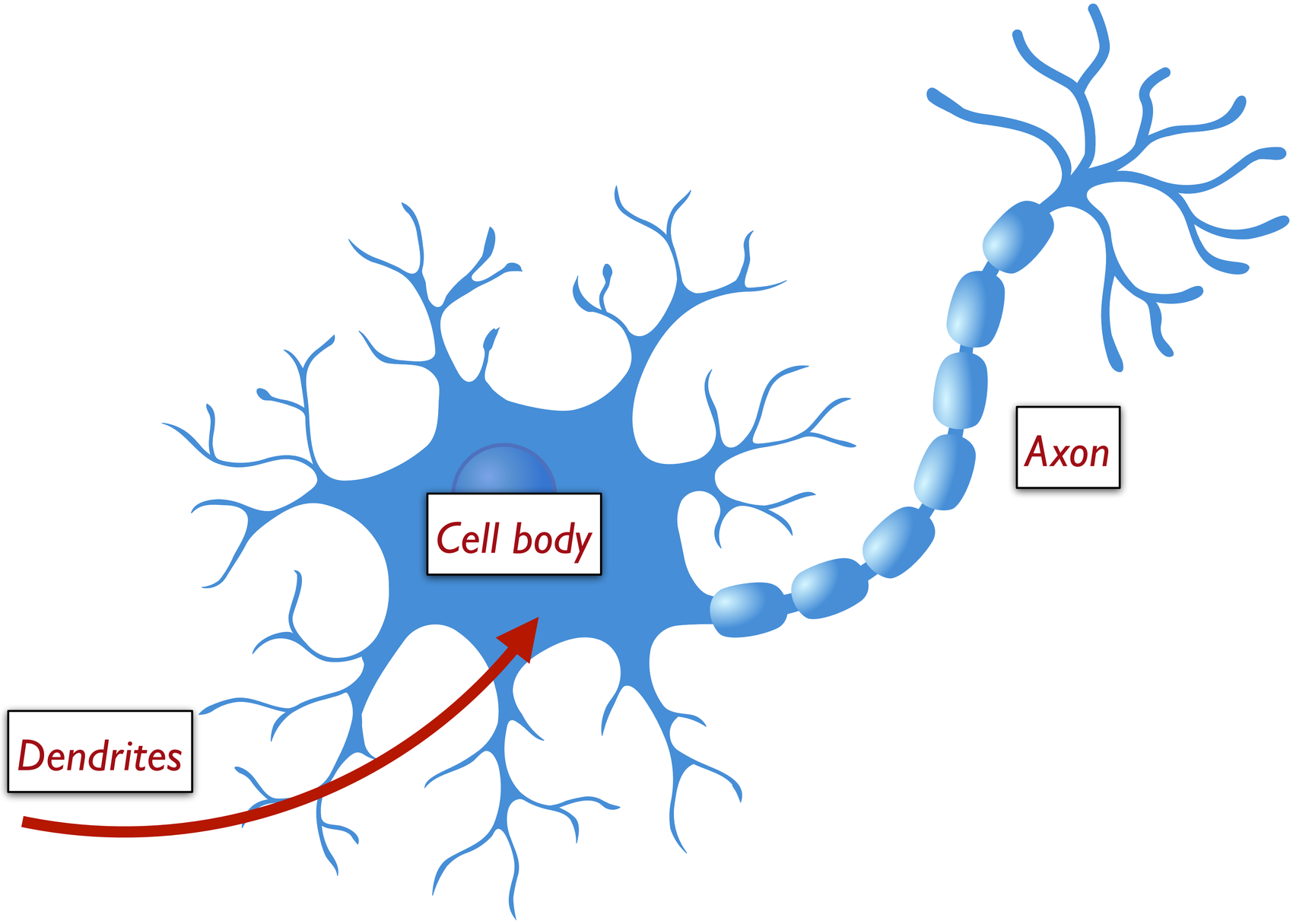
Neurons

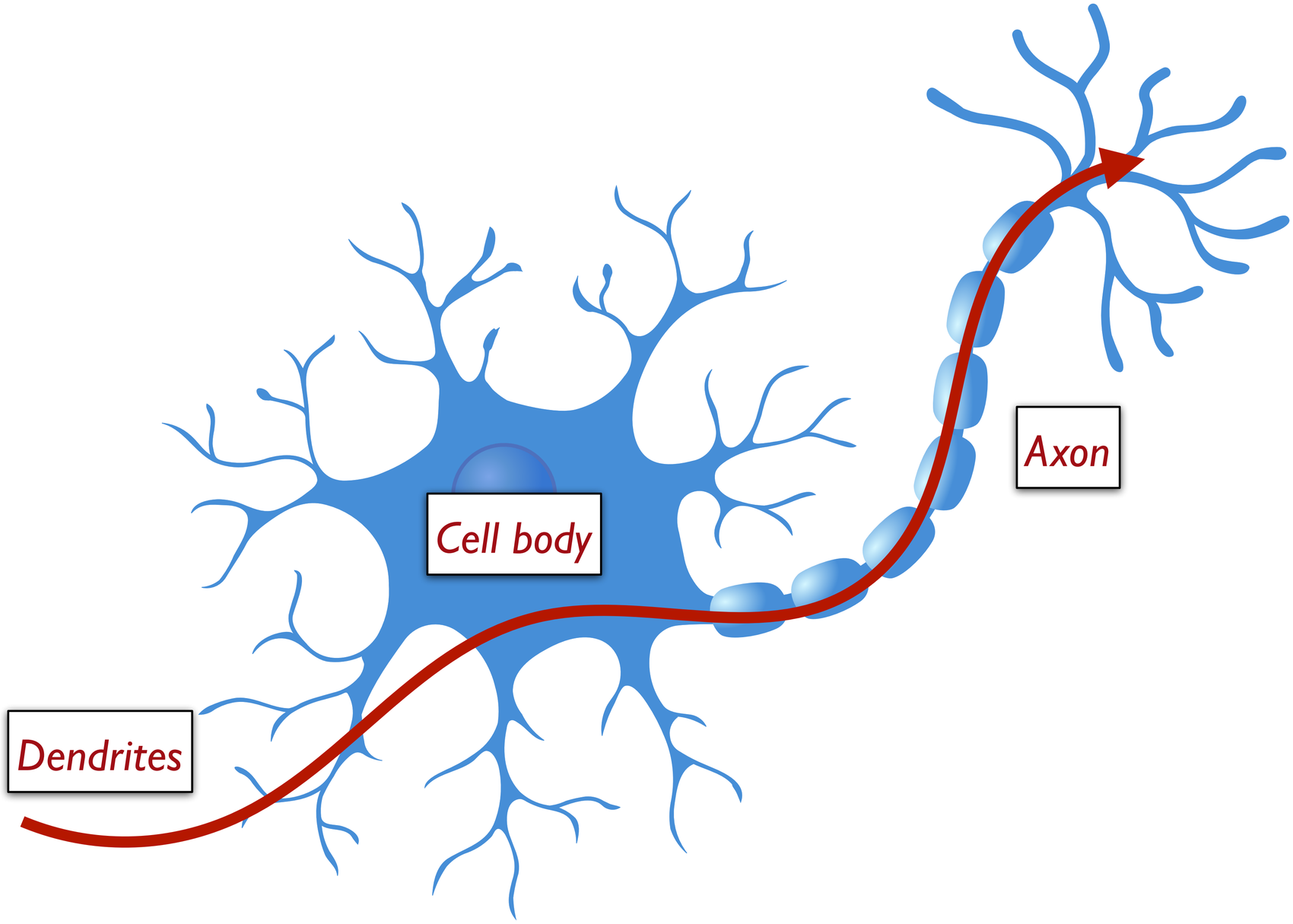



Cell body


Cell body
Dendrites


Axon
Cell body
Dendrites

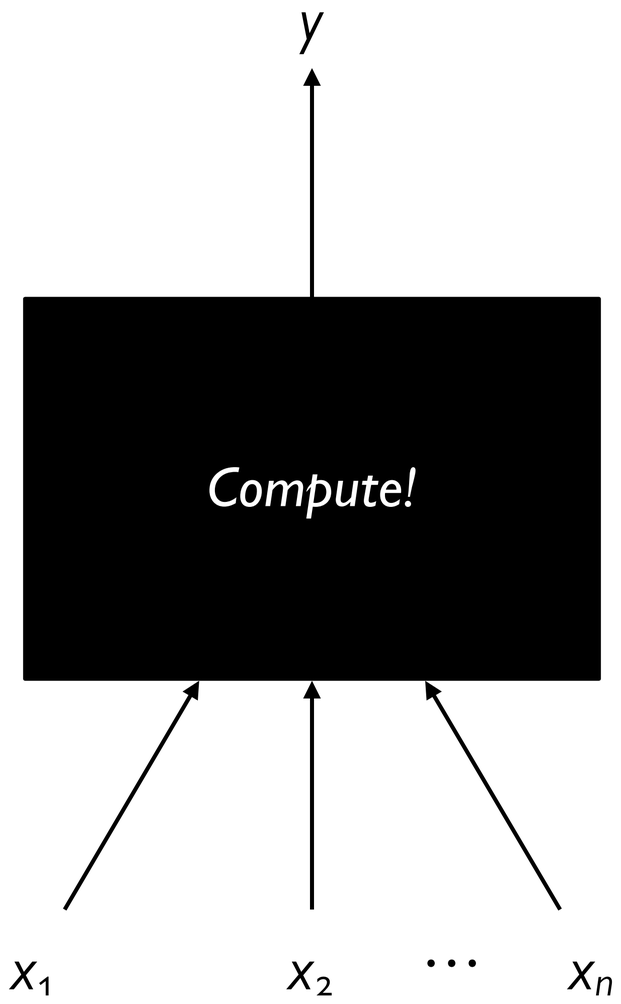

Axon
Cell body
Dendrites

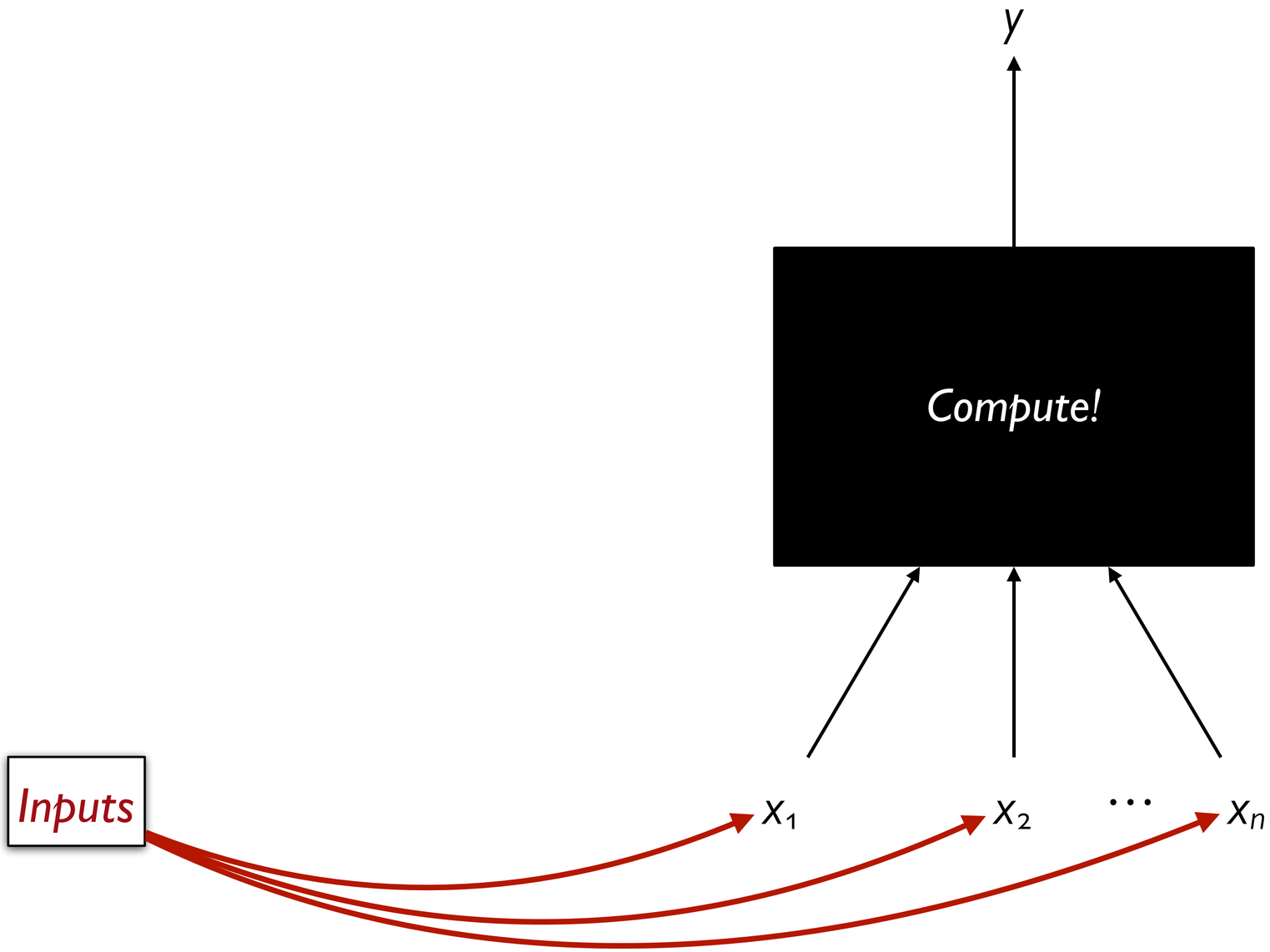

Dendrites
Cell body
Axon

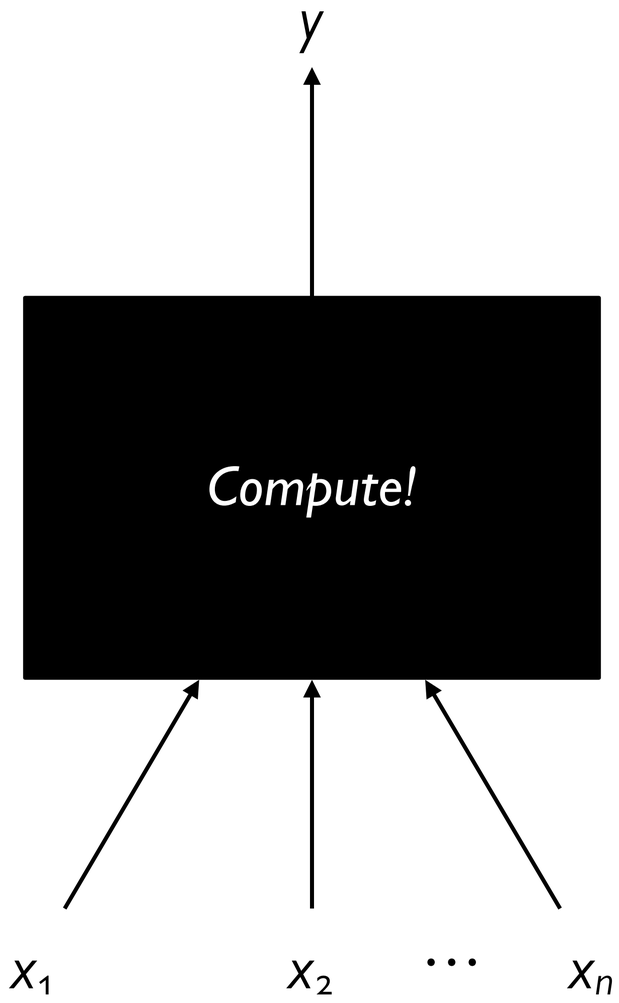
Different connections from other neurons to a given neuron have different strengths.

In calculating the sum of its inputs, the neuron gives *more weight* to inputs from *stronger connections* than inputs from weaker connections.

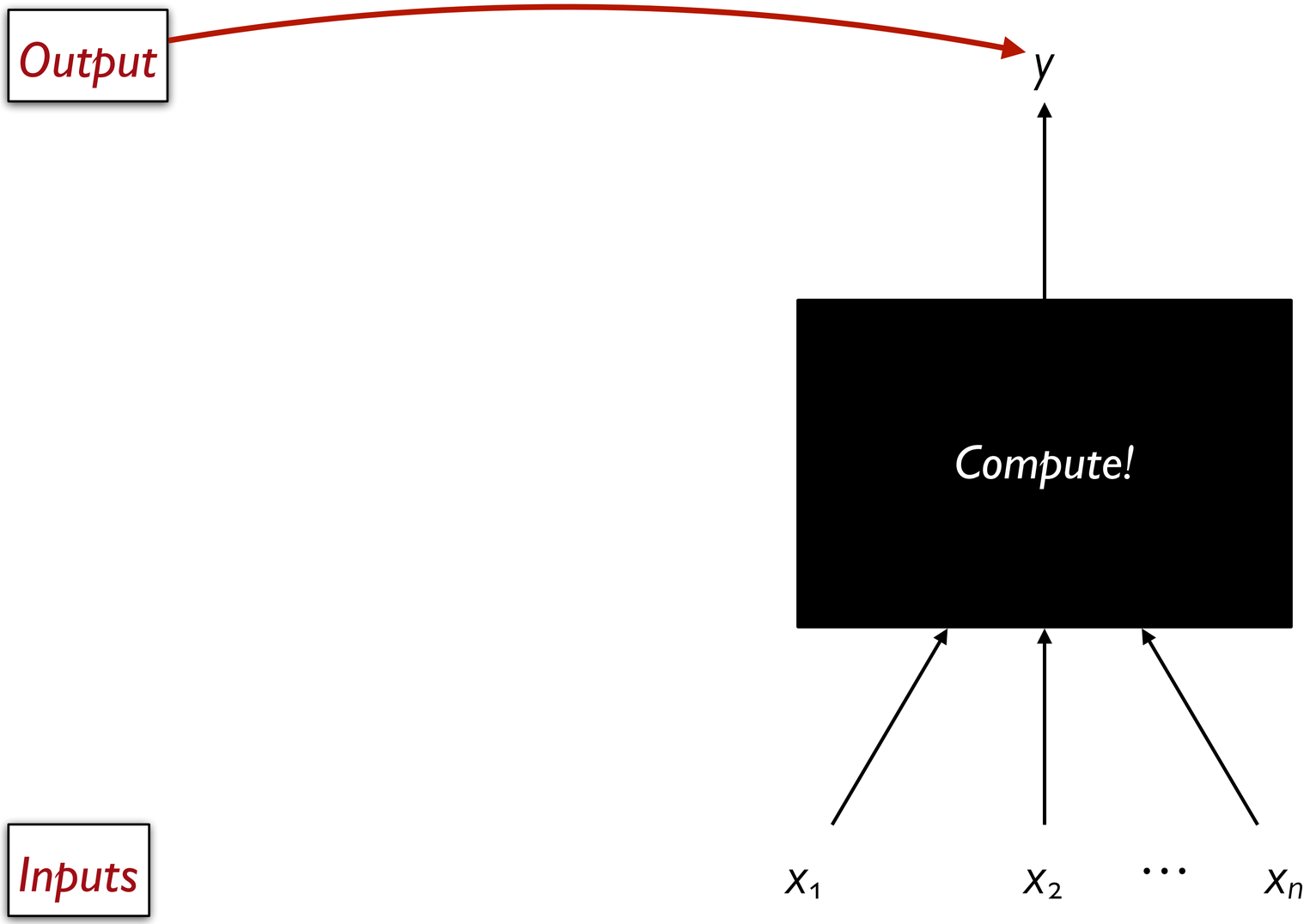
Neuroscientists believe that adjustments to the strength of connections between neurons is a key part of how learning takes place in the brain.


y
Compute!
x1
x2
...
xn

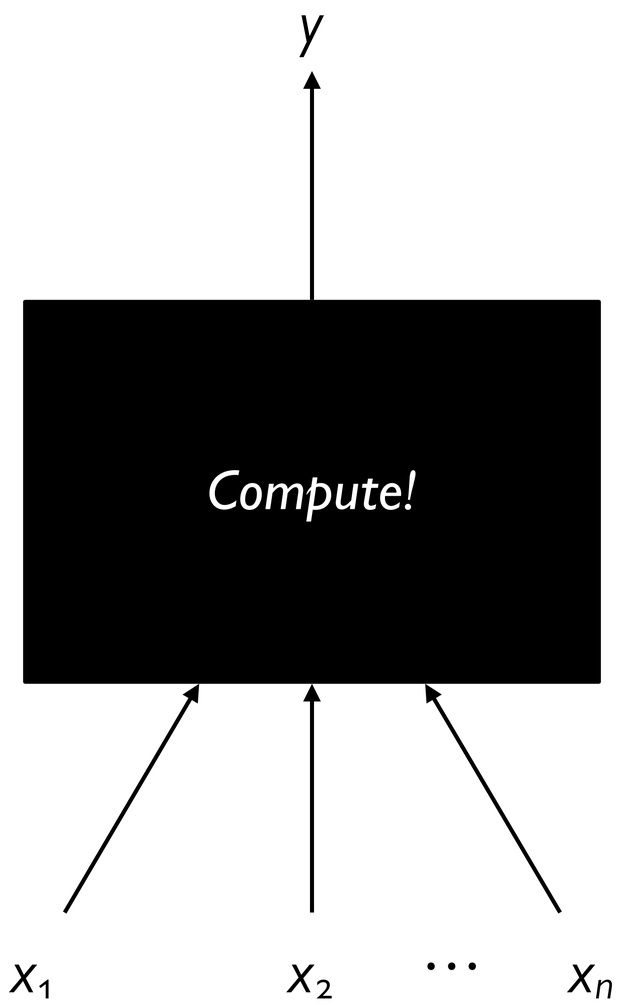

y
Compute!
Inputs
x_1
x_2
...
x_n



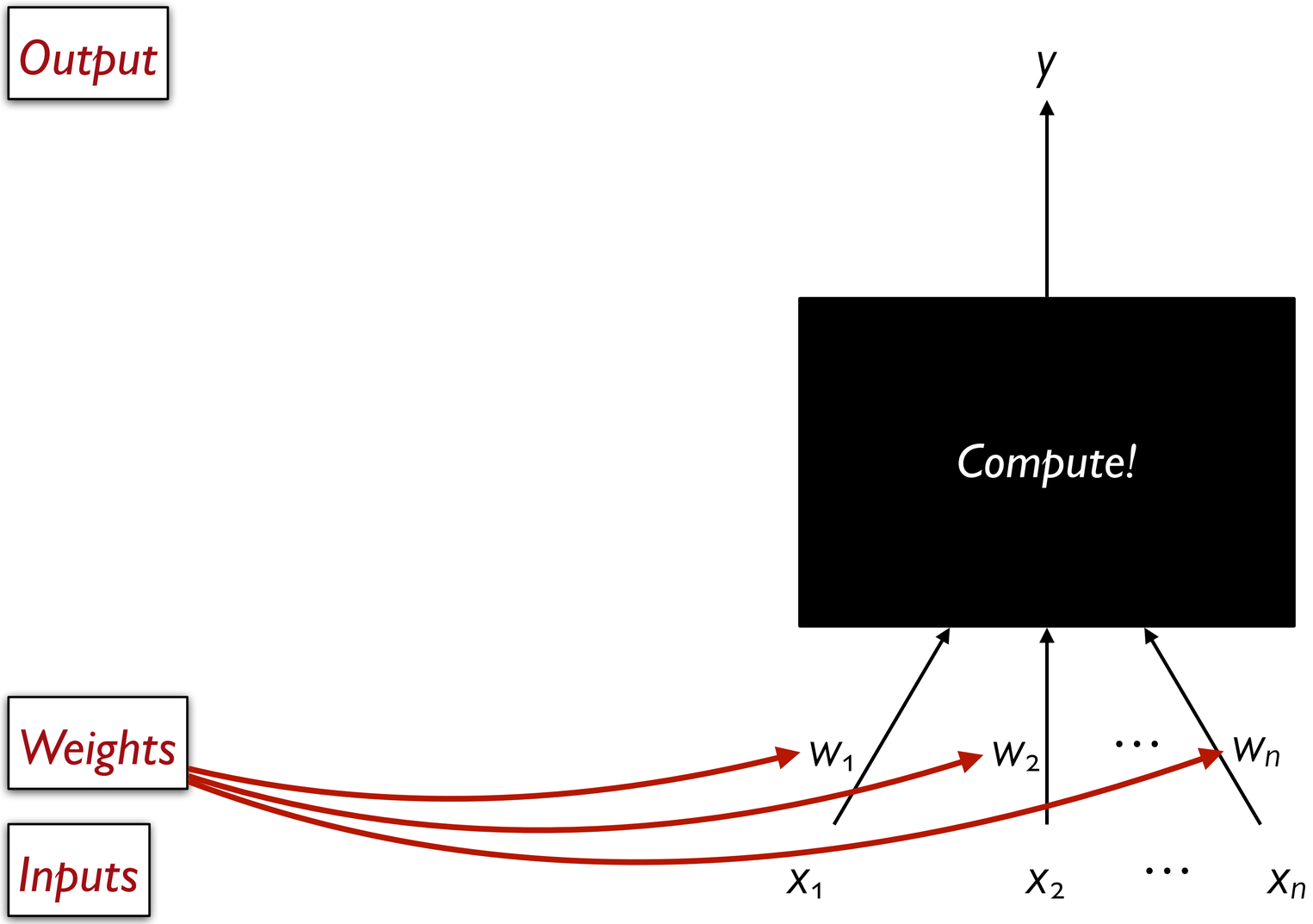
Inputs


Output
y
Compute!
Inputs
x_1
x_2
...
x_n

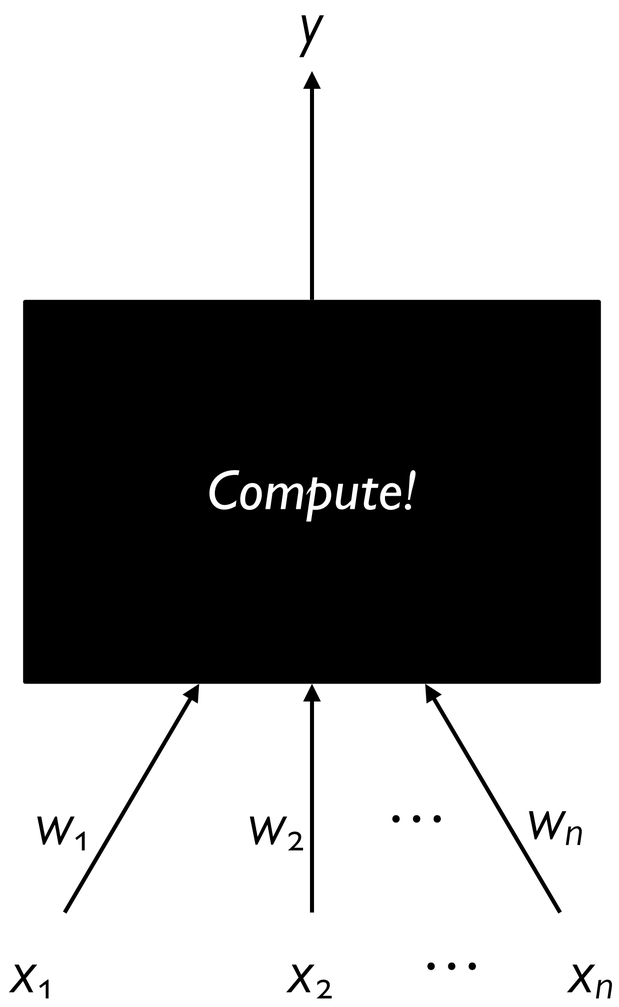
Output



Inputs


Output
y
Compute!
Weights
w_1
w_2
w_n
Inputs
x_1
x_2
x_n

Output



Weights

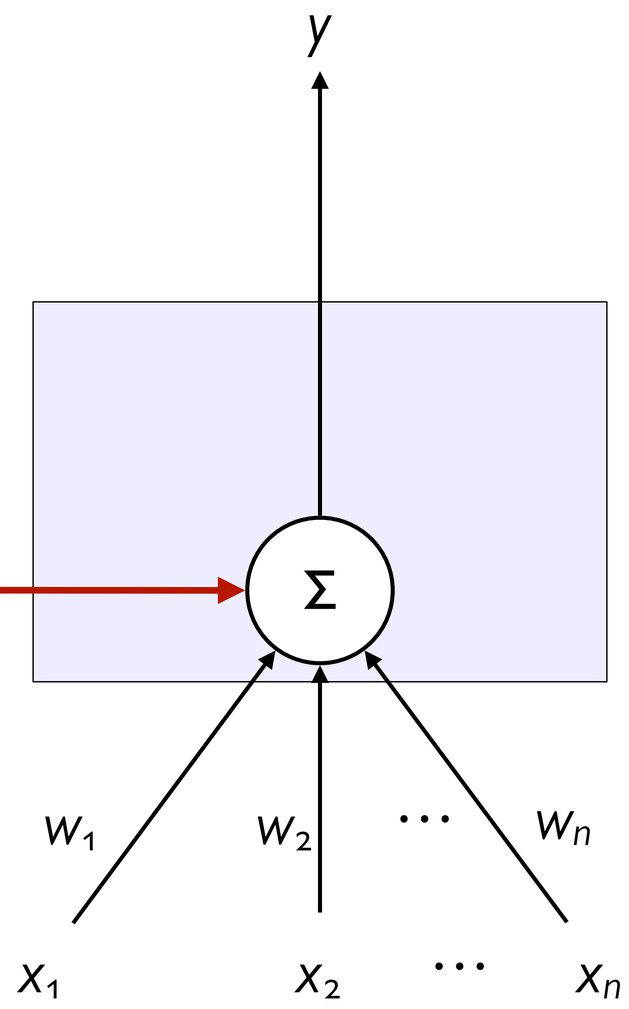
Inputs

Output

Weighted sum

Weights

Inputs



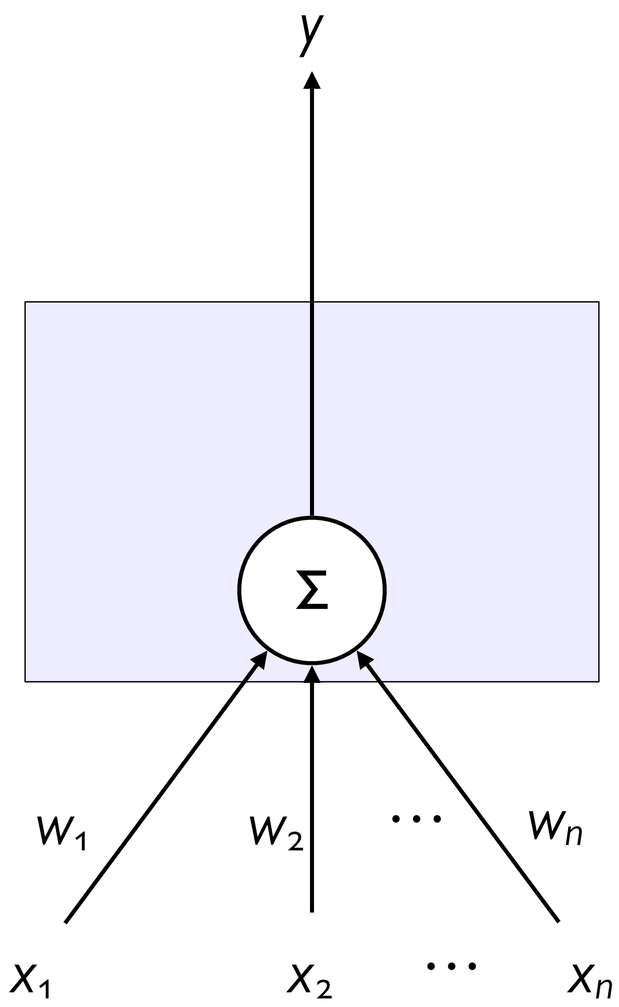
$$y = w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

Output

Weighted sum

Weights

Inputs



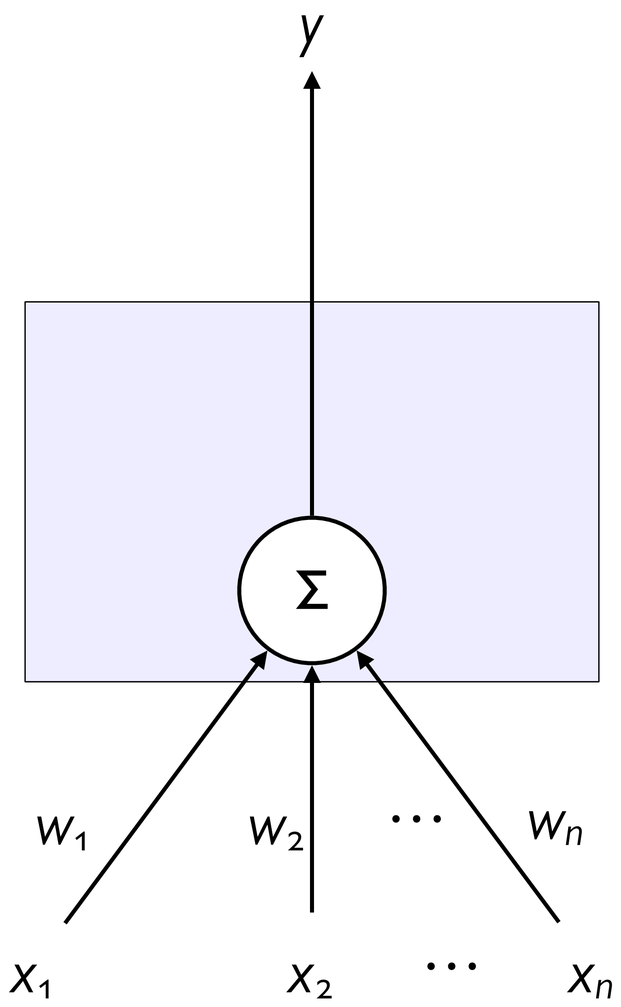
$$y = w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

Output

Weighted sum

Weights

Inputs



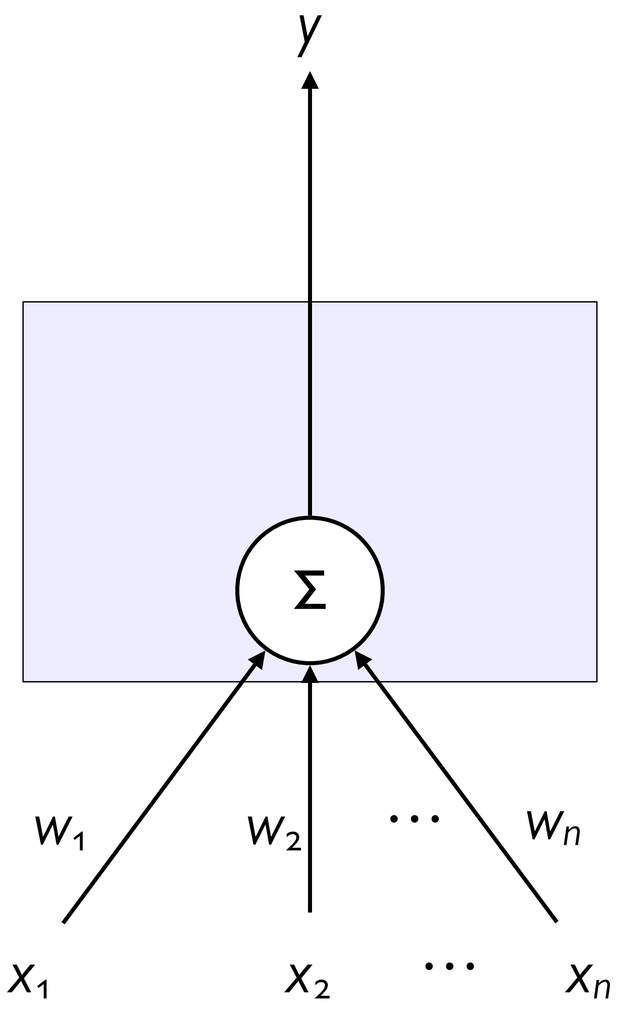
$y = \Sigma_i \, w_i x_i$

Output

Weighted sum

Weights Vector **w**

Inputs Vector **x**



$y = \mathbf{w} \cdot \mathbf{x}$

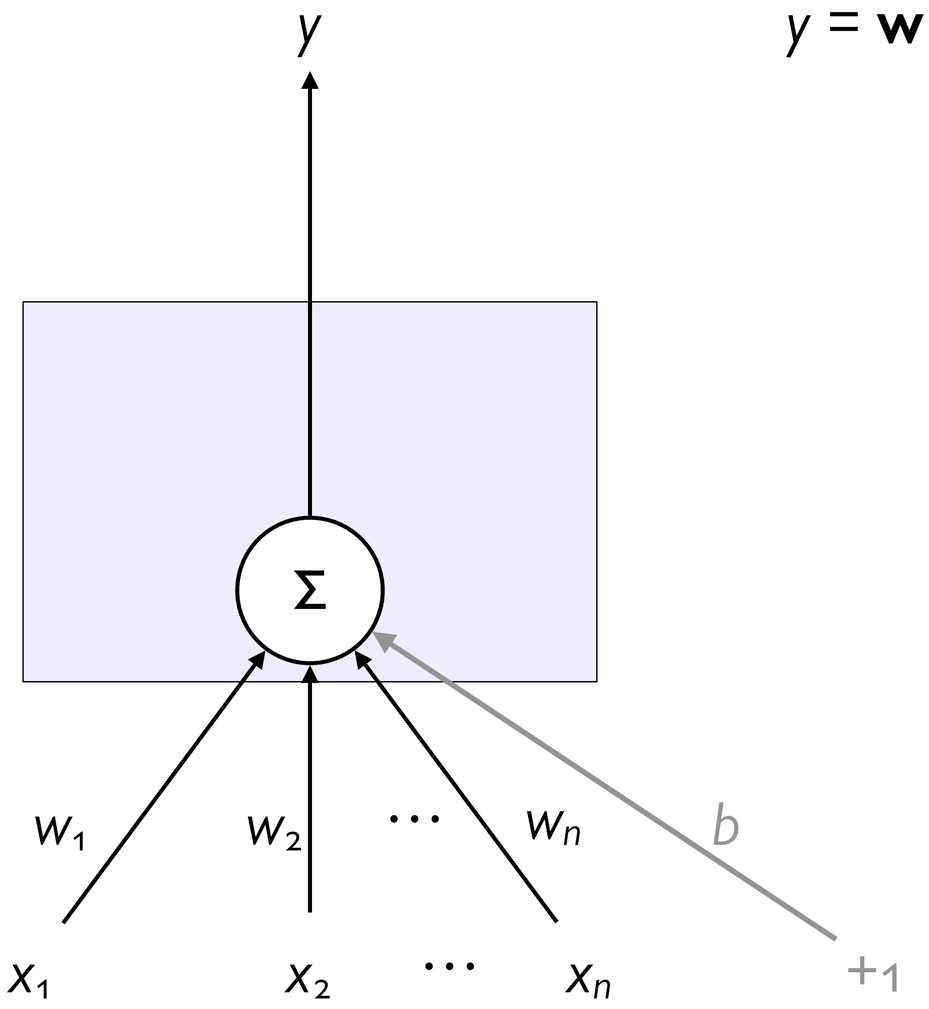
Output

$y = \mathbf{w} \cdot \mathbf{x} + b$

Weighted sum

Weights *Vector* **w** and scalar *b*

Inputs *Vector* **x**



Output

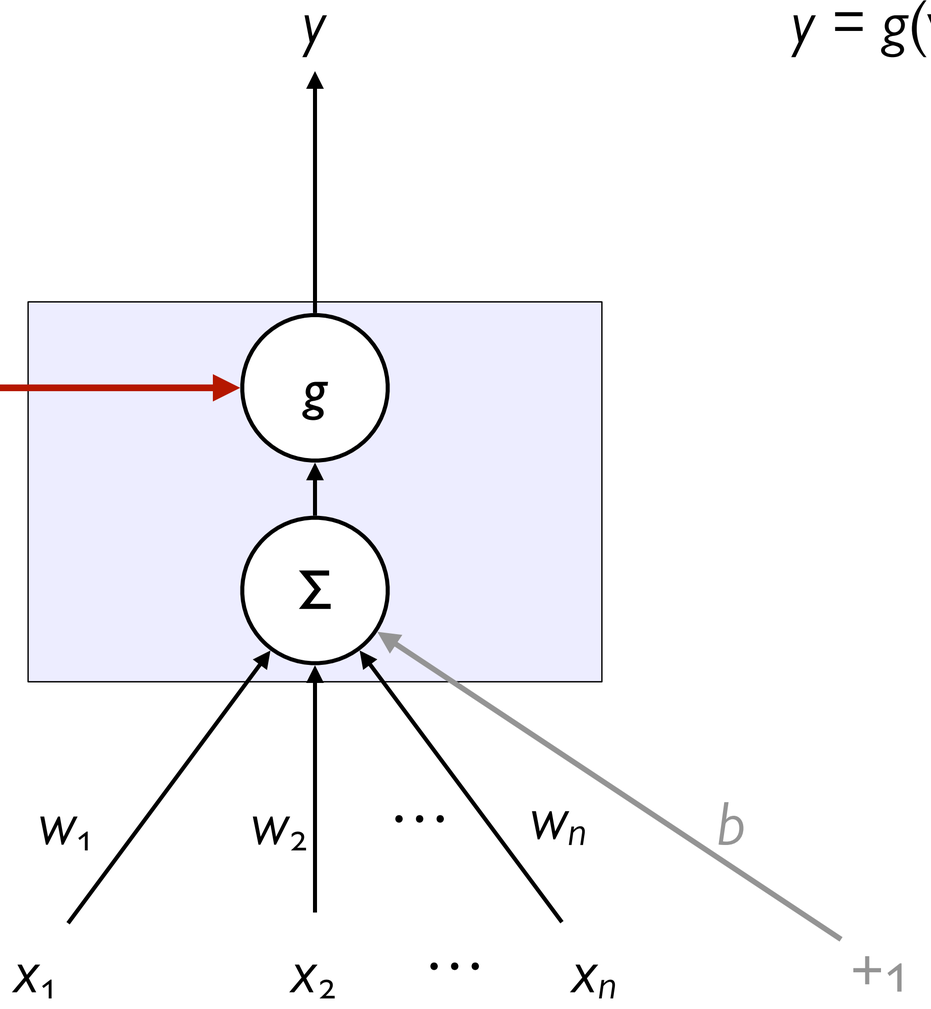
Non-linear activation function

Weighted sum

Weights *Vector* **w** and scalar *b*

Inputs *Vector* **x**

$y = g(\mathbf{w} \cdot \mathbf{x} + b)$



Output

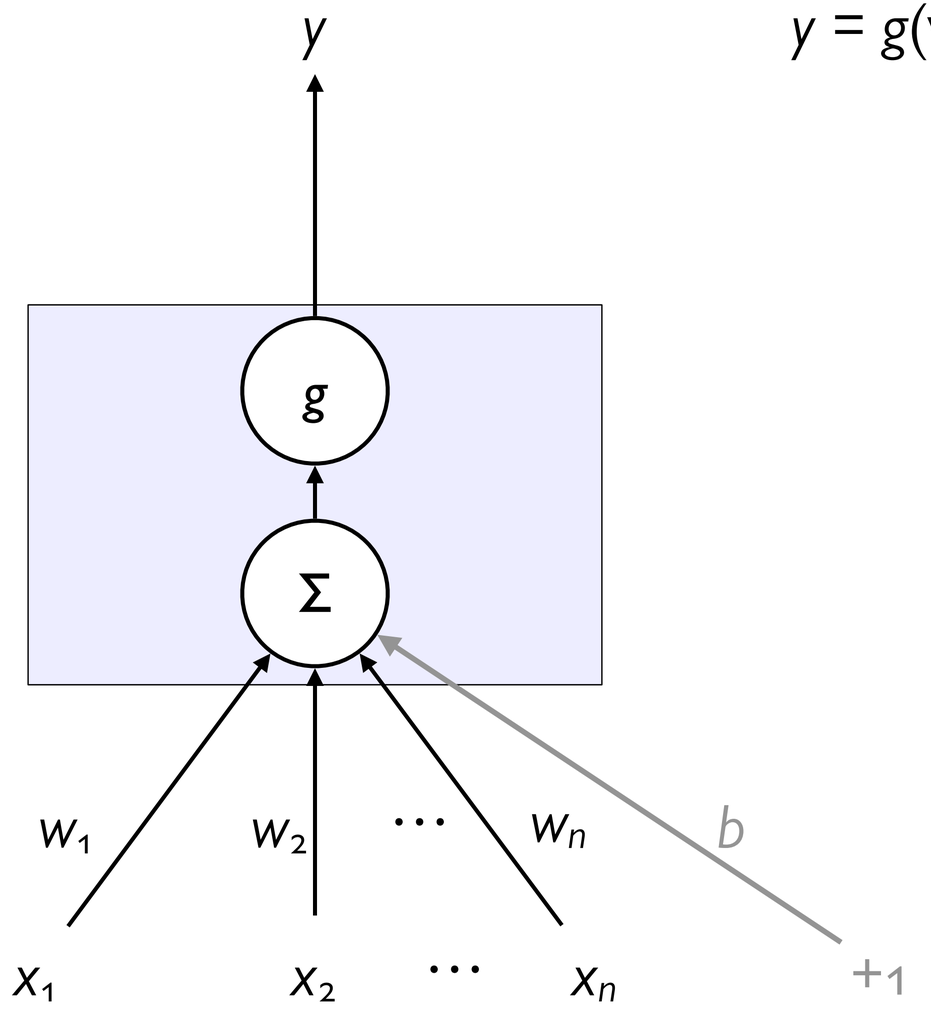
Non-linear activation function

Weighted sum

Weights Vector **w** and scalar *b*

Inputs Vector **x**

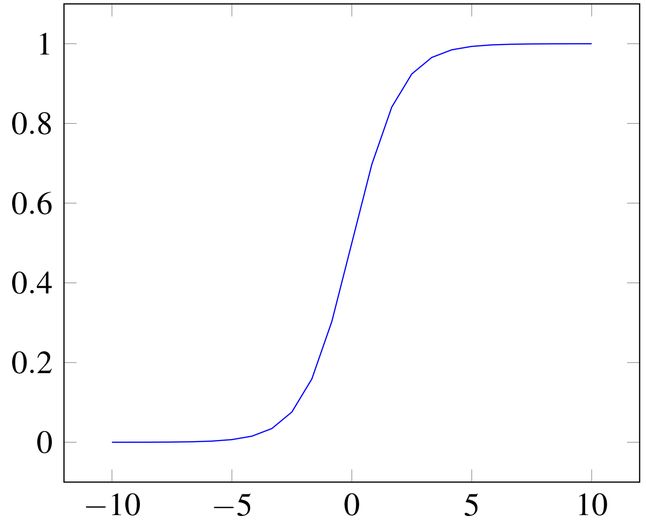
$$y = g(\mathbf{w} \cdot \mathbf{x} + b)$$



A traditional choice of non-linear activation function is the *sigmoid*, like we used for logistic regression.

Given an input between $-\infty$ and ∞,

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



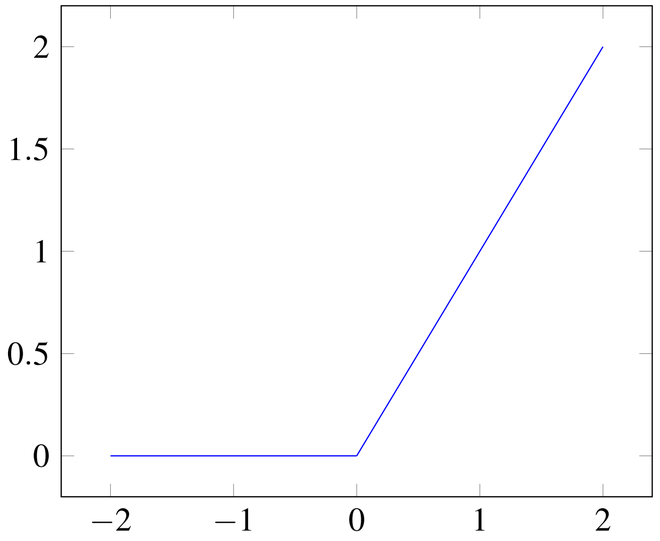
squishes the extremes so the output is between 0 and 1.

However, years of experience has taught us that other nonlinear functions work even better.

One good choice is *ReLU* – a rectified linear unit.

It's the same as its input except it turns flat when the input would fall below o:

$$f(z) = \max(z, 0)$$



Computing functions with neural units

Can neural units compute simple functions of the input?

AND

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

OR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Minsky & Papert, 1969

Perceptrons

A very simple neural unit:

Binary output (0 or 1)

No non-linear activation function

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

Solving AND

Deriving AND

*Goal: Return 1 if x_1 and x_2 are **both** 1.*

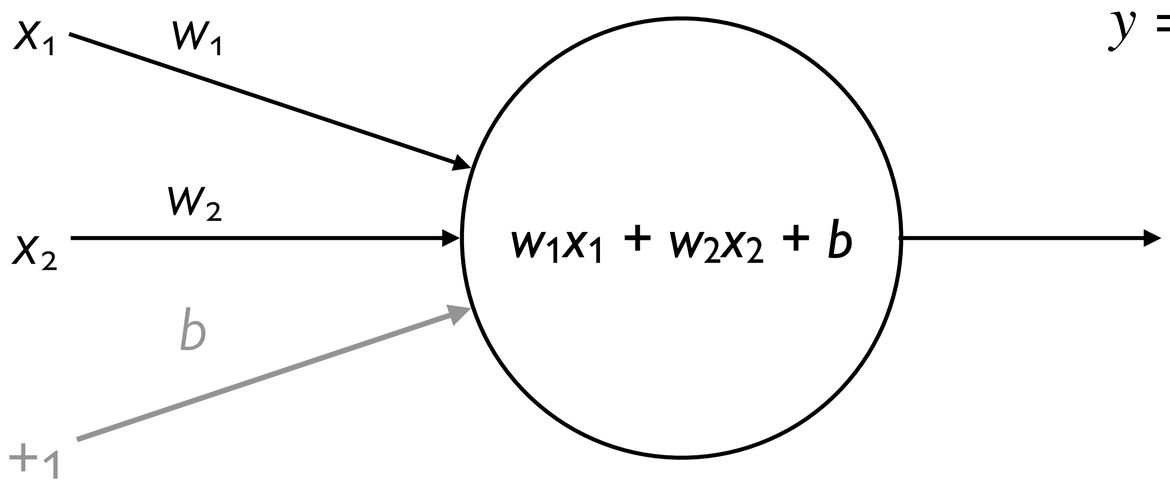
x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

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

Deriving AND

*Goal: Return 1 if x_1 and x_2 are **both** 1.*

x_1	x_2	y



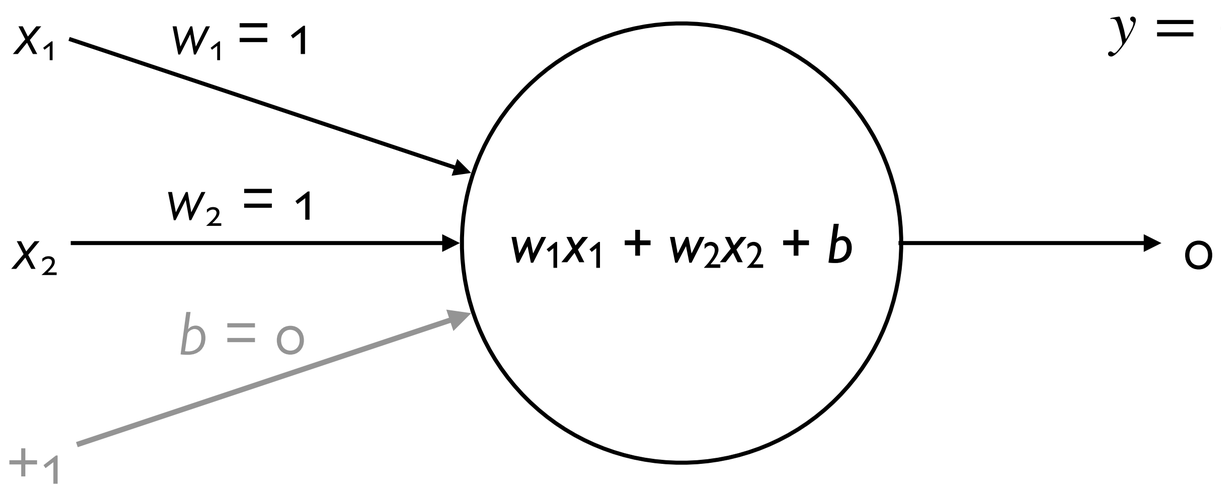
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

What should w_1, w_2, and b be?

Deriving AND

*Goal: Return 1 if x_1 and x_2 are **both** 1.*

x_1	x_2	y
0	0	0



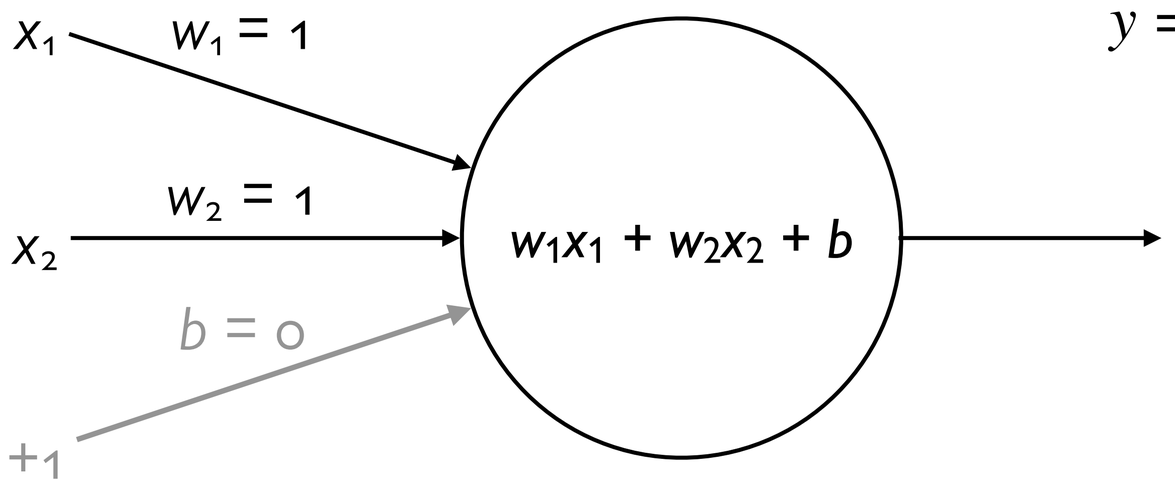
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

What should w_1, w_2, and b be?

Deriving AND

*Goal: Return 1 if x_1 and x_2 are **both** 1.*

x_1	x_2	y
0	0	0
0	1	0



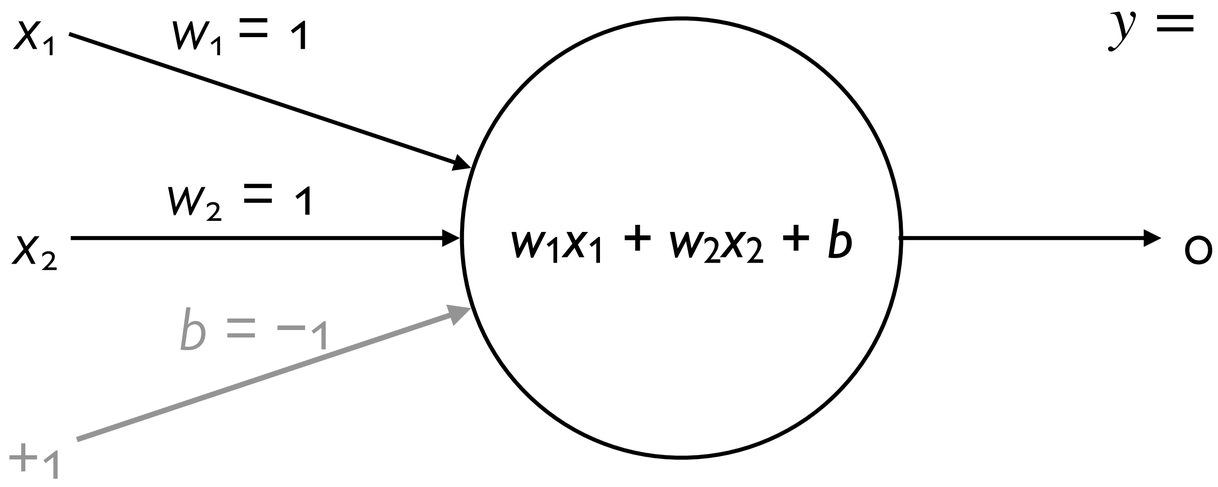
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

What should w_1, w_2, and b be?

Deriving AND

*Goal: Return 1 if x_1 and x_2 are **both** 1.*

x_1	x_2	y
0	0	0
0	1	0



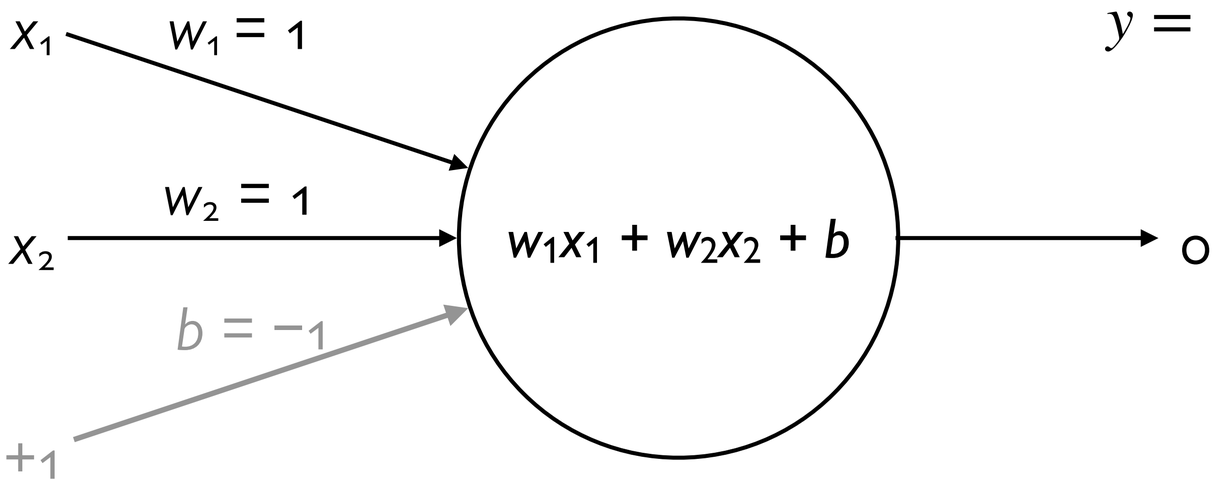
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

What should w_1, w_2, and b be?

Deriving AND

*Goal: Return 1 if x_1 and x_2 are **both** 1.*

x_1	x_2	y
0	0	0
0	1	0
1	0	0



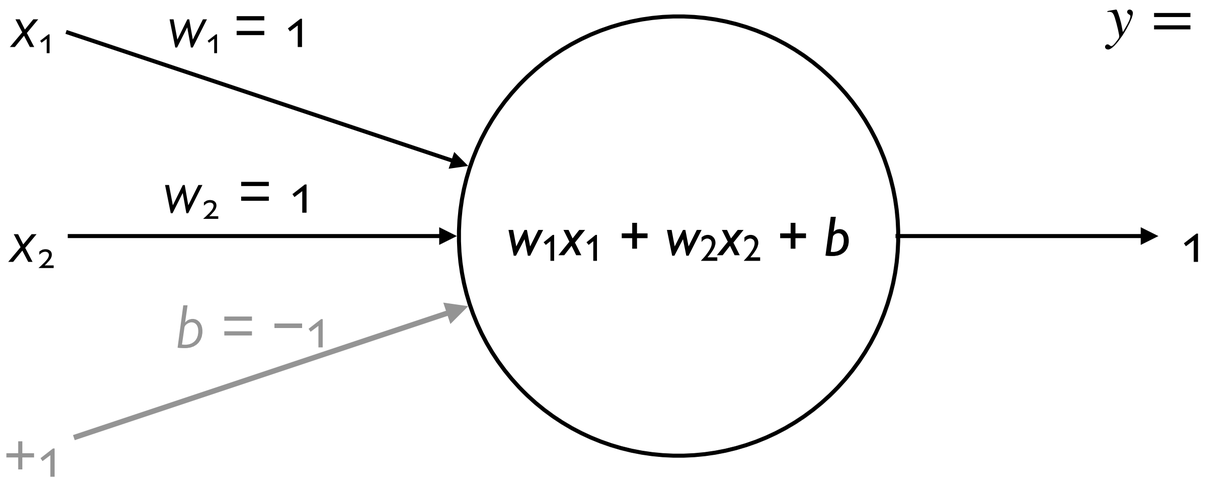
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

What should w_1, w_2, and b be?

Deriving AND

*Goal: Return 1 if x_1 and x_2 are **both** 1.*

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



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

What should w_1, w_2, and b be?

Solving OR

Deriving OR

Goal: Return 1 if either x_1 or x_2 is 1.

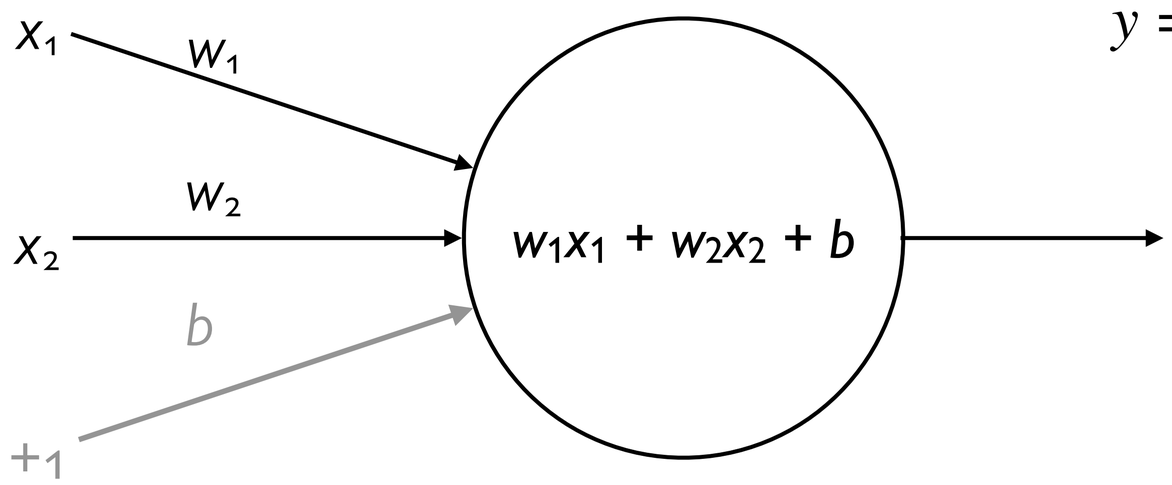
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

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

Deriving OR

Goal: Return 1 if either x_1 or x_2 is 1.

x_1	x_2	y



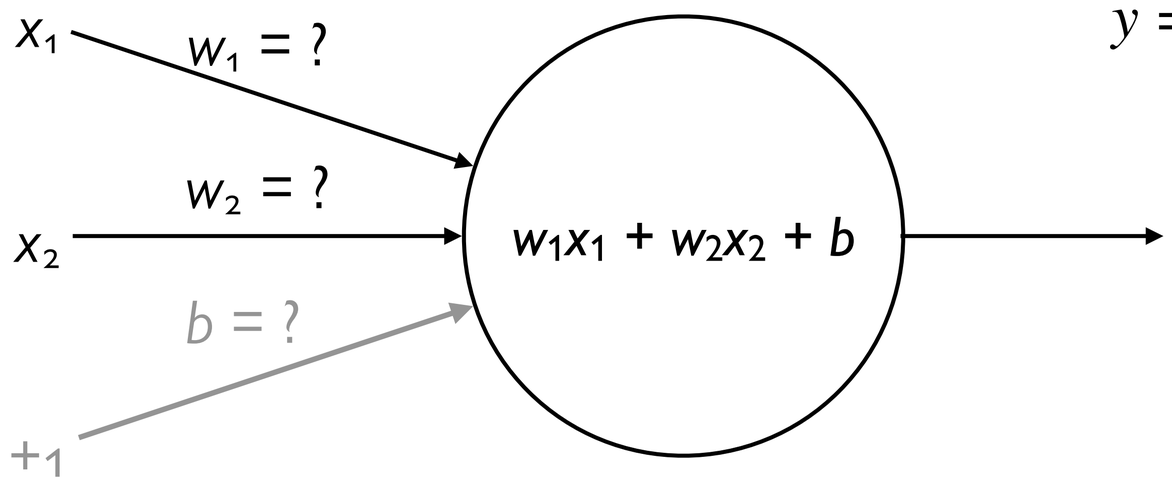
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

What should w_1, w_2, and b be?

Deriving OR

Goal: Return 1 if either x_1 or x_2 is 1.

x_1	x_2	y
0	0	0



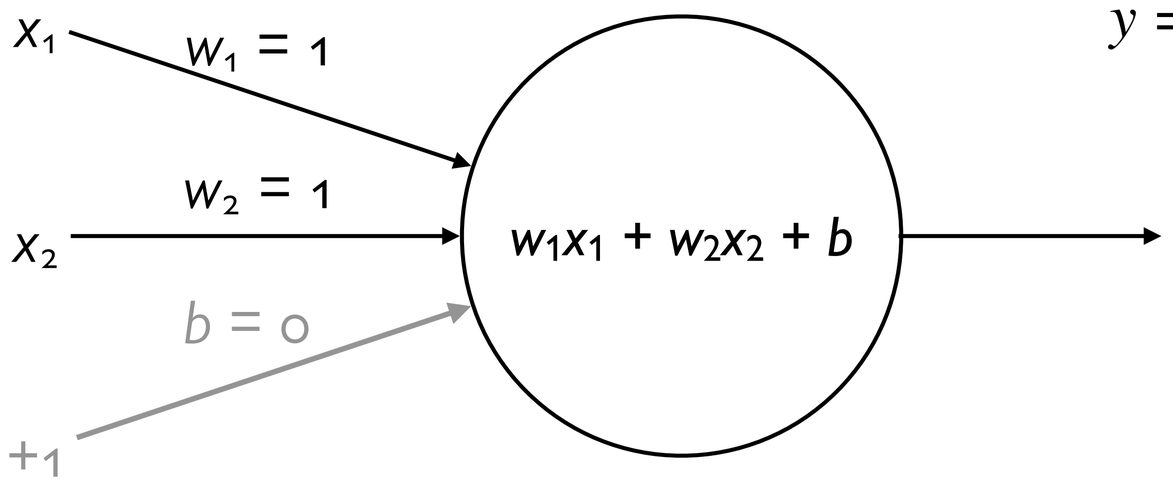
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

What should w_1, w_2, and b be?

Deriving OR

Goal: Return 1 if either x_1 or x_2 is 1.

x_1	x_2	y
0	0	0



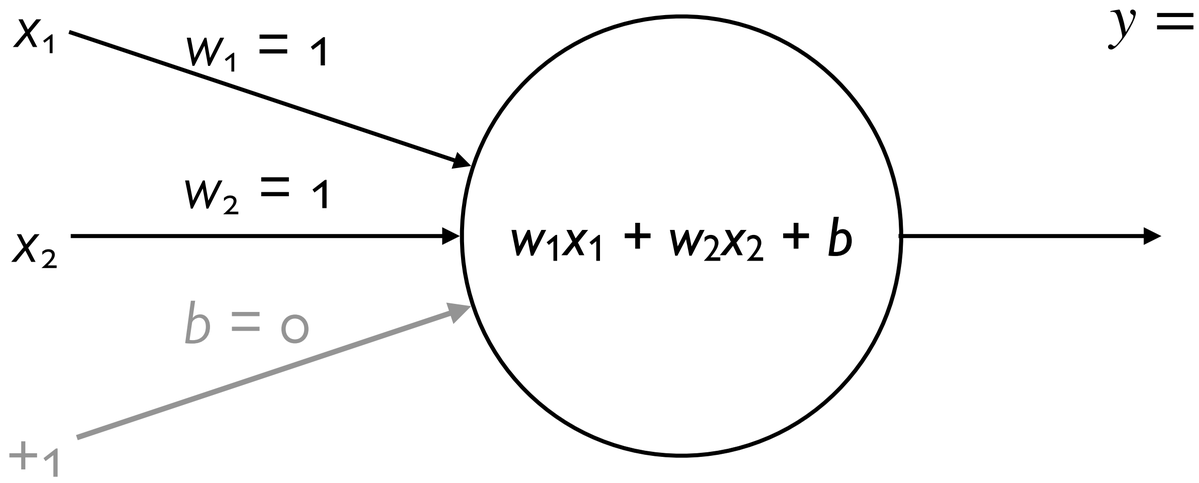
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

What should w_1, w_2, and b be?

Deriving OR

Goal: Return 1 if either x_1 or x_2 is 1.

x_1	x_2	y
0	0	0
0	1	1



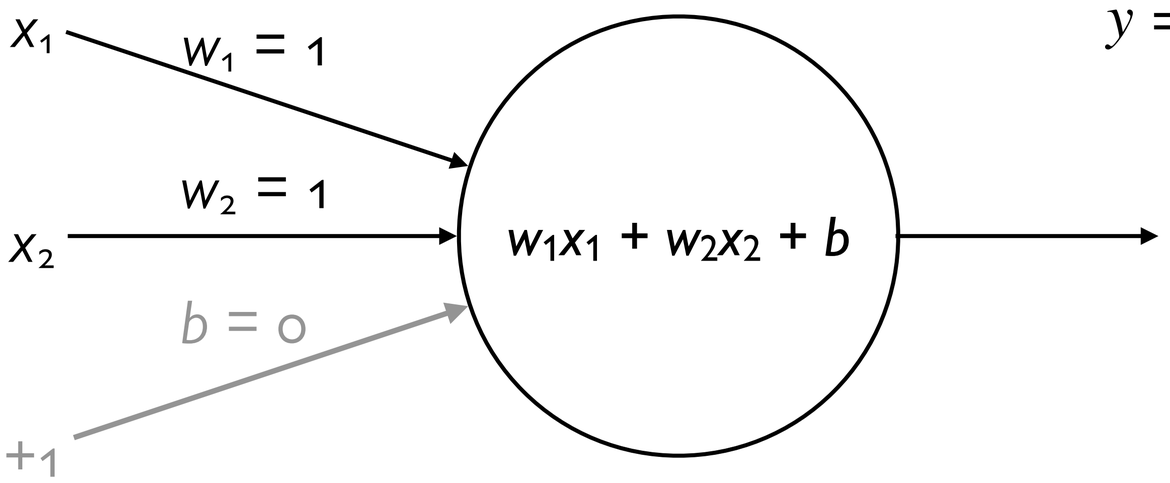
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

What should w_1, w_2, and b be?

Deriving OR

Goal: Return 1 if either x_1 or x_2 is 1.

x_1	x_2	y
0	0	0
0	1	1
1	0	1



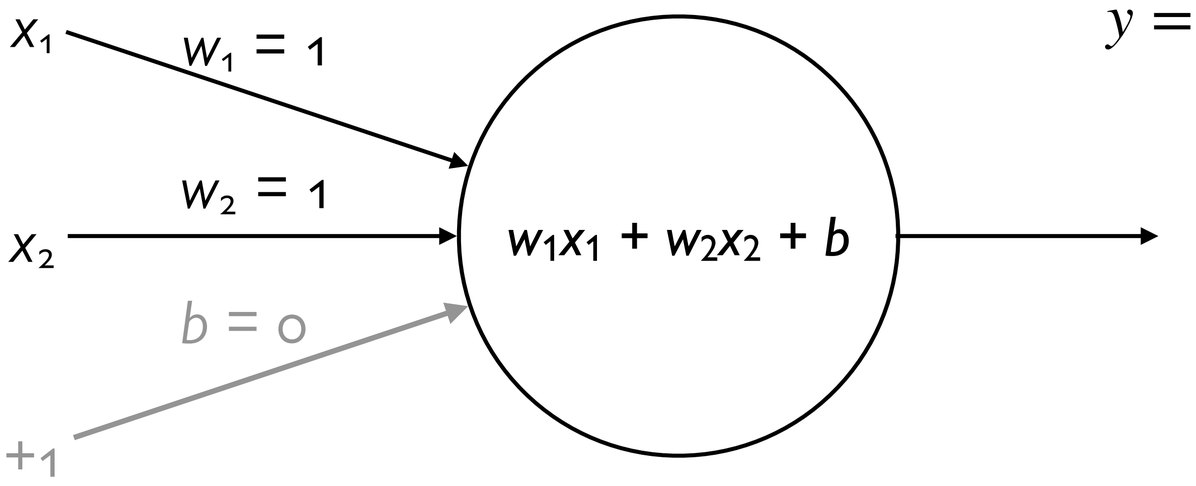
$$y = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

What should w_1, w_2, and b be?

Deriving OR

Goal: Return 1 if either x_1 or x_2 is 1.

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



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

What should w_1, w_2, and b be?

Solving XOR

Deriving XOR

Goal: Return 1 if either x_1 or x_2 is 1 but not both of them.

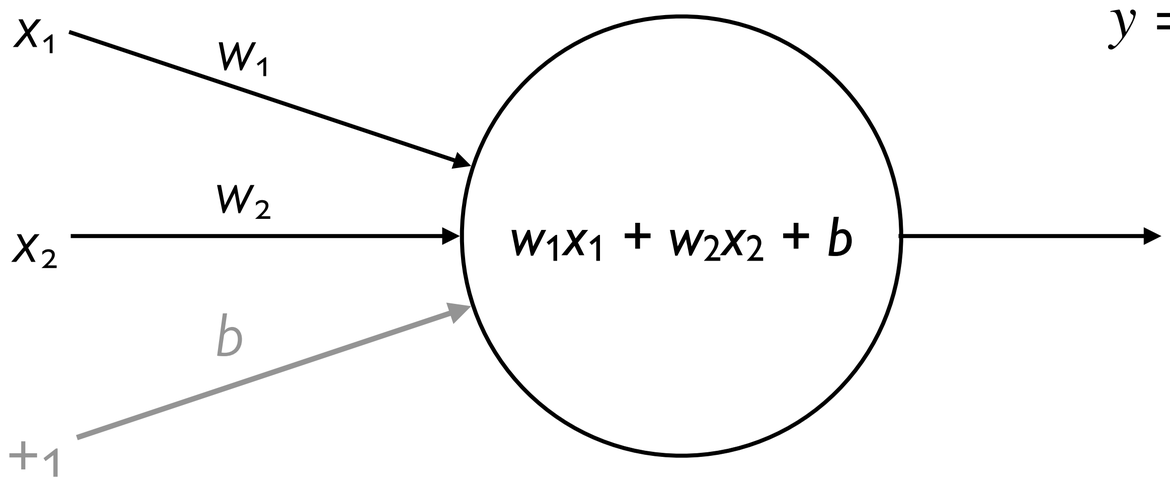
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

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

Deriving XOR

Goal: Return 1 if either x_1 or x_2 is 1 but not both of them.

x_1	x_2	y



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

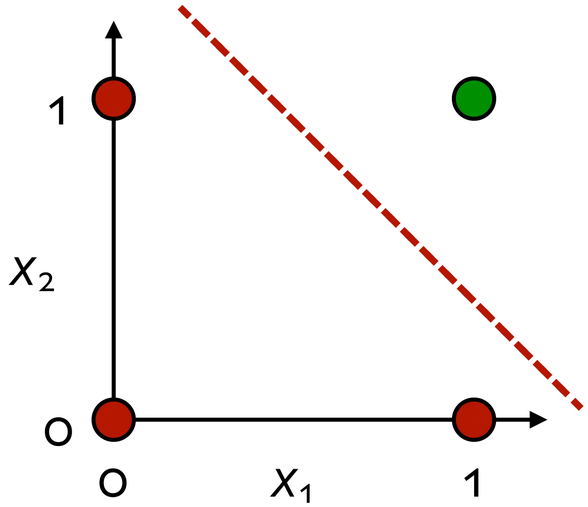
What should w_1, w_2, and b be?

Trick question – you can't capture XOR with perceptrons!

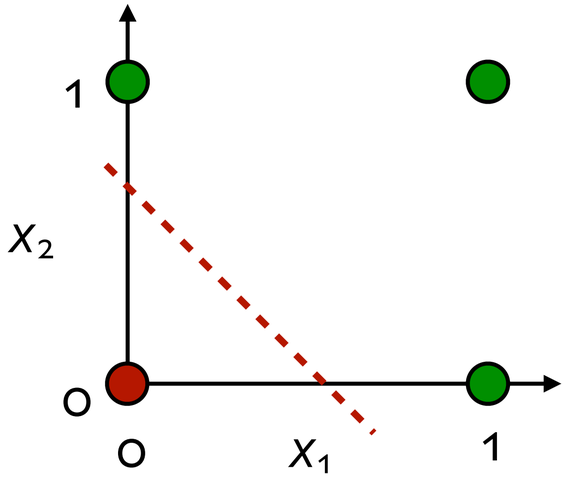
Perceptrons are linear classifiers

The perceptron equation is the equation of a line – the ***decision boundary****, separating the outputs 0 and 1.*

x_1 AND x_2



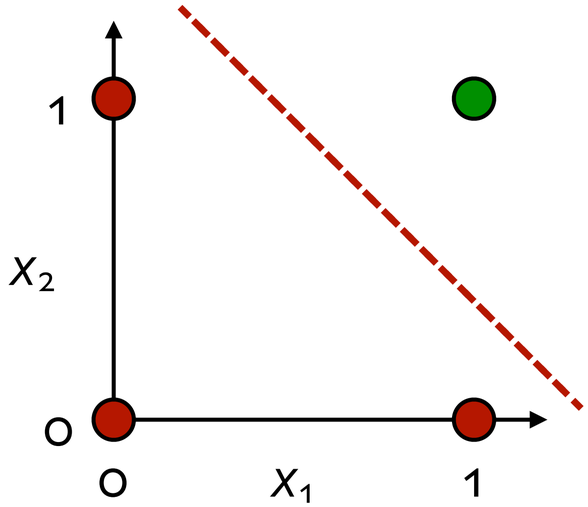
x_1 OR x_2



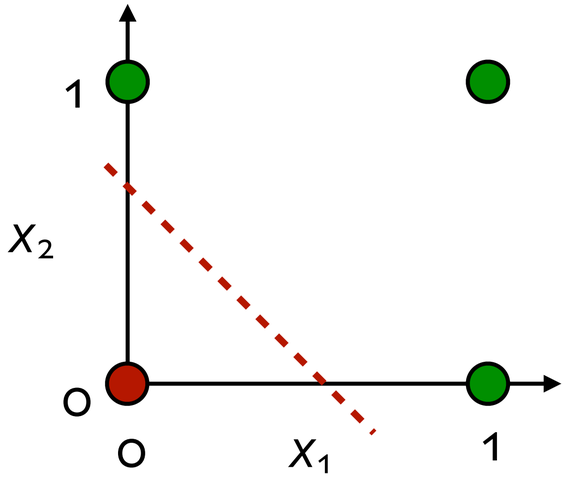
Perceptrons are linear classifiers

The perceptron equation is the equation of a line – the ***decision boundary****, separating the outputs 0 and 1.*

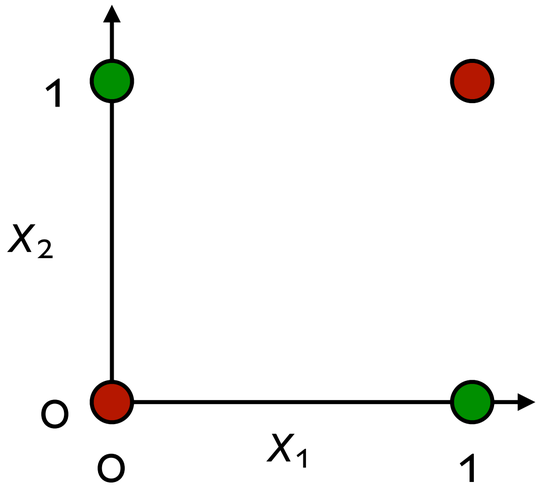
x_1 AND x_2



x_1 OR x_2

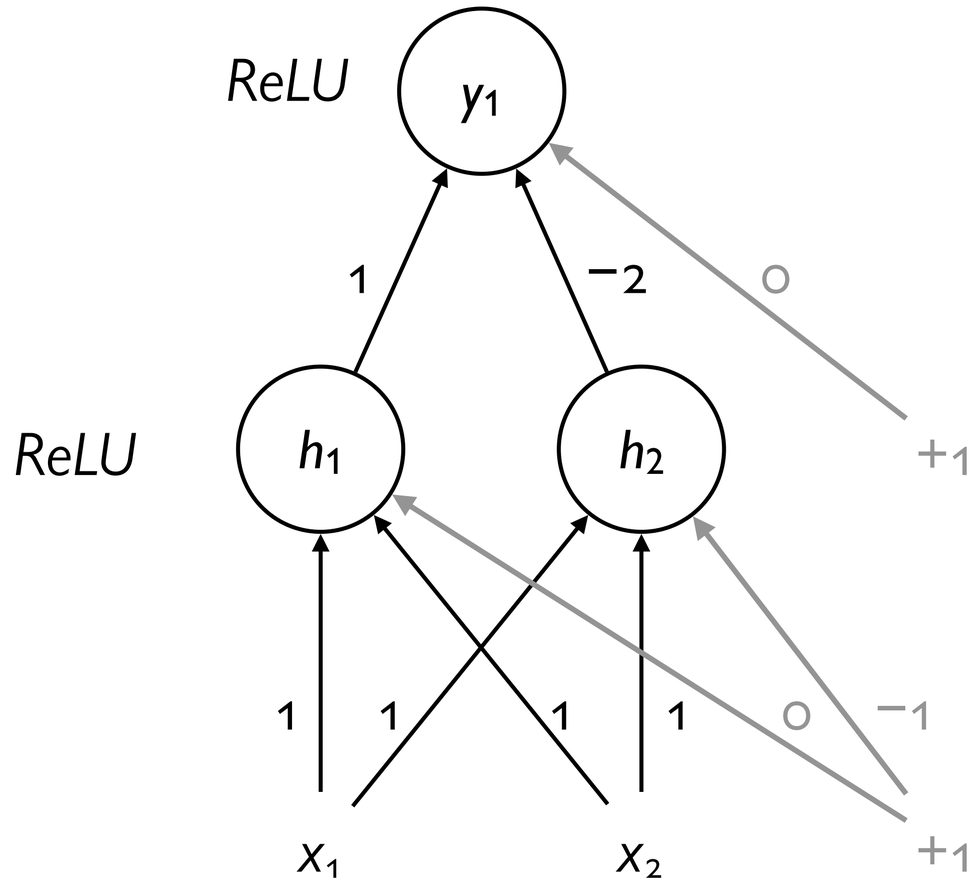


x_1 XOR x_2

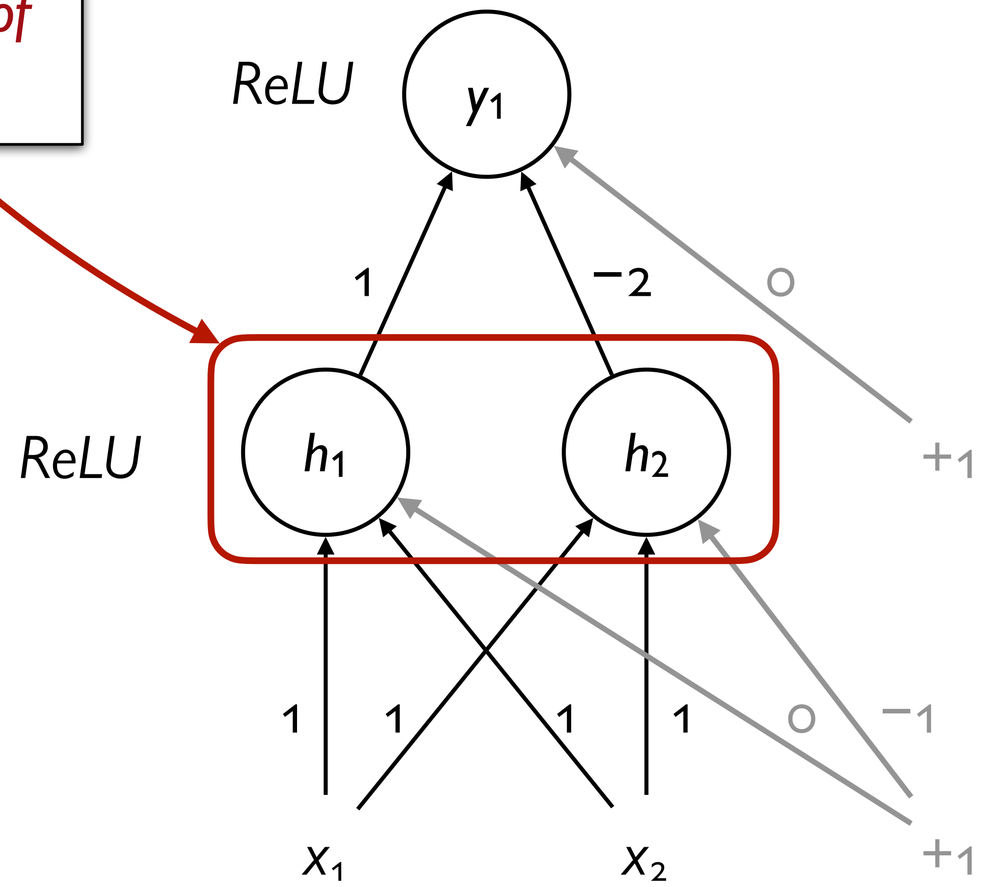


All hope is not lost.

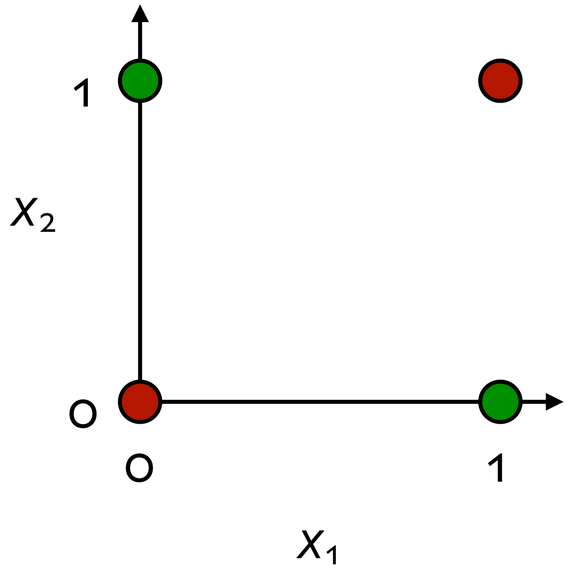
While XOR can't be calculated by a *single* perceptron, it *can* be be calculated by a layered network of units.

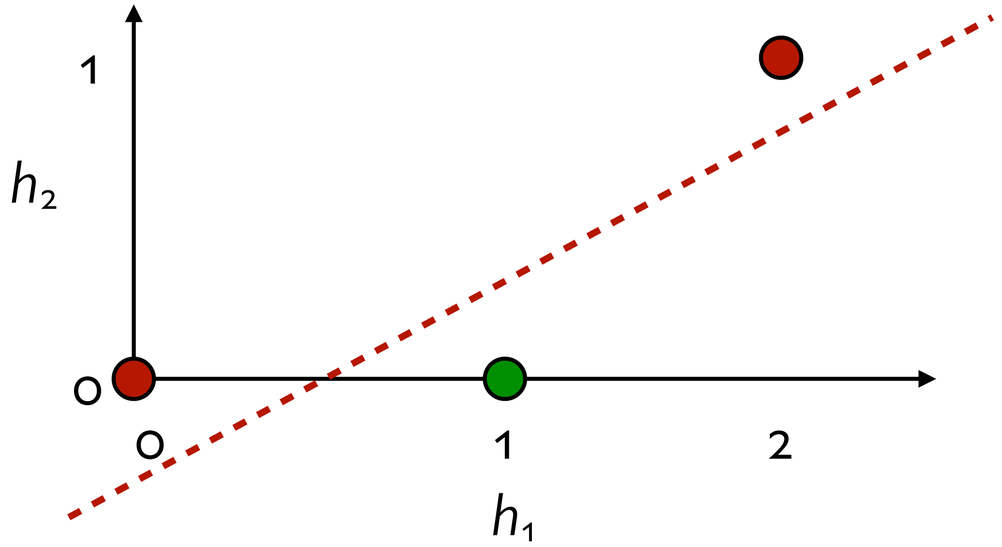

ReLU
y_1
1
−2
0
+1
ReLU
h_1
h_2
1
1
1
1
0
−1
+1
x_1
x_2

*This is a **hidden layer**, where intermediate units learn transformations of the features.*



The hidden representation *h*

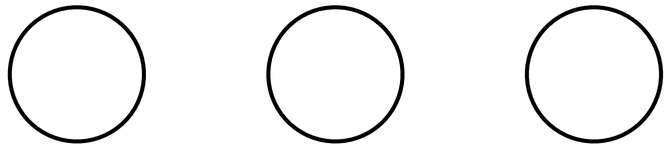


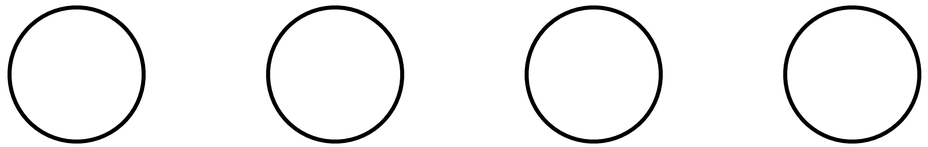


Feedforward neural networks

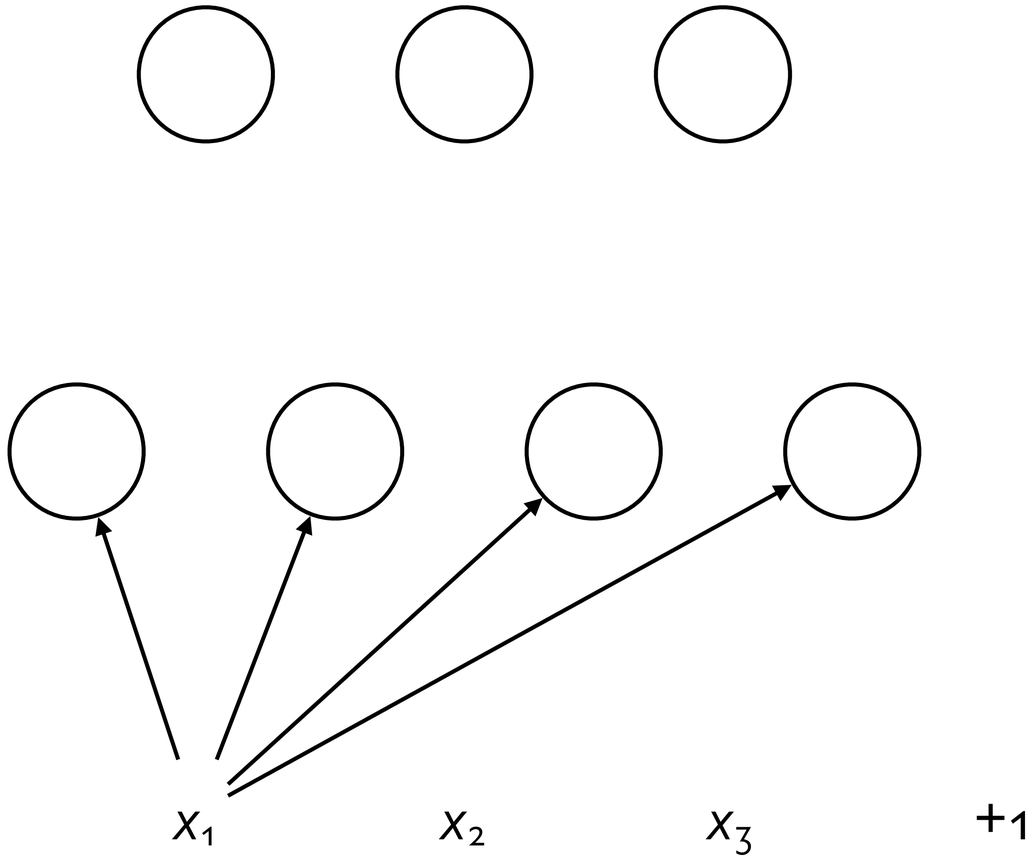
A *feedforward neural network* is the simplest and most common kind of neural network in NLP.

Units are organized in layers

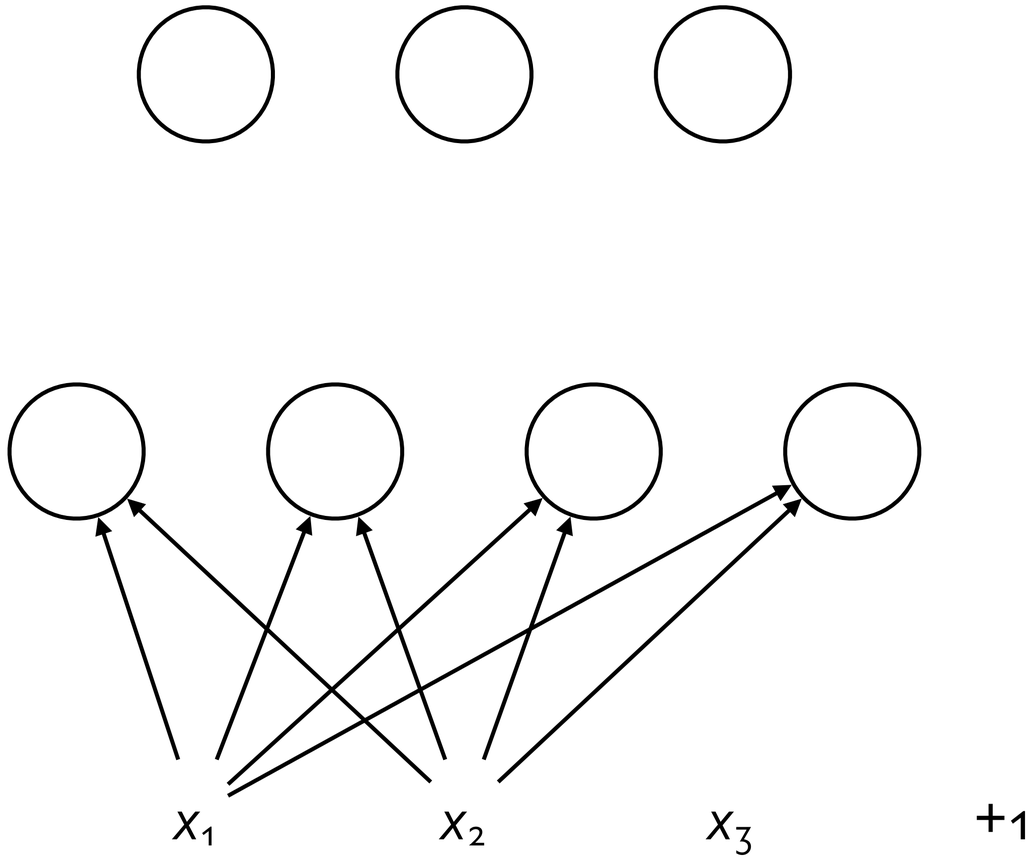




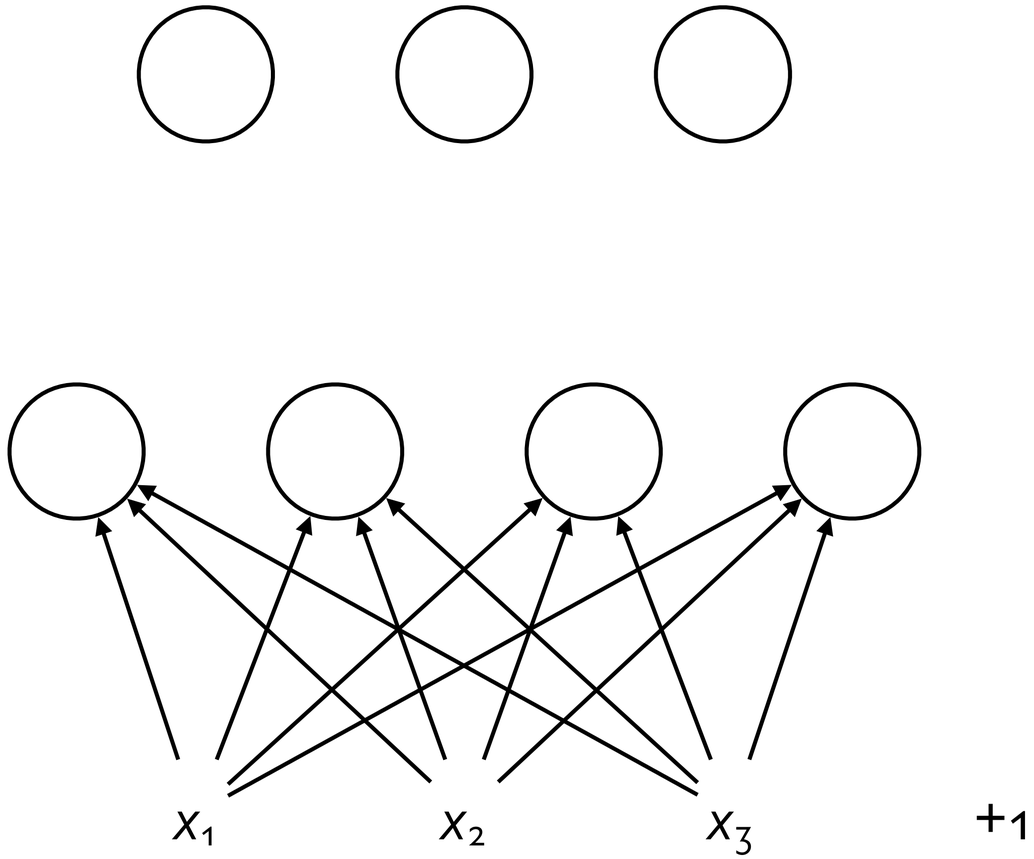
The output of each layer serves as the input to the next higher layer



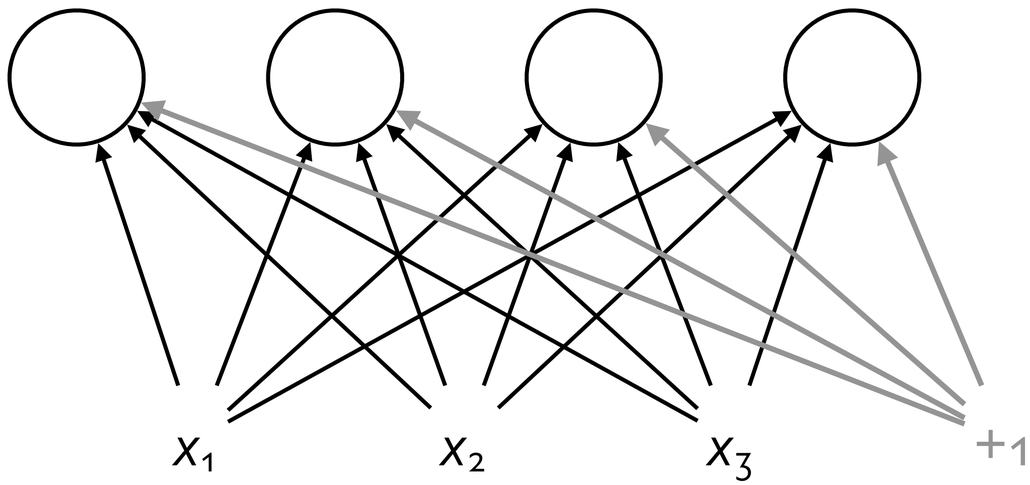
The output of each layer serves as the input to the next higher layer



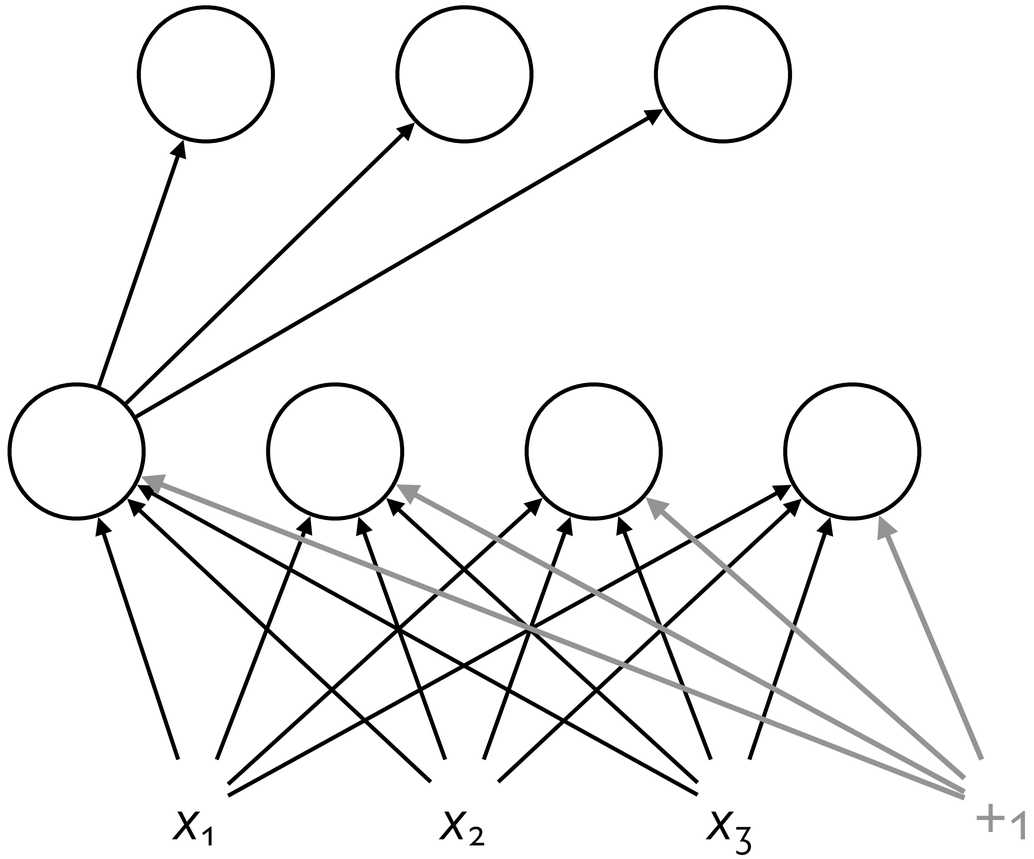
The output of each layer serves as the input to the next higher layer



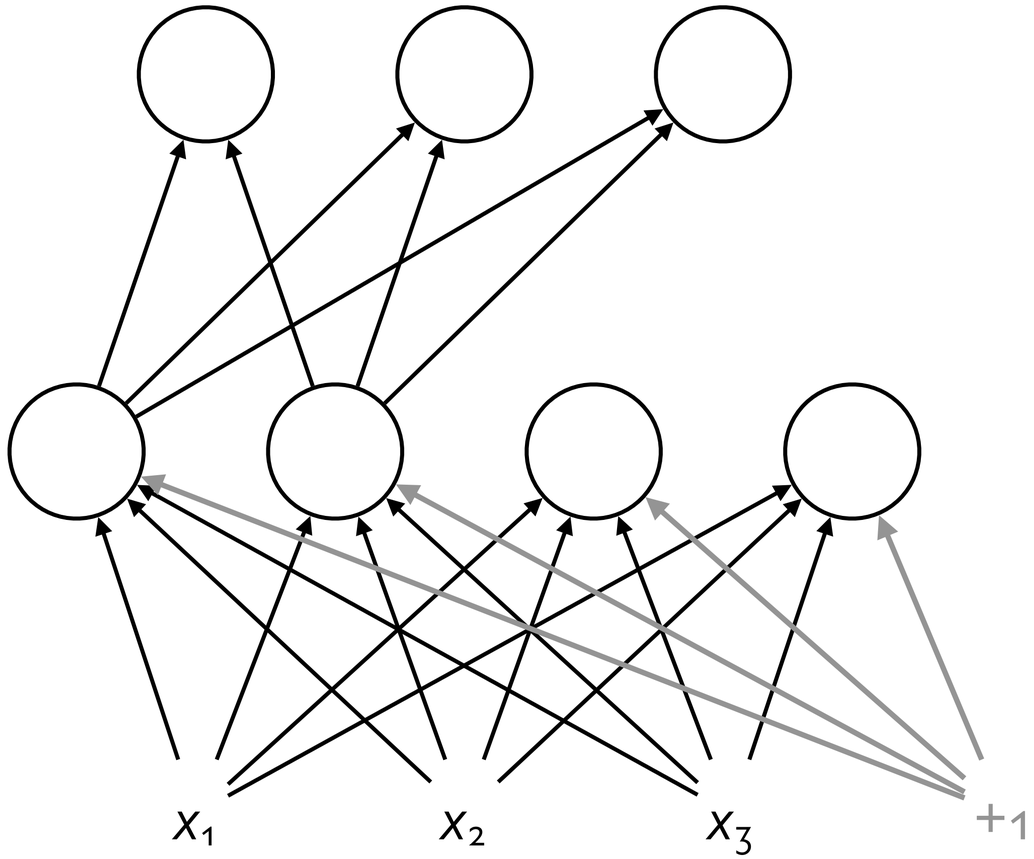
The output of each layer serves as the input to the next higher layer



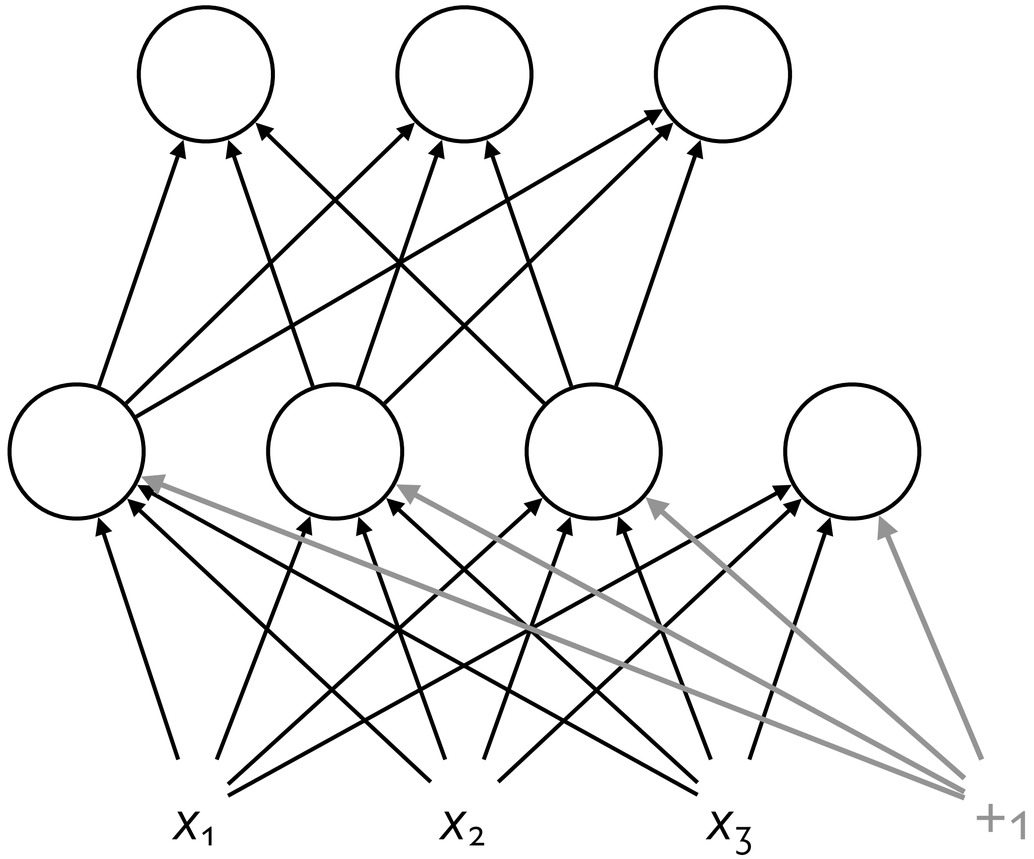
The output of each layer serves as the input to the next higher layer



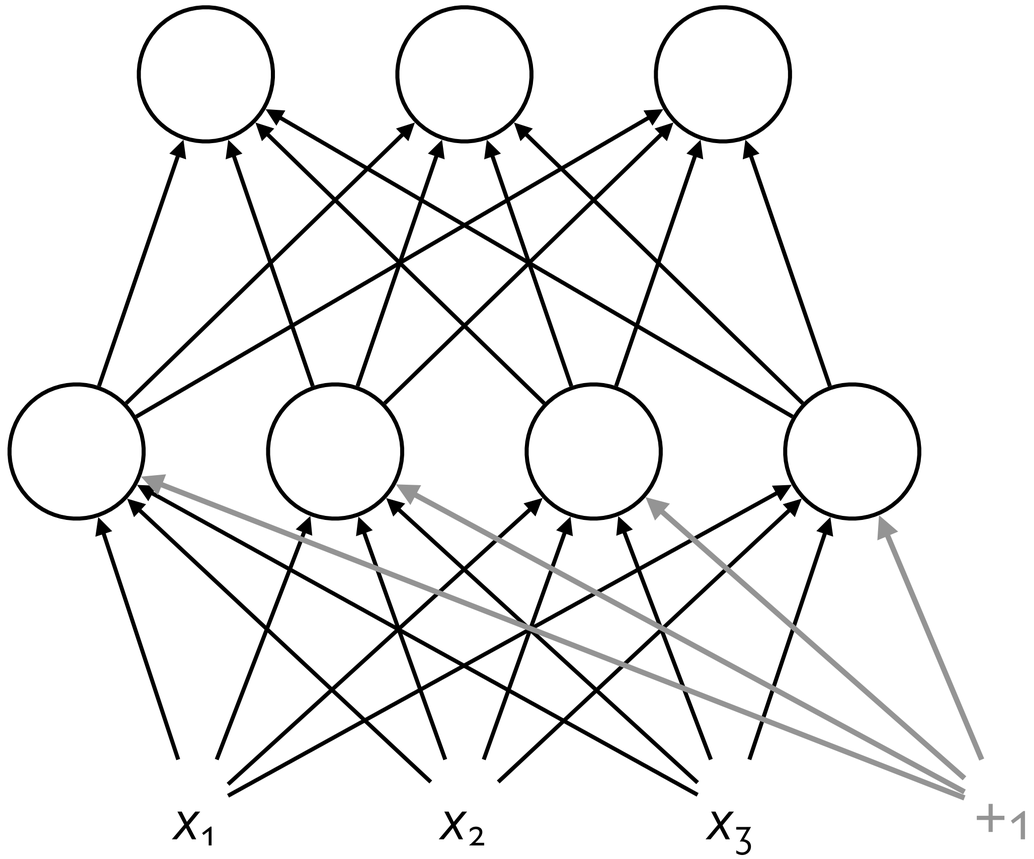
The output of each layer serves as the input to the next higher layer



The output of each layer serves as the input to the next higher layer

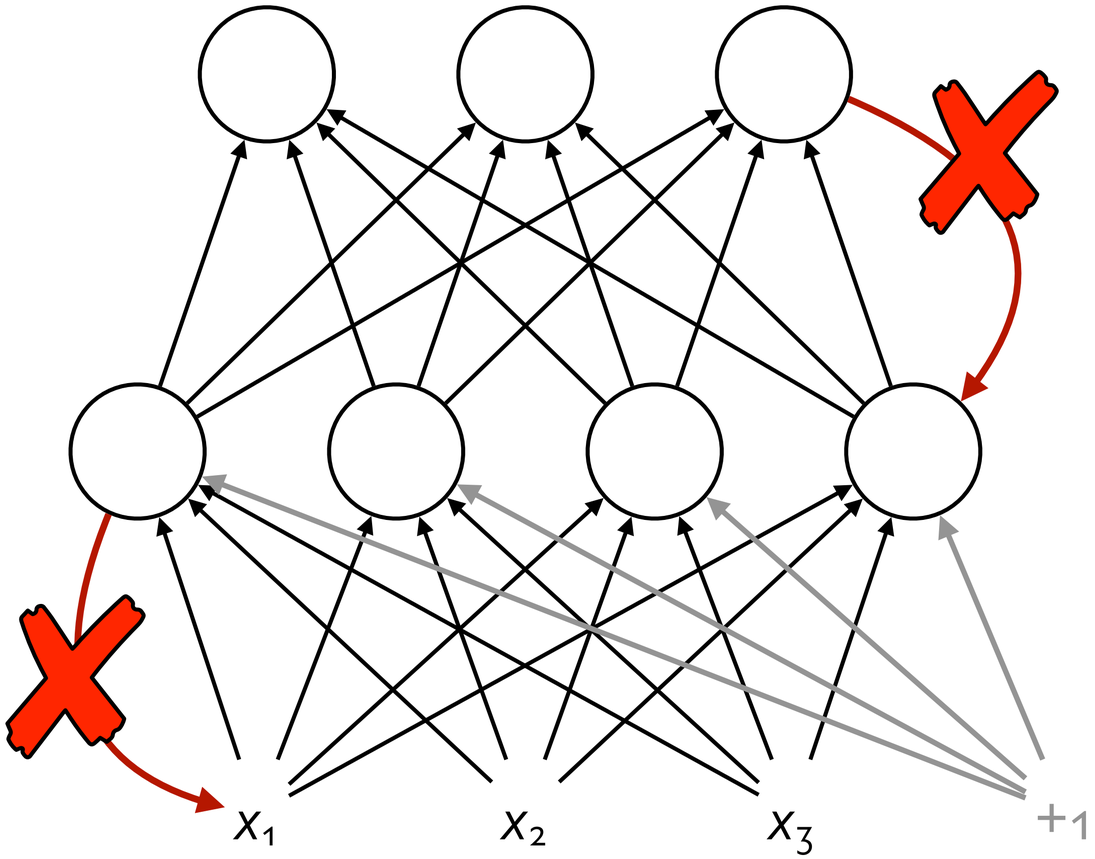


The output of each layer serves as the input to the next higher layer



There are no cycles.

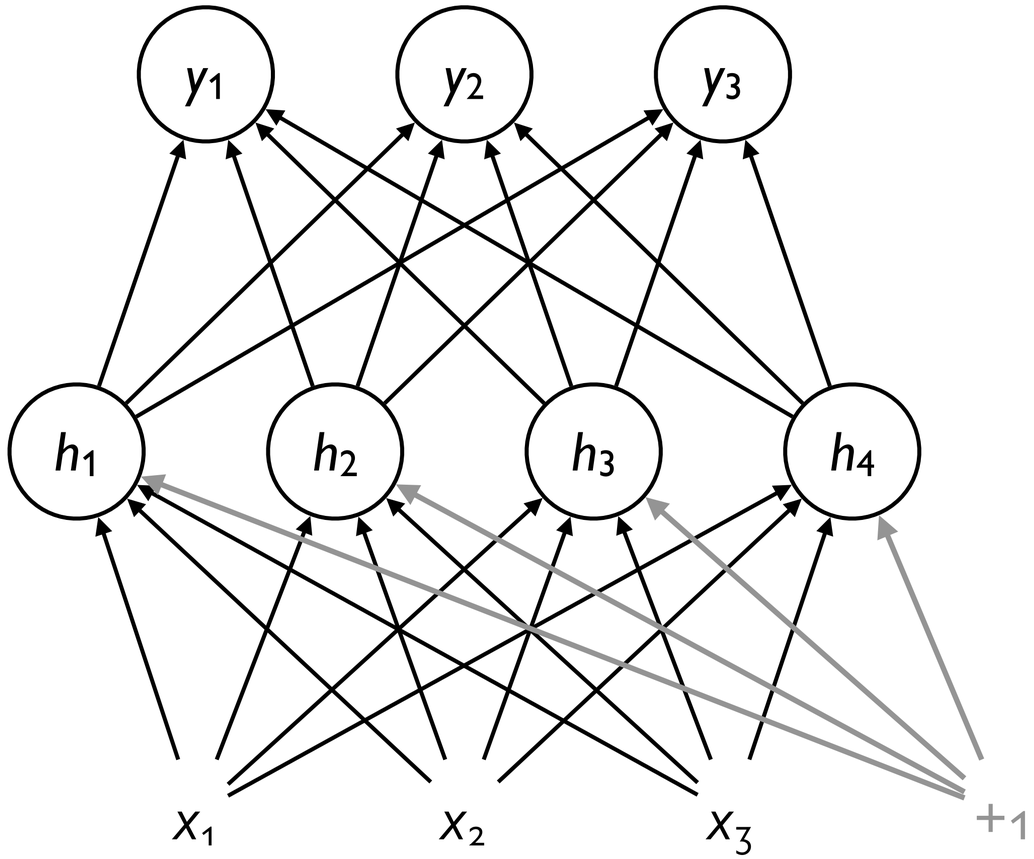
No outputs are passed back to lower layers.



Output layer

Hidden layer

Input layer

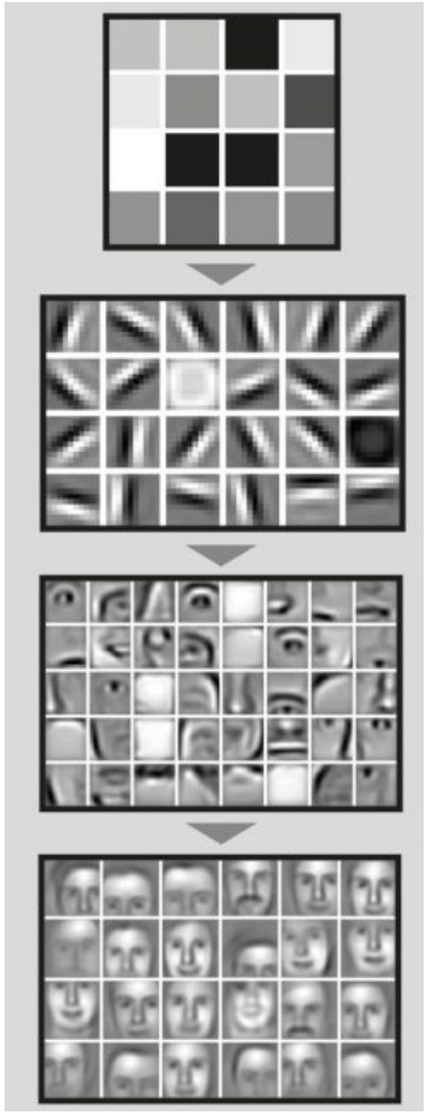


In the standard architecture, every layer is *fully connected*:

- Each unit in each layer takes as input the outputs from *all* the units in the previous layer.
- There is a link between every pair of units from two adjacent layers.

What are hidden layers for?

Hidden layers are mathematical functions, each designed to produce an output specific to an intended result



First hidden layer detects pixels of light and dark

Second hidden layer identifies edges and simple shapes

Third hidden layer comprises more complex objects from edges and simple shapes

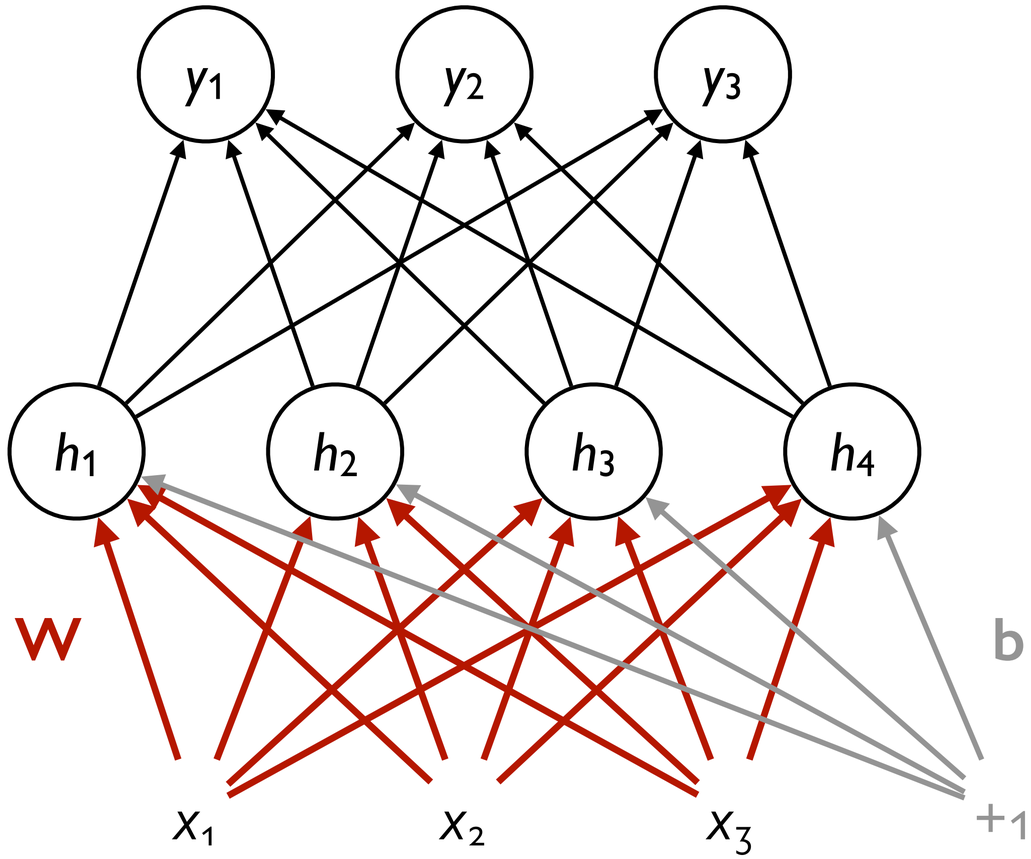
Output layer recognizes a human face with some confidence

For the hidden layer, we can combine the weight $\mathbf{w}_i$ and bias $\mathbf{b}_i$ for each unit i into

a single weight matrix $\mathbf{W}$, where

each element $\mathbf{W}_{i,j}$ represents the weight of the connection from the ith input unit x_i to the jth hidden unit h_j, and

a single bias vector $\mathbf{b}$ for the whole layer.



Then the hidden layer computation can be done very efficiently with simple matrix operations:

$$\mathbf{h} = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$$

One-layer network with scalar output

This is logistic regression!

$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$

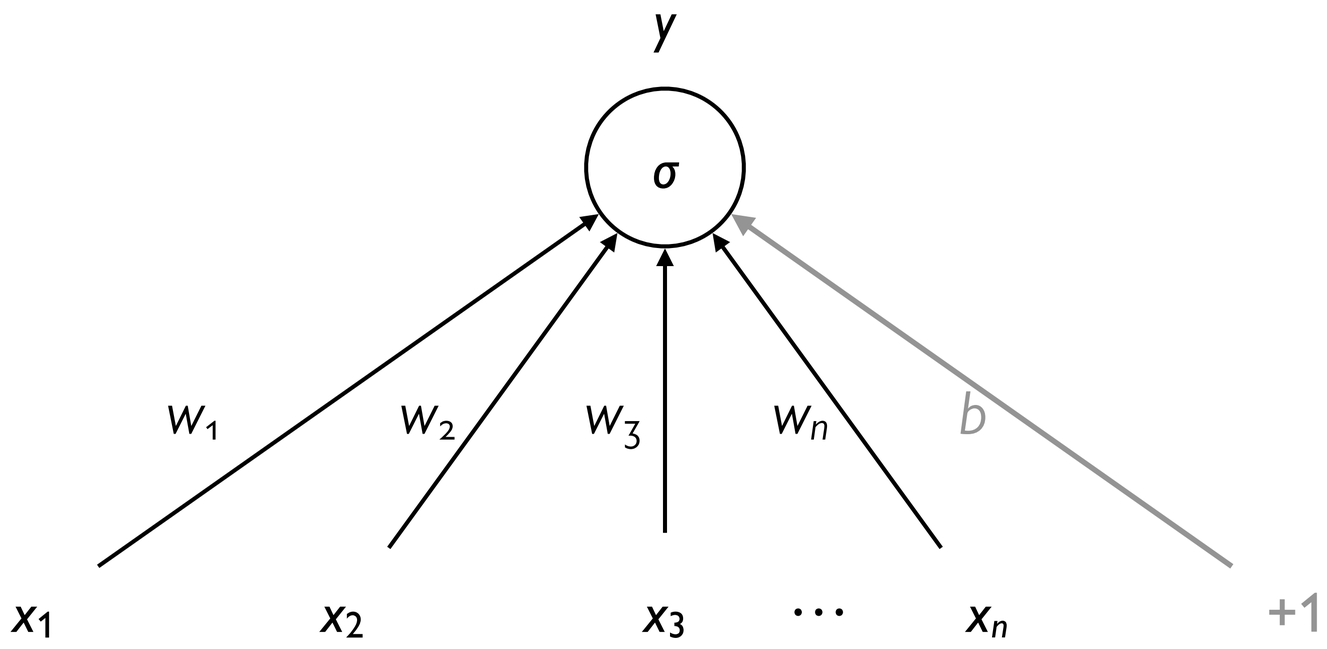
Output layer

Sigmoid node

Vector **w** *and scalar b*

Input layer

Vector **x**



Two-layer network with scalar output

Output layout

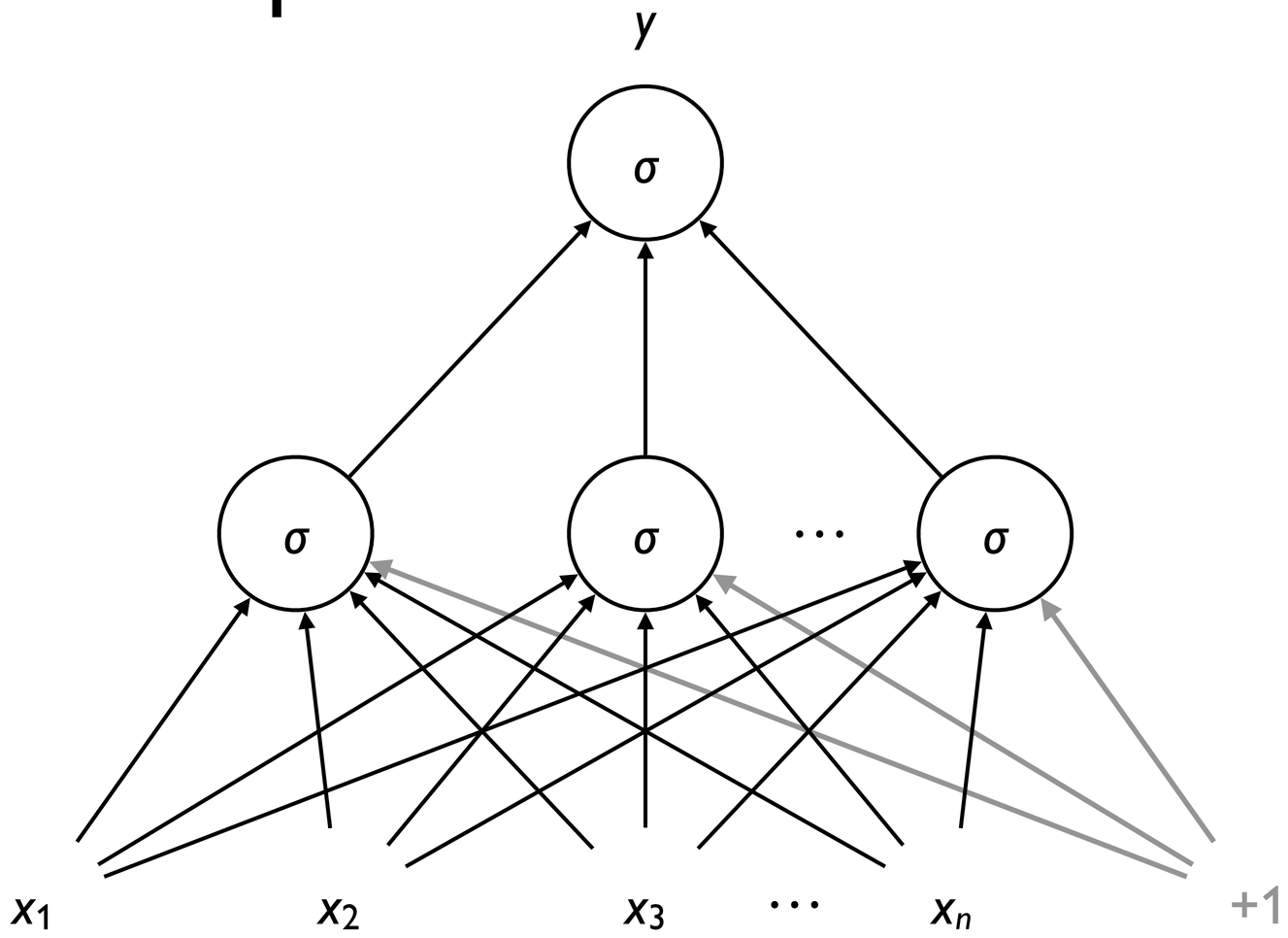
Matrix **U**

Hidden layout

Matrix **W** *and vector* **b**

Input layer

Vector **x**



$y = \sigma(\mathbf{U} \cdot \mathbf{h})$

$\mathbf{h} = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$

We can think of a neural network classifier with one hidden layer as

building a vector **h** – the hidden layer representation of the input – and

running standard logistic regression on the features that the network develops in **h**.

For multinomial classification, we need probabilities, but we have a vector of real-valued numbers.

Solution: *Softmax*!

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{d} \exp(z_j)}, \text{ where } 1 \leq i \leq d$$

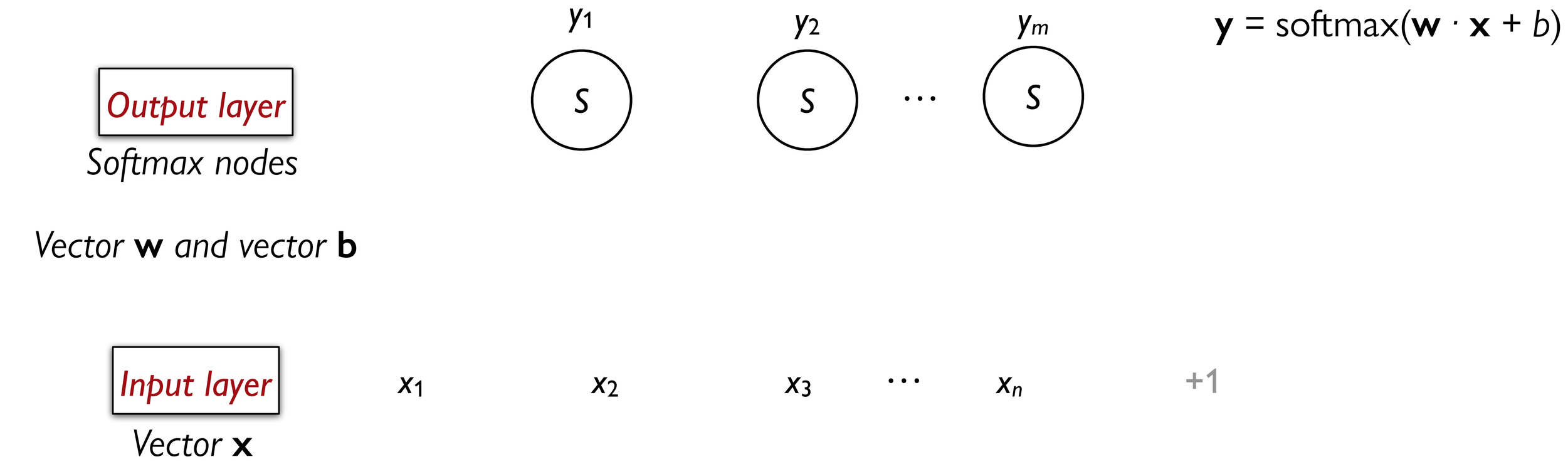
Function for normalizing a vector of real values into a vector encoding a probability distribution:

All the numbers lie between 0 and 1, and

they sum to 1.

One-layer network with softmax output

This is multinomial logistic regression!



One-layer network with softmax output

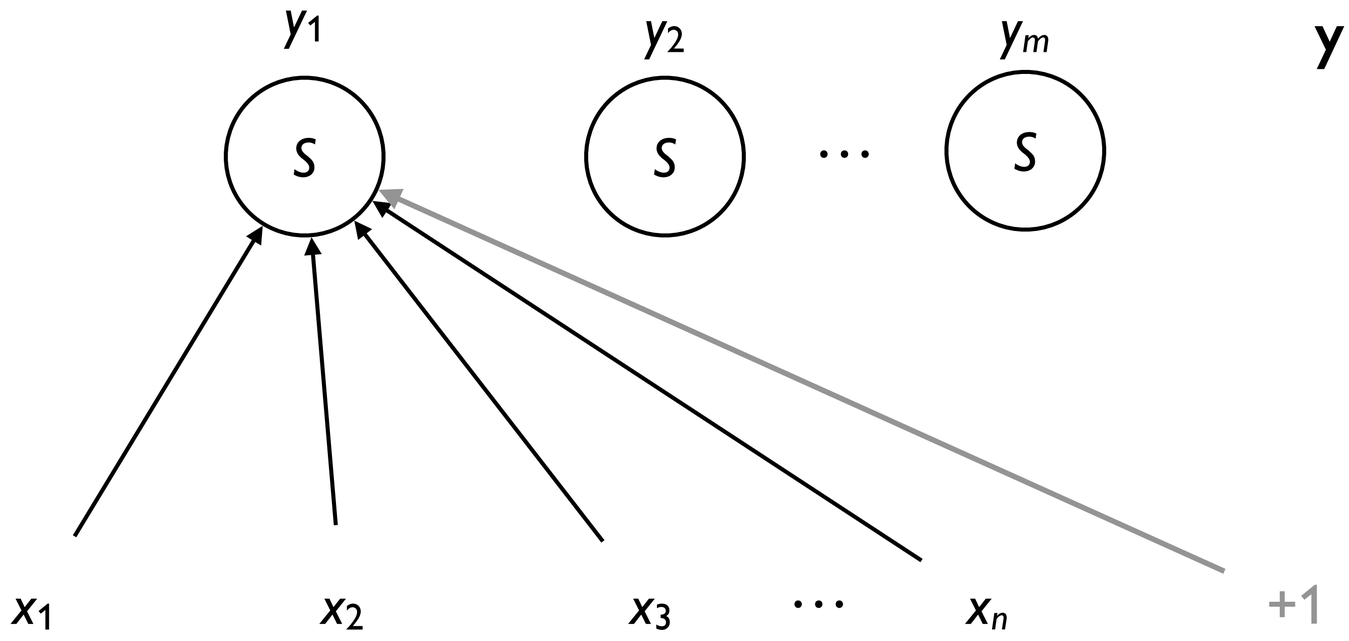
This is multinomial logistic regression!

$\mathbf{y} = \text{softmax}(\mathbf{w} \cdot \mathbf{x} + b)$

Output layer

Softmax nodes

Vector **w** *and vector* **b**



Input layer

Vector **x**

One-layer network with softmax output

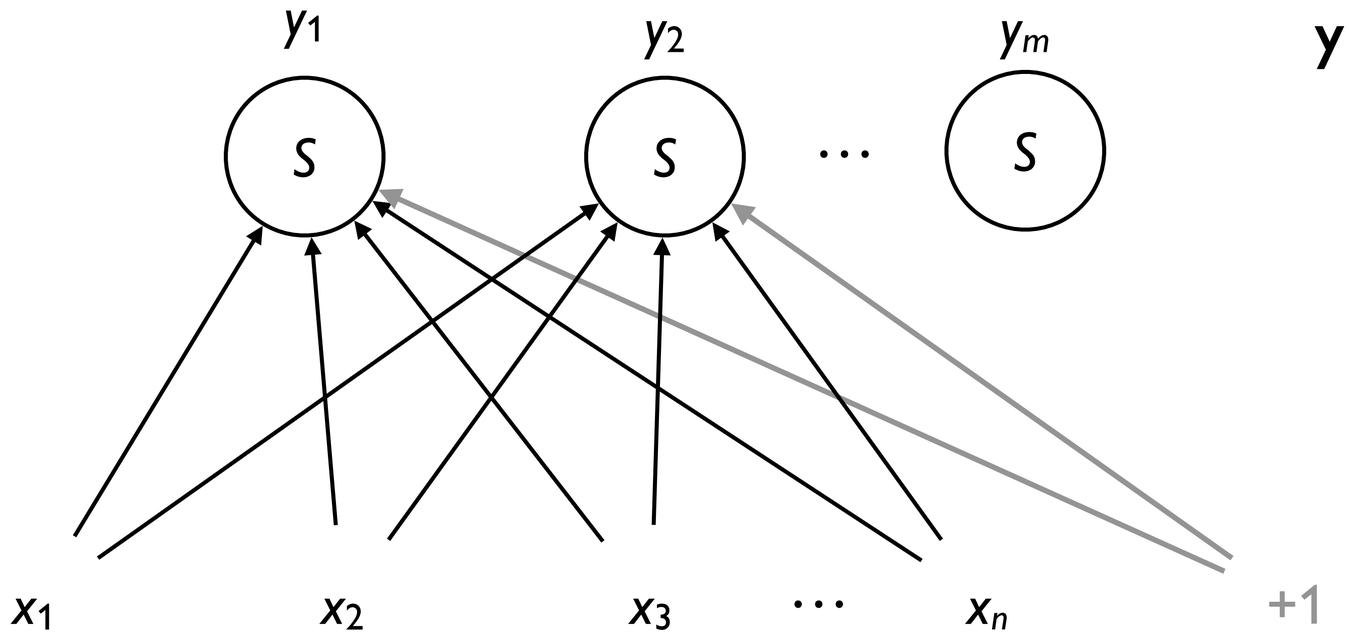
This is multinomial logistic regression!

$\mathbf{y} = \text{softmax}(\mathbf{w} \cdot \mathbf{x} + b)$

Output layer

Softmax nodes

Vector **w** *and vector* **b**



Input layer

Vector **x**

One-layer network with softmax output

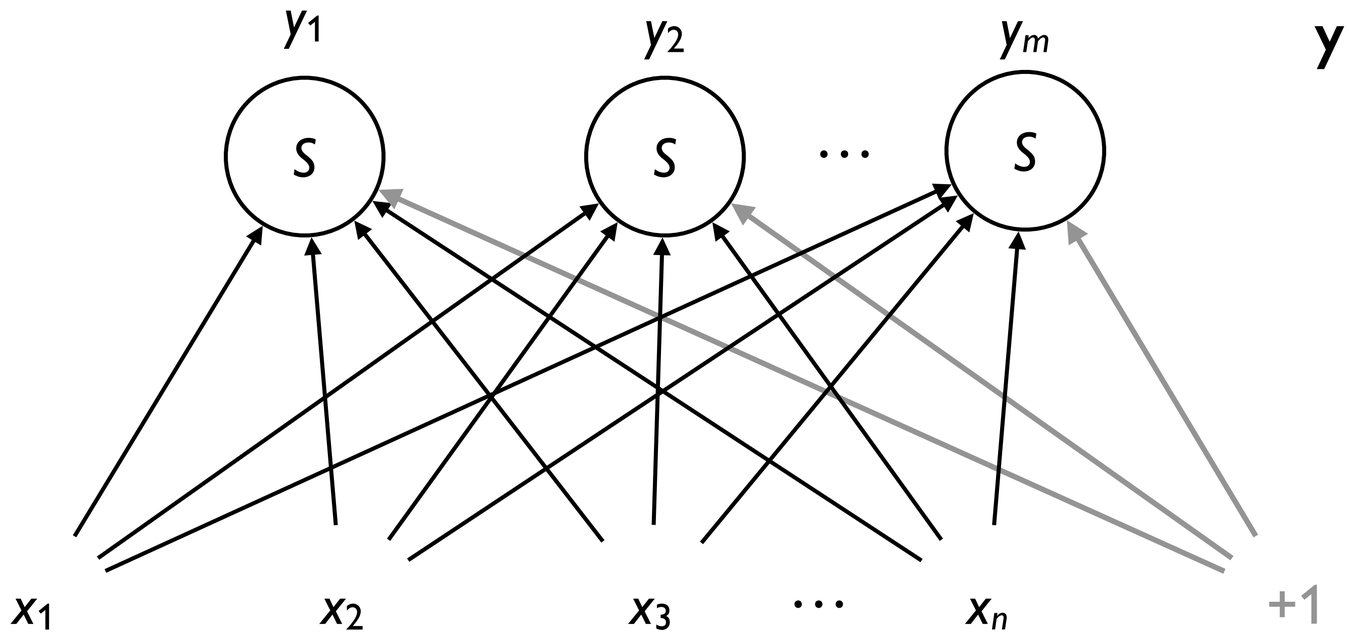
This is multinomial logistic regression!

$\mathbf{y} = \text{softmax}(\mathbf{w} \cdot \mathbf{x} + b)$

Output layer

Softmax nodes

Vector **w** *and vector* **b**



Input layer

Vector **x**

Two-layer network with softmax output

Output layer

Softmax nodes

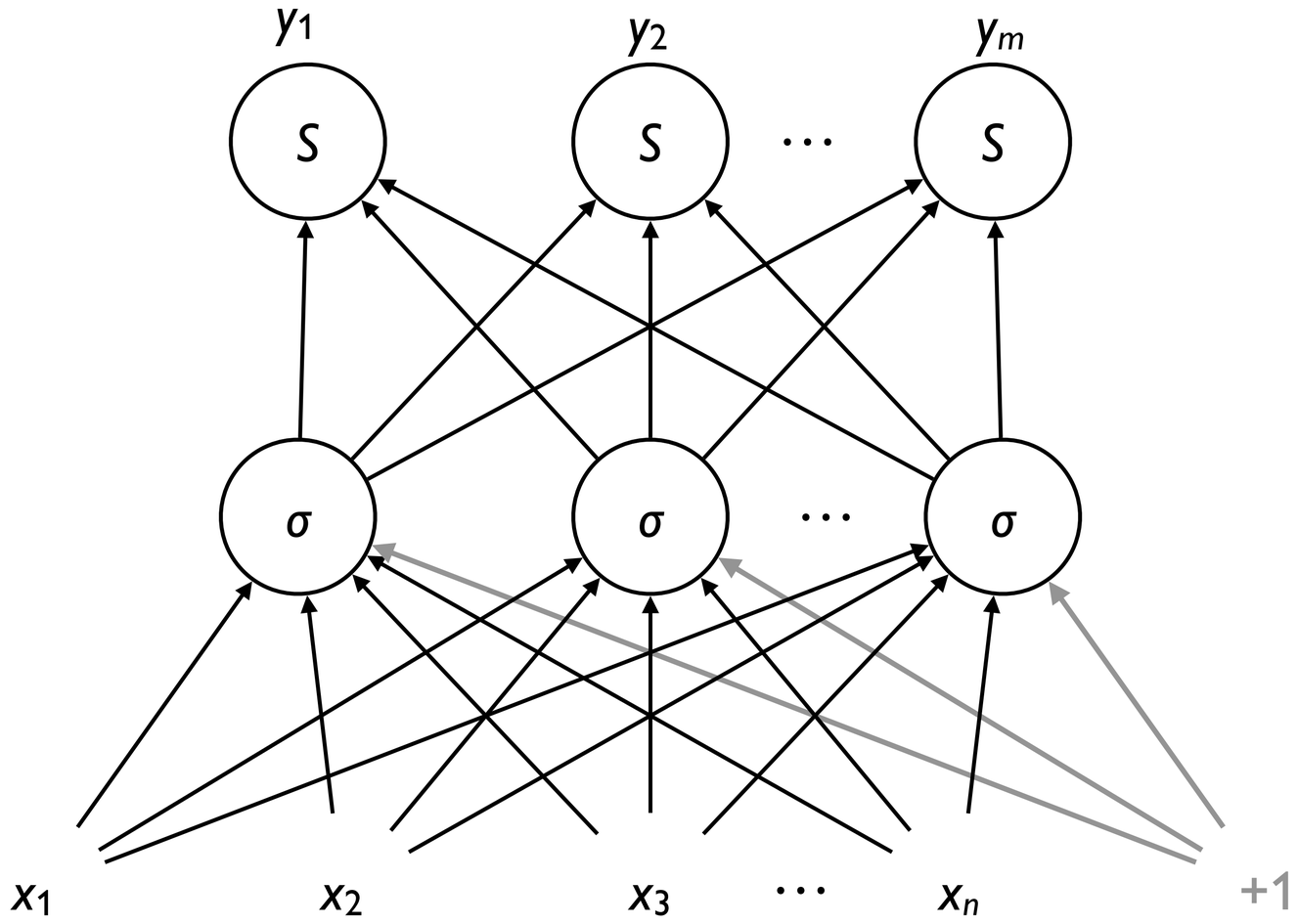
Matrix **U**

Hidden layer

Matrix **W** *and vector* **b**

Input layer

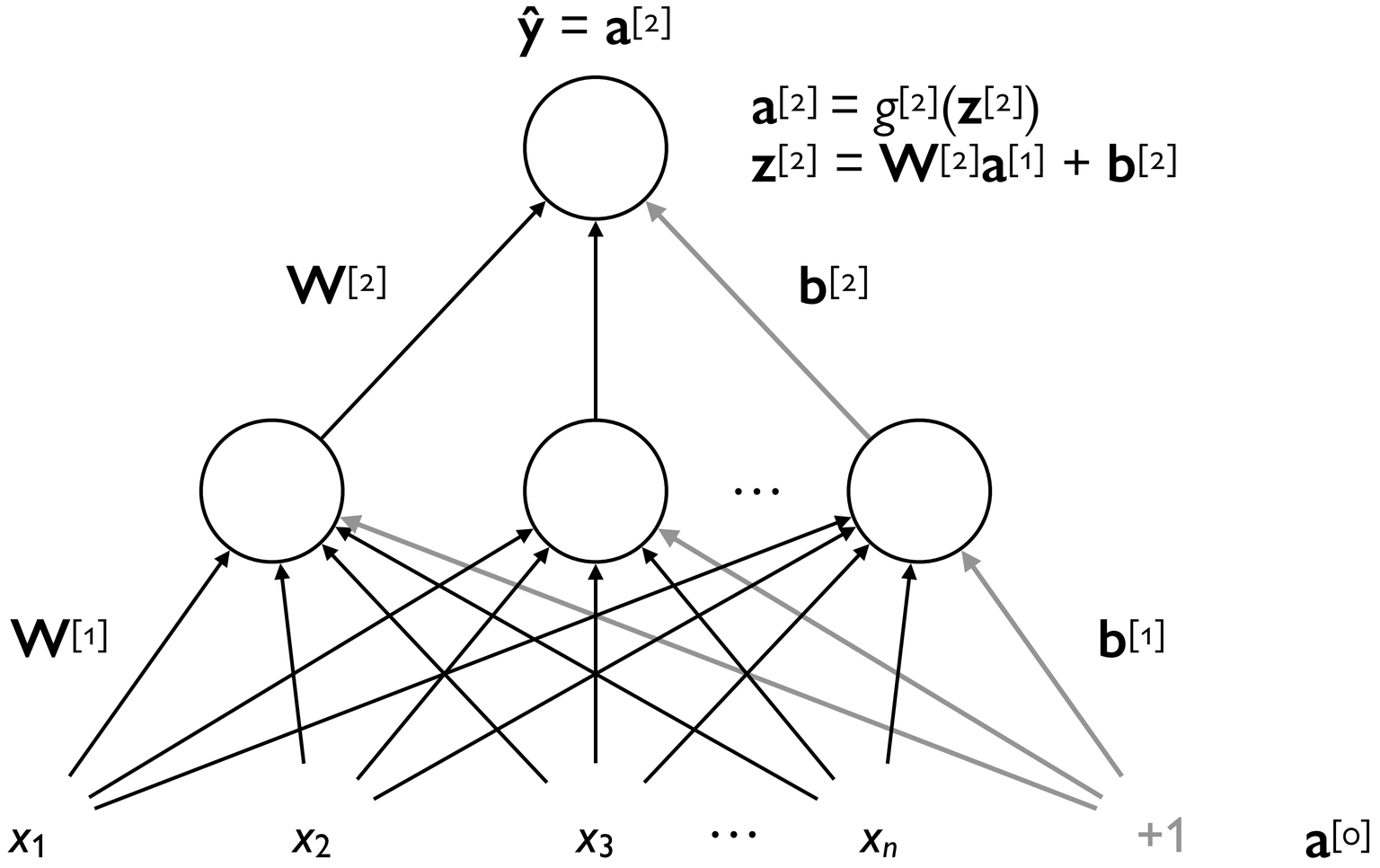
Vector **x**



$$\mathbf{y} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$$

$$\mathbf{h} = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$$

Multi-layer notation



Multi-layer notation

At every layer the network performs the same computation:

for i in $1, \ldots, n$:

$$\mathbf{z}^{[i]} = \mathbf{W}^{[i]} \cdot \mathbf{a}^{[i-1]} + \mathbf{b}^{[i]}$$

$$\mathbf{a}^{[i]} = g^{[i]}(\mathbf{z}^{[i]})$$

$$\hat{\mathbf{y}} = \mathbf{a}^{[n]}$$

Replacing the bias unit

Let's switch to a notation without the bias unit.

This is just a notational change:

1. Add a dummy node $\mathbf{a}_0 = 1$ to each layer
2. Its weight $\mathbf{w}_0$ will be the bias
3. So, input layer $\mathbf{a}_0^{[0]} = 1$, and $\mathbf{a}_0^{[1]} = 1$, $\mathbf{a}_0^{[2]} = 1$, etc.

Replacing the bias unit

Instead of

$$x = x_1, x_2, \ldots x_{n_0}$$

$$\mathbf{h} = \sigma(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$$

$$\mathbf{h}_j = \sigma\left(\sum_{i=1}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i + \mathbf{b}_j \right)$$

We'll do this

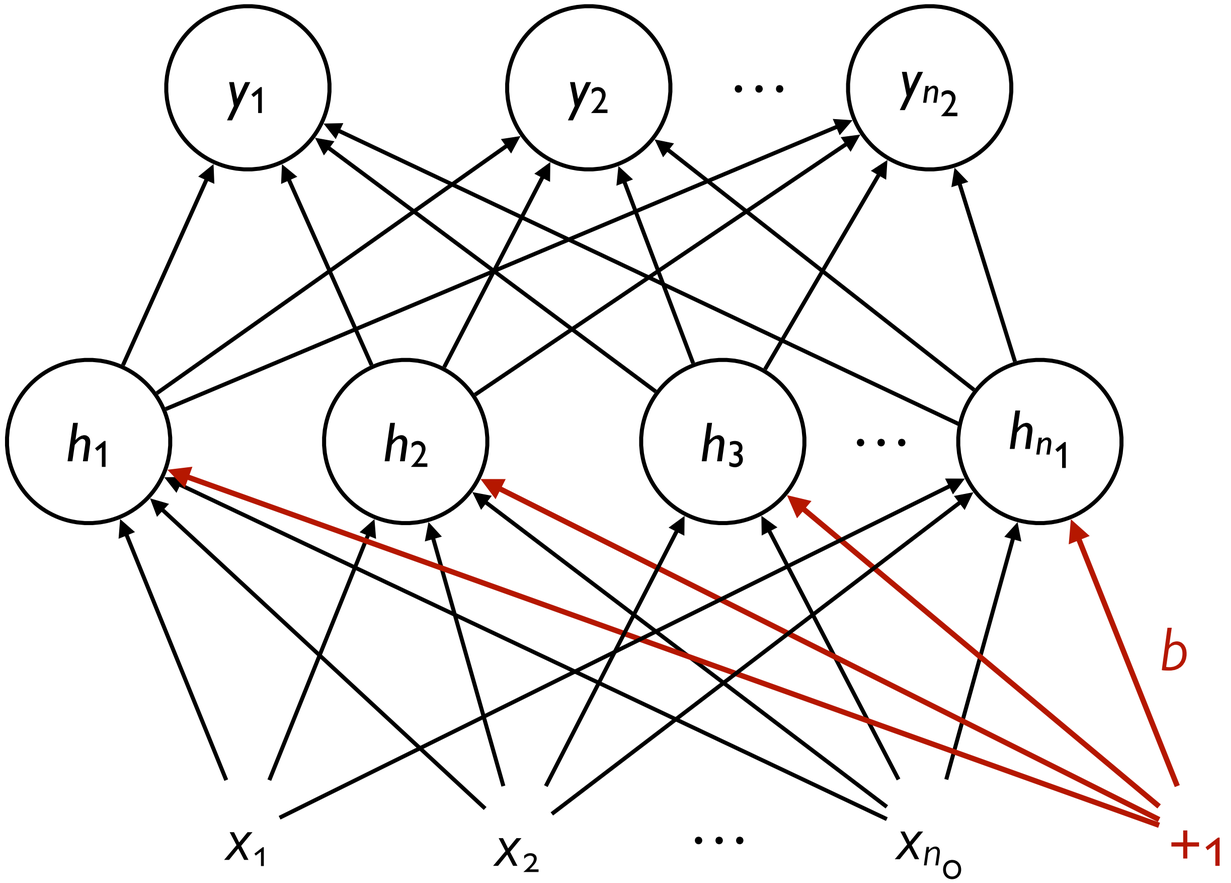
$$x = x_0, x_1, \ldots x_{n_0}$$

$$\mathbf{h} = \sigma(\mathbf{W} \cdot \mathbf{x})$$

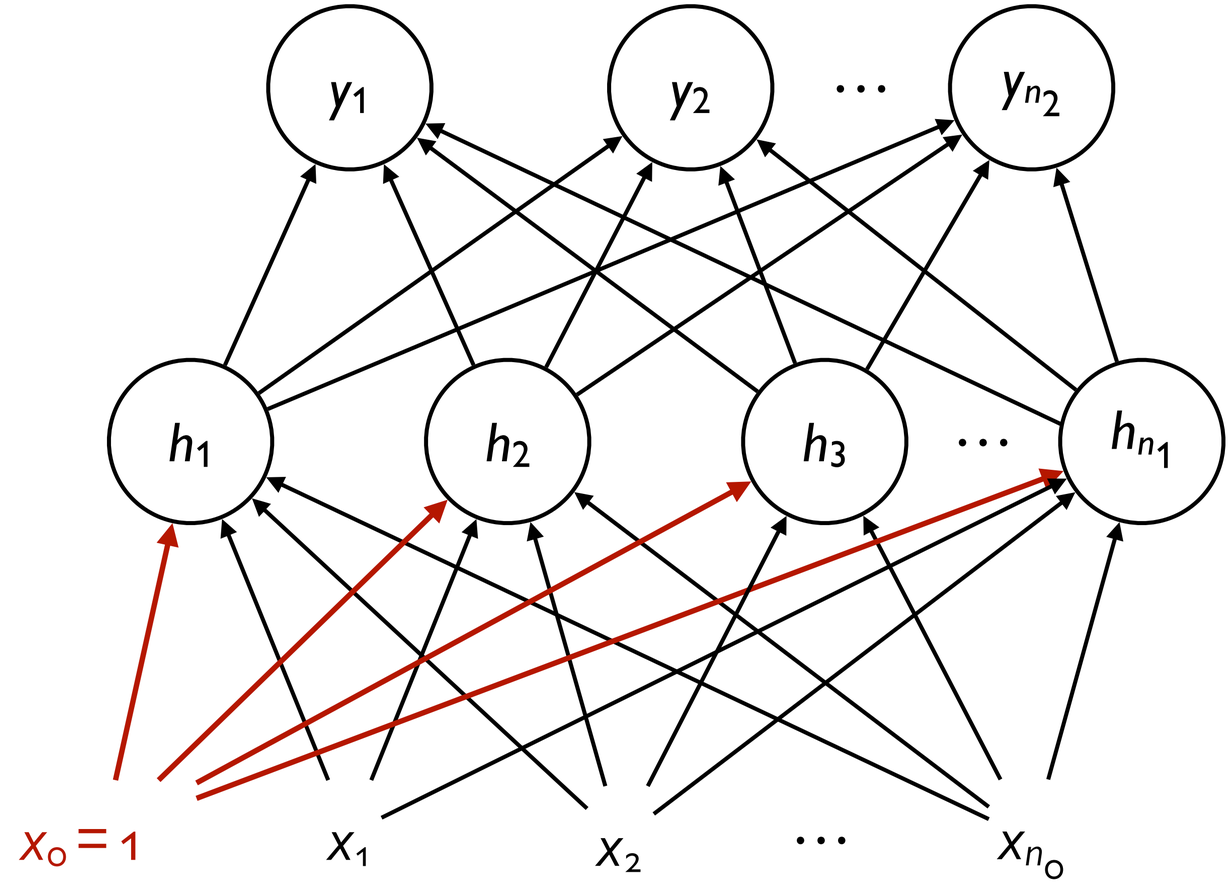
$$\mathbf{h}_j = \sigma\left(\sum_{i=0}^{n_0} \mathbf{W}_{ji}\mathbf{x}_i \right)$$

Replacing the bias unit

Instead of



We'll do this



Using feedforward networks

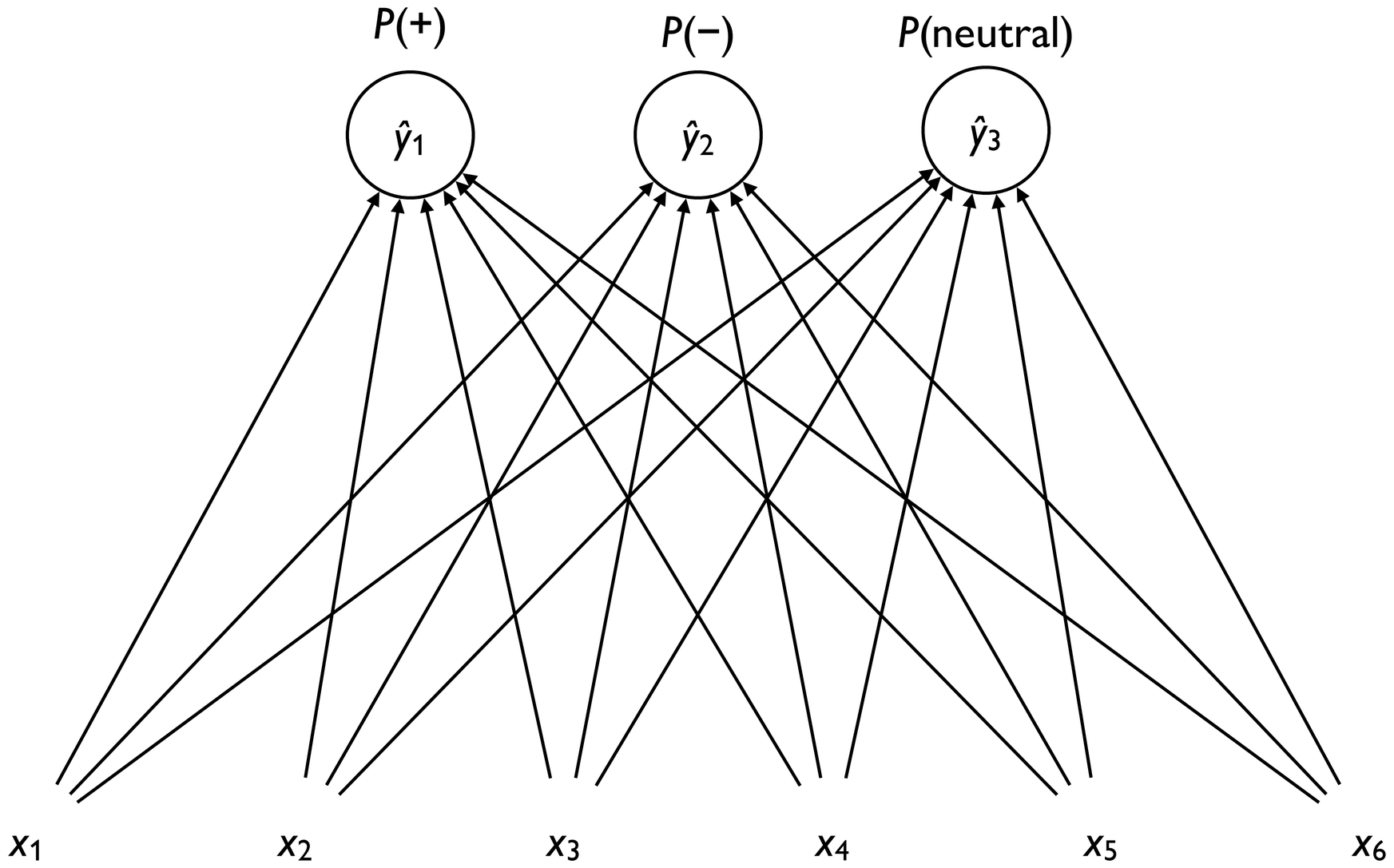
Let's reconsider text classification.

We can start with a logistic regression classifier,
which corresponds to a one-layer network.

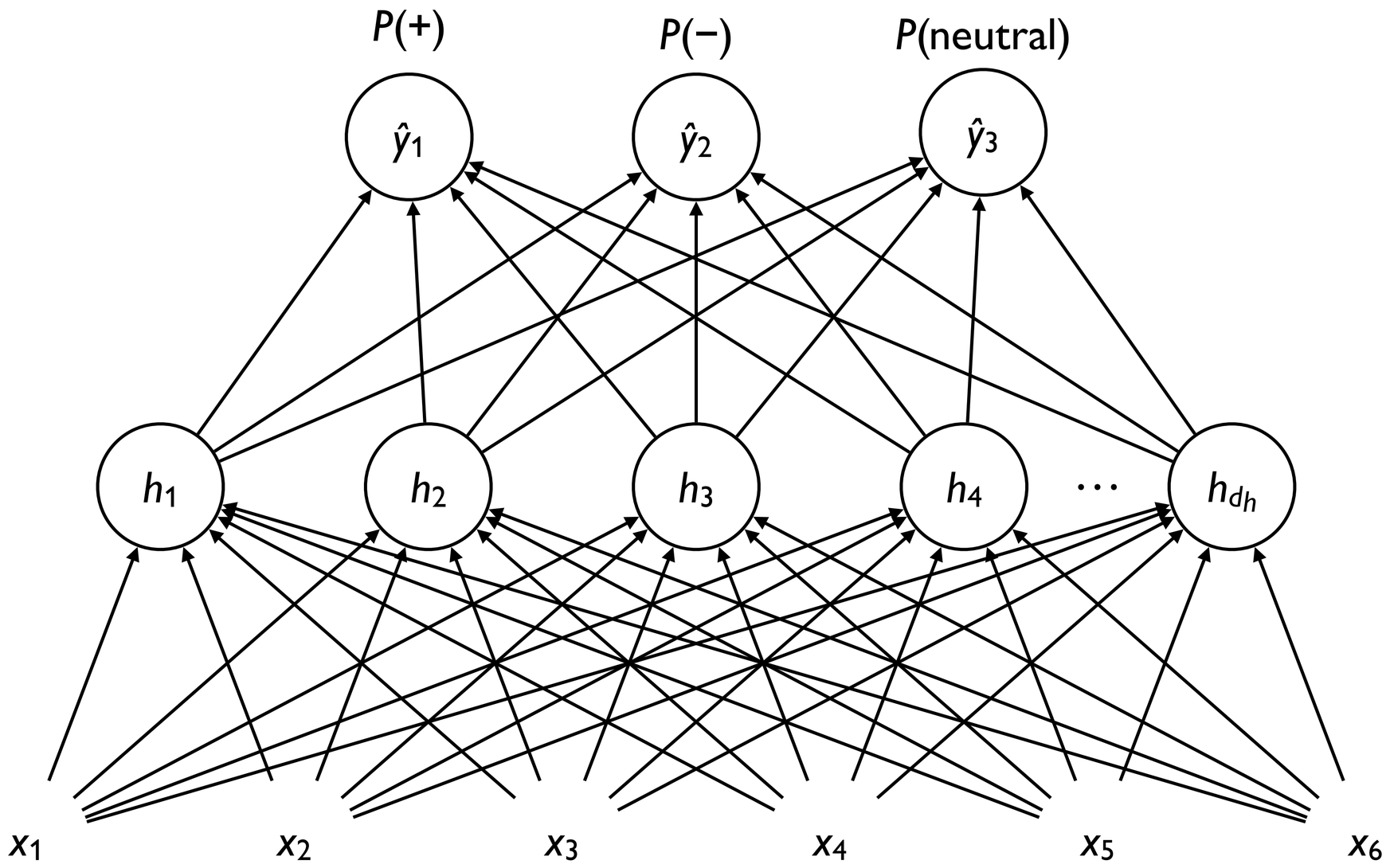
The input layer can consist of scalar features, like before:

Var	Definition
x_1	count(positive lexicon words $\in$ doc)
x_2	count(negative lexicon words $\in$ doc)
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_4	count(1st and 2nd pronouns $\in$ doc)
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_6	log(word count of doc)

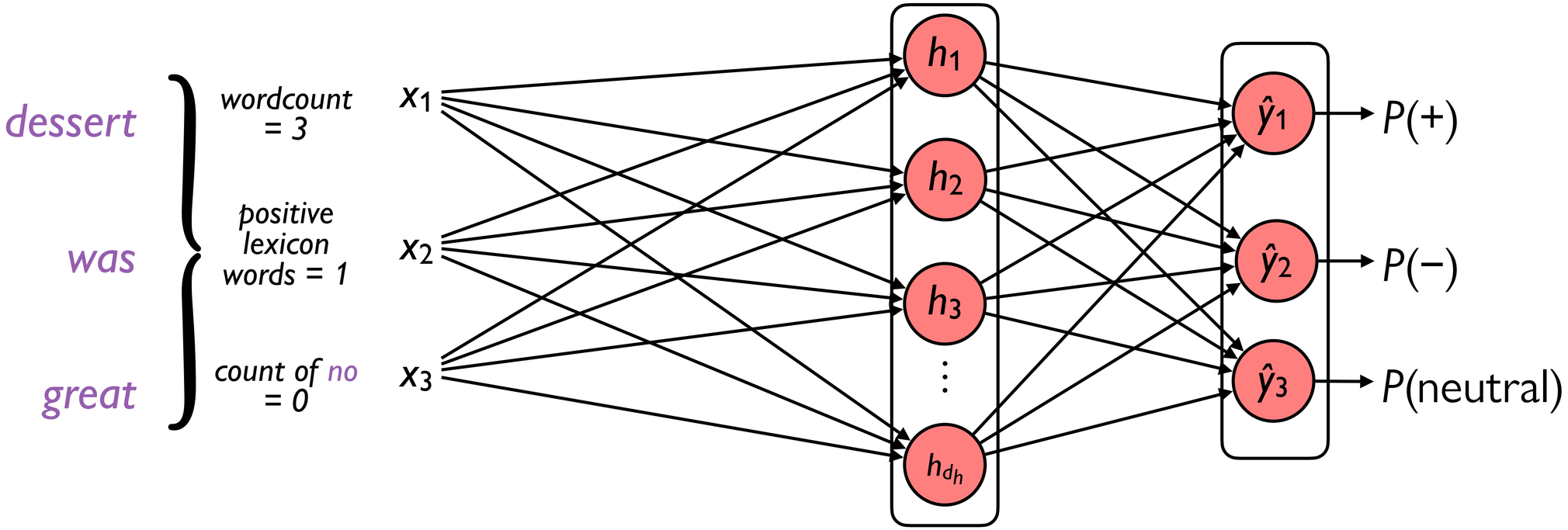
And the output layer has a node for each label:



And then we can add a hidden layer:



Neural net classification with embeddings as input features



x $[d \times 1]$

W $[d_h \times 3d]$

h $[d_h \times 1]$

U $[3 \times d_h]$

y $[3 \times 1]$

Input layer

Hidden layer

Output layer

d = 3 features

Softmax

The real power of deep learning comes from the ability to *learn features* from the data.

Instead of using hand-built human-engineered features for classification, use learned representations like embeddings!

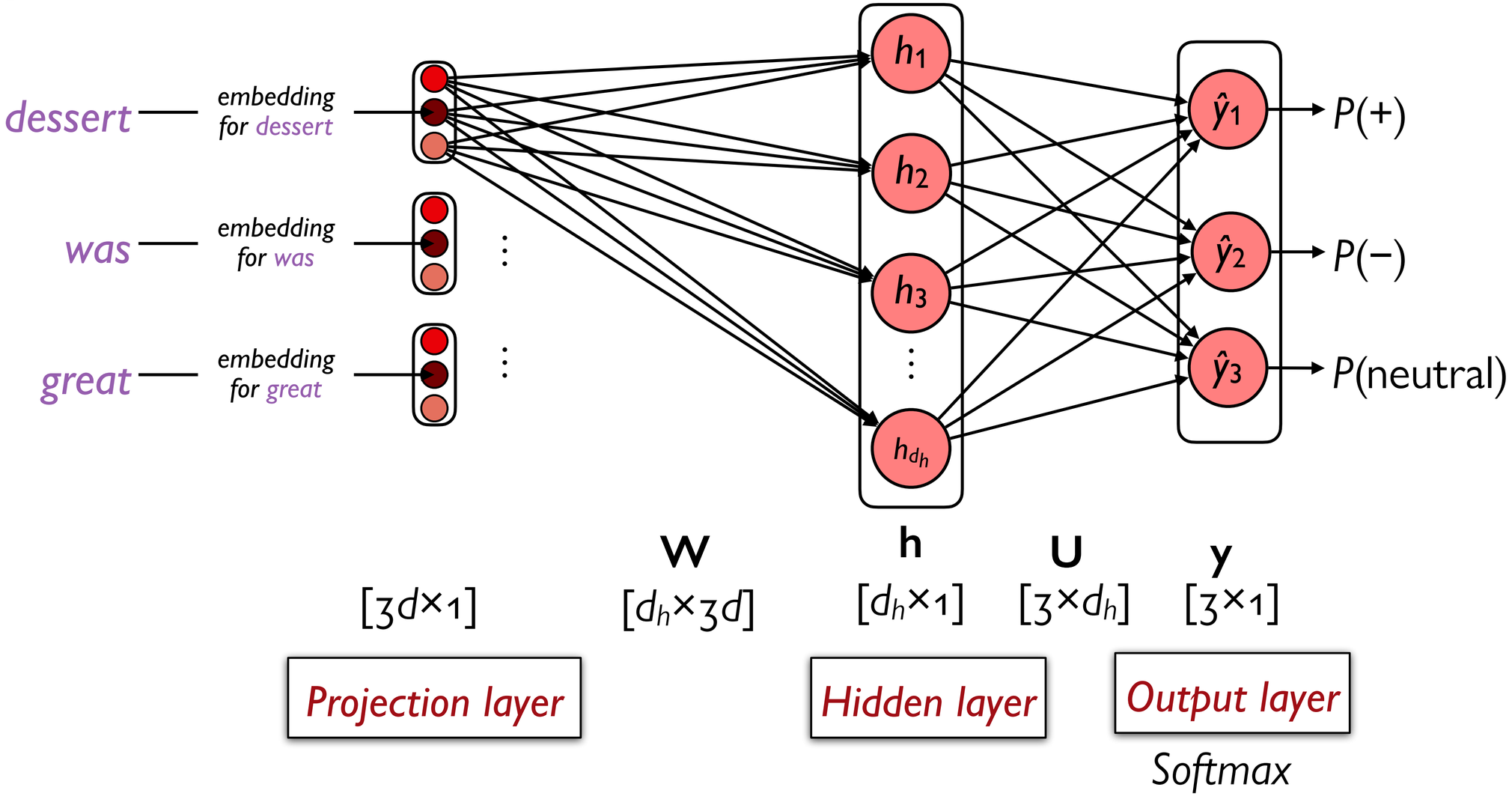
An embedding is a vector of dimension $[1 \times d]$ that represents the input token.

An embedding matrix **E** is a dictionary, one row per token of vocab. V.

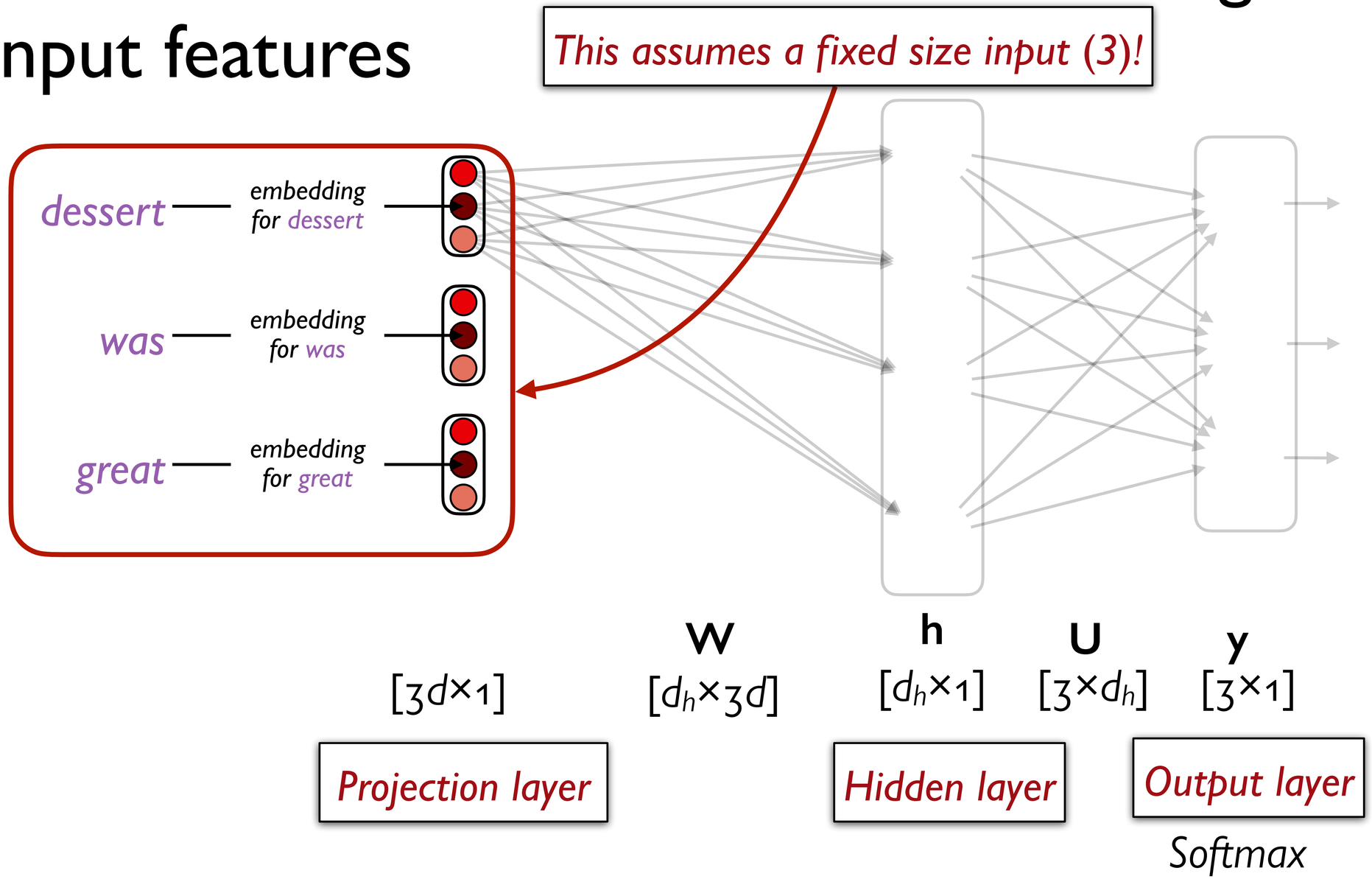
E has shape $[|V| \times d]$

Embedding matrices are central to NLP; they represent input text in LLMs and all modern NLP tools.

Neural net classification with embeddings as input features



Neural net classification with embeddings as input features



One approach: Make the input the length of the longest input document

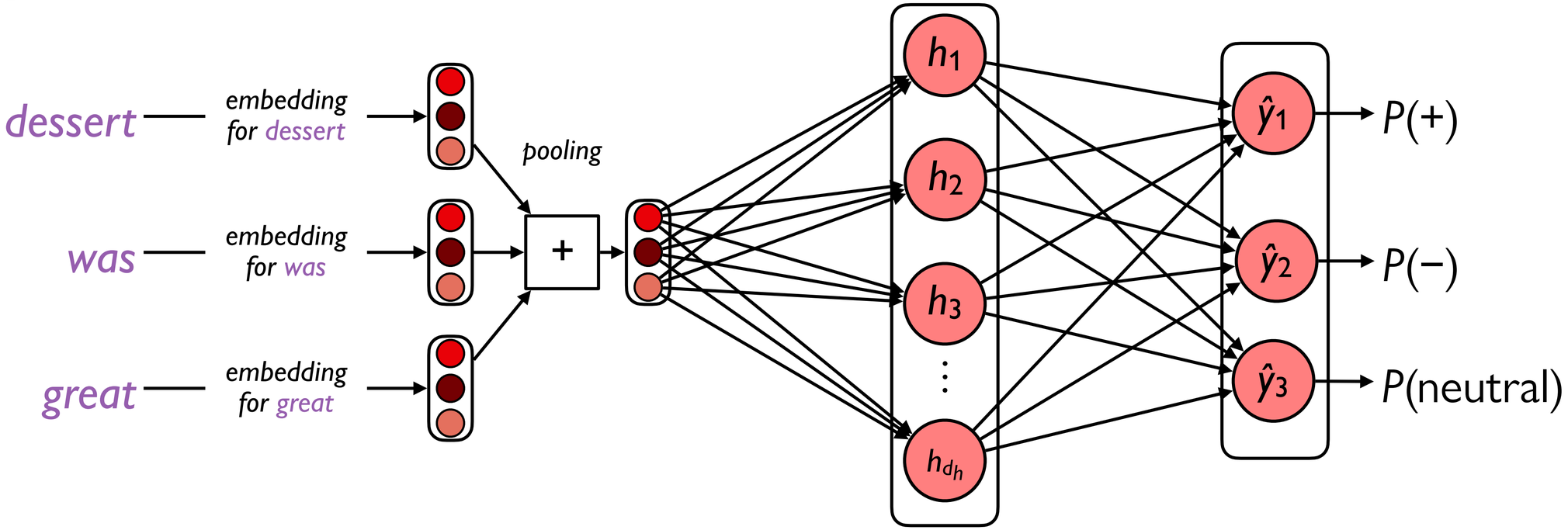
- If it's shorter, pad it with zero embeddings
- If you get a longer input at test time, truncate it ☹️

Alternatively, we can take word embeddings (like Word2vec) and apply a *pooling* function to the embeddings of all the words in the input, making a single embedding for the entire input.

Simple pooling functions:

- add up all the embedding vectors
- average all the embedding vectors

Neural net classification with embeddings as input features



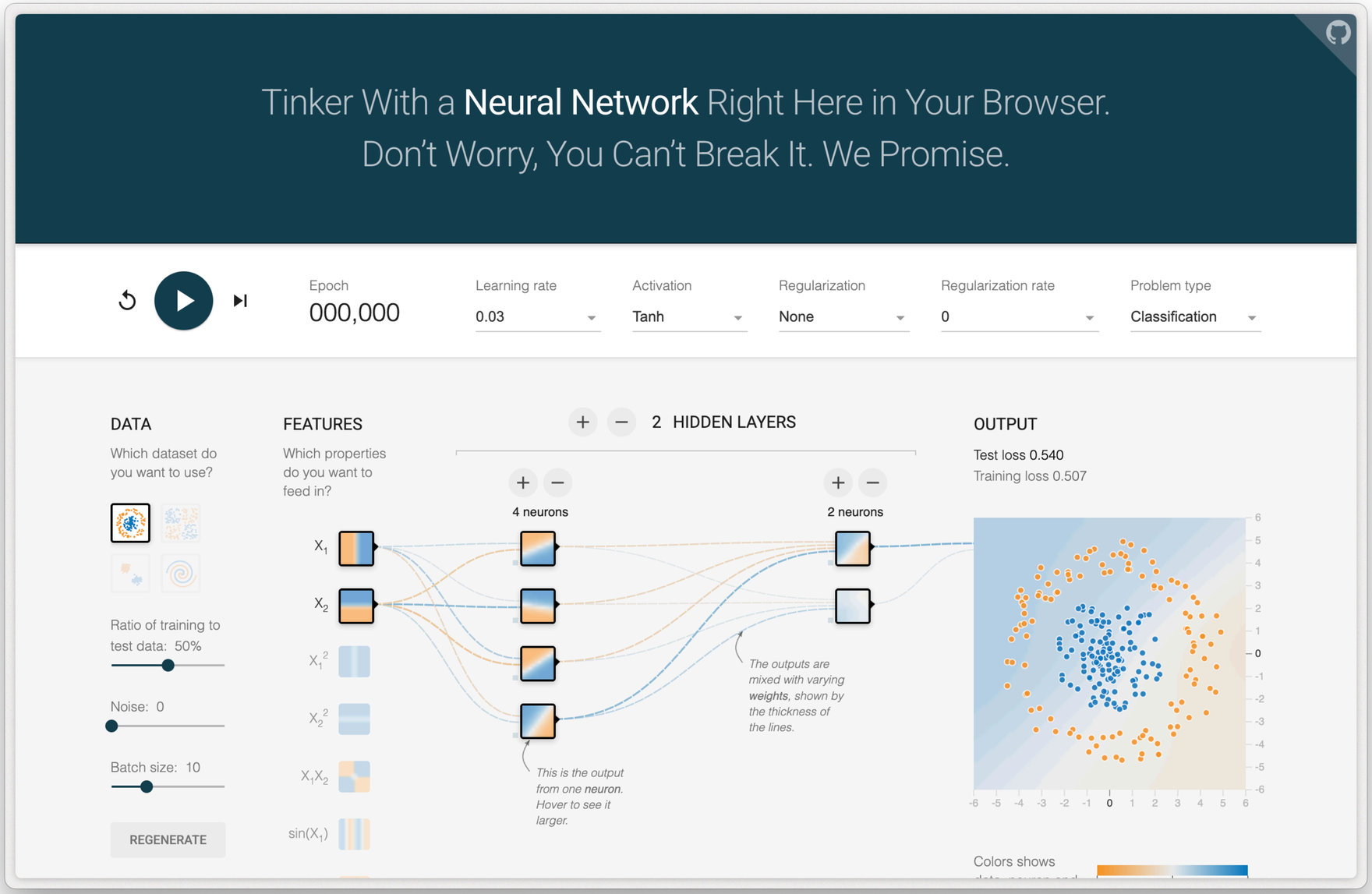
The idea of relying on another algorithm to have already learned an embedding representation for our input words – as when we use Word2vec embeddings of the input – is called *pretraining*.

Deep feedforward neural networks

A simple way to increase the range of features that a model can express is to add more hidden layers, each of which is a combination of a linear transformation and a nonlinear function.

A deep feedforward neural network is a type of neural network that has more than one hidden layer.

The depth refers to the number of hidden layers, and the width refers to the number of neurons in each layer.



playground.tensorflow.org

Are more layers always better?

Not necessarily!

More parameters and more computation

More prone to overfitting and underfitting

More difficult to optimize and converge

Techniques for improving the performance and generalization of deep networks:

Regularization – add some penalty or constraint to the network to reduce its complexity and prevent overfitting

Dropout – randomly dropping out some neurons and connections during training to reduce the co-dependence of neurons and increase the robustness of the network

Acknowledgments

This class incorporates material from:

Jurafsky & Martin, *Speech and Language Processing*, 3rd ed. draft