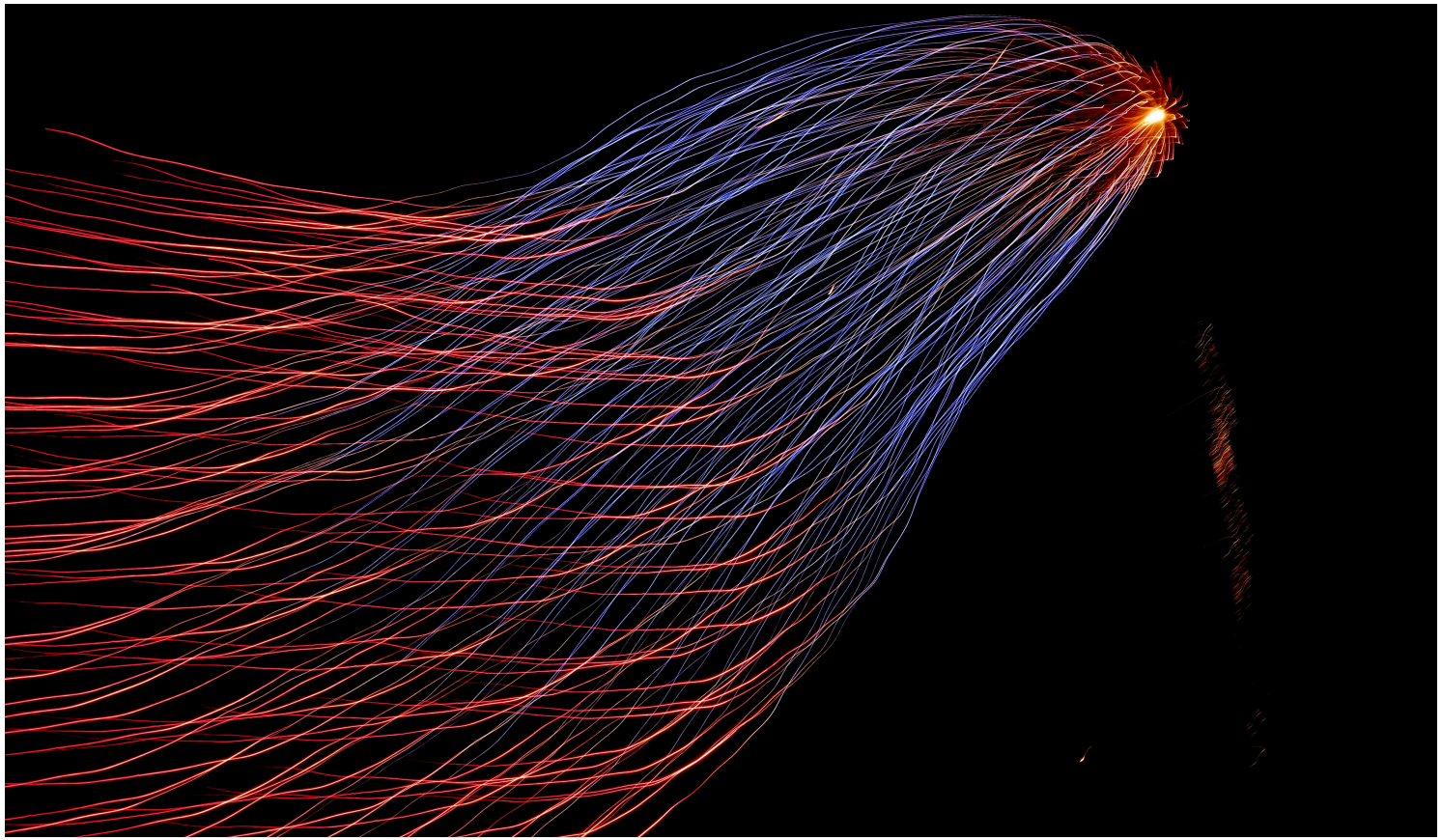


Feedforward Neural Networks

Natalie Parde, Ph.D.

Department of Computer Science

University of Illinois at Chicago

CS 521: Statistical Natural Language Processing

Spring 2020

Many slides adapted from Jurafsky and Martin (https://web.stanford.edu/~jurafsky/slp3/).

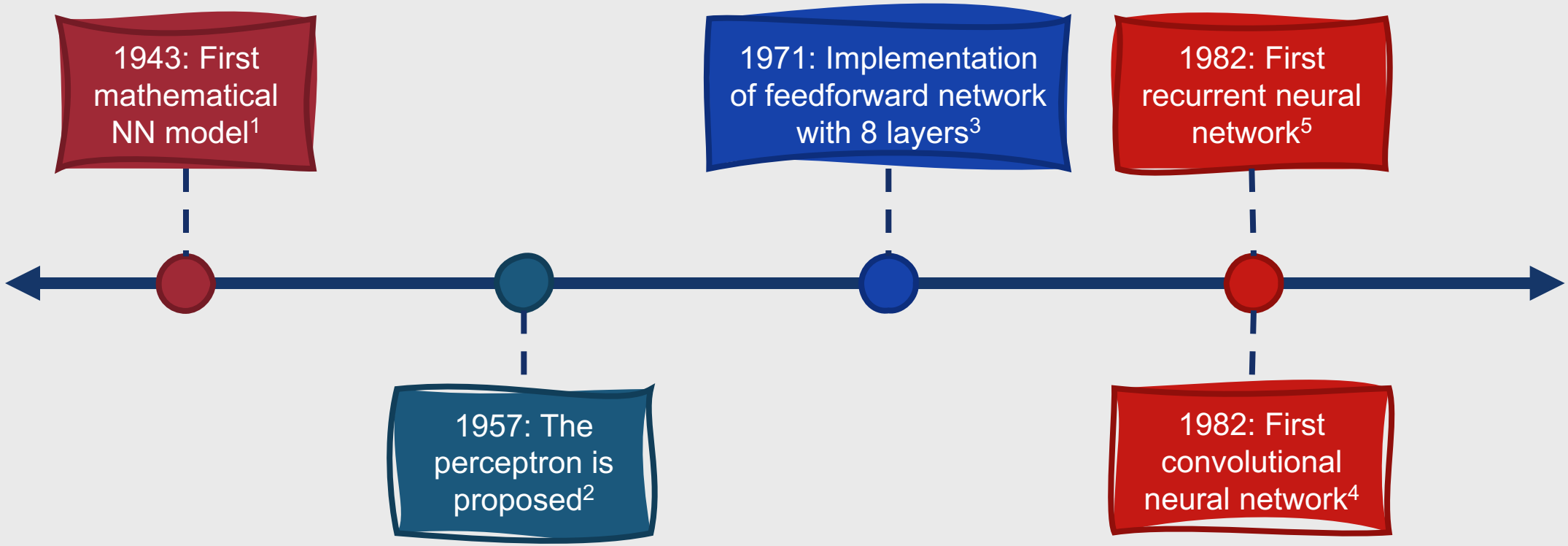
What are neural networks?

- Classification models comprised of interconnected computing units, or **neurons**, (loosely!) mirroring the interconnected neurons in the human brain

Neural networks are an increasingly fundamental tool for natural language processing.

ACL Year	# Paper Titles with "Neural"	% Paper Titles with "Neural"
2000	0	0
2001	0	0
2002	0	0
2003	0	0
2004	1	1/137 = 0.7%
2005	0	0
2006	0	0
2007	1	1/207 = 0.5%
2008	0	0
2009	1	1/248 = 0.4%
2010	0	0
2011	0	0
2012	0	0
2013	5	5/399 = 1.3%
2014	11	11/333 = 3.3%
2015	36	36/363 = 9.9%
2016	49	49/390 = 12.6%
2017	81	81/357 = 22.7%
2018	138	138/674 = 20.5%
2019	197	197/1449 = 13.6%

Are neural networks new?



[1]McCulloch, W. S., and W. Pitts. "A logical calculus of the ideas immanent in nervous activity." *The bulletin of mathematical biophysics* 5.4 (1943): 115-133.

[2]Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory.

[3]Ivakhnenko, A. G. (1971). Polynomial theory of complex systems. *IEEE transactions on Systems, Man, and Cybernetics*, (4), 364-378.

[4]Fukushima, K., & Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets* (pp. 267-285). Springer, Berlin, Heidelberg.

[5]Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, *79*(8), 2554-2558.

Why haven’t they been a big deal until recently then?

- Data
- Computing power

Neural networks are everywhere!

Augmenting Neural Networks with First-order Logic

Tao Li
University of Utah
tli@cs.utah.edu

Vivek Srikumar
University of Utah
svivek@cs.utah.edu

Abstract

Today, the dominant paradigm for training neural networks involves minimizing task loss on a large dataset. Using world knowledge to inform a model, and yet retain the ability to perform end-to-end training remains an open question. In this paper, we present a novel framework for introducing declarative knowledge to neural network architectures in order to guide training and prediction. Our framework systematically compiles logical statements into computation graphs that augment

Paragraph: Gaius Julius Caesar (July 100 BC – 15 March 44 BC), Roman general, statesman, Consul and notable writer of Latin prose, played a critical role in the events that led to the demise of the Roman Republic and the rise of the Roman Empire through his various military campaigns.

Question: Which Roman general is known for writing prose?

Figure 1: An example of reading comprehension that illustrates alignments/attention. In this paper, we consider the problem of incorporating external knowledge about such alignments into training neural networks.

Neural Relation Extraction for Knowledge Base Enrichment

Bayu Distiawan Trisedya[1], Gerhard Weikum[2], Jianzhong Qi[1], Rui Zhang[1]*
[1] The University of Melbourne, Australia
[2] Max Planck Institute for Informatics, Saarland Informatics Campus, Germany
{btrisedya@student, jianzhong.qi@, rui.zhang@}unimelb.edu.au
weikum@mpi-inf.mpg.de

Abstract

We study relation extraction for knowledge base (KB) enrichment. Specifically, we aim to extract entities and their relationships from sentences in the form of triples and map the elements of the extracted triples to an existing KB in an end-to-end manner. Previous studies focus on the extraction itself and rely on Named Entity Disambiguation (NED) to map triples into the KB space. This way, NED errors may cause extraction errors that affect the overall precision and recall. To address this

Input sentence:
"New York University is a private university in Manhattan."
Unsupervised approach output:
⟨NYU, is, private university⟩
⟨NYU, is private university in, Manhattan⟩
Supervised approach output:
⟨NYU, instance of, Private University⟩
⟨NYU, located in, Manhattan⟩
Canonicalized output:
⟨Q49210, P31, Q902104⟩
⟨Q49210, P131, Q11299⟩

Table 1: Relation extraction example.

Cross-Domain Generalization of Neural Constituency Parsers

Daniel Fried* **Nikita Kitaev*** **Dan Klein**
Computer Science Division
University of California, Berkeley
{dfried,kitaev,klein}@cs.berkeley.edu

Abstract

Neural parsers obtain state-of-the-art results on benchmark treebanks for constituency parsing—but to what degree do they generalize to other domains? We present three results about the generalization of neural parsers in a zero-shot setting: training on trees from one corpus and evaluating on out-of-domain corpora. First, neural and non-neural parsers generalize comparably to new domains. Second, incorporating pre-trained encoder representations into neural parsers substantially improves their performance across all domains, but does not give a larger relative improvement for out-of-domain treebanks. Finally, despite the rich input representations they learn, neural parsers still benefit from structured output

treebanks still transfer to out-of-domain improvements (McClosky et al., 2006).

Is the success of neural constituency parsers (Henderson 2004; Vinyals et al. 2015; Dyer et al. 2016; Cross and Huang 2016; Choe and Charniak 2016; Stern et al. 2017; Liu and Zhang 2017; Kitaev and Klein 2018, *inter alia*) similarly transferable to out-of-domain treebanks? In this work, we focus on *zero-shot generalization*: training parsers on a single treebank (e.g. WSJ) and evaluating on a range of broad-coverage, out-of-domain treebanks (e.g. Brown (Francis and Kučera, 1979), Genia (Tateisi et al., 2005), the English Web Treebank (Petrov and McDonald, 2012)). We ask three questions about zero-shot generalization properties of state-of-the-art neural constituency parsers:

Do Neural Dialog Systems Use the Conversation History Effectively? An Empirical Study

Chinnadhurai Sankar[1,2,4]* **Sandeep Subramanian**[1,2,5]
Christopher Pal[1,3,5] **Sarath Chandar**[1,2,4] **Yoshua Bengio** [1,2]
[1]Mila [2]Université de Montréal [3]École Polytechnique de Montréal
[4]Google Research, Brain Team [5]Element AI, Montréal

Abstract

Neural generative models have been become increasingly popular when building conversational agents. They offer flexibility, can be easily adapted to new domains, and require minimal domain engineering. A common criticism of these systems is that they seldom understand or use the available dialog history effectively. In this paper, we take an empirical approach to understanding how these models use the available dialog history by study-

they still lack the ability to "understand" and process the dialog history to produce coherent and interesting responses. They often produce boring and repetitive responses like "Thank you." (Li et al., 2015; Serban et al., 2017a) or meander away from the topic of conversation. This has been often attributed to the manner and extent to which these models use the dialog history when generating responses. However, there has been little empirical investigation to validate these speculations.

Effective Adversarial Regularization for Neural Machine Translation

Motoki Sato[1], **Jun Suzuki**[2,3], **Shun Kiyono**[3,2]
[1]Preferred Networks, Inc., [2]Tohoku University,
[3]RIKEN Center for Advanced Intelligence Project
sato@preferred.jp, jun.suzuki@ecei.tohoku.ac.jp, shun.kiyono@riken.jp

Abstract

A regularization technique based on adversarial perturbation, which was initially developed in the field of image processing, has been successfully applied to text classification tasks and has yielded attractive improvements. We aim to further leverage this promising methodology into more sophisticated and critical neural models in the natural language processing field, i.e., neural machine translation (NMT) models. However, it is not trivial to apply this

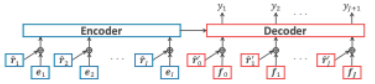


Figure 1: An intuitive sketch that explains how we add adversarial perturbations to a typical NMT model structure for adversarial regularization. The definitions of $\boldsymbol{e}_i$ and $\boldsymbol{f}_j$ can be found in Eq. 2. Moreover, those of $\hat{\boldsymbol{r}}_i$ and $\hat{\boldsymbol{r}}'_j$ are in Eq. 8 and 13, respectively.

Neural Network Basics

- Neural networks are comprised of small computing **units**
- Each computing unit takes a **vector of input values**
- Each computing unit produces a **single output value**
- Many different types of neural networks exist

Types of Neural Networks

- Feedforward Neural Network
- Convolutional Neural Network
- Recurrent Neural Network
- Generative Adversarial Network
- Sequence-to-Sequence Network
- Autoencoder
- Transformer

Types of Neural Networks

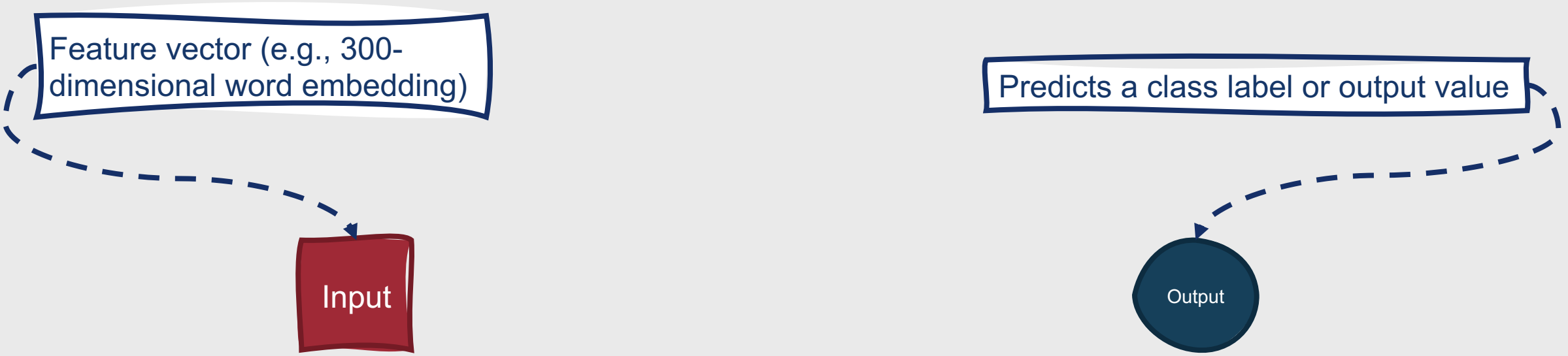
- Feedforward Neural Network
- Convolutional Neural Network
- Recurrent Neural Network
- Generative Adversarial Network
- Sequence-to-Sequence Network
- Autoencoder
- Transformer

Today's lecture!

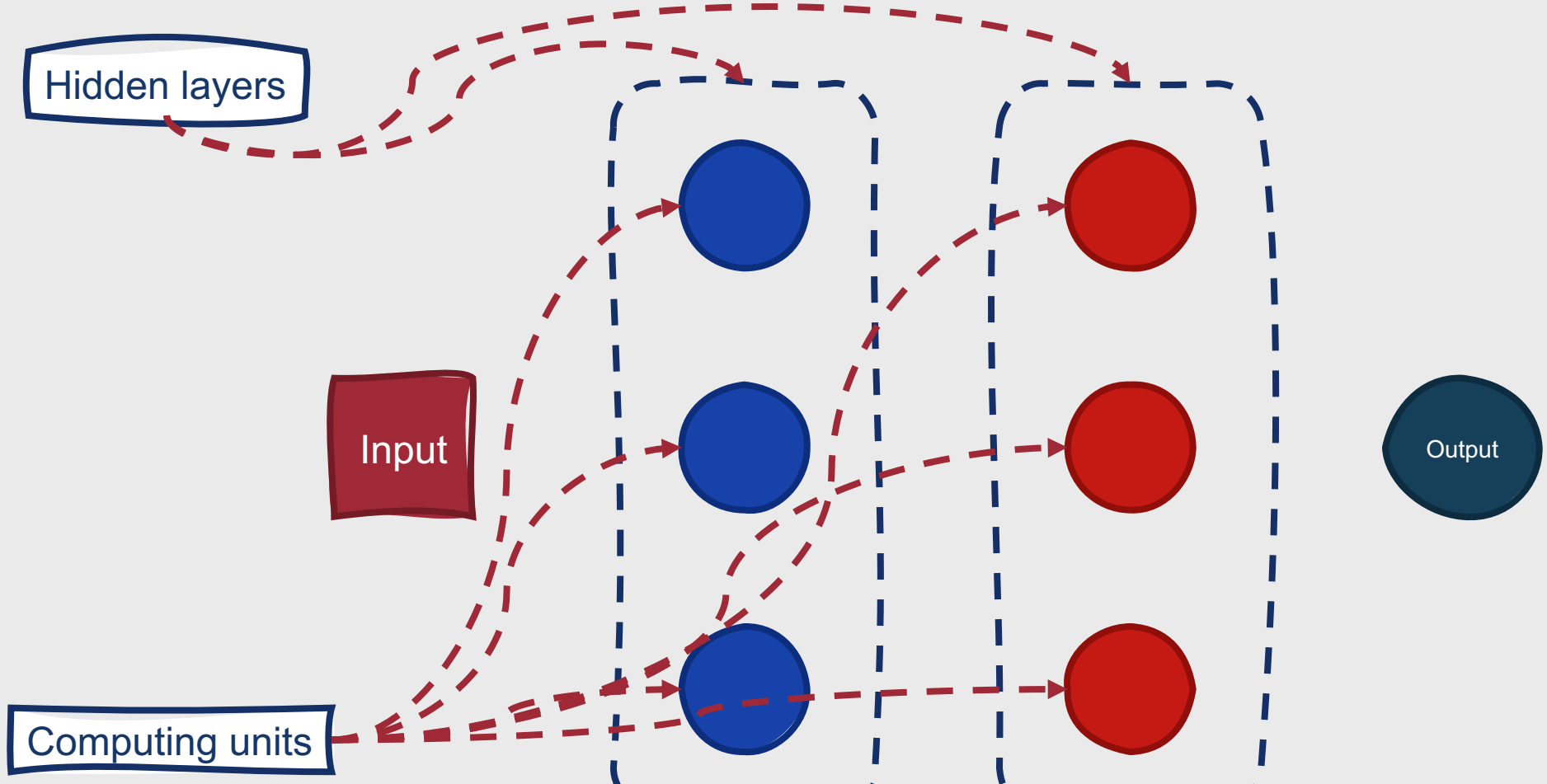
Feedforward Neural Networks

- Earliest and simplest form of neural network
- Data is fed forward from one layer to the next
- Each layer:
  - One or more units
  - A unit in layer *n* receives input from all units in layer *n*-1 and sends output to all units in layer *n*+1
  - A unit in layer *n* does not communicate with any other units in layer *n*
- The outputs of all units except for those in the last layer are **hidden** from external viewers

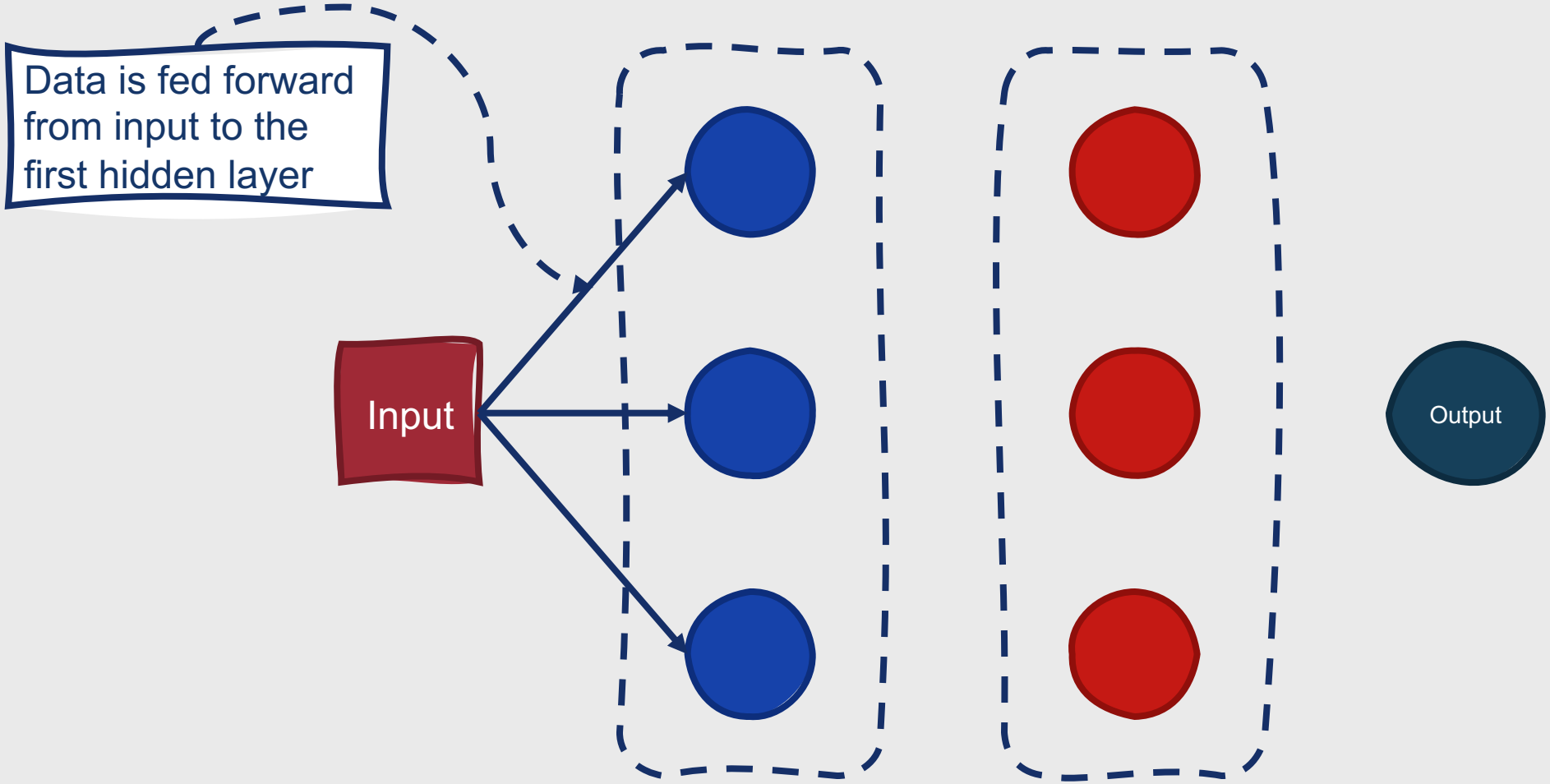
Feedforward Neural Networks



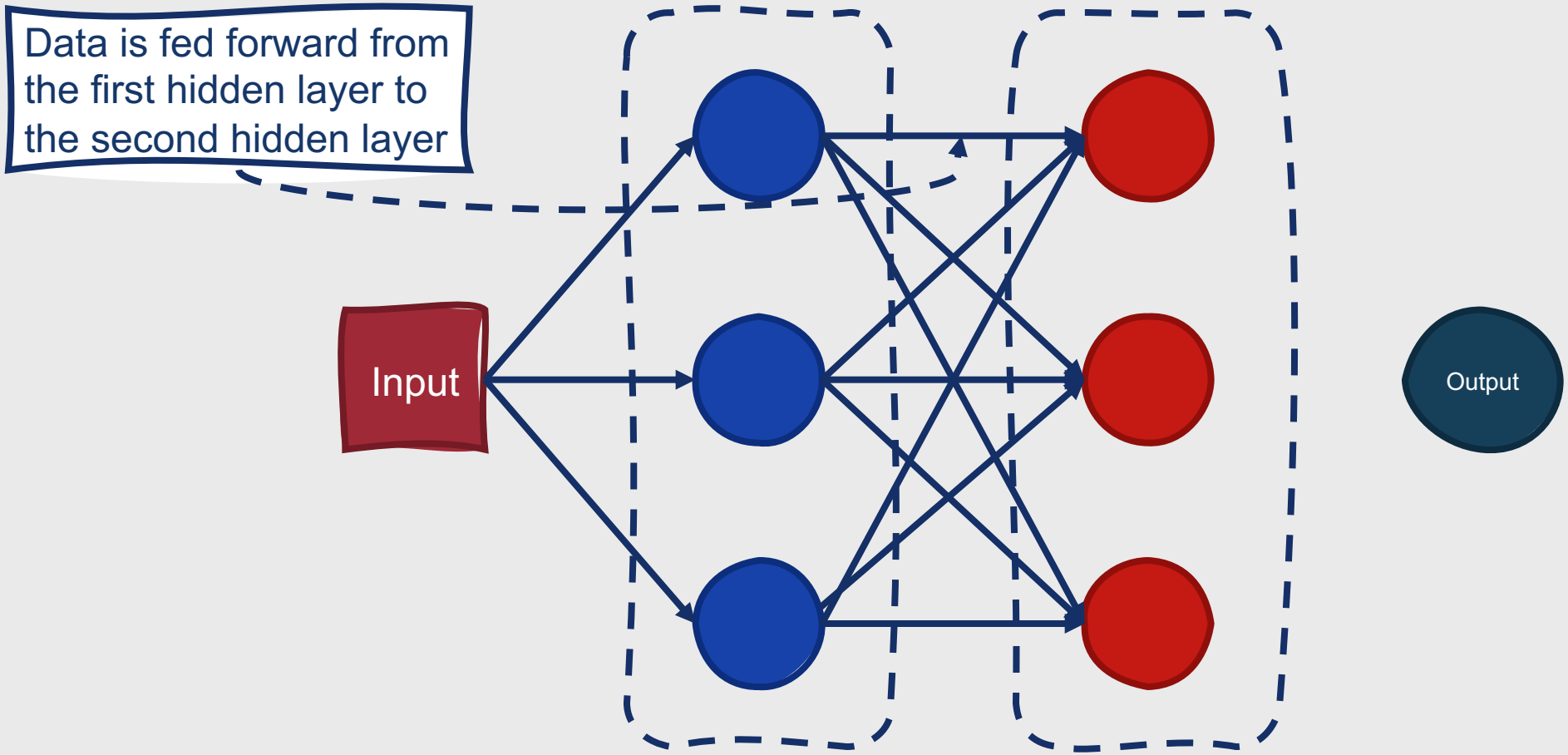
Feedforward Neural Networks



Feedforward Neural Networks

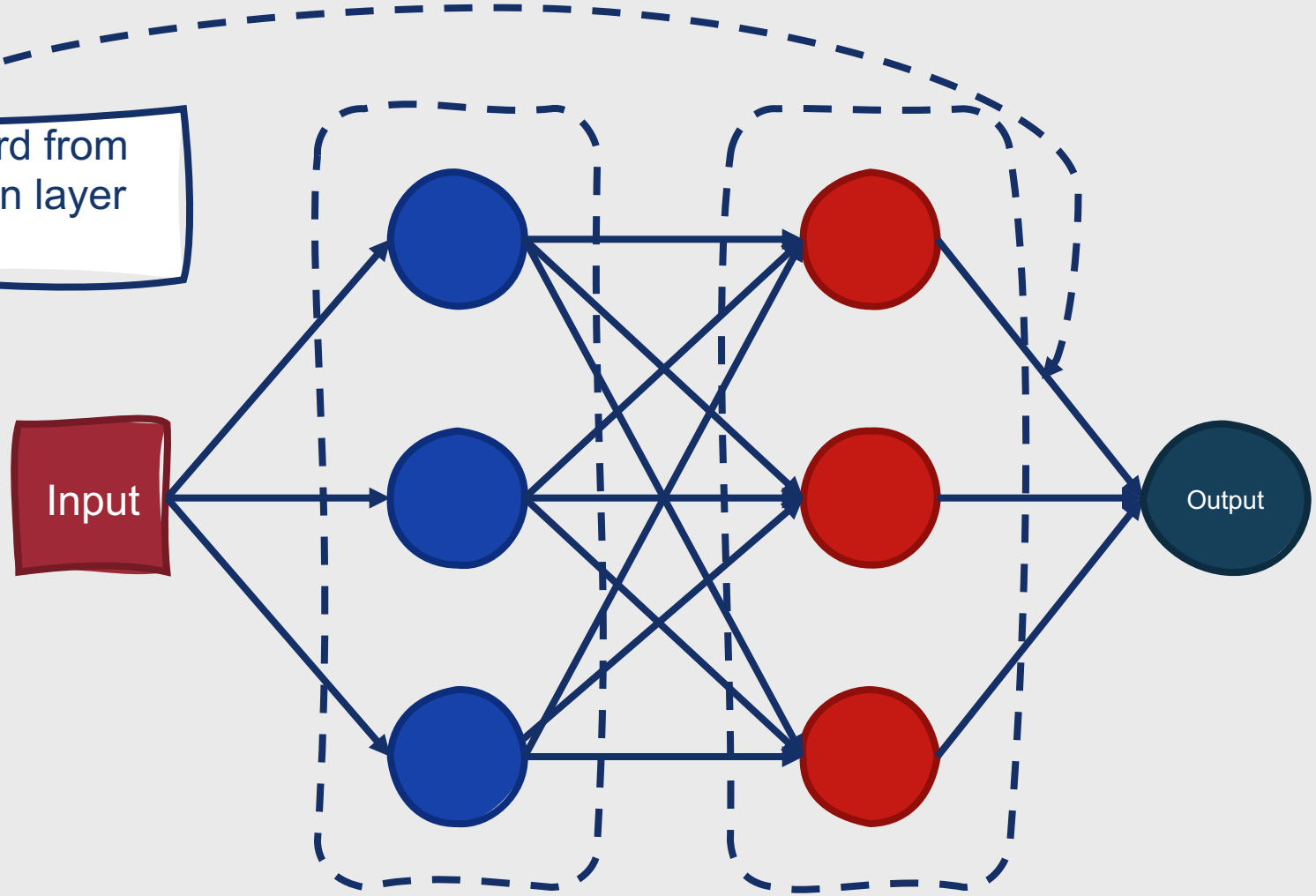


Feedforward Neural Networks

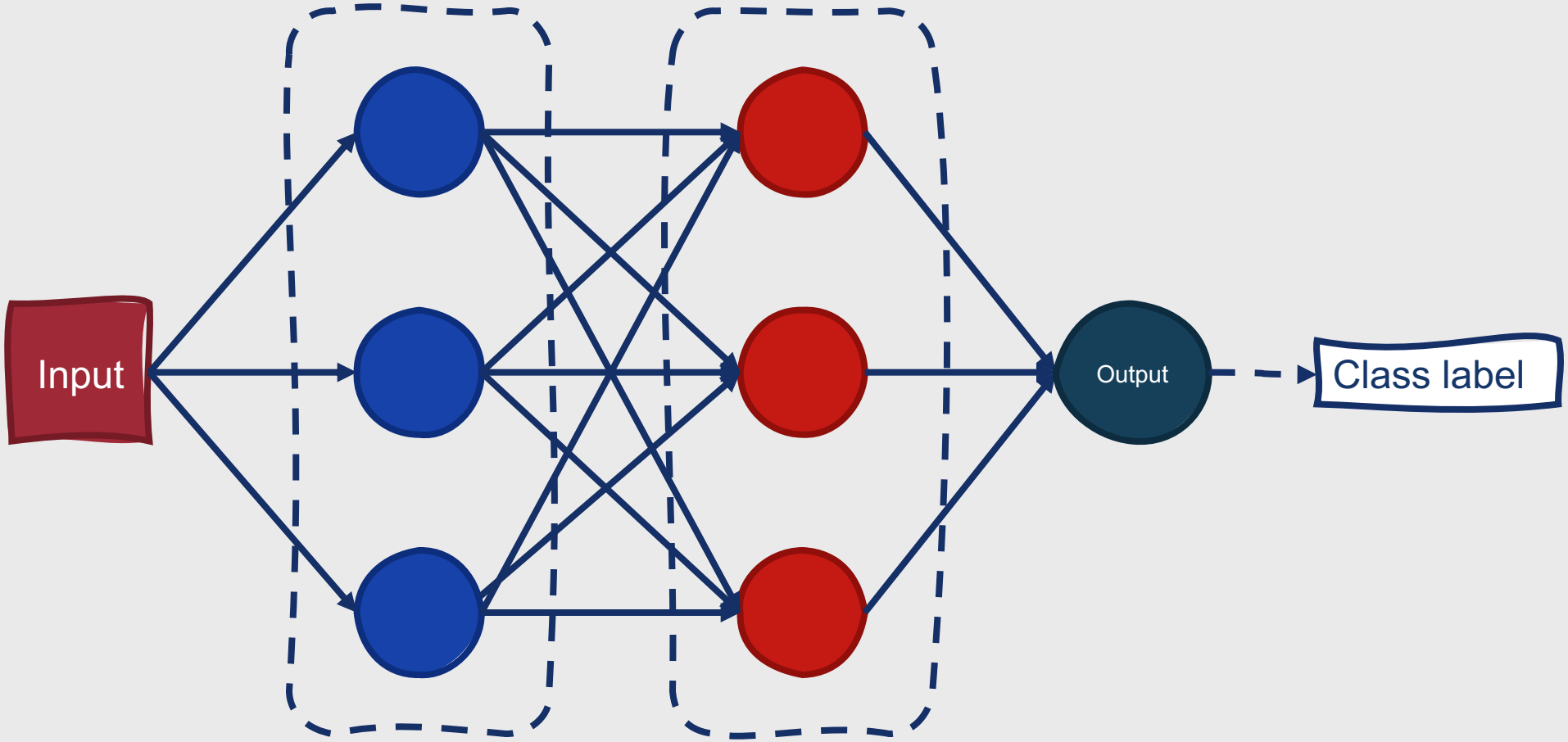


Feedforward Neural Networks

Data is fed forward from the second hidden layer to the output unit



Feedforward Neural Networks



Are feedforward neural networks an example of deep learning?

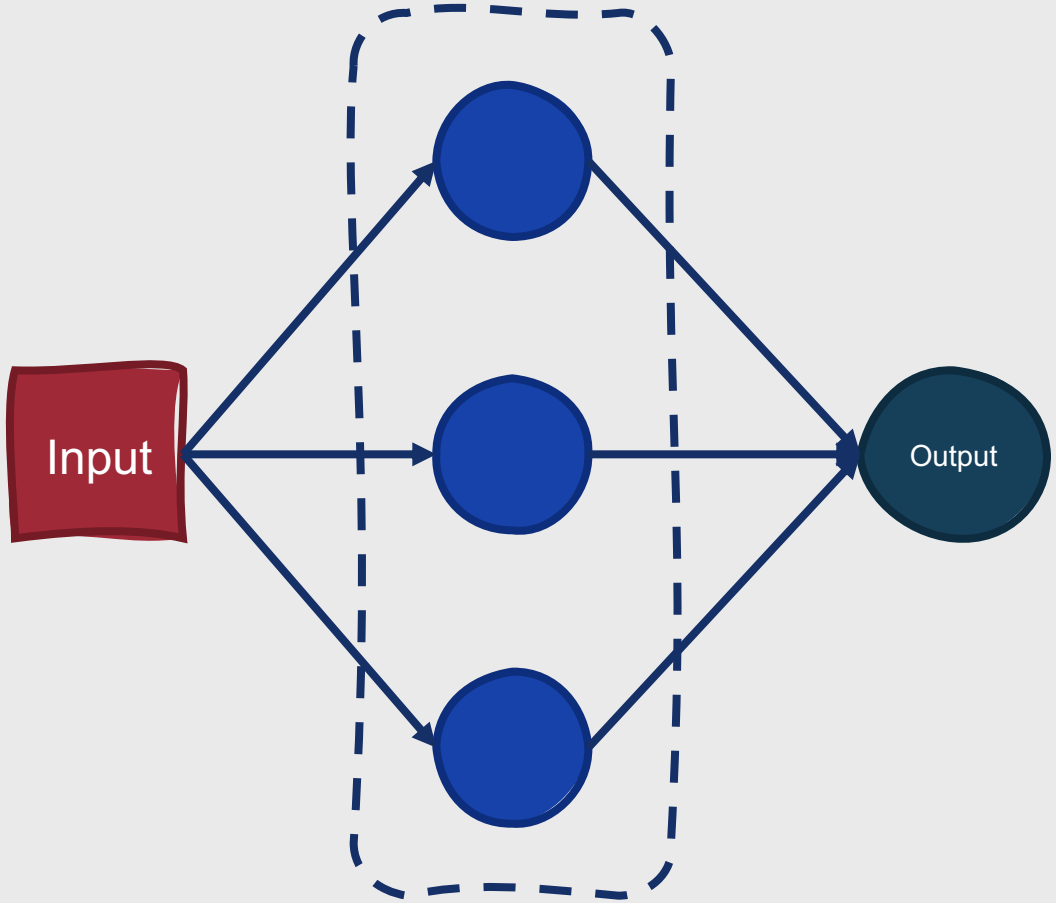
Yes ...if they have multiple layers

People often tend to refer to neural network-based machine learning as **deep learning**

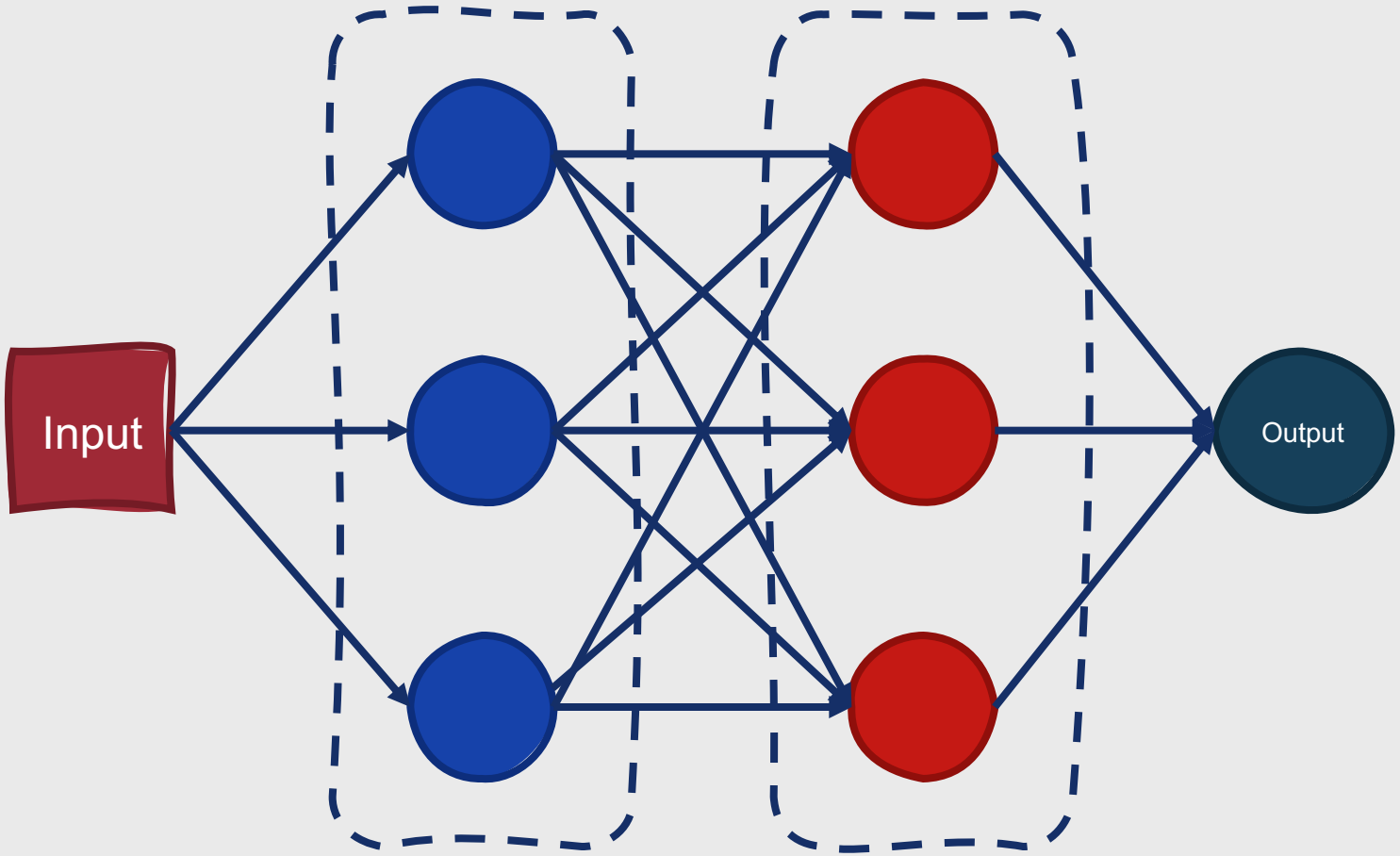
Why?

- Modern networks often have many layers (in other words, they're **deep**)

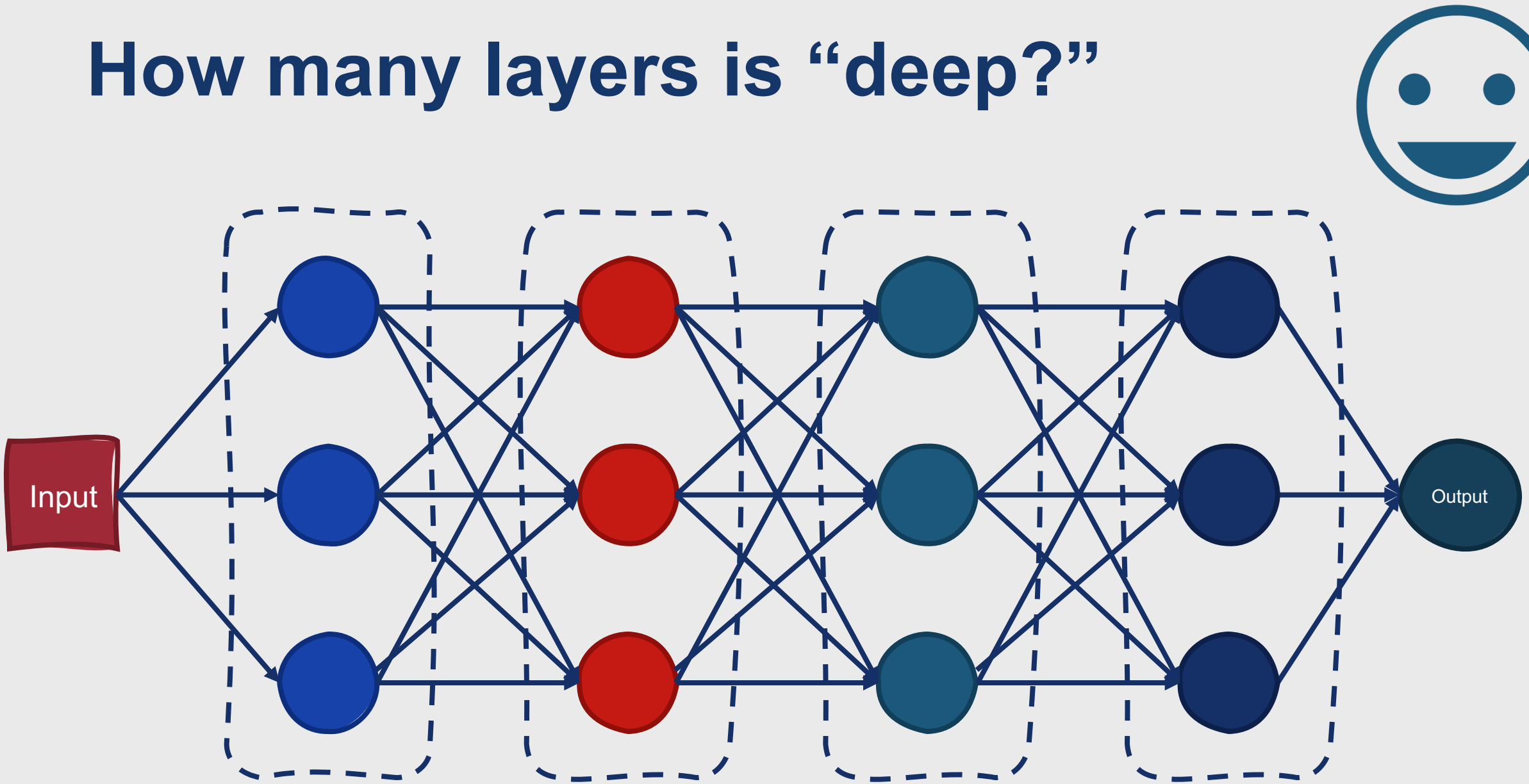
How many layers is “deep?”



How many layers is “deep?”

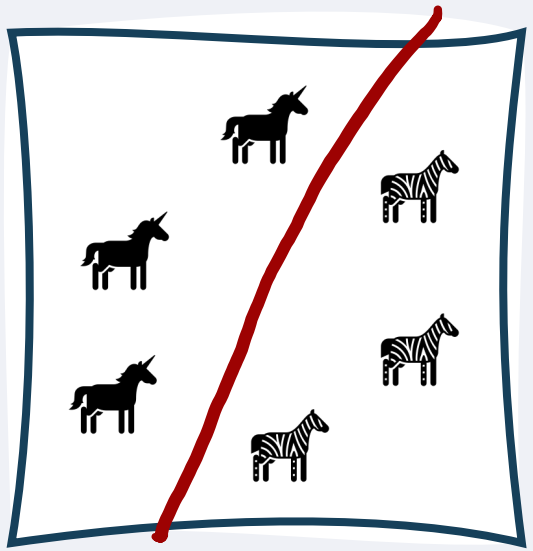


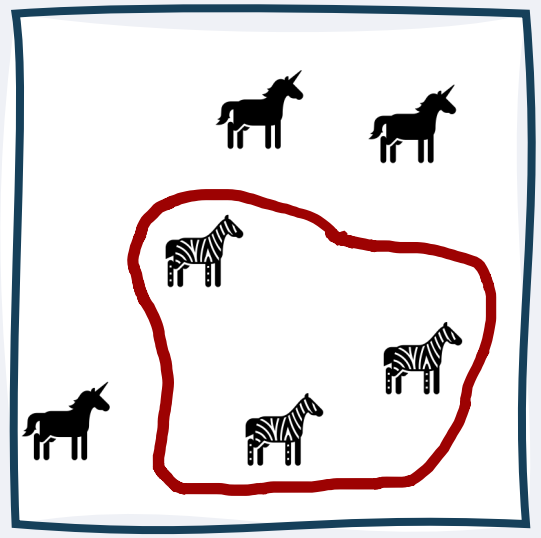
How many layers is “deep?”



Neural networks tend to be more powerful than traditional classification algorithms.

- Traditional classification algorithms usually assume that data is **linearly separable**
- In contrast, neural networks learn **nonlinear functions**





Neural networks also commonly use different types of features from traditional classification algorithms.

Traditional classification

- **Manually engineer** a set of features and extract them for each instance
  - Part-of-speech label
  - Number of exclamation marks
  - Sentiment score

Neural networks

- **Implicitly learn** features and extract those for each instance
  - Word embeddings

Neural networks aren't necessarily the best classifier for all tasks!

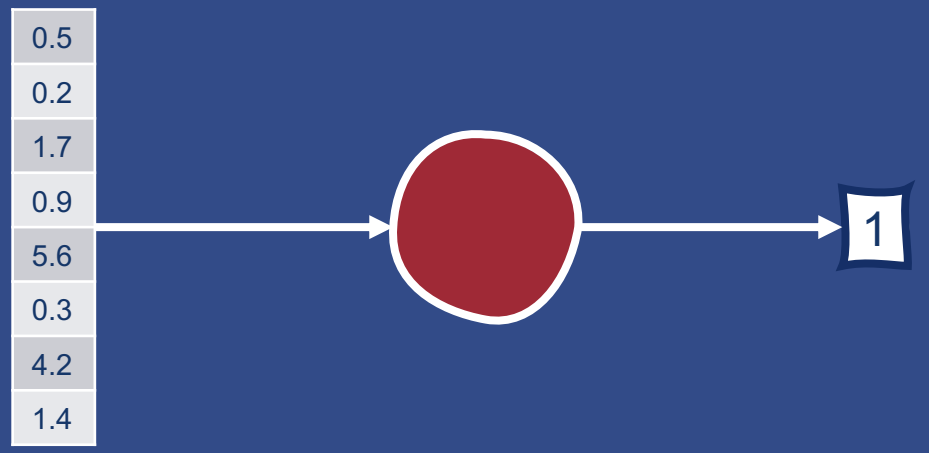
Learning features **implicitly** requires a lot of data

In general, deeper network → more data needed

Thus, neural nets tend to work very well for large-scale problems, but not that well for small-scale problems

Building Blocks for Neural Networks

- At their core, neural networks are comprised of **computational units**
- Computational units:
  1. Take a set of real-valued numbers as input
  2. Perform some computation on them
  3. Produce a single output



Computational Units

- The computation performed by each unit is a weighted sum of inputs
  - Assign a weight to each input
  - Add one additional bias term
- More formally, given a set of inputs $x_1, \ldots, x_n$, a unit has a set of corresponding weights $w_1, \ldots, w_n$ and a bias b, so the weighted sum z can be represented as:
  - $z = b + \sum_i w_i x_i$

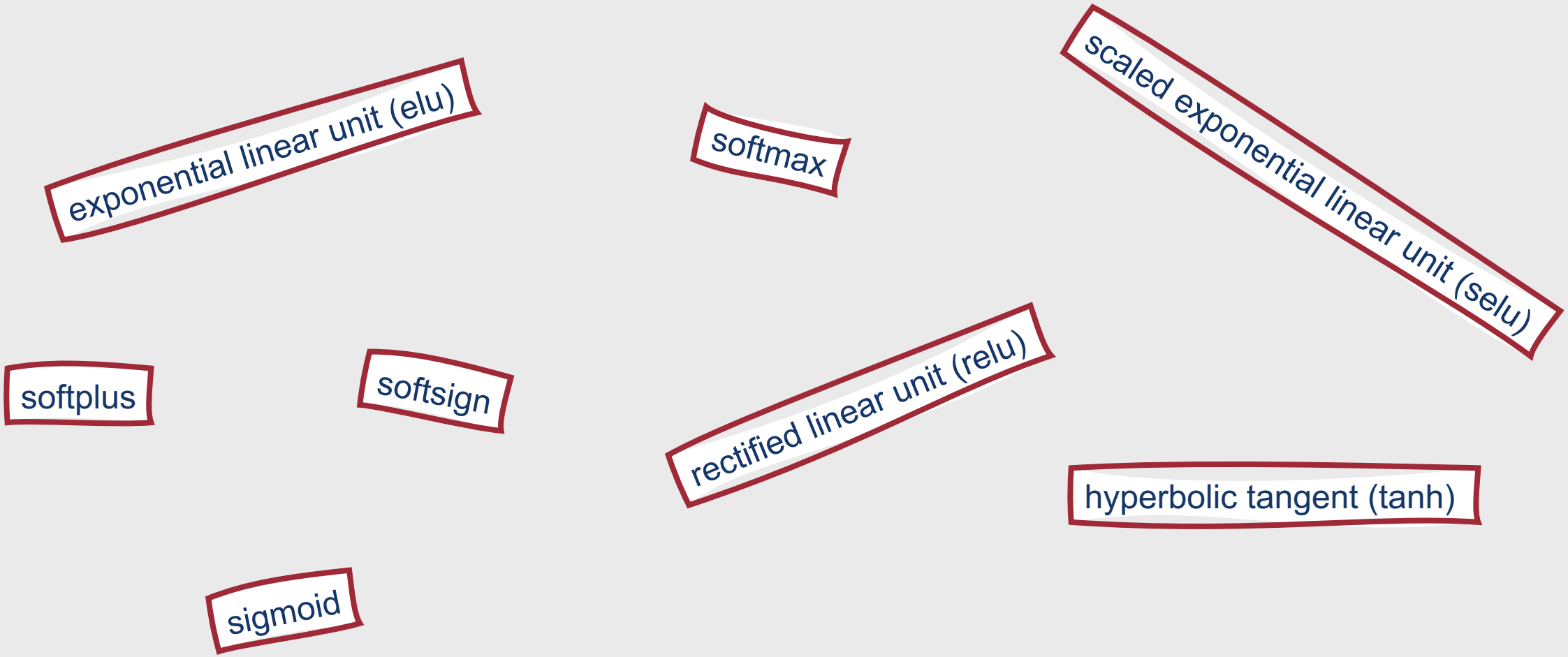
Sound familiar?

- This is exactly the same sort of weighted sum of inputs that we needed to find with logistic regression!
- Recall that we can also represent the weighted sum z using vector notation:
  - $z = w \cdot x + b$

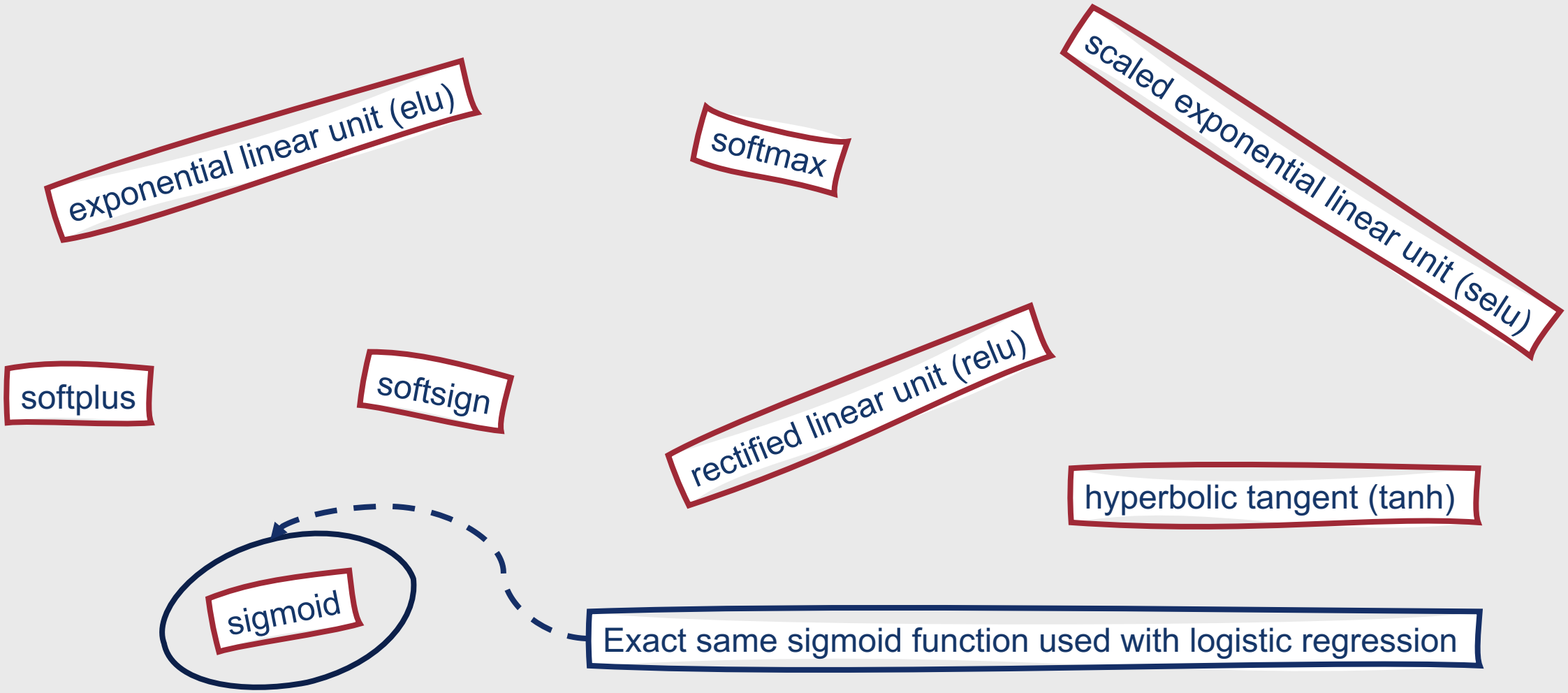
Computational Units

- The weighted sum of inputs computes a **linear function** of x
- As we already saw, neural networks learn **nonlinear functions**
- These nonlinear functions are commonly referred to as **activations**
- The output of a computation unit is thus the **activation value** for the unit, y
  - $y = f(z) = f(w \cdot x + b)$

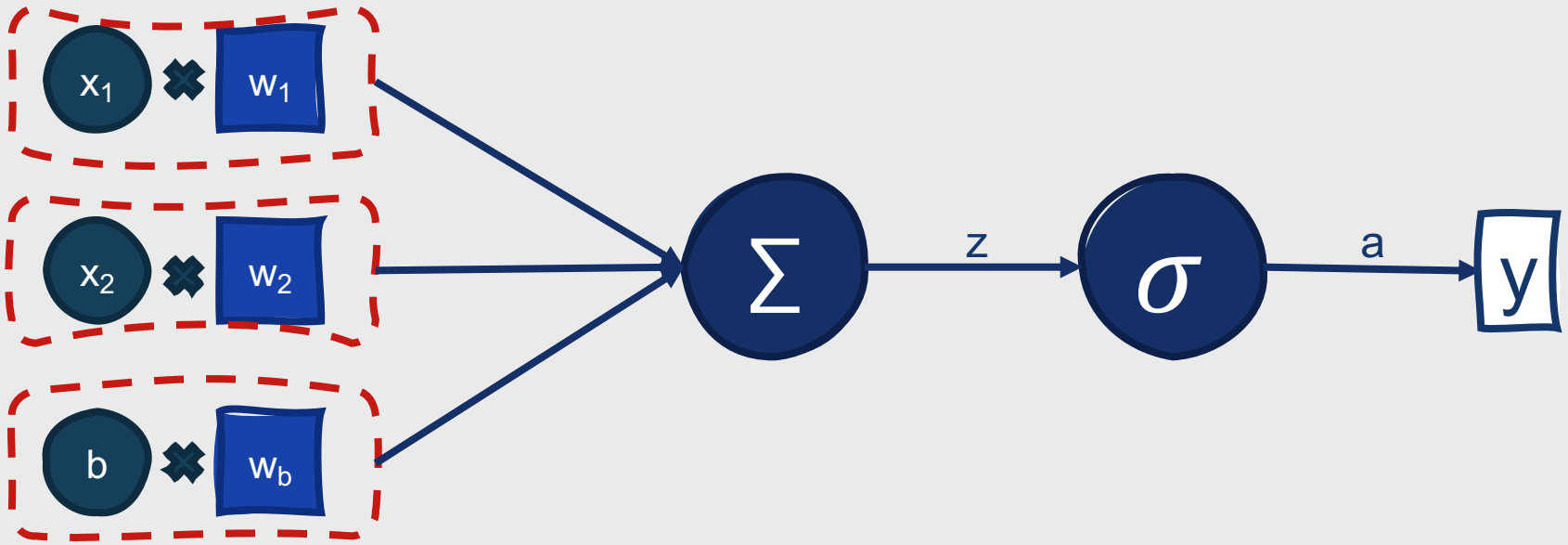
There are many different activation functions!



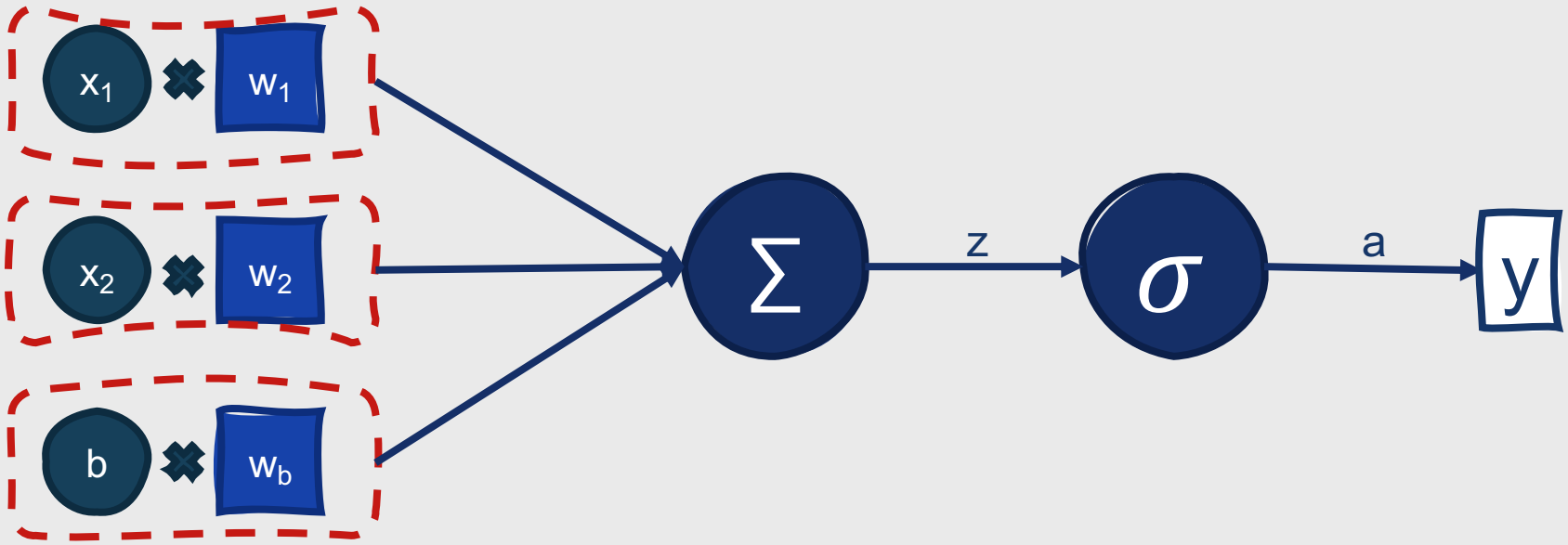
There are many different activation functions!



Computational Unit with Sigmoid Activation



Example: Computational Unit with Sigmoid Activation

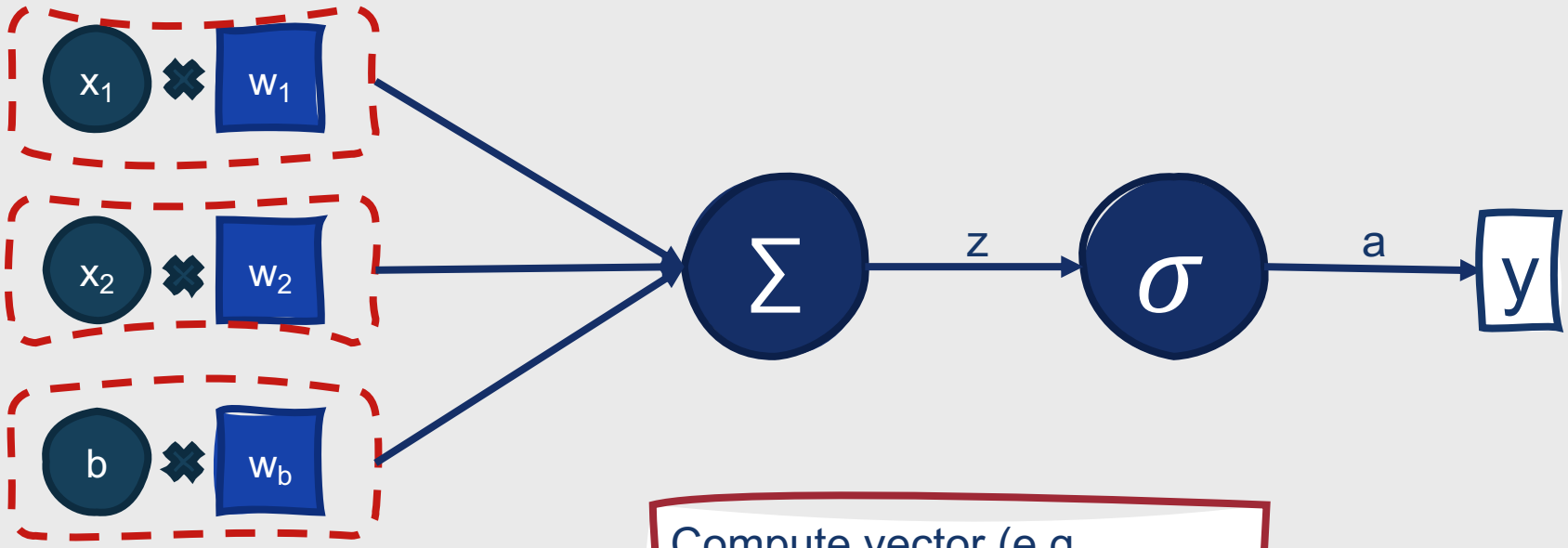


Input: “beautiful brutalist architecture”

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with Sigmoid Activation



Input: “beautiful brutalist architecture”

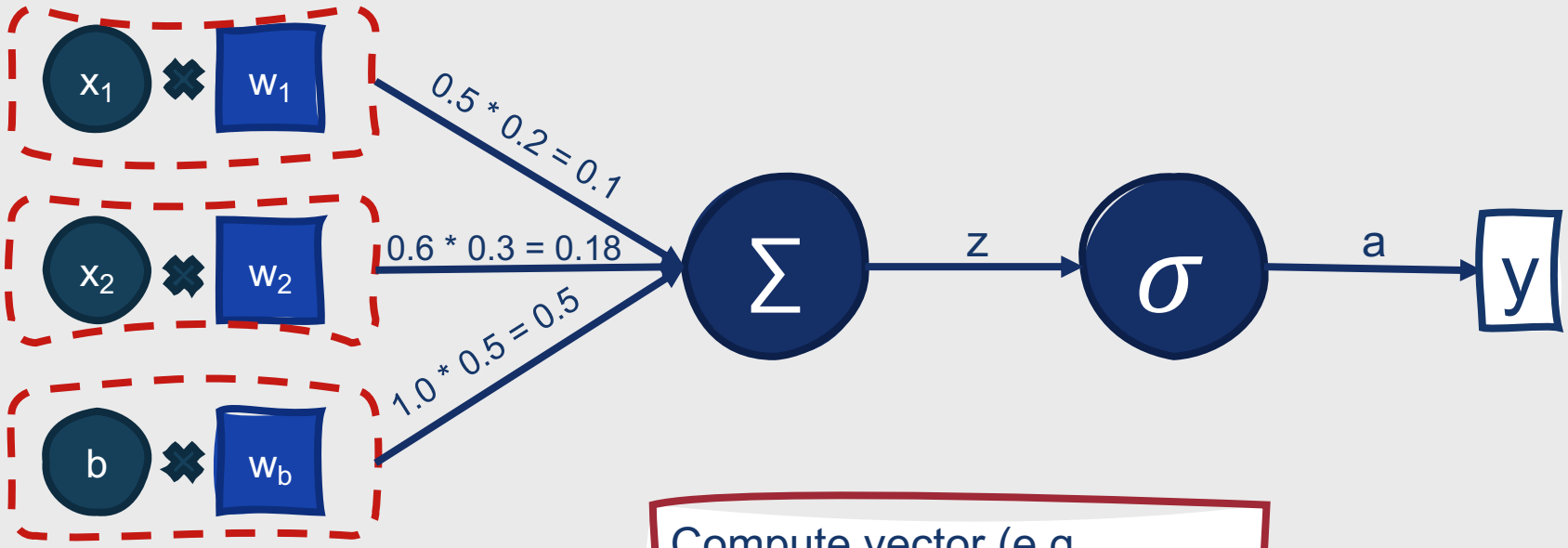
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with Sigmoid Activation



Input: “beautiful brutalist architecture”

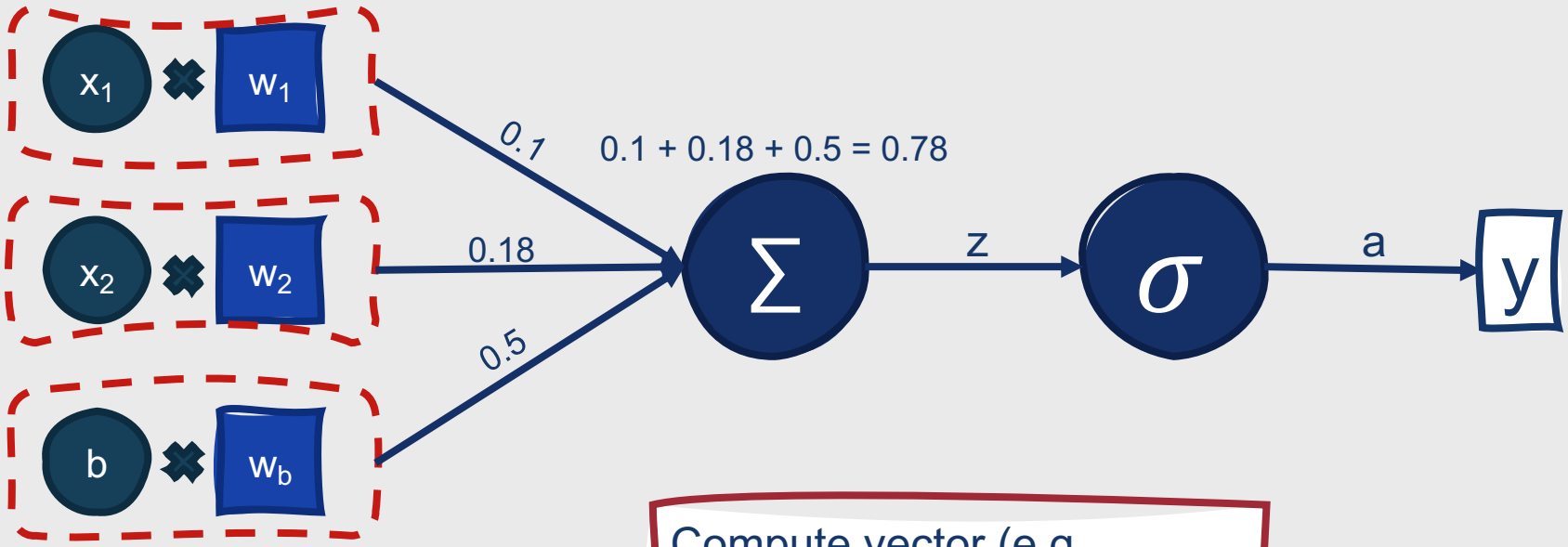
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with Sigmoid Activation



Input: “beautiful brutalist architecture”

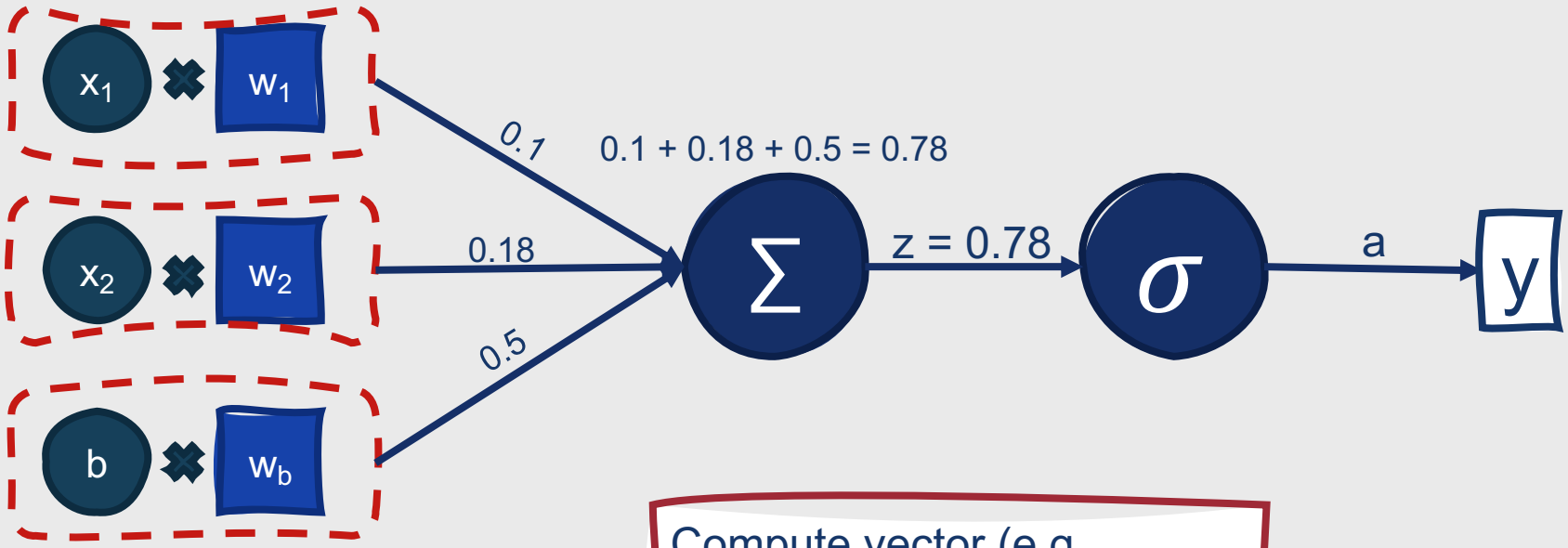
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with Sigmoid Activation



Input: “beautiful brutalist architecture”

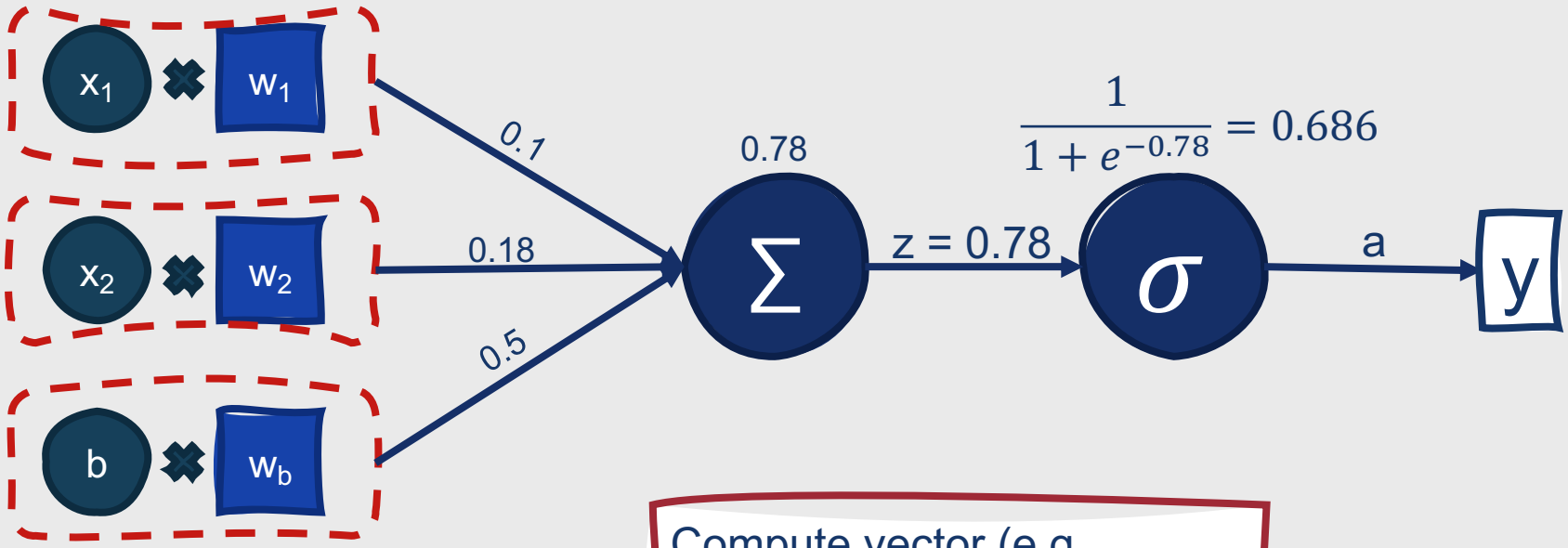
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with Sigmoid Activation



Input: “beautiful brutalist architecture”

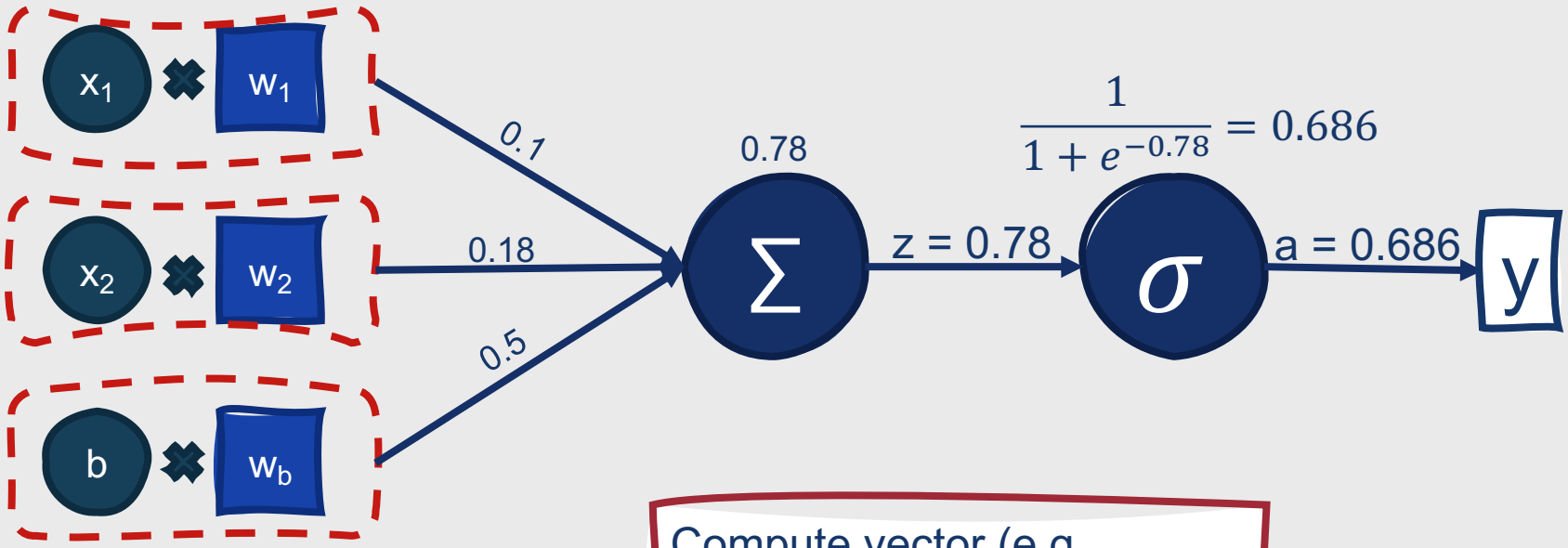
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with Sigmoid Activation



Input: “beautiful brutalist architecture”

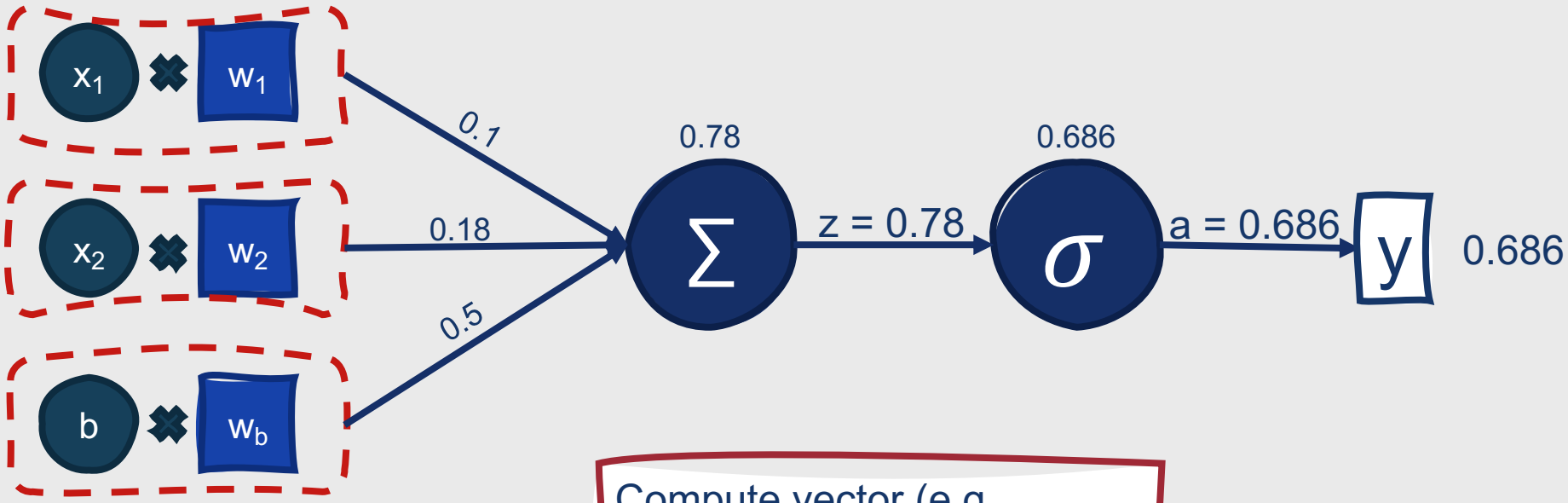
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with Sigmoid Activation



Input: “beautiful brutalist architecture”

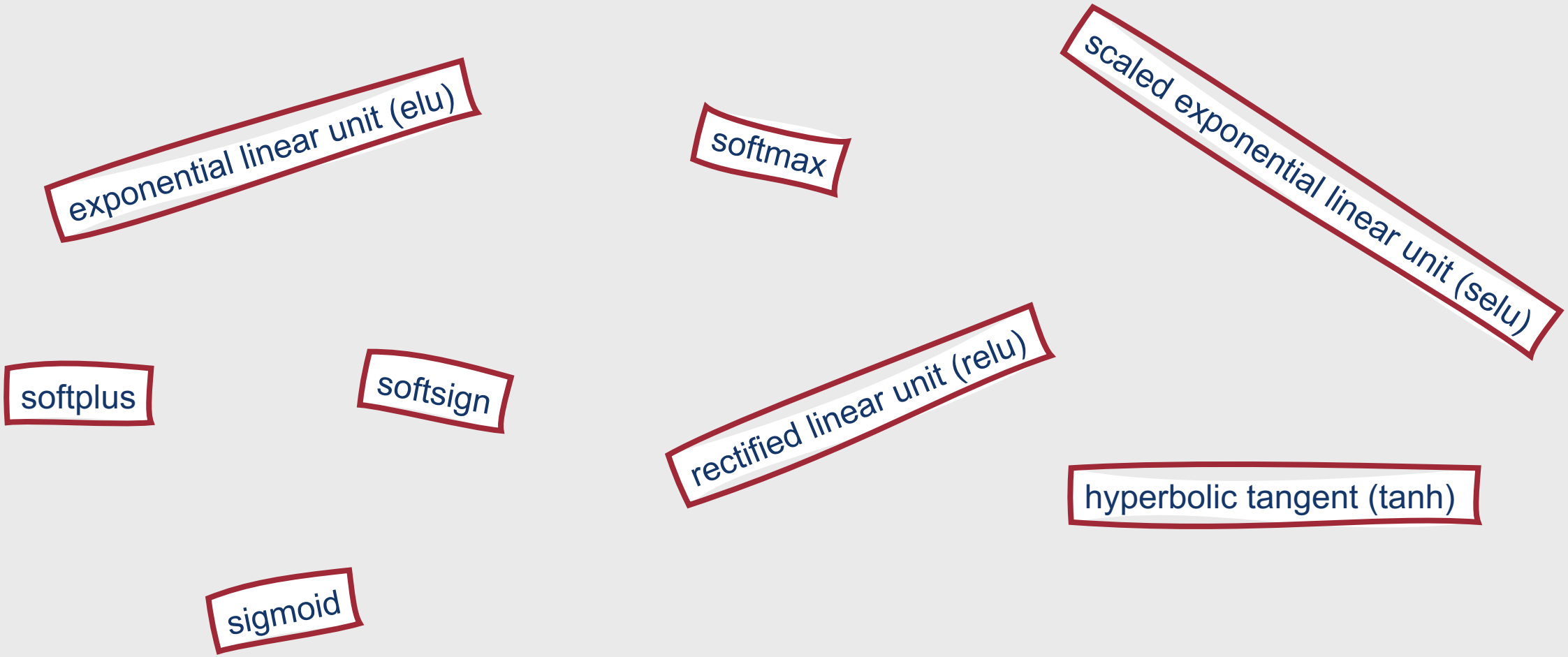
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

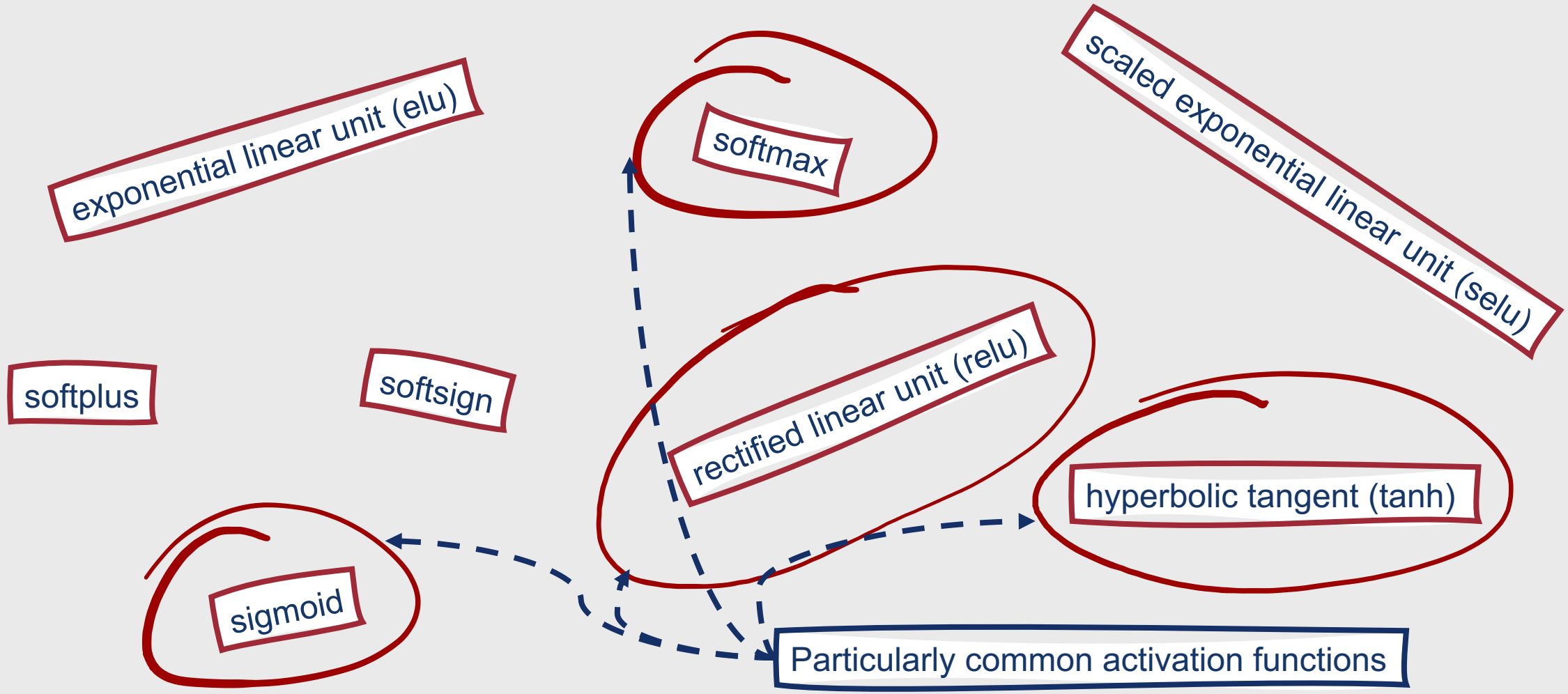
Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Remember, there are many different activation functions!



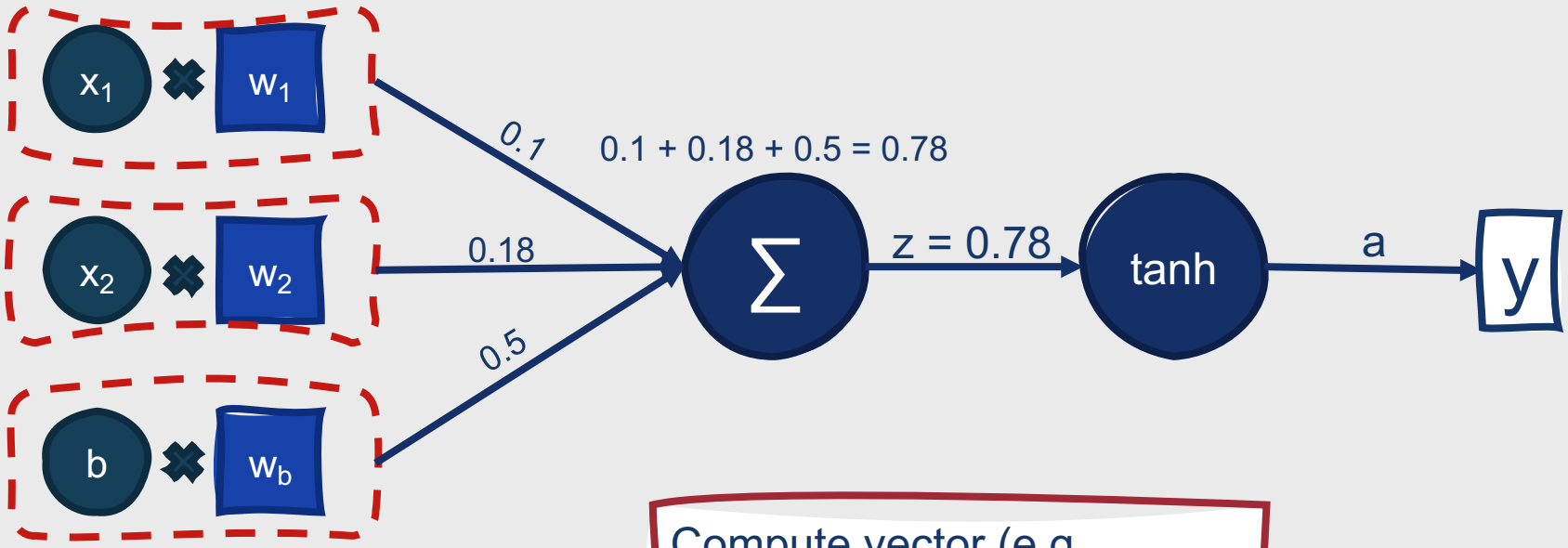
Remember, there are many different activation functions!



Activation: tanh

- Variant of sigmoid that ranges from -1 to +1
  - $y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- Once again differentiable
- Larger derivatives → generally faster convergence

Example: Computational Unit with tanh Activation



Input: “beautiful brutalist architecture”

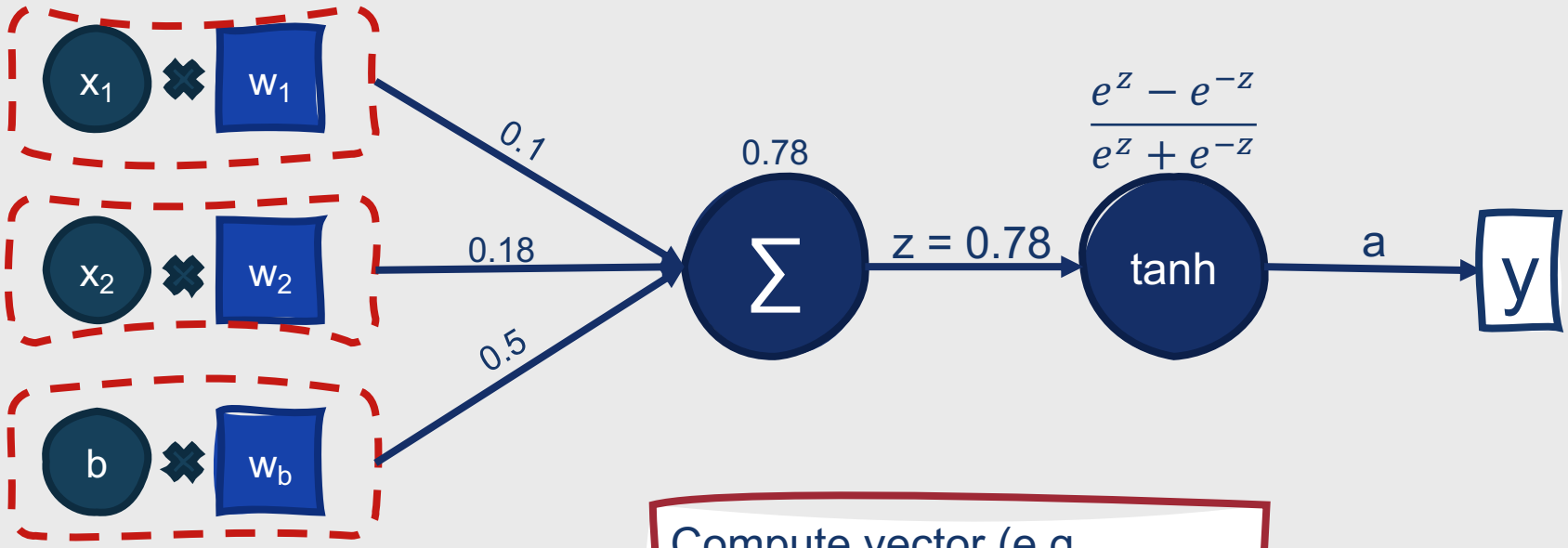
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with tanh Activation



Input: “beautiful brutalist architecture”

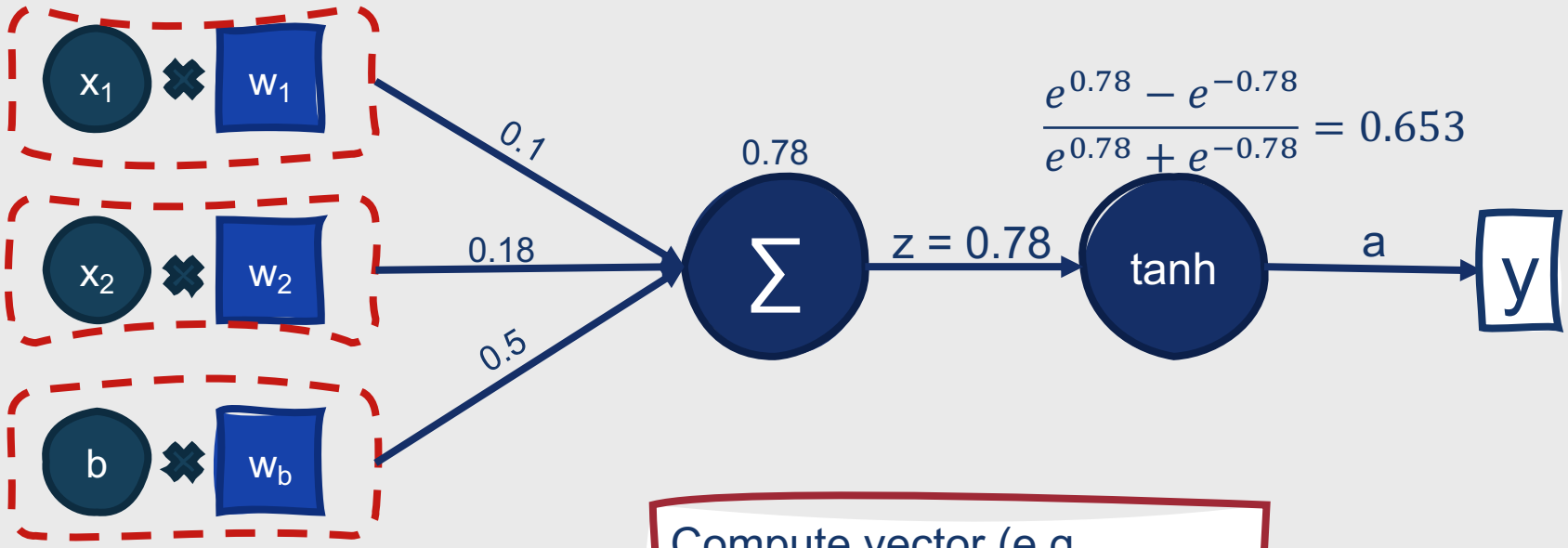
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with tanh Activation



Input: “beautiful brutalist architecture”

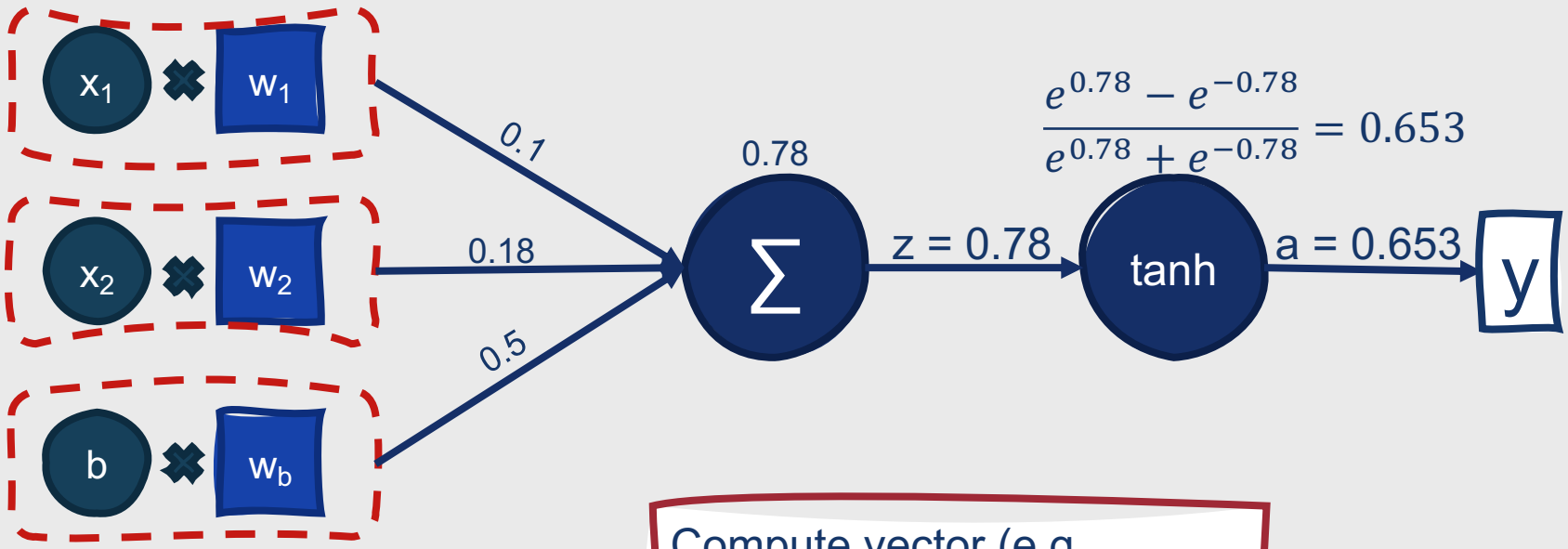
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with tanh Activation



Input: “beautiful brutalist architecture”

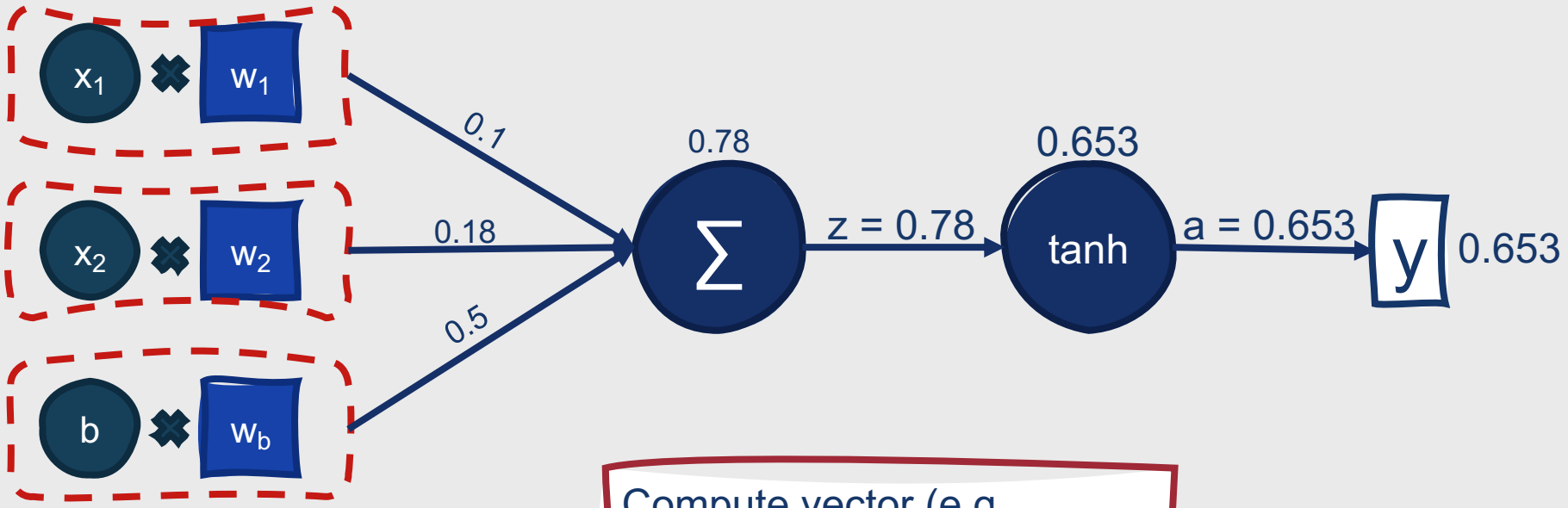
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with tanh Activation



Input: “beautiful brutalist architecture”

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

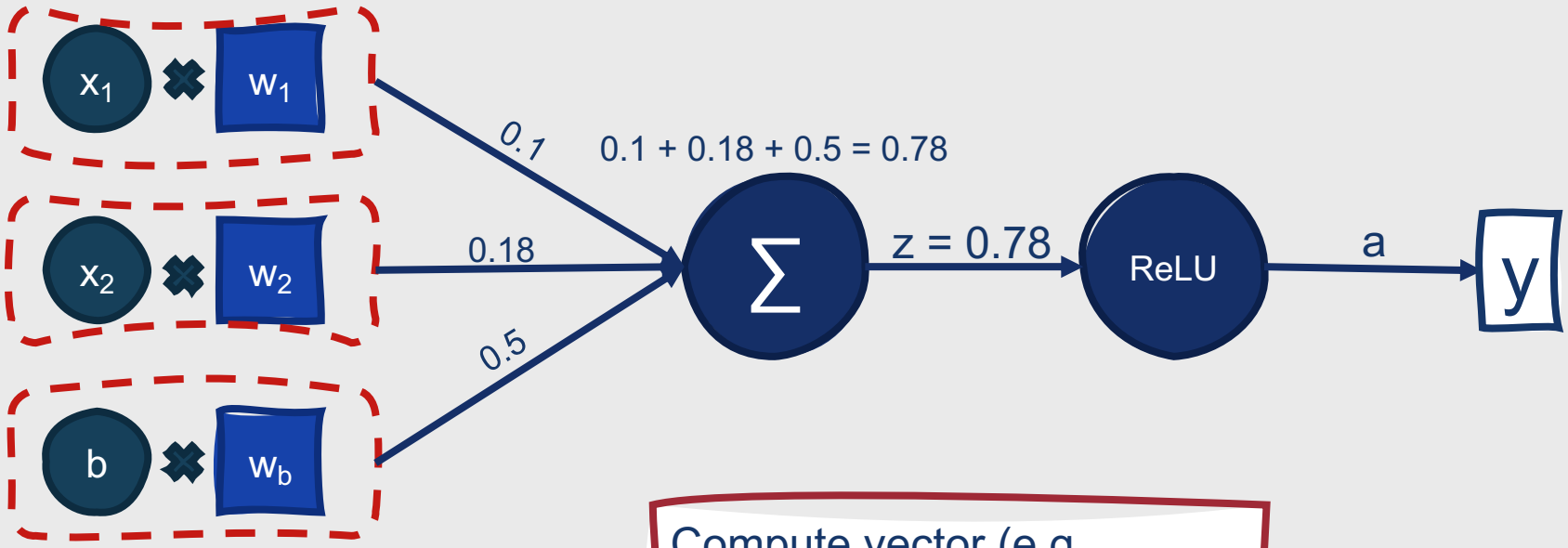
Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Activation: ReLU

- Ranges from 0 to ∞
- Simplest activation function:
  - $y = \max(z, 0)$
- Very close to a linear function!
- Quick and easy to compute

Example: Computational Unit with ReLU Activation



Input: “beautiful brutalist architecture”

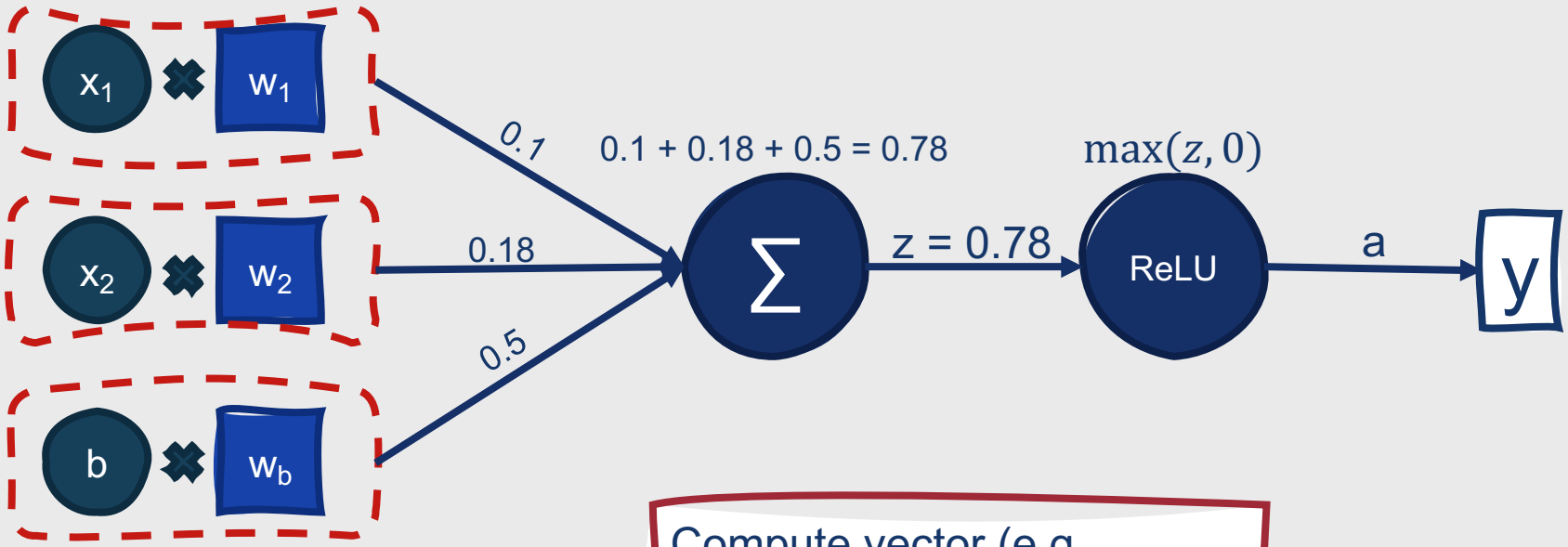
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with ReLU Activation



Input: “beautiful brutalist architecture”

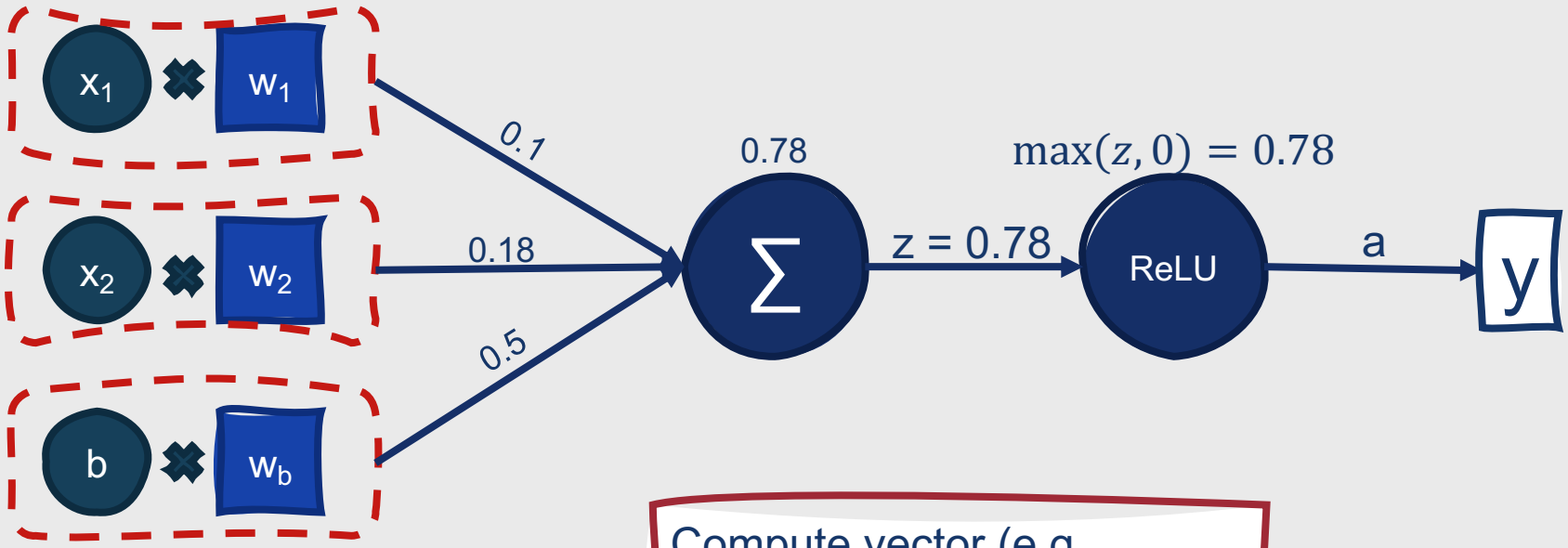
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with ReLU Activation



Input: “beautiful brutalist architecture”

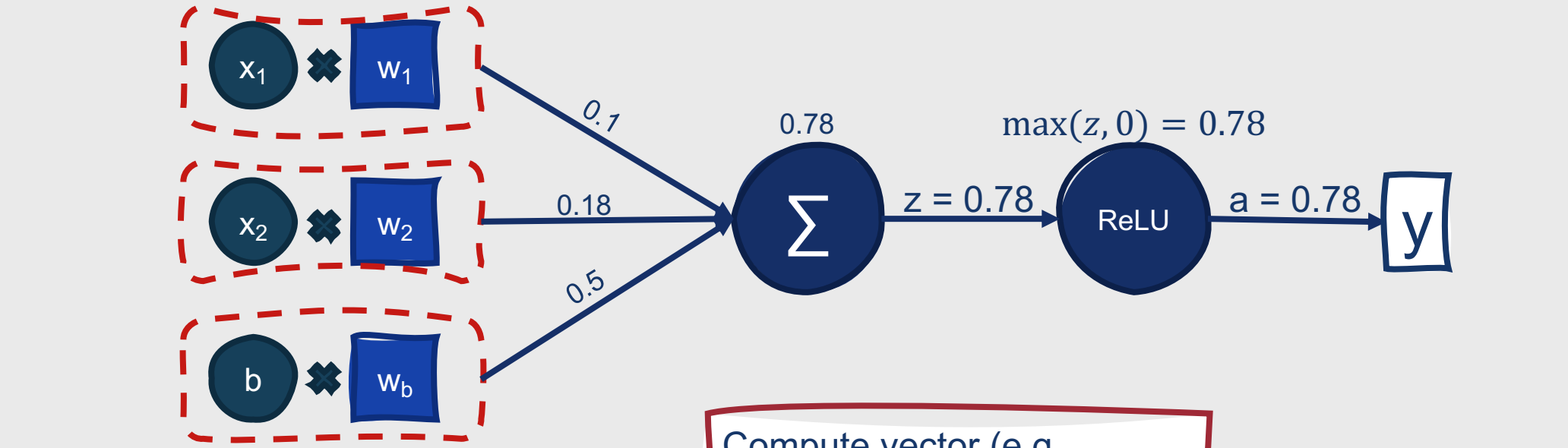
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with ReLU Activation



Input: “beautiful brutalist architecture”

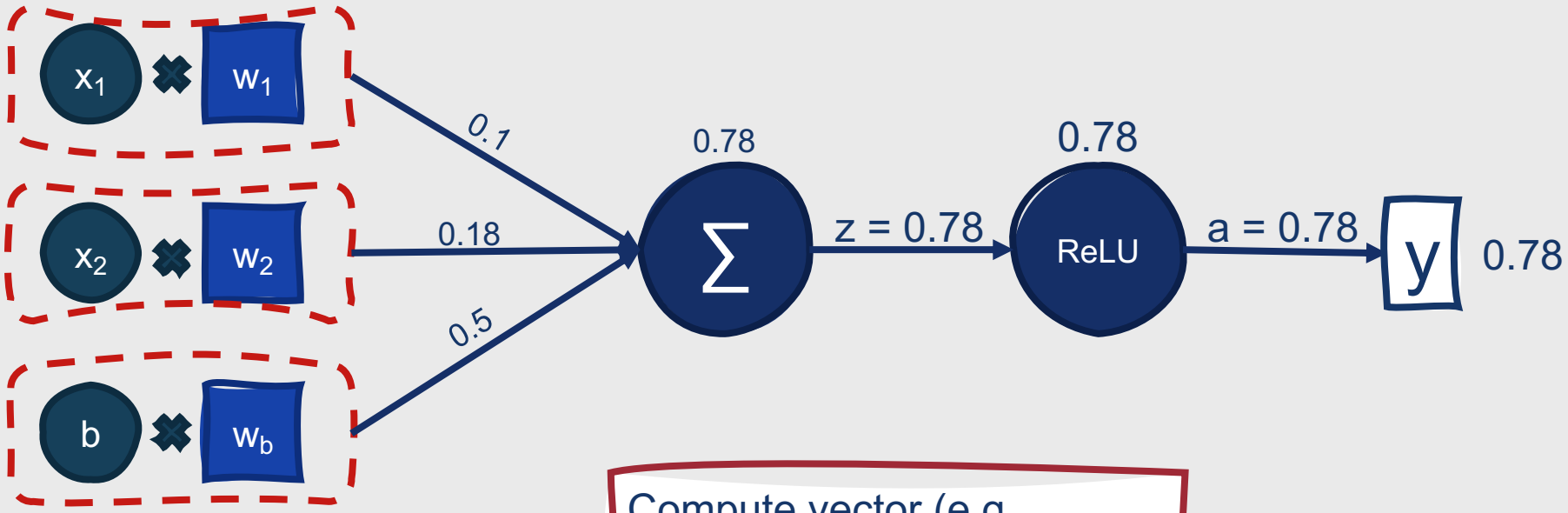
Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Example: Computational Unit with ReLU Activation



Input: “beautiful brutalist architecture”

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Comparing sigmoid, tanh, and ReLU

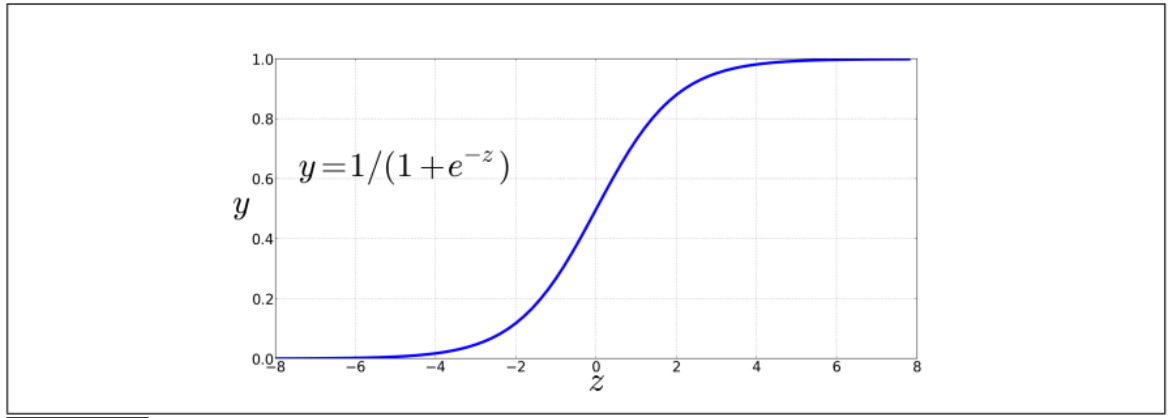


Figure 7.1 The sigmoid function takes a real value and maps it to the range $[0,1]$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

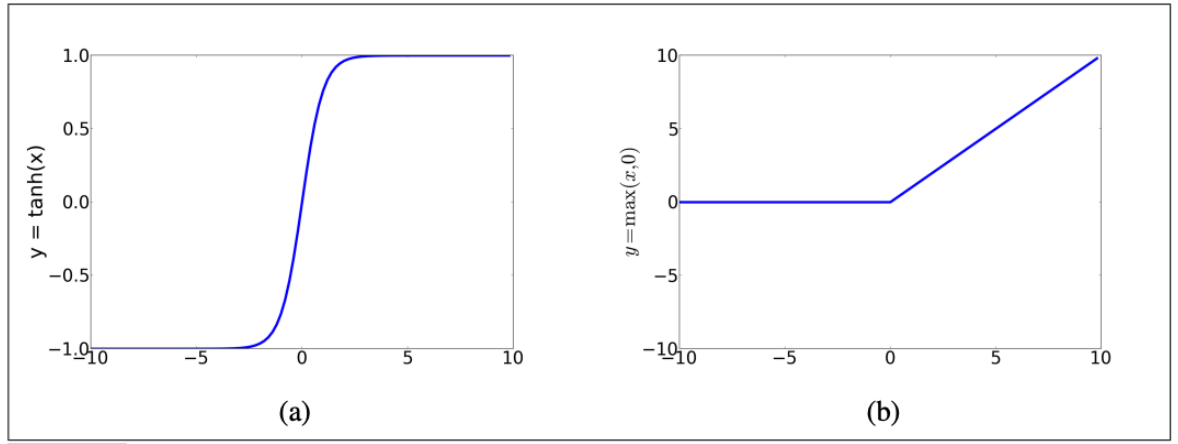


Figure 7.3 The tanh and ReLU activation functions.

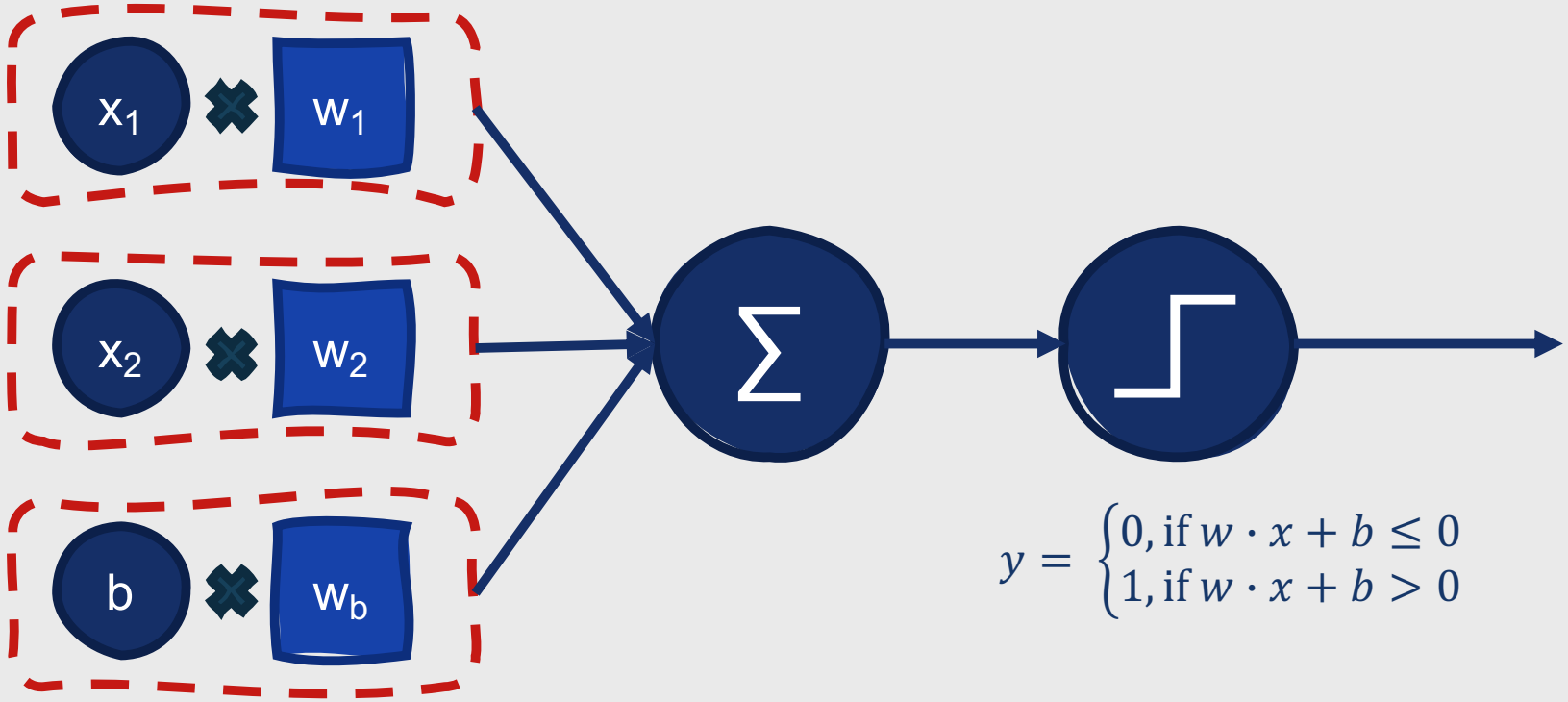
Combining Computational Units

Neural networks are powerful primarily because they are able to **combine multiple computational units into larger networks**

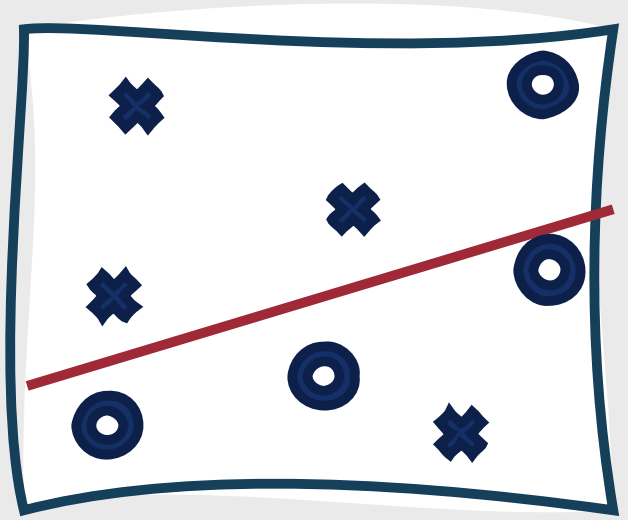
Many problems cannot be solved using a single computational unit

Early example of this: The XOR problem

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0



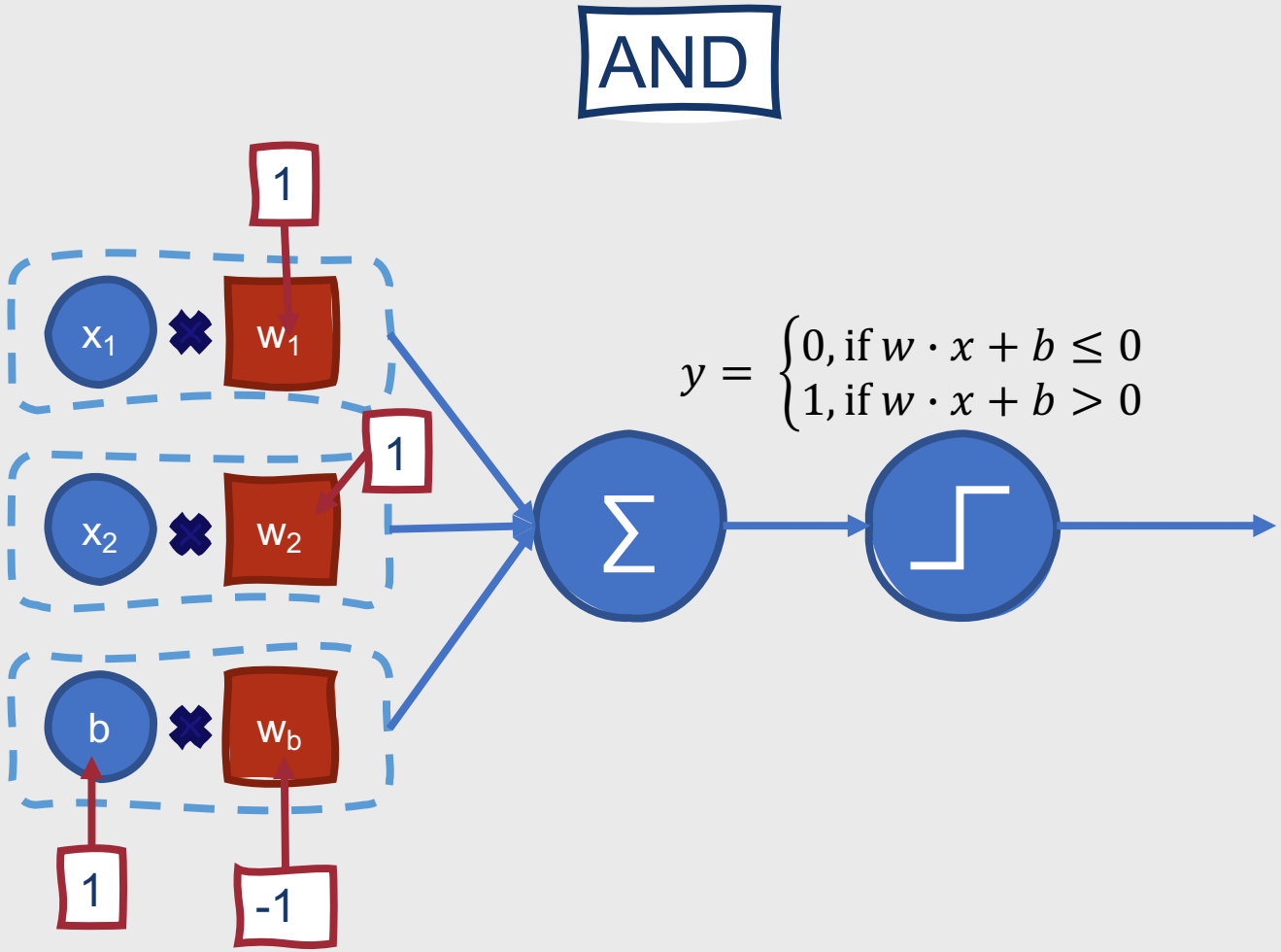
$$y = \begin{cases} 0, \text{if } w \cdot x + b \leq 0 \\ 1, \text{if } w \cdot x + b > 0 \end{cases}$$



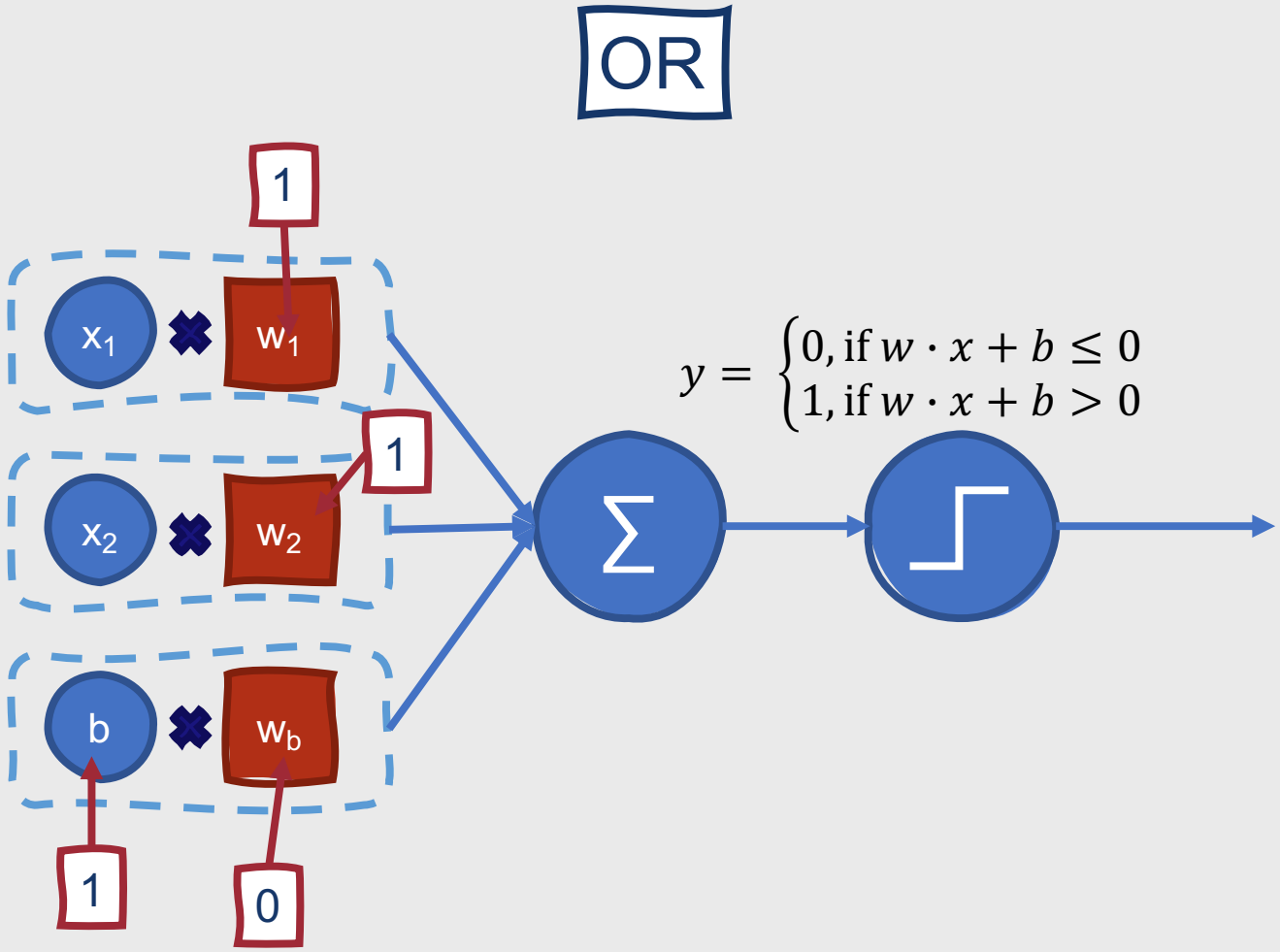
AND and OR can both be solved using a single perceptron.

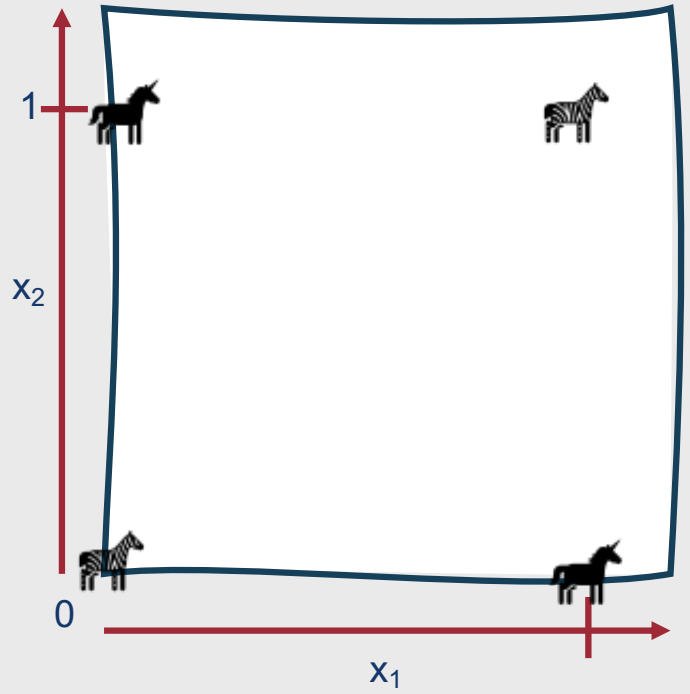
- **Perceptron:** A function that outputs a binary value based on whether the product of its inputs and associated weights surpasses a threshold
  - Learns this threshold iteratively by trying to find the boundary that is best able to distinguish between data of different categories

It's easy to compute AND and OR using perceptrons.



It's easy to compute AND and OR using perceptrons.



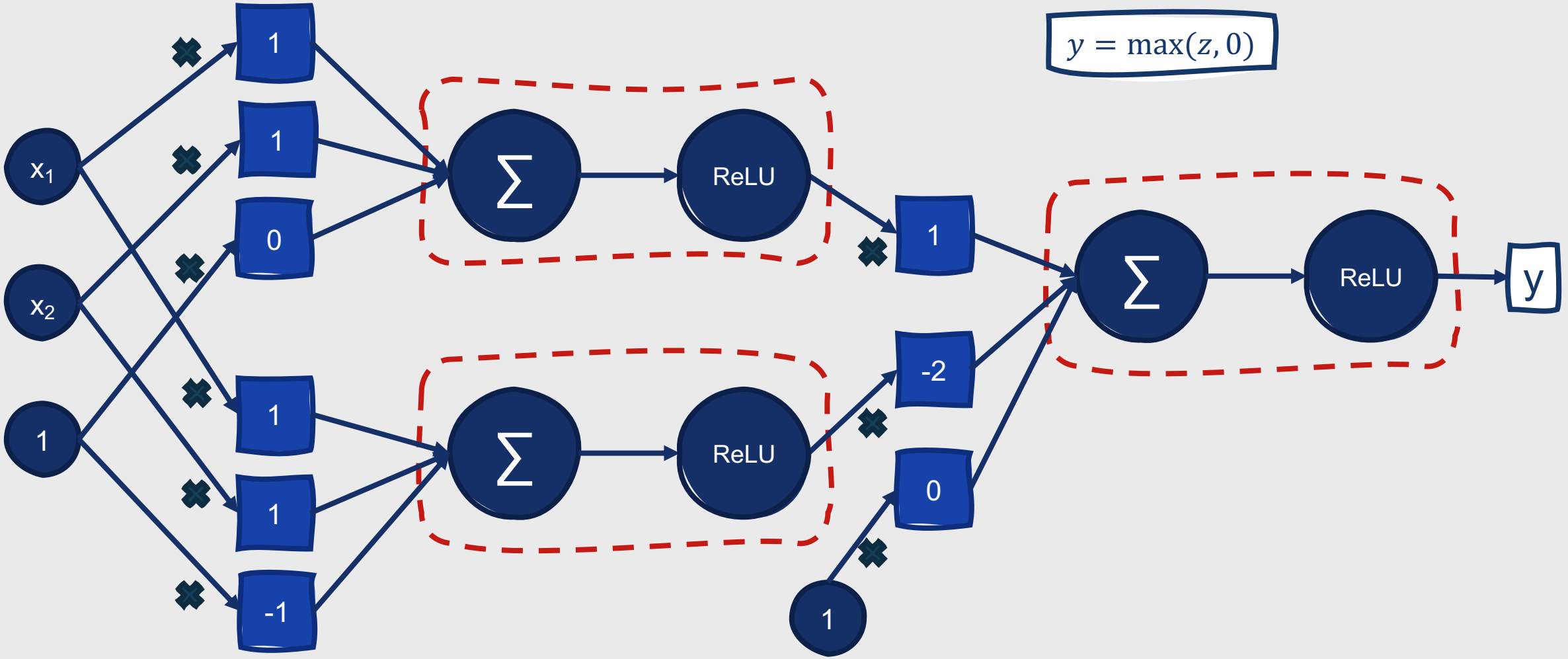


AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

However, it's impossible to compute XOR using a single perceptron.

- Why?
  - Perceptrons are **linear classifiers**
  - XOR is not a **linearly separable function**

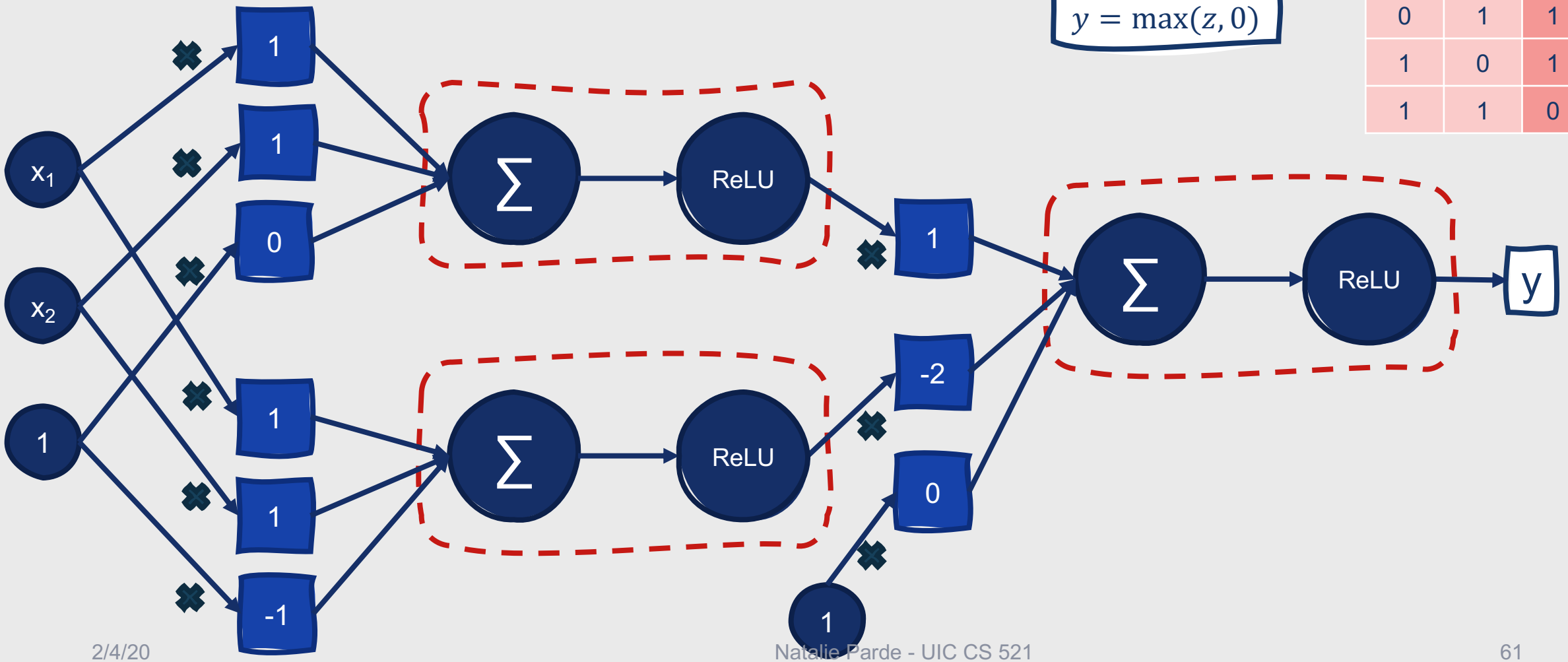
The only successful way to compute XOR is by combining these smaller units into a larger network.



Truth Table Examples: XOR

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

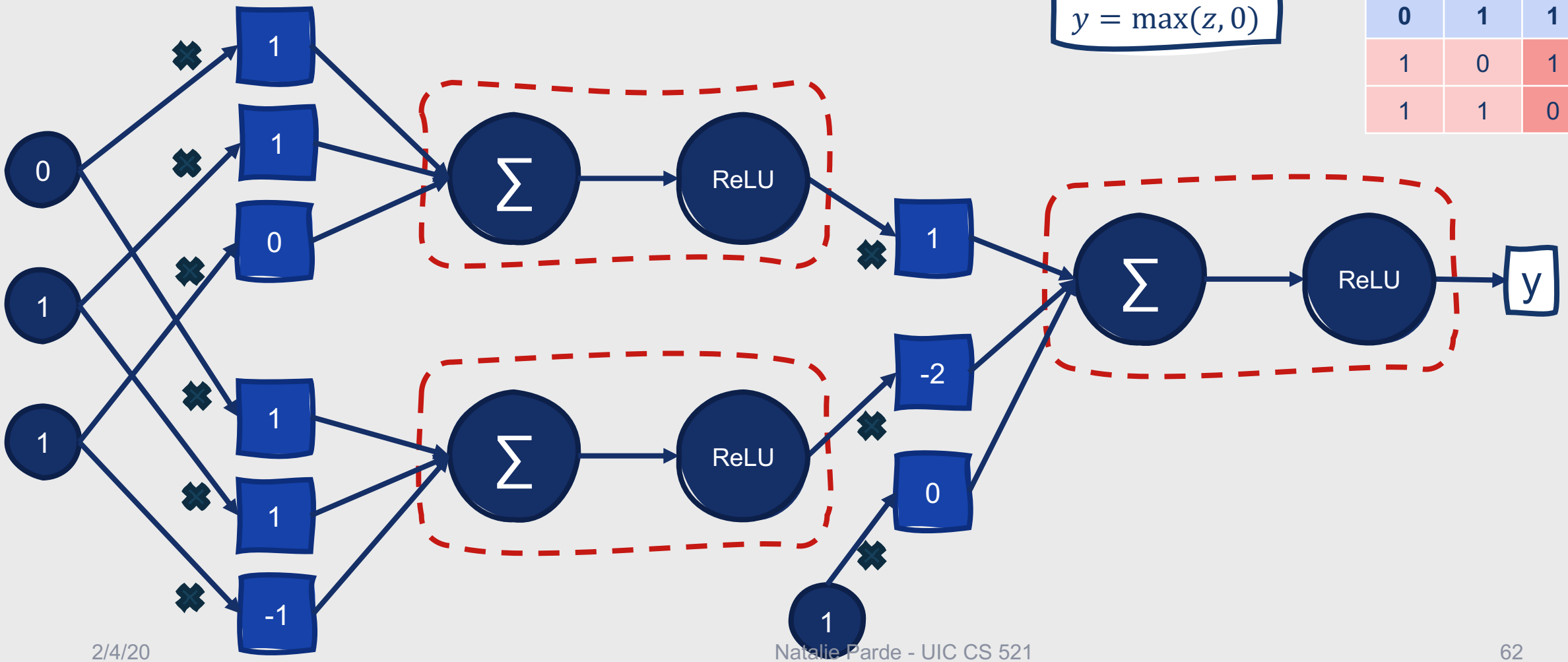
$y = \max(z, 0)$



Truth Table Examples: XOR

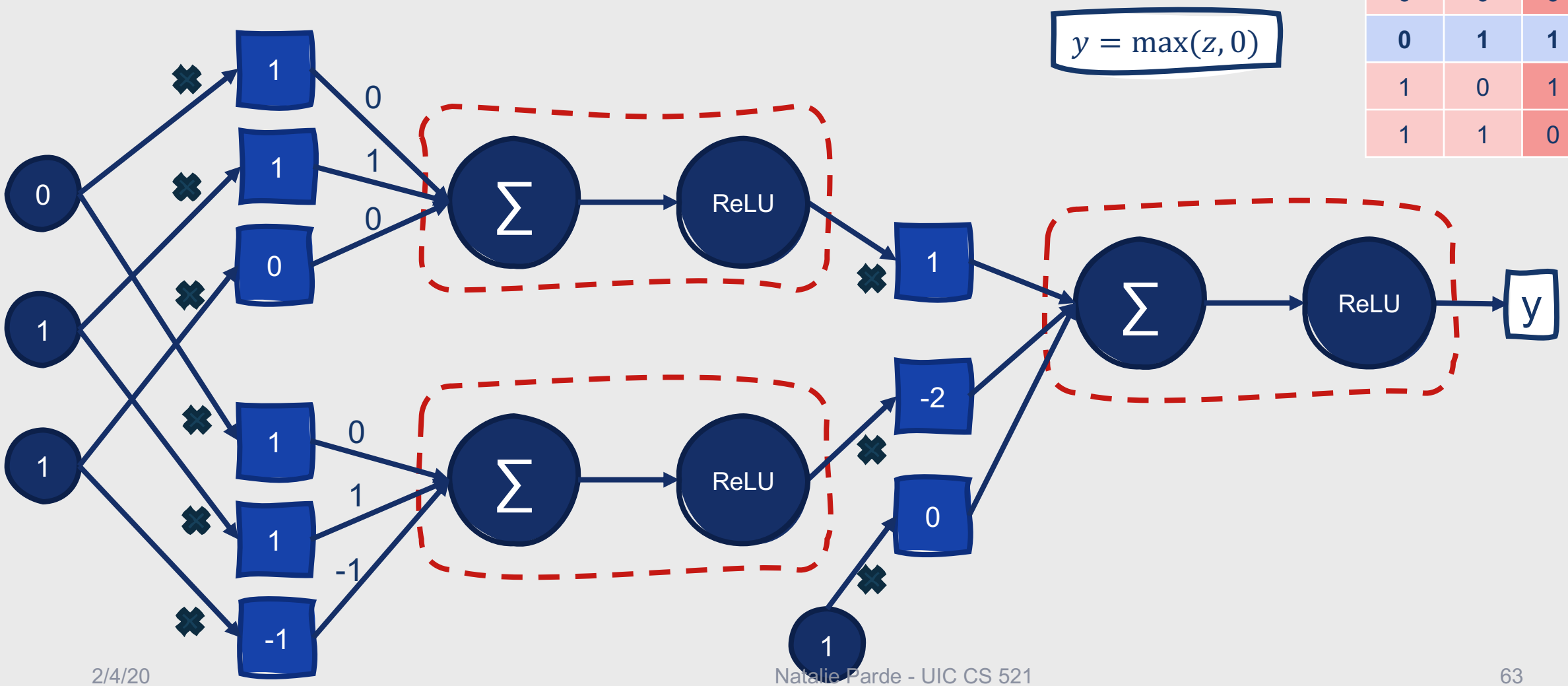
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

$y = \max(z, 0)$



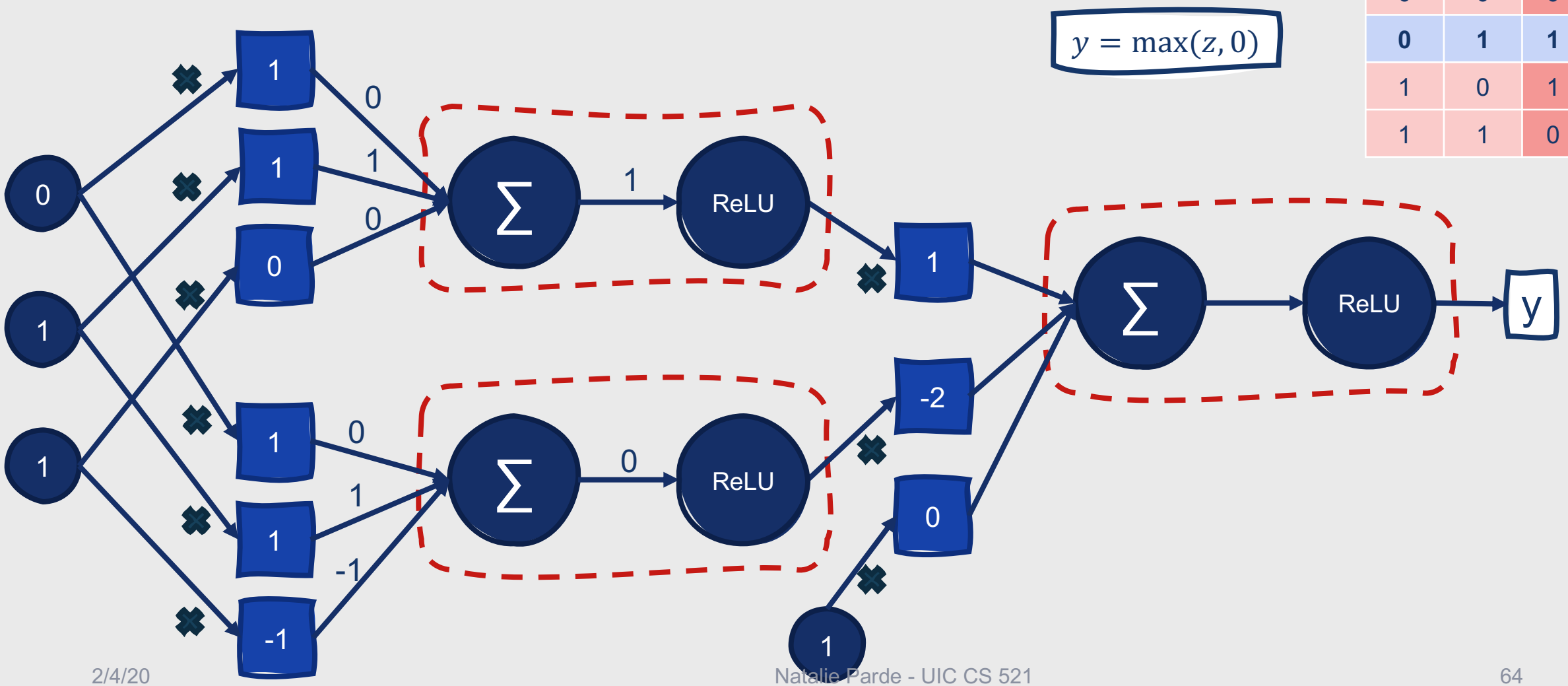
Truth Table Examples: XOR

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



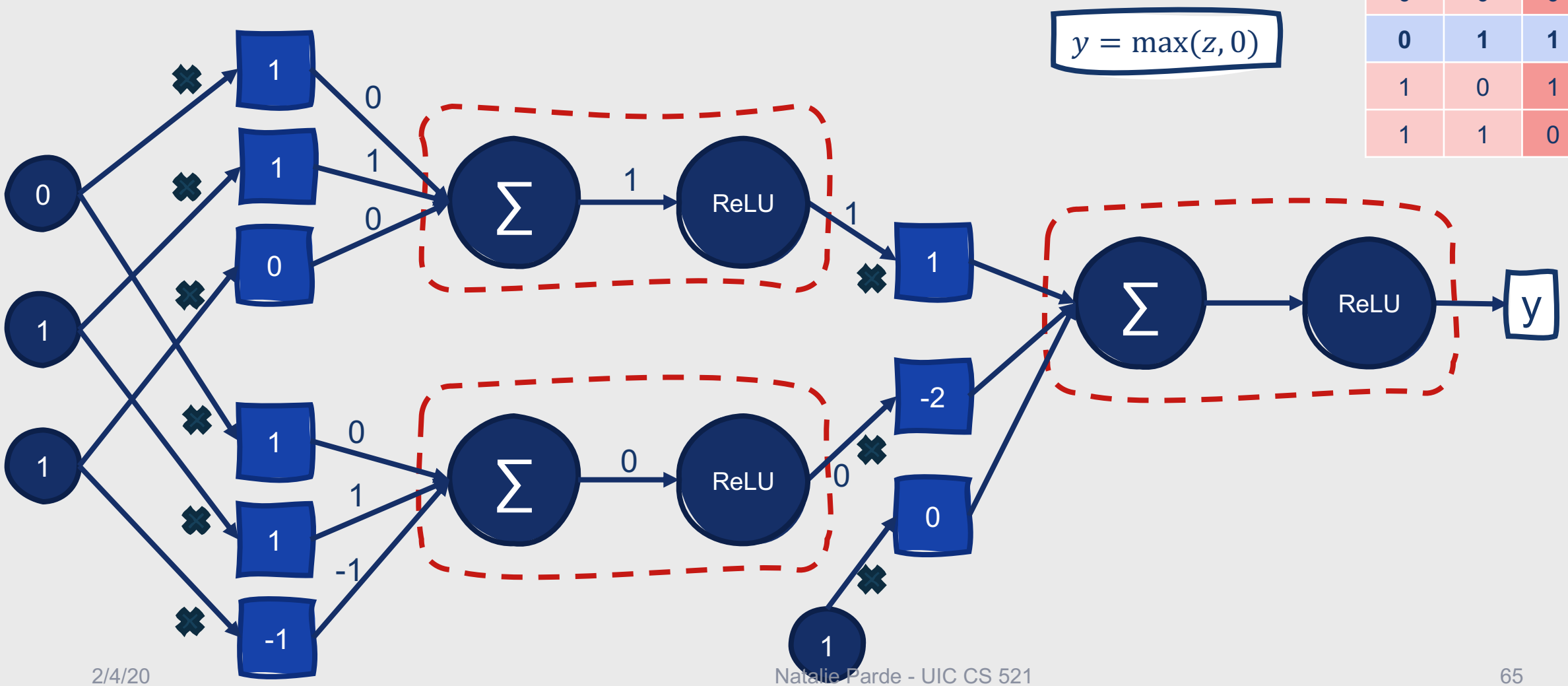
Truth Table Examples: XOR

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



Truth Table Examples: XOR

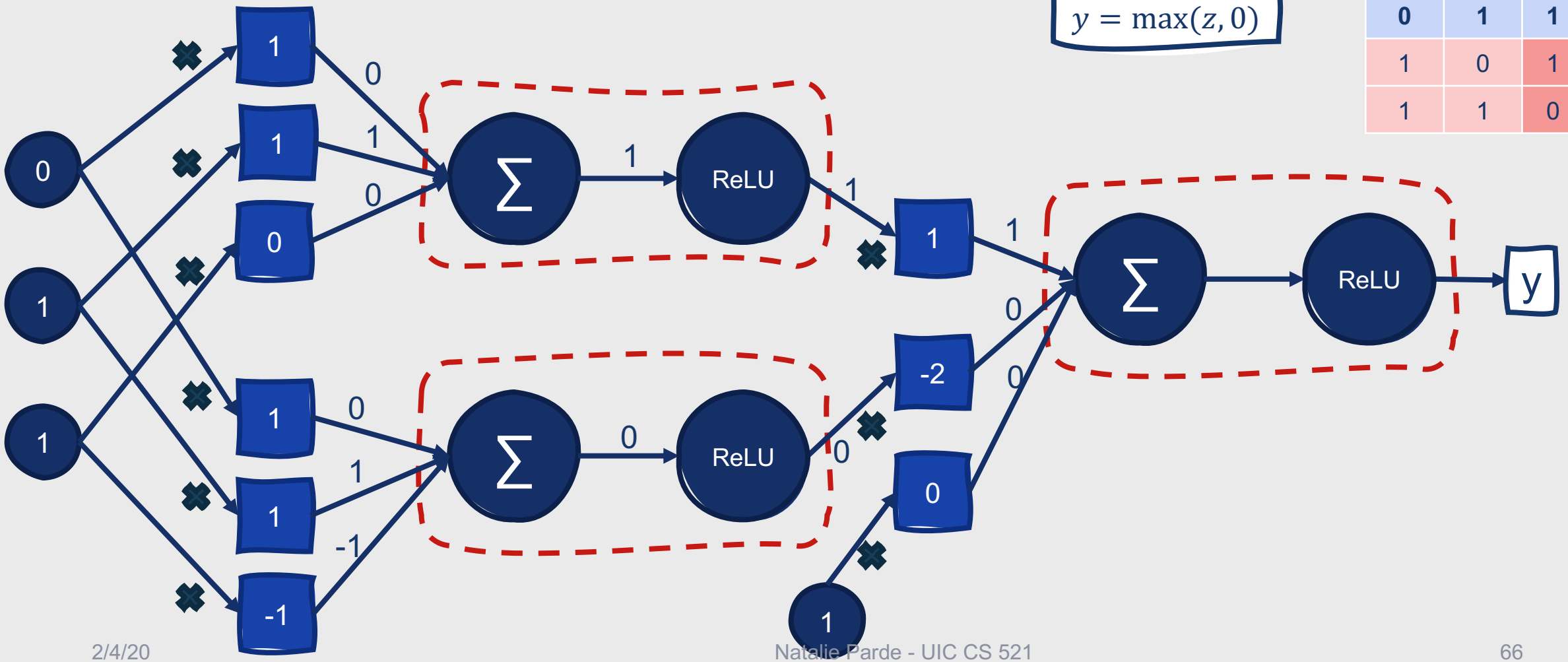
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



Truth Table Examples: XOR

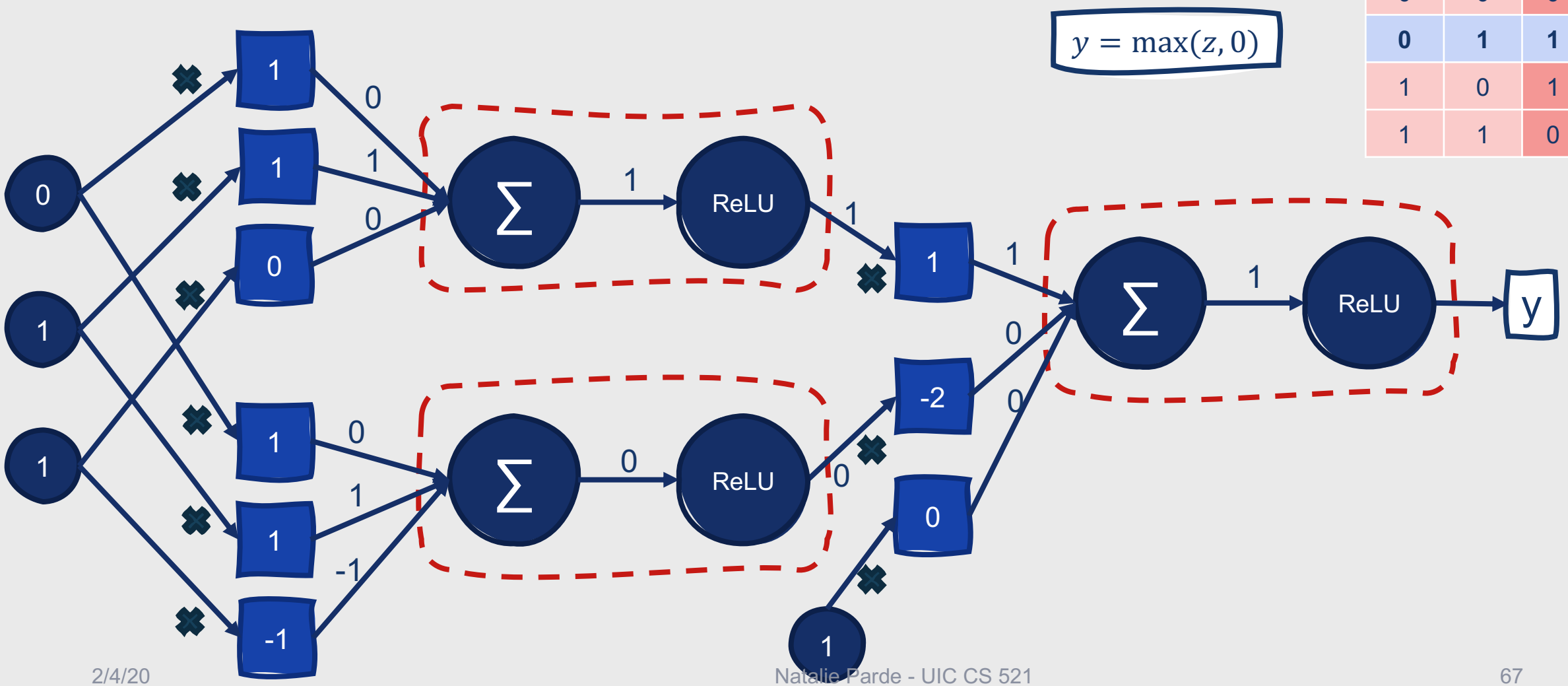
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

$y = \max(z, 0)$



Truth Table Examples: XOR

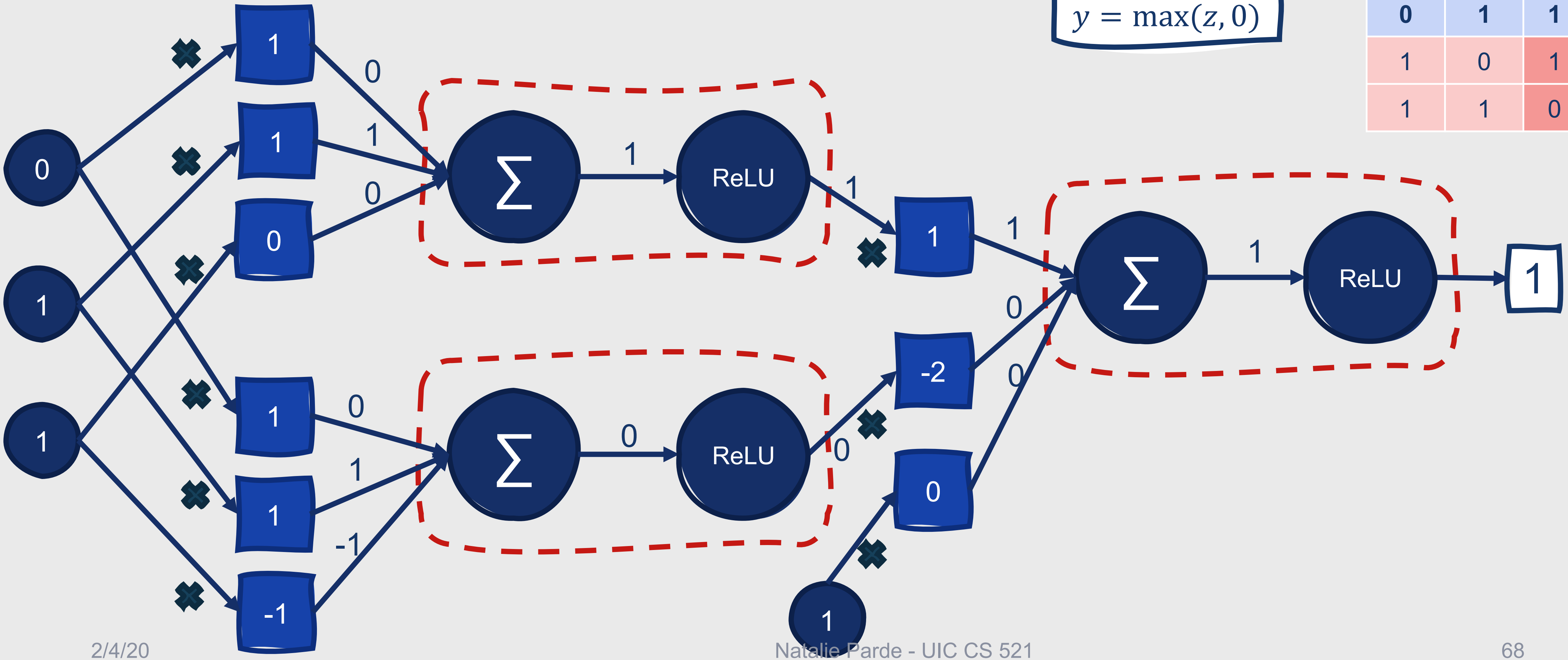
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



Truth Table Examples: XOR

$y = \max(z, 0)$

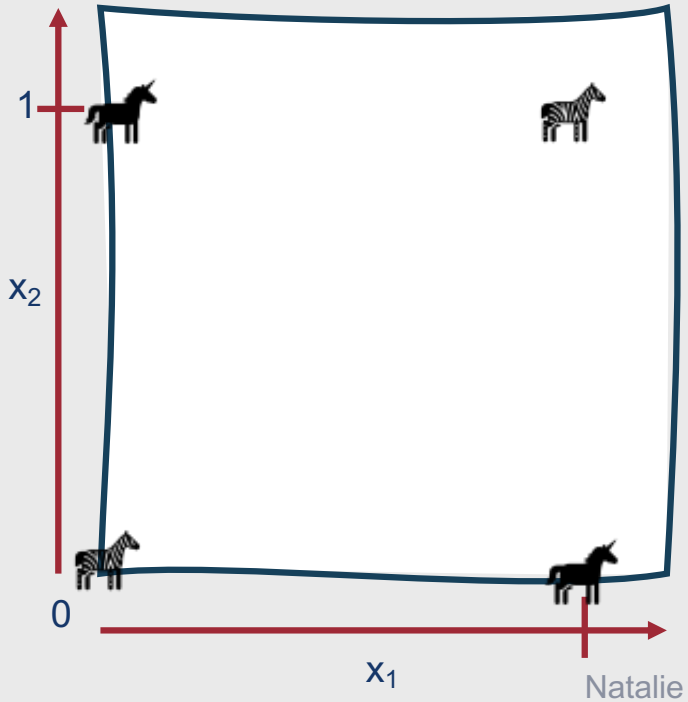
XOR		
x1	**x2**	**y**
0	0	0
0	**1**	**1**
1	0	1
1	1	0



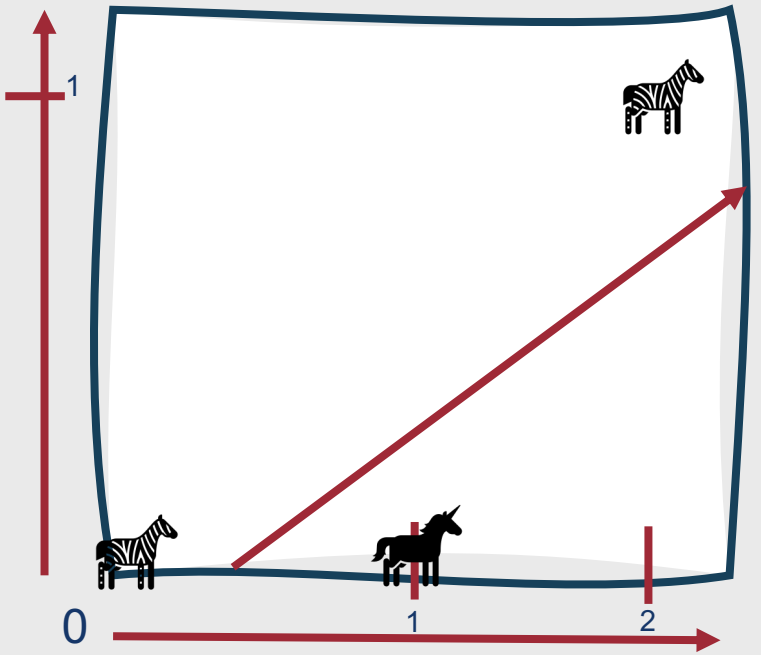
Why does this work?

- When computational units are combined, the outputs from each successive layer provide **new representations** for the input
- These new representations are **linearly separable**

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



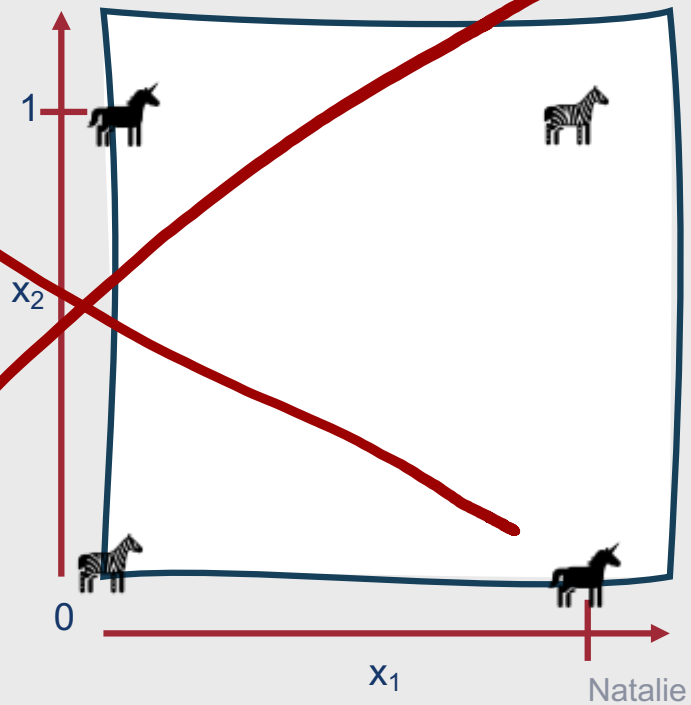
XOR		
h0	h1	y
0	0	0
1	0	1
2	1	0



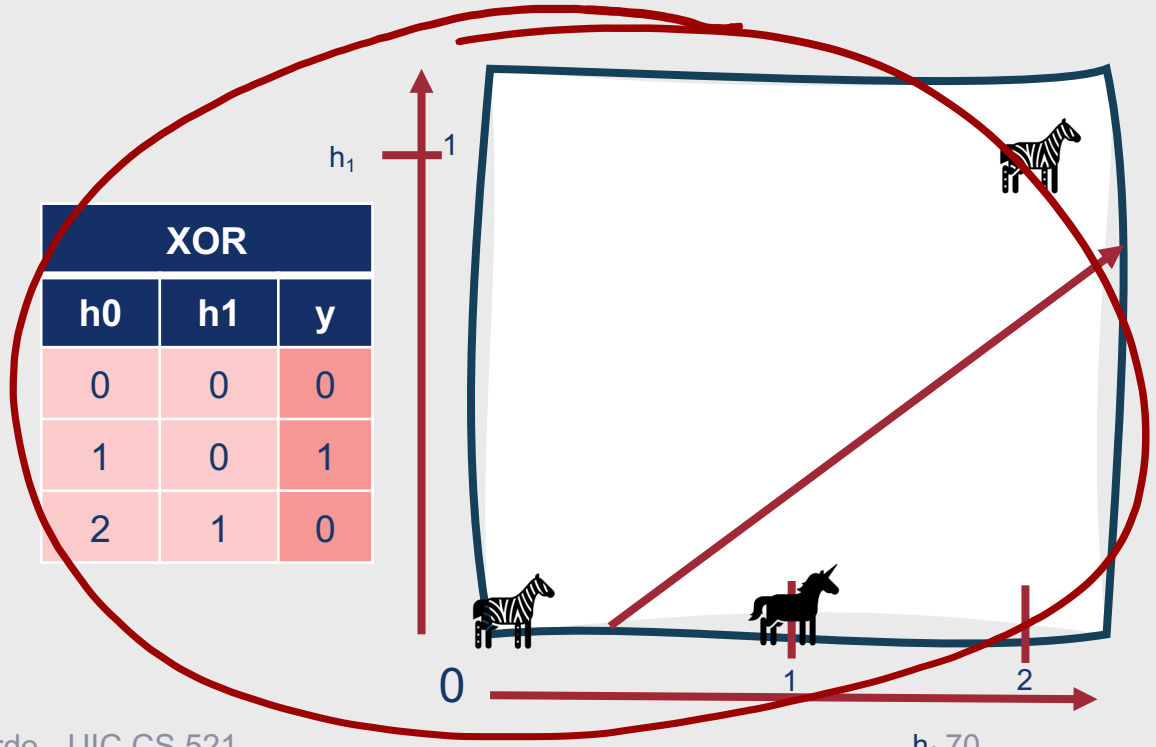
Why does this work?

- When computational units are combined, the outputs from each successive layer provide **new representations** for the input
- These new representations are **linearly separable**

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



XOR		
h0	h1	y
0	0	0
1	0	1
2	1	0



Combining Computational Units

- In our XOR example, we manually assigned weights to each unit
- In real-world examples, these weights are learned automatically using a **backpropagation** algorithm
- Thus, the network is able to learn a useful representation of the input training data on its own
  - Key advantage of neural networks

More about specific unit types in feedforward networks….

- Three main unit types:
  - Input units
  - Hidden units
  - Output units

Input Units

0.5	0.2	0.1	0.7	0.4

- Vector of scalar values
  - Word embedding
  - Other feature vector
- No computations performed in input units

Hidden Units

- Computation units
  - As described previously, take a weighted sum of inputs and apply a nonlinear function to it
- Contained in one or more layers
- Layers are **fully connected**
  - All units in layer *n* receive inputs from all units in layer *n*-1
    - Layer *n*-1 can be the input layer or an earlier hidden layer

Hidden Layers

- Remember: Individual computation units have parameters **w** (the weight vector) and *b* (the bias)
- The parameters for an entire hidden layer (including all computation units within that layer) can then be represented as:
  - *W*: Weight matrix containing the weight vector $\mathbf{w_i}$ for each unit *i*
  - **b**: Bias vector containing the bias value b_i for each unit *i*
    - Single bias for layer, but each unit can associate a different weight with the bias
- W_{ij} represents the weight of the connection from input unit x_i to hidden unit h_j

Why represent W as a single matrix?

- More efficient computation across the entire layer
- Use matrix operations!
  - Multiply the weight matrix by input vector **x**
  - Add the bias vector **b**
  - Apply the activation function *g* (e.g., sigmoid, tanh, or ReLU)
- This means that we can compute a vector **h** representing the output of a hidden layer as follows:
  - $\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$

Formal Definitions

- An input (layer 0) vector x has a dimensionality of n_0, where n_0 is the number of inputs
  - So, $x \in \mathbb{R}^{n_0}$
- The subsequent hidden layer (layer 1) has dimensionality n_1, where n_1 is the number of hidden units in the layer
  - So, $h \in \mathbb{R}^{n_1}$ and $b \in \mathbb{R}^{n_1}$ (remember, b contains the different weighted bias values associated with each hidden unit)
- The weight matrix thus has the dimensionality $W \in \mathbb{R}^{n_1 \times n_0}$

Output Units

- Provide probabilities indicating whether the input belongs to a given class
- Number of output units can vary:
  - Binary classification might have a single output unit
  - Multinomial classification (e.g., part-of-speech tagging) might have an output unit for each class

Output Layer

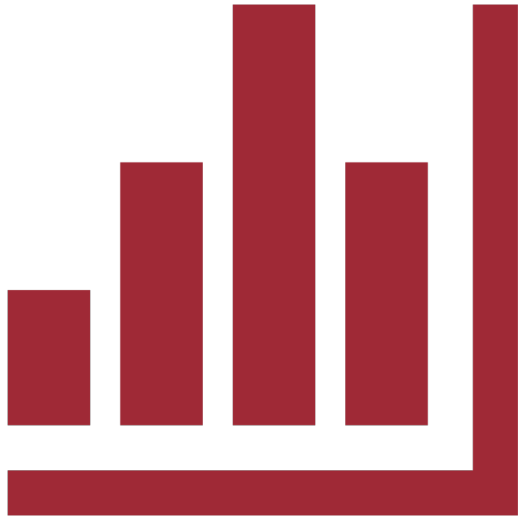
- Provides a probability distribution across the output nodes
- How?
  - Output layer also has a weight matrix, U
  - Bias vector is optional
  - Following intuition/examples, $z = U\mathbf{h}$, where **h** is the vector of outputs from the previous hidden layer

Formal Definitions

- Letting n_2 be the number of output nodes, $z \in \mathbb{R}^{n_2}$
- The weight matrix U thus has the dimensionality $U \in \mathbb{R}^{n_2 \times n_1}$, where n_1 is the number of hidden units in the previous layer
  - U_{ij} is the weight from unit j in the hidden layer to unit i in the output layer

Just like with logistic regression, the values in z are just real-valued numbers.

- We need to convert them to probabilities instead!
- We do this using activation functions
  - Sigmoid
  - Softmax
  - Etc.
- Popular choice in multinomial feedforward networks: **Softmax**
  - Increase the probability of the highest value in the vector
  - Decrease the probabilities of the other values
  - $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{|\mathbf{Z}|} e^{z_j}}$



Feedforward Network

- Final set of equations:
  - $\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$
  - $\mathbf{z} = U\mathbf{h}$
  - $y = \text{softmax}(\mathbf{z})$
- This represents a two-layer feedforward neural network
  - When numbering layers, count the hidden and output layers but not the input layer

What if we want our network to have more than two layers?

- Let $W^{[n]}$ be the weight matrix for layer n, $\mathbf{b}^{[n]}$ be the bias vector for layer n, and so forth
- Let $g(\cdot)$ be an activation function
  - ReLU
  - tanh
  - softmax
  - Etc.
- Let $\mathbf{a}^{[n]}$ be the output from layer n, and $\mathbf{z}^{[n]}$ be the combination of weights and biases $W^{[n]}\ \mathbf{a}^{[n-1]} + \mathbf{b}^{[n]}$
- Let the input layer be $\mathbf{a}^{[0]}$

What if we want our network to have more than two layers?

- With this representation, a two-layer network becomes:
  - $z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$
  - $a^{[1]} = g^{[1]}(z^{[1]})$
  - $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
  - $a^{[2]} = g^{[2]}(z^{[2]})$
  - $y' = a^{[2]}$
- With this notation, we can easily generalize to networks with more layers:
  - For i in 1..*n*
    - $z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}$
    - $a^{[i]} = g^{[i]}(z^{[i]})$
  - $y' = a^{[n]}$

One final note….

- The activation function $g(\cdot)$ generally differs for the final layer
- Earlier layers will more commonly be ReLU or tanh
- Final layers will more commonly be softmax (for multinomial classification) or sigmoid (for binary classification)

Summary: Feedforward Neural Networks

- **Neural networks** are classification models comprised of interconnected computing units
- **Feedforward neural networks** are a subset of neural networks in which information is passed forward from one **fully-connected** layer to the next
- Individual computing units in neural networks calculate weighted sums of input values
- **Activation functions** are applied to these linear combinations to produce **non-linear** representations
- Feedforward neural networks contain three types of units:
  - **Input**
  - **Hidden**
  - **Output**
- When neural networks contain multiple layers stacked on top of one another, they are often referred to as **deep neural networks**