# Neural Networks and Neural Language Models

**Natalie Parde, Ph.D.**

Department of Computer Science

University of Illinois at Chicago

CS 421: Natural Language Processing

Fall 2019

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

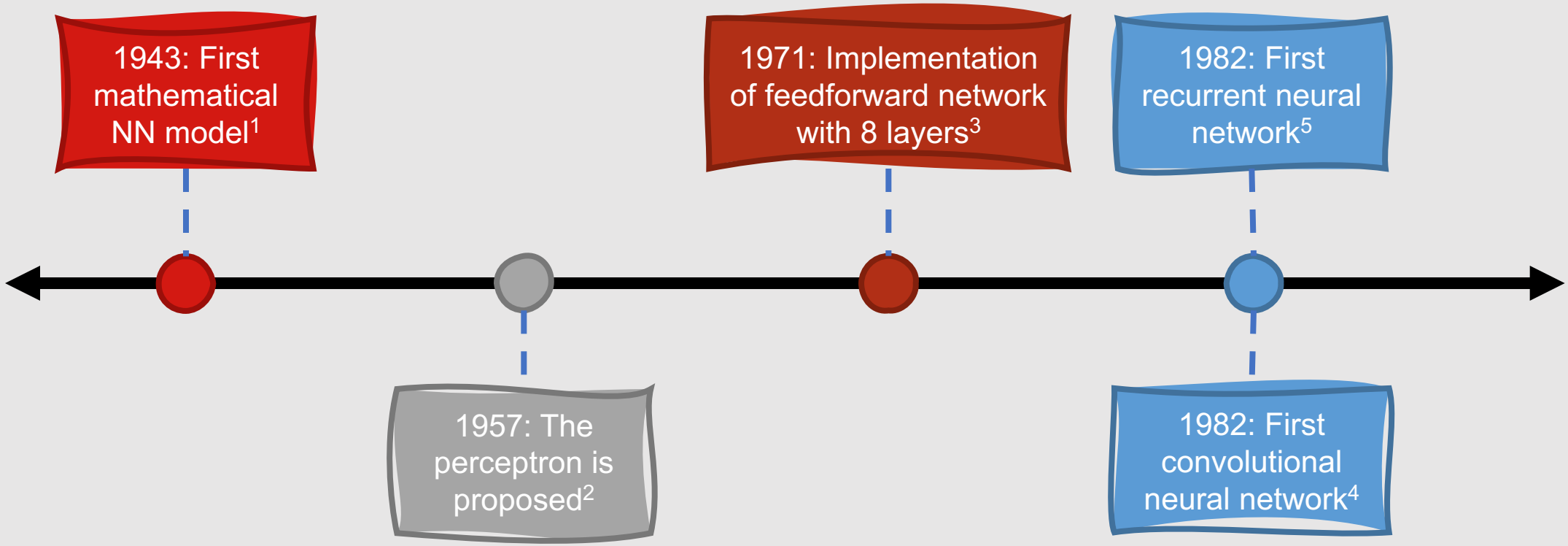
# What are neural networks?

- 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

**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 mod-

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 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.

**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 author 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 networks are everywhere!

# 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

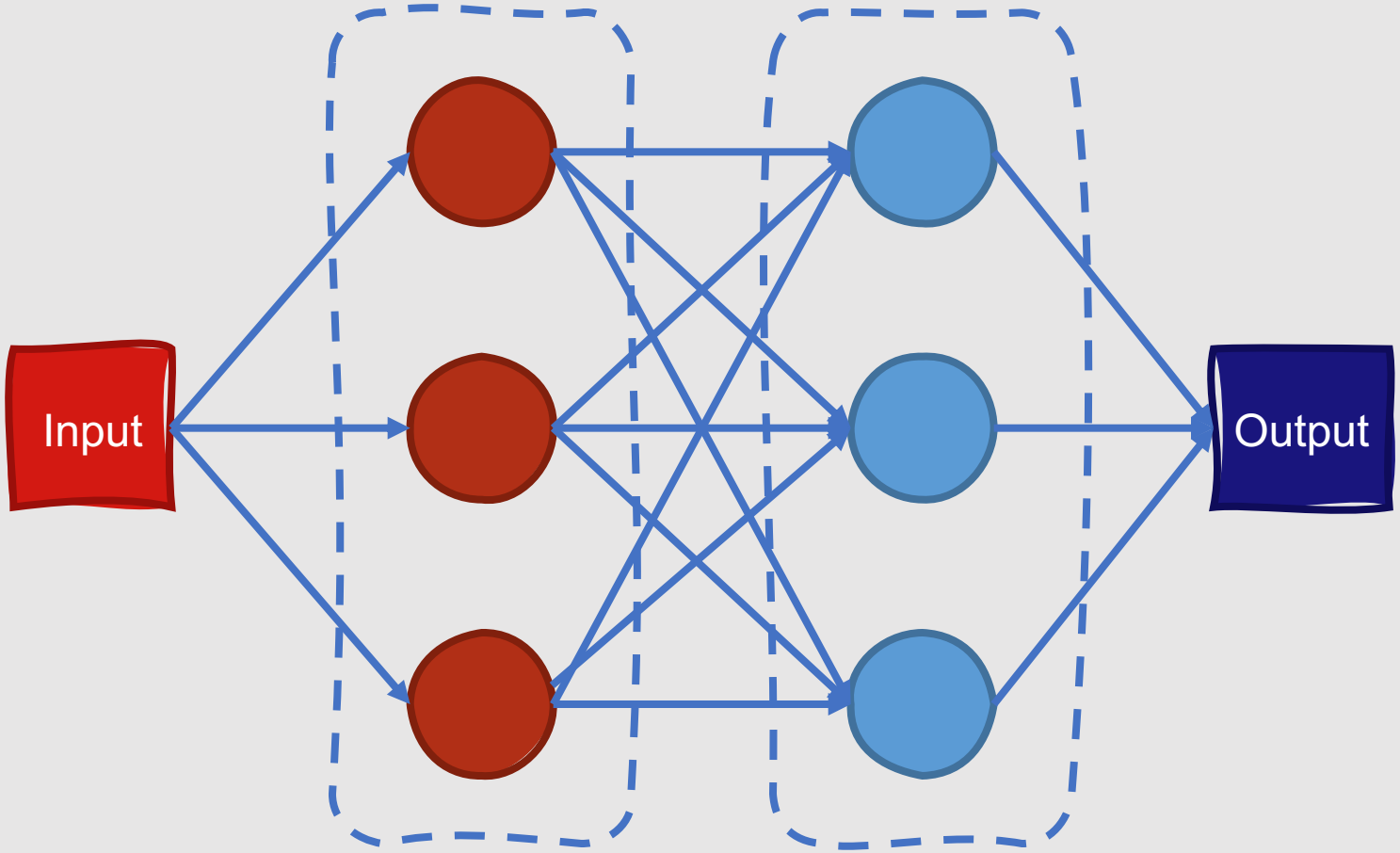
# Types of Neural Networks

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

# 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



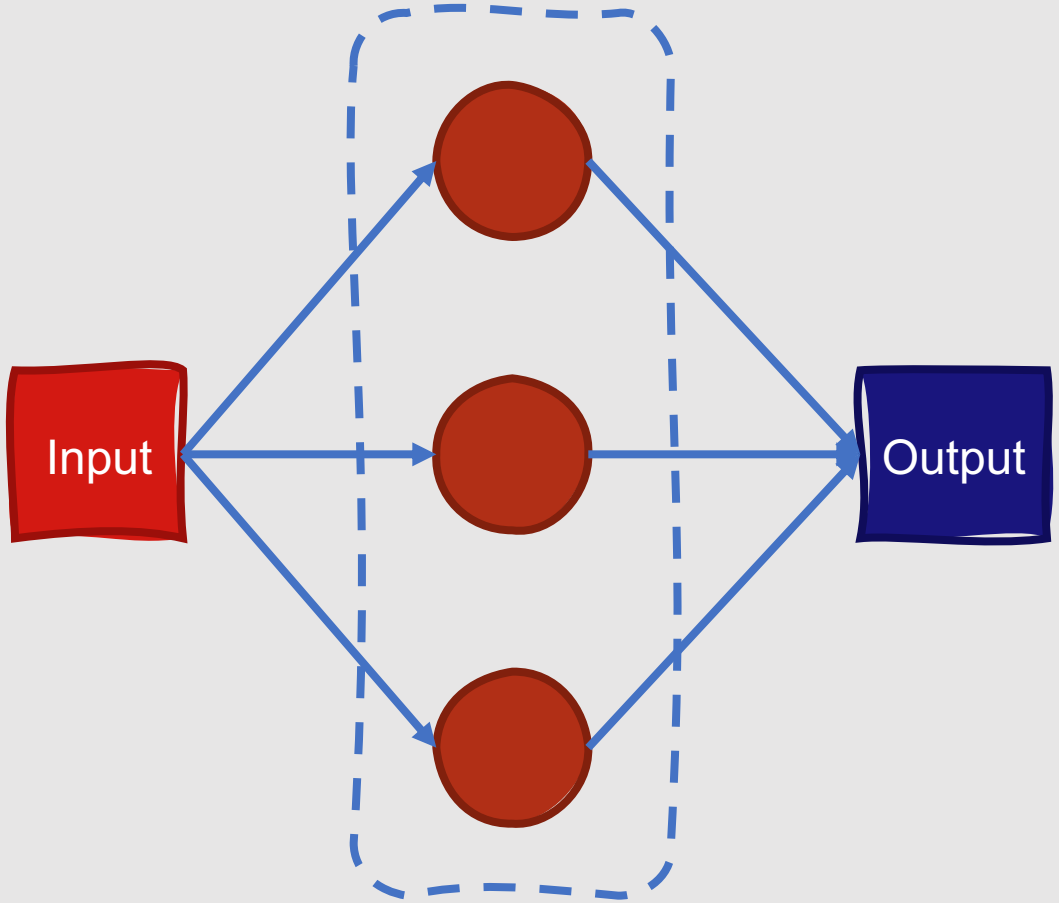
# What is deep learning?

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

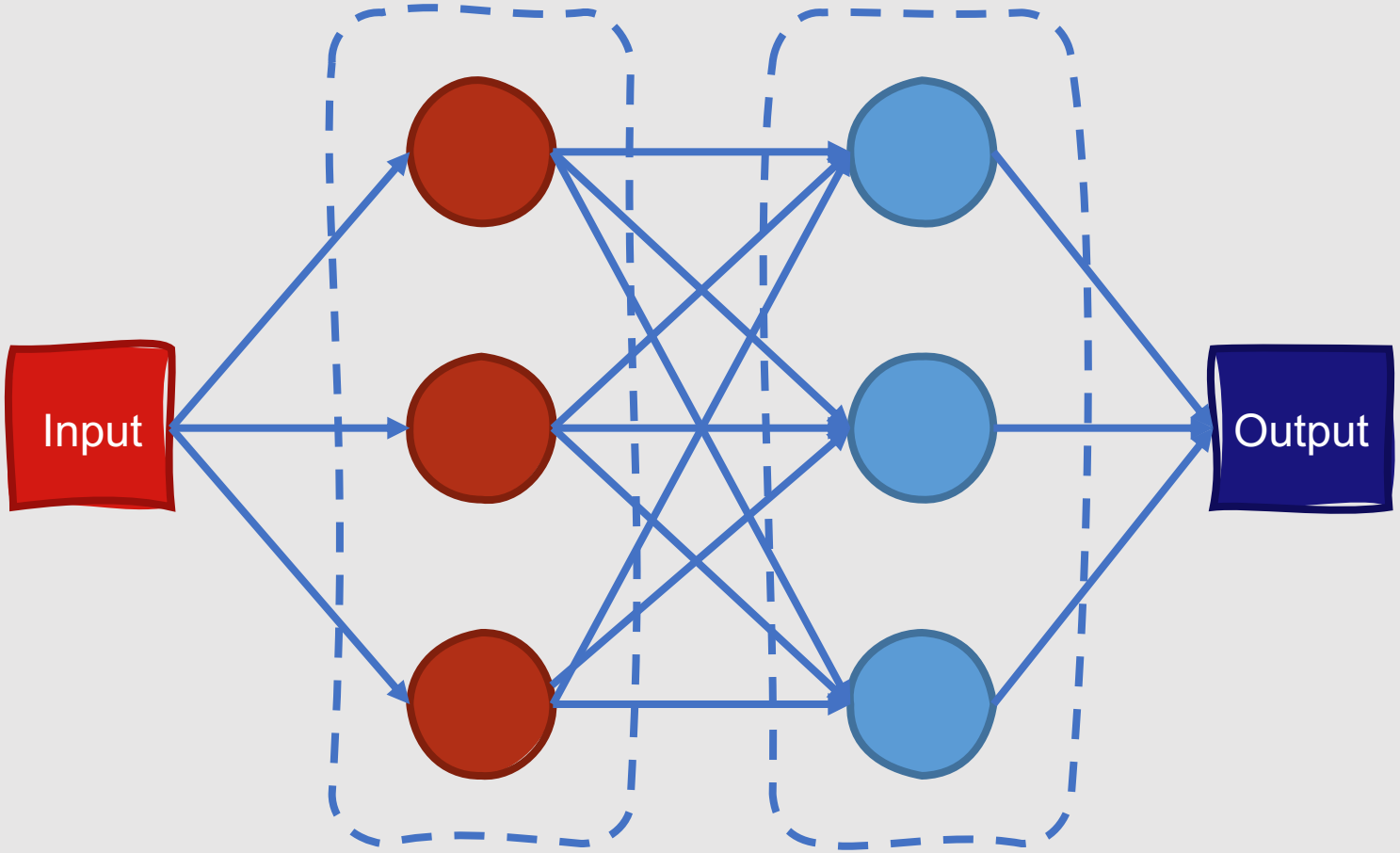
Why?

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

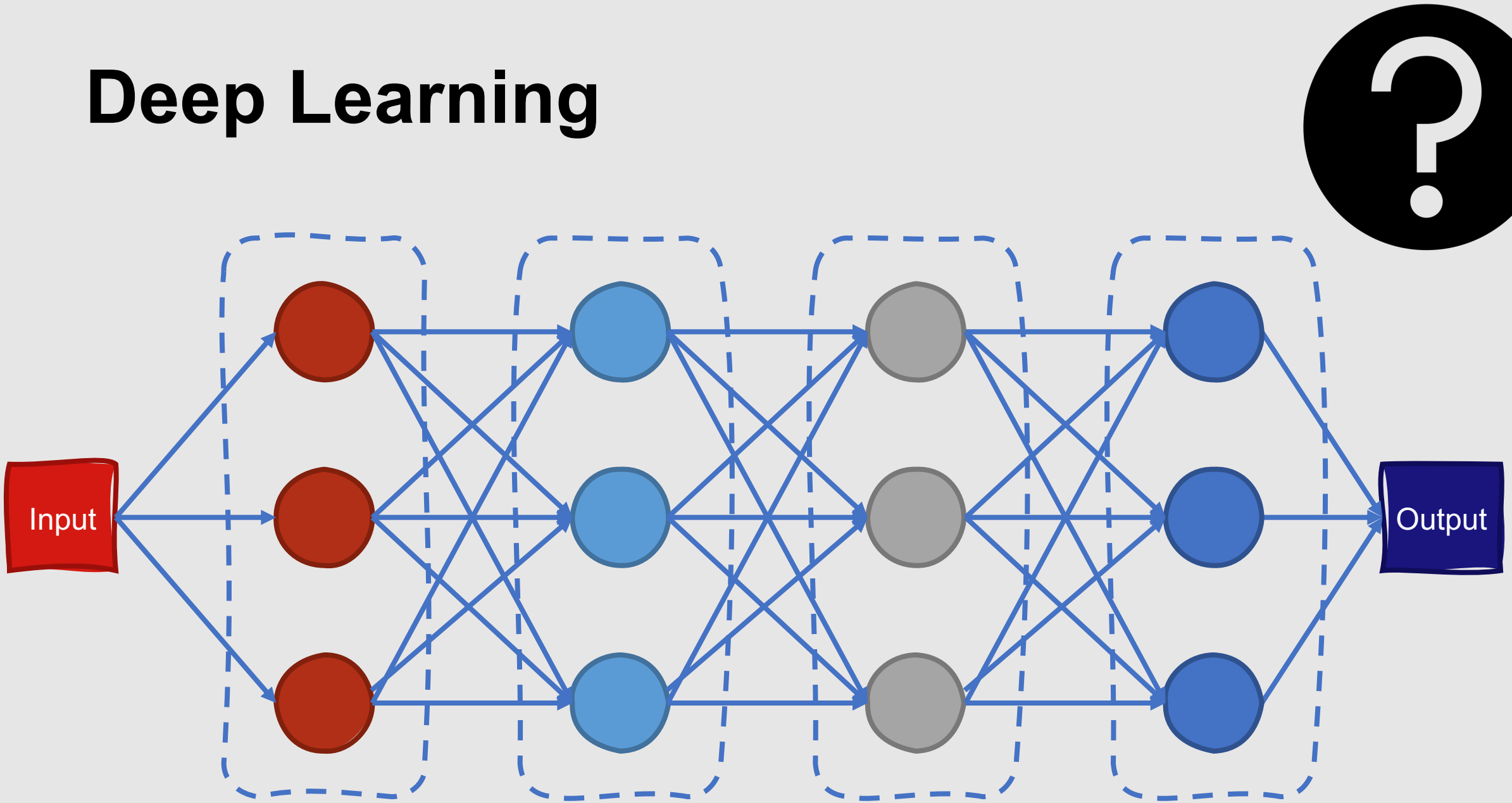
# Deep Learning

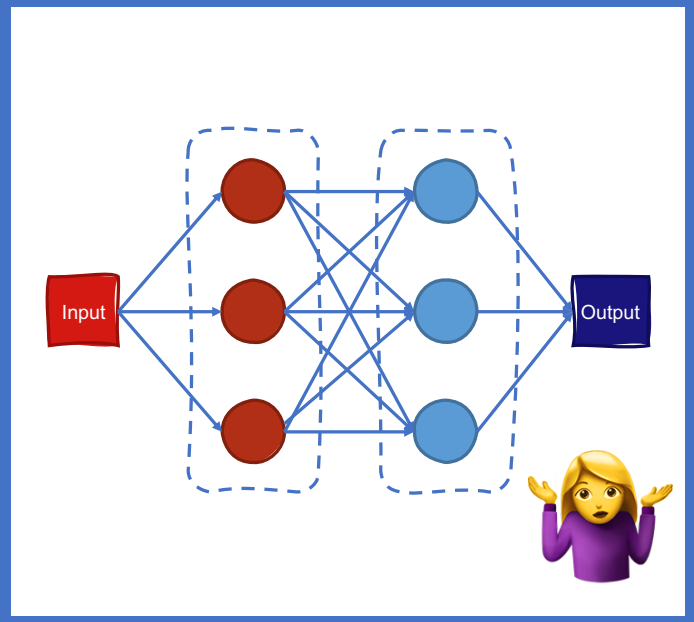


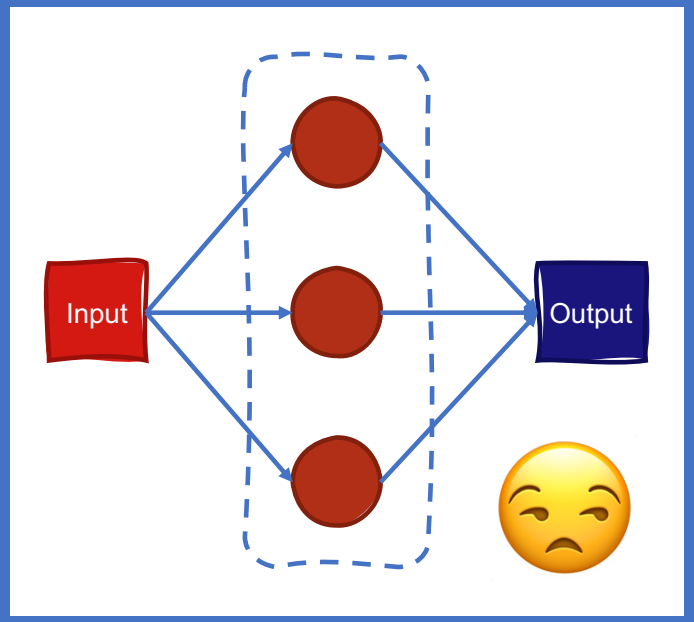

# Deep Learning

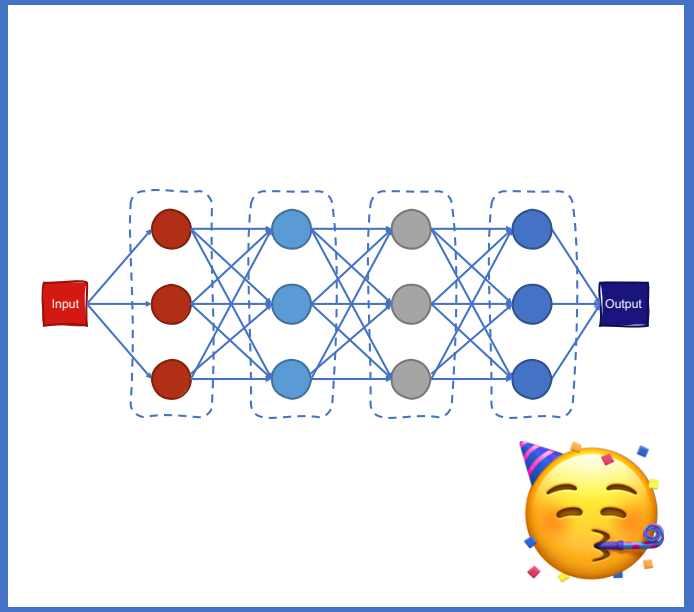



# Deep Learning





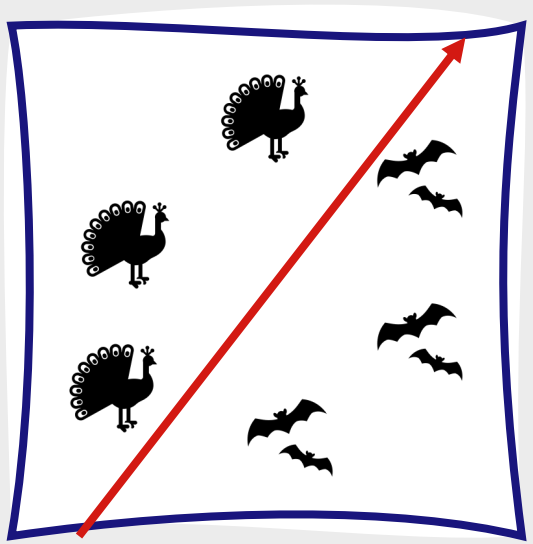



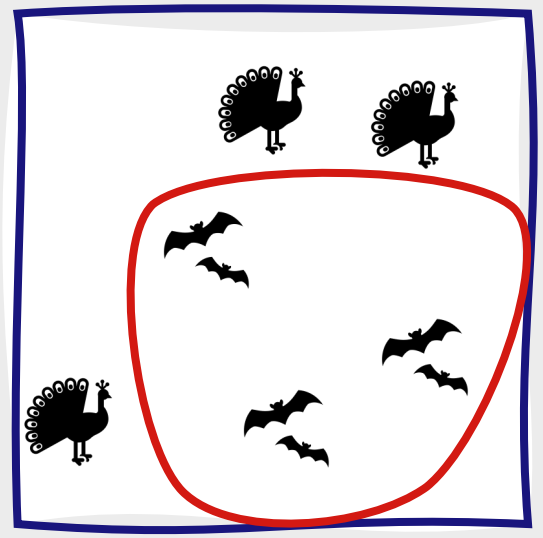




# Deep Learning

**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 as part of the classifier's training process
  - Just use word embeddings as input

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

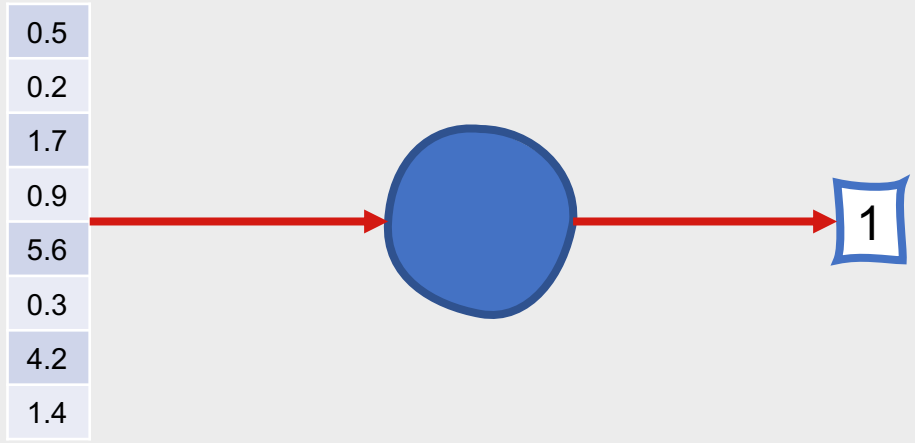
Learning features **implicitly** requires a lot of data

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:
  - $z = b + \sum_i w_i x_i$

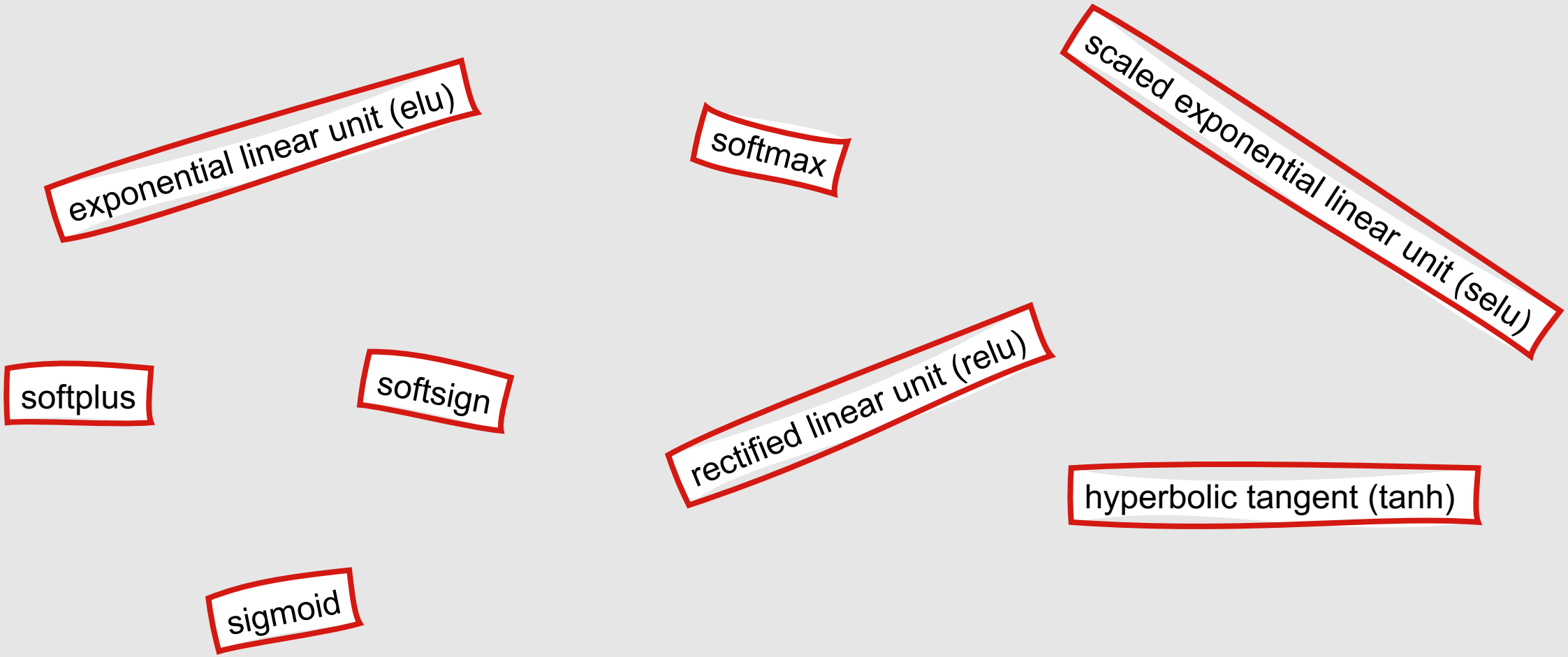
# Computational Units

- Recall that our classification input is some sort of **word embedding** or other **feature vector**
- Thus, letting $w$ be the weight vector and $x$ be the input vector (a word embedding or feature vector), we can also represent the weighted sum $z$ using vector notation:
  - $z = w \cdot x + b$

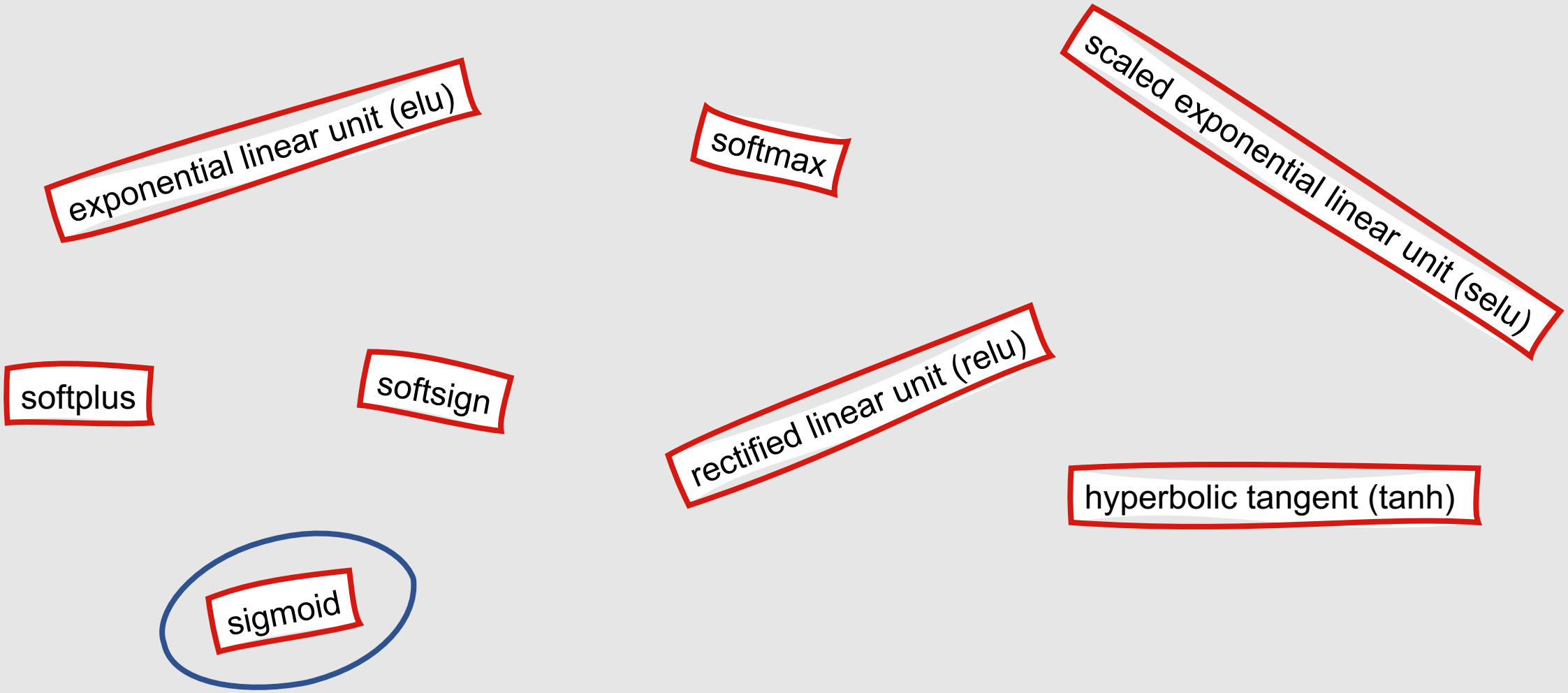
# Computational Units

- The equation thus far still computes a **linear function** of $x$!
- Recall that 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!



# Sigmoid Activation

- Recall the sigmoid function from Word2Vec:
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
- The sigmoid activation simply sets $x$ to the linear combination of weights and bias $z$
  - $y = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-w \cdot x+b}}$

# Advantages of Sigmoid Activation

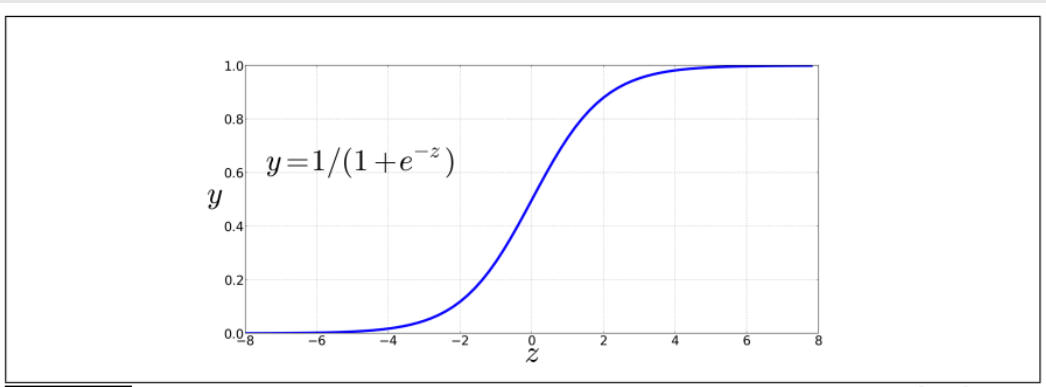
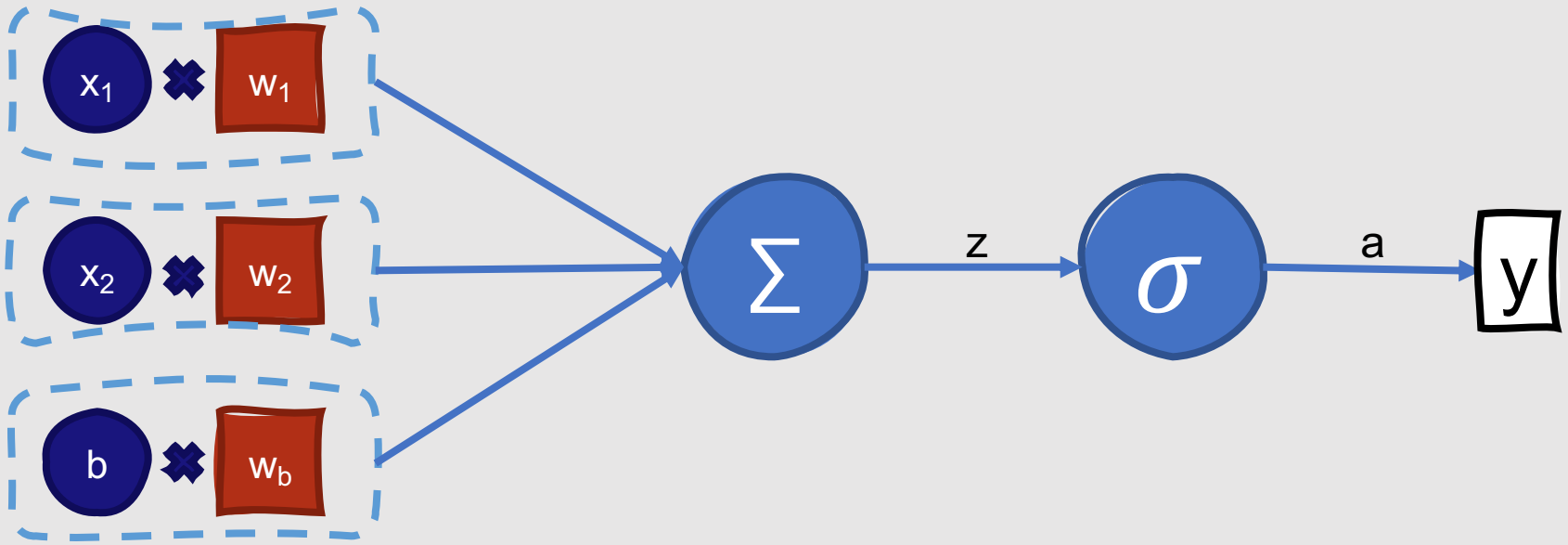


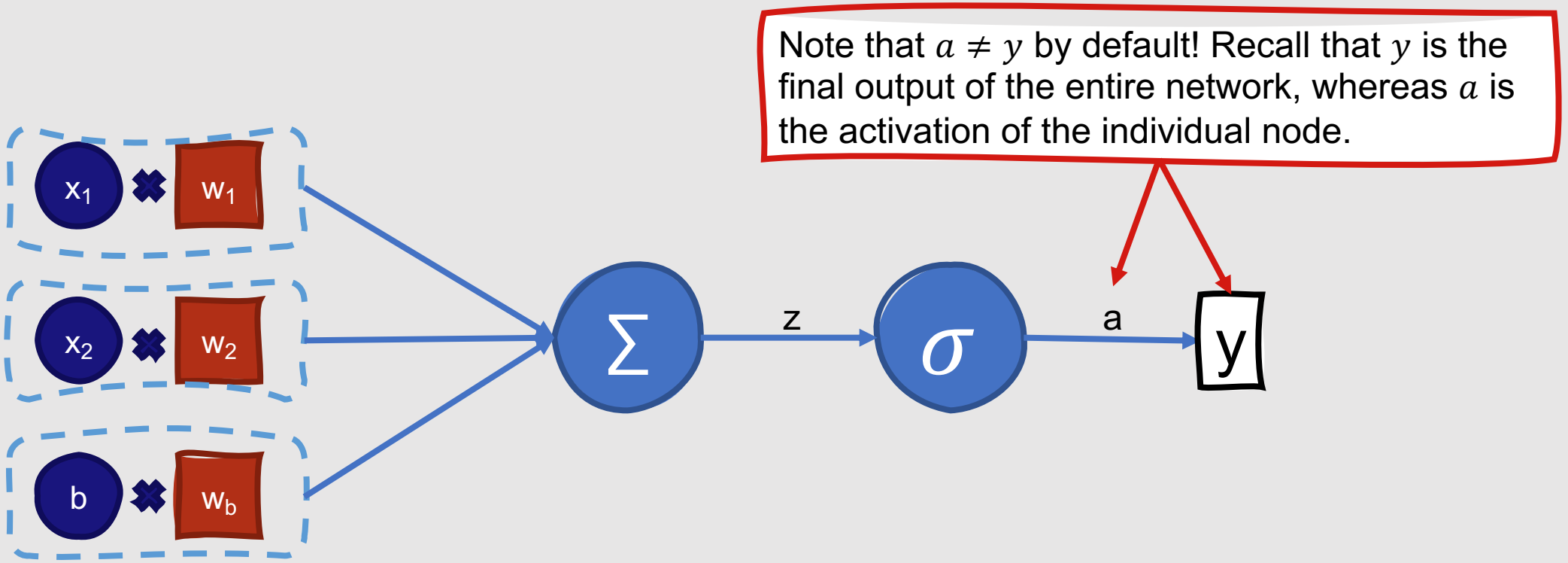

**Figure 7.1** The sigmoid function takes a real value and maps it to the range $[0,1]$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

- Maps the unit's output to a [0,1] range
  - Squashes outliers
- Differentiable (useful for learning)
  - You need to be able to differentiate functions to optimize (minimize/maximize) them

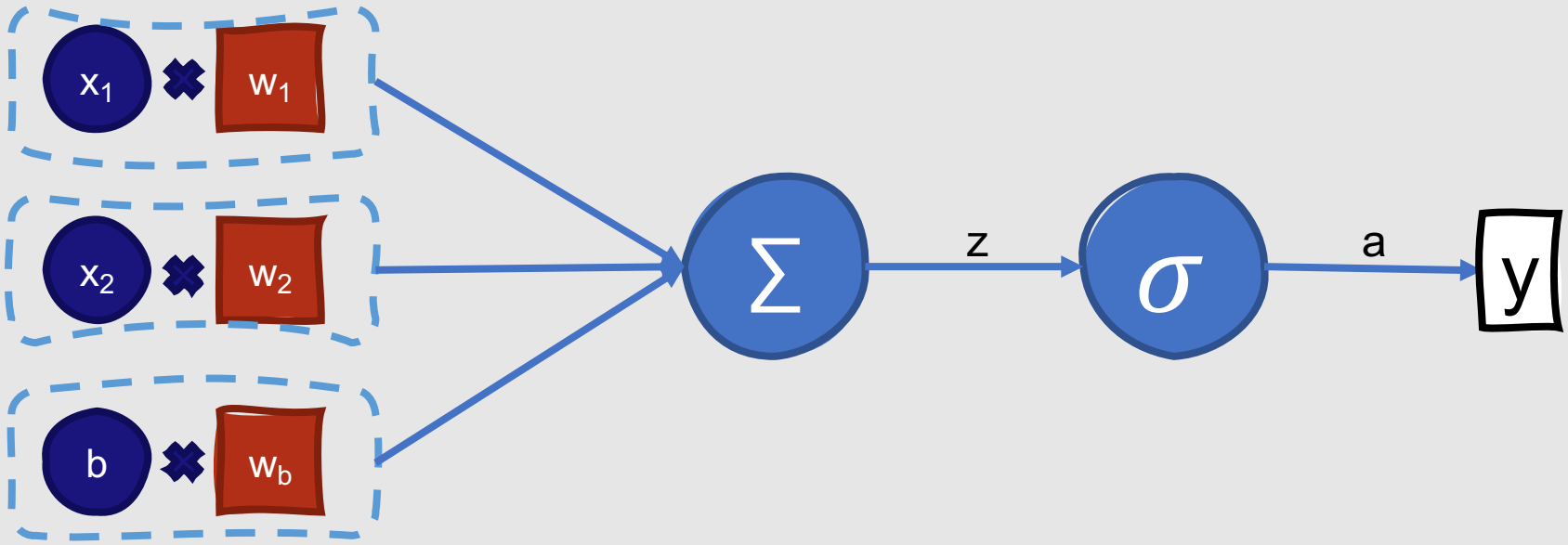
# Computational Unit with Sigmoid Activation



# Computational Unit with Sigmoid Activation



# Example: Computational Unit with Sigmoid Activation

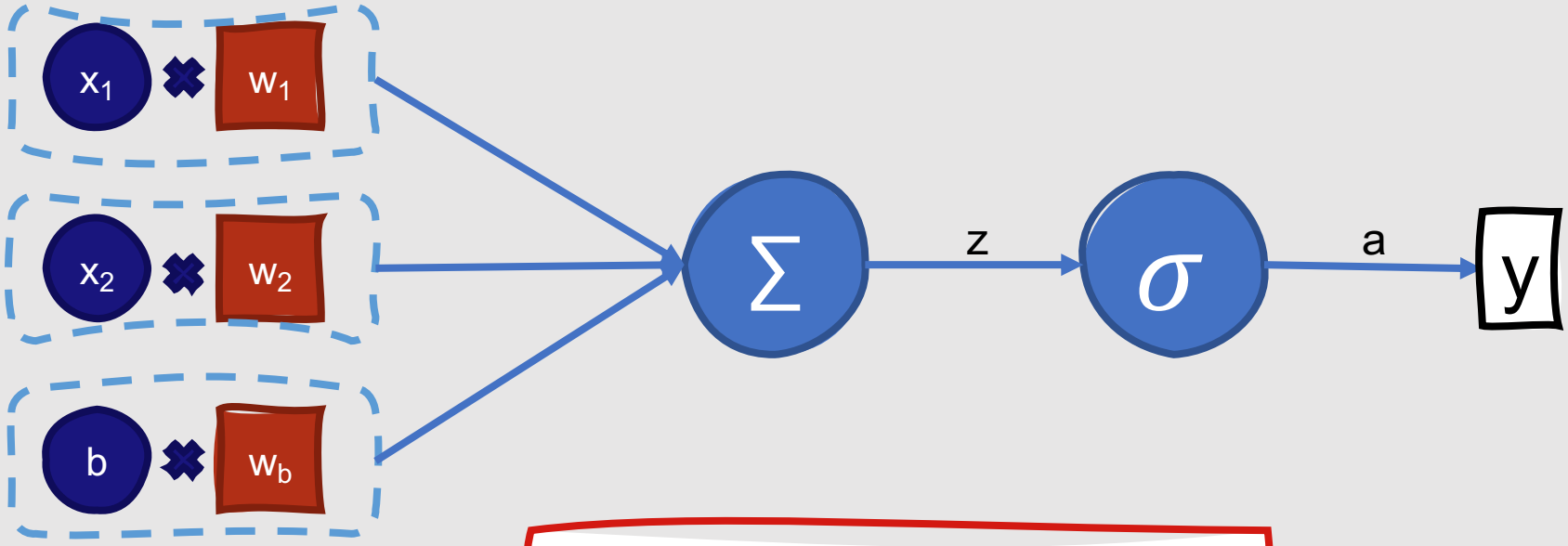



Input: “pumpkin spice latte”

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

Bias: 1.0

# Example: Computational Unit with Sigmoid Activation




Input: “pumpkin spice latte”

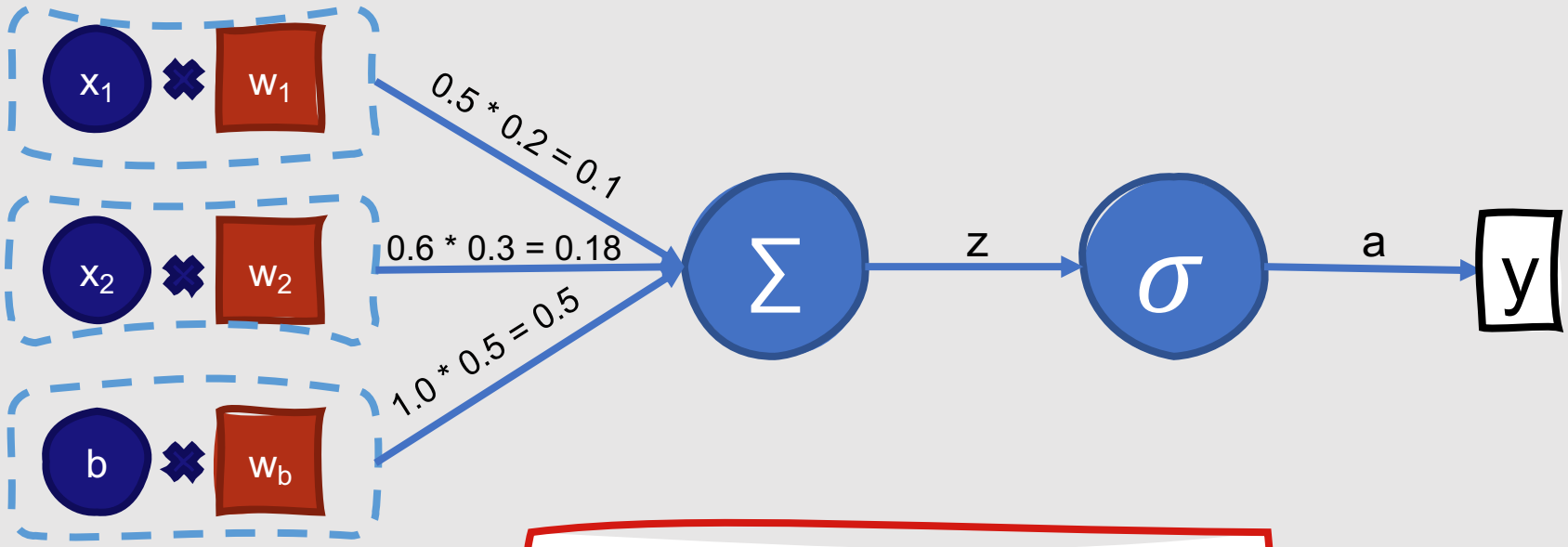
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with Sigmoid Activation




Input: "pumpkin spice latte"

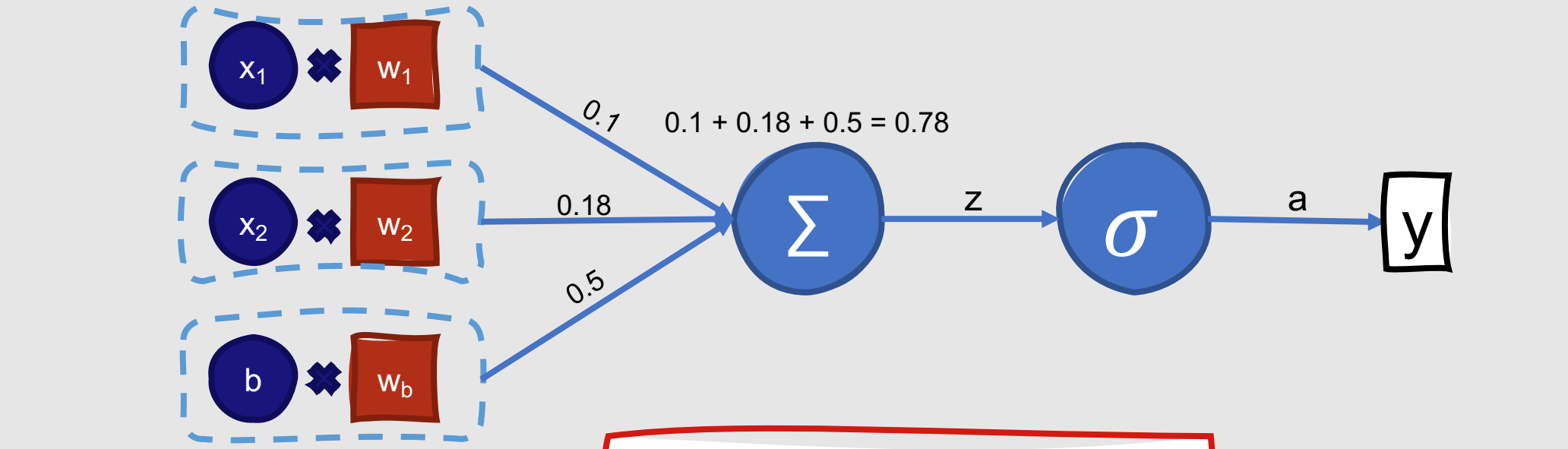
Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte")

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with Sigmoid Activation




Input: “pumpkin spice latte”

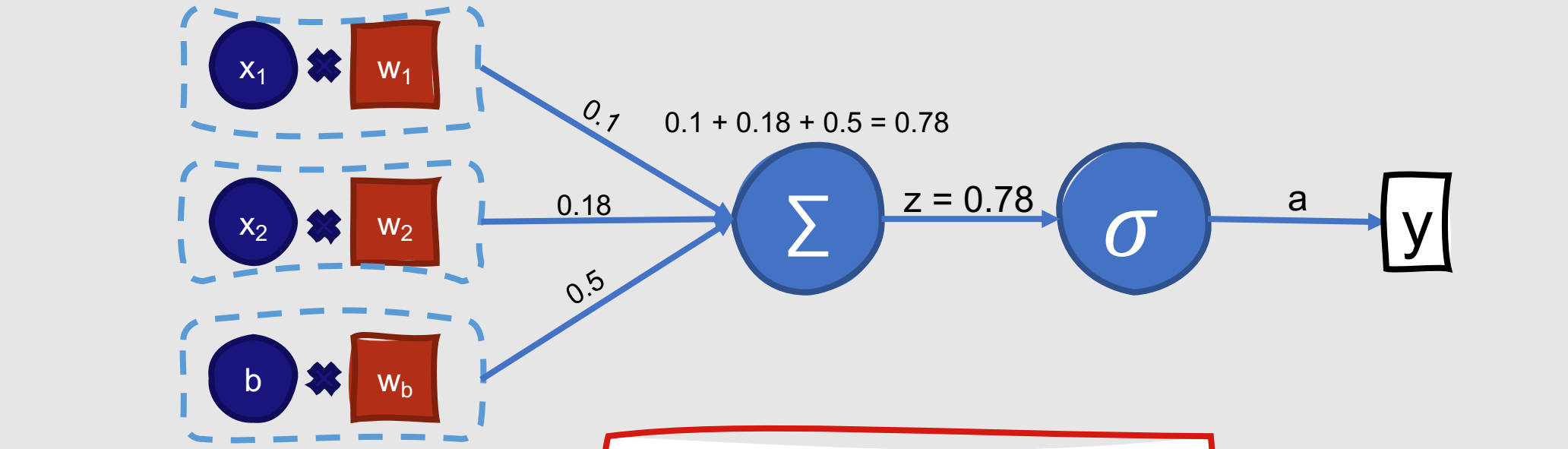
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with Sigmoid Activation




Input: “pumpkin spice latte”

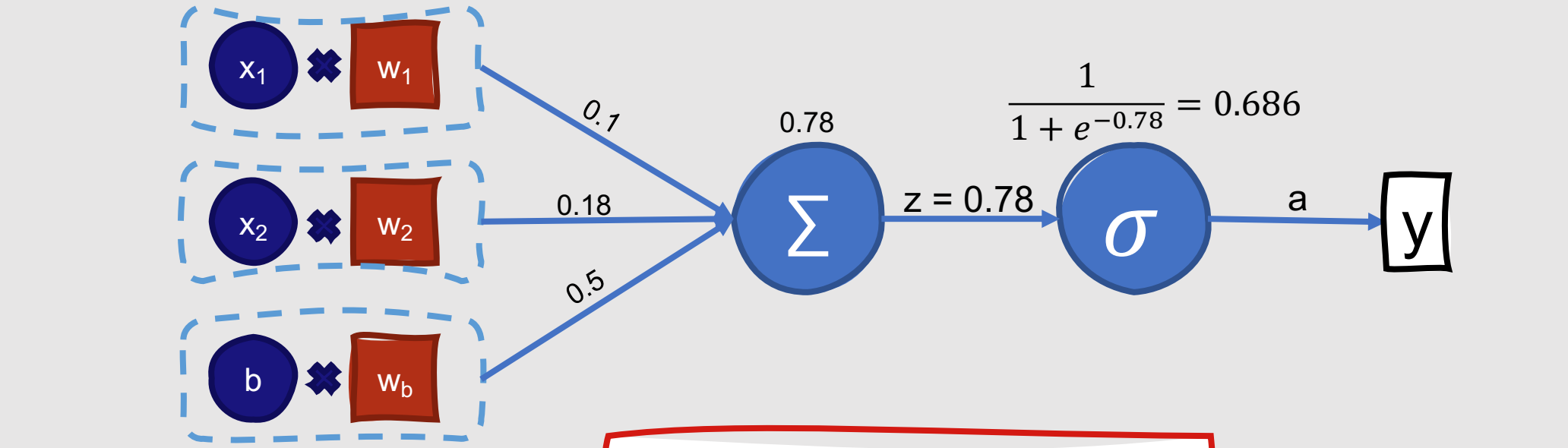
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with Sigmoid Activation




Input: "pumpkin spice latte"

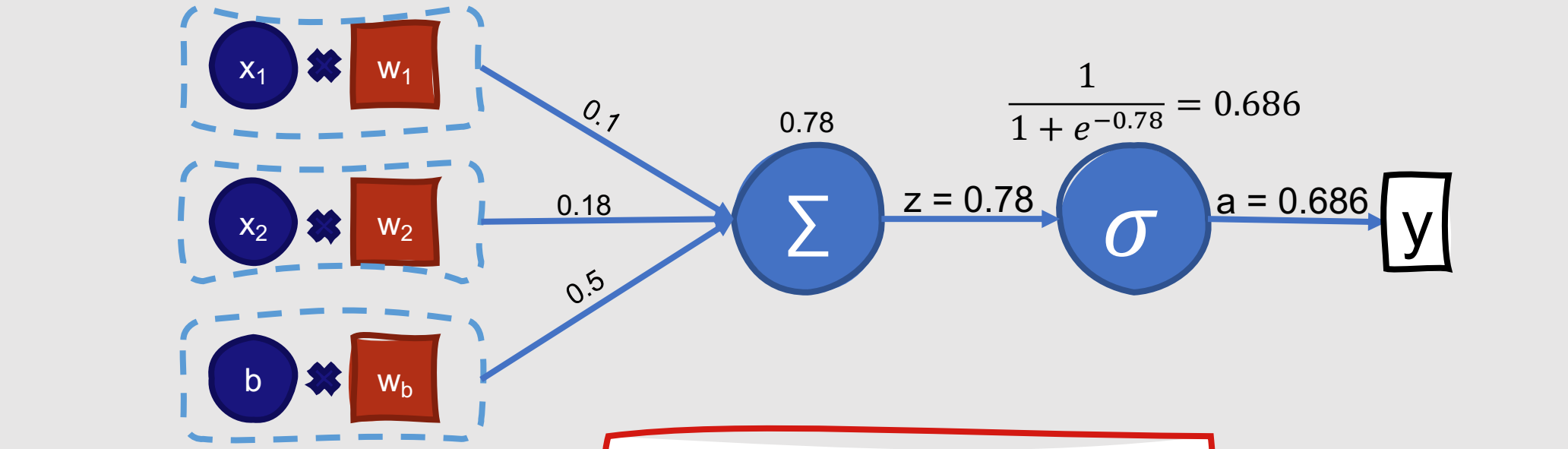
Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte")

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with Sigmoid Activation




Input: “pumpkin spice latte”

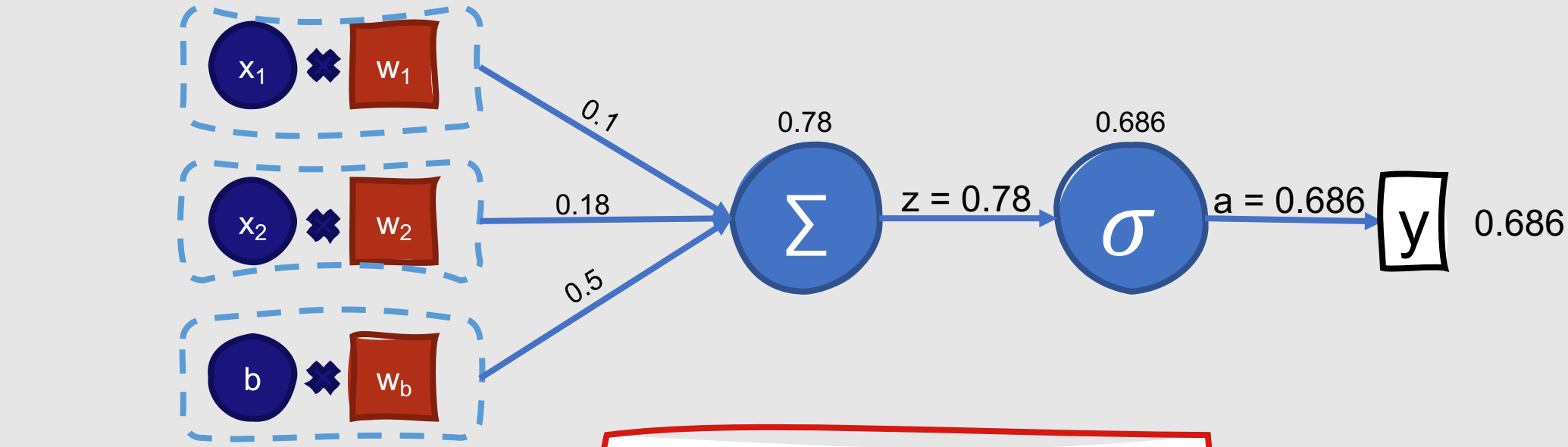
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with Sigmoid Activation




Input: “pumpkin spice latte”

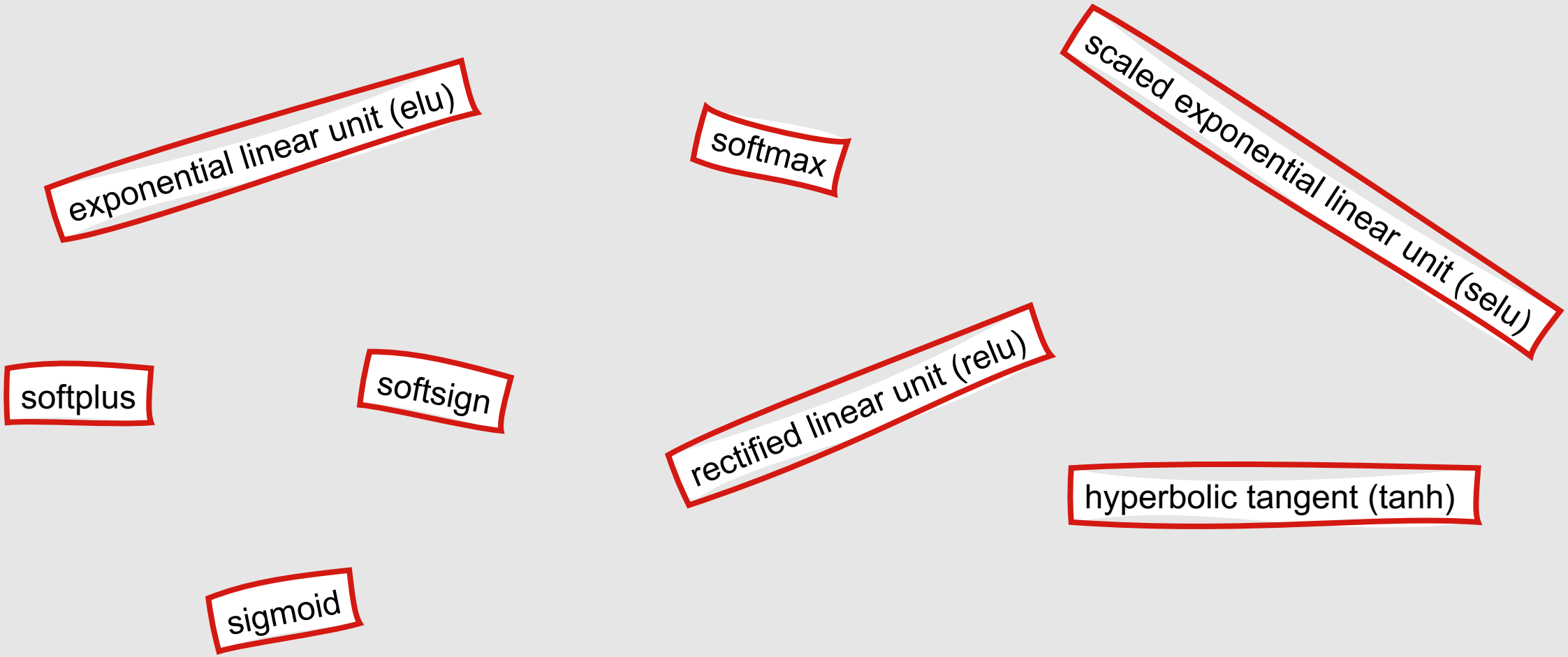
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

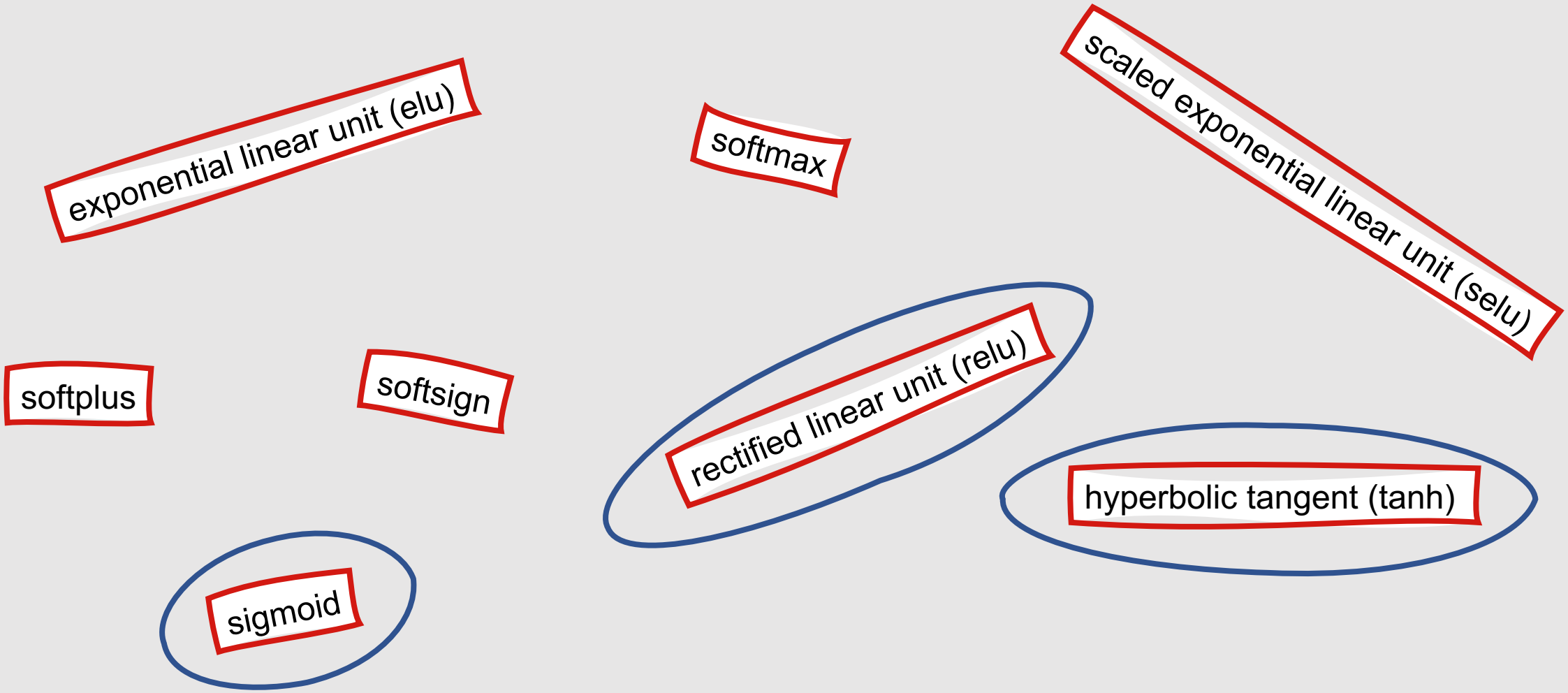
Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

# Remember, there are many different activation functions!



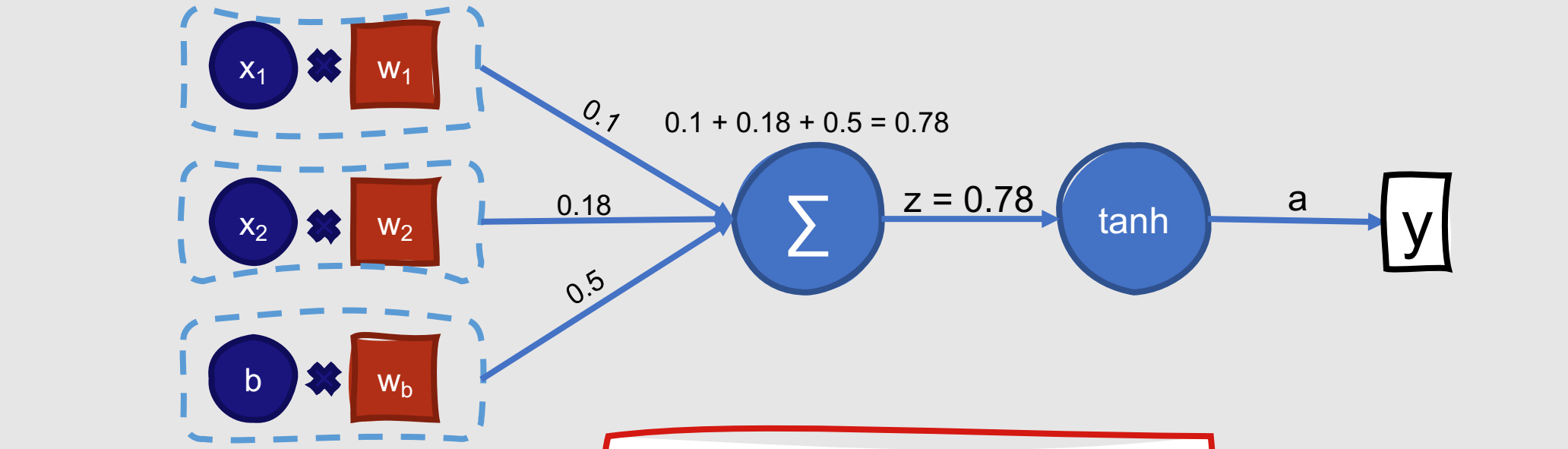
# Particularly Common Activation Functions



# 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: “pumpkin spice latte”

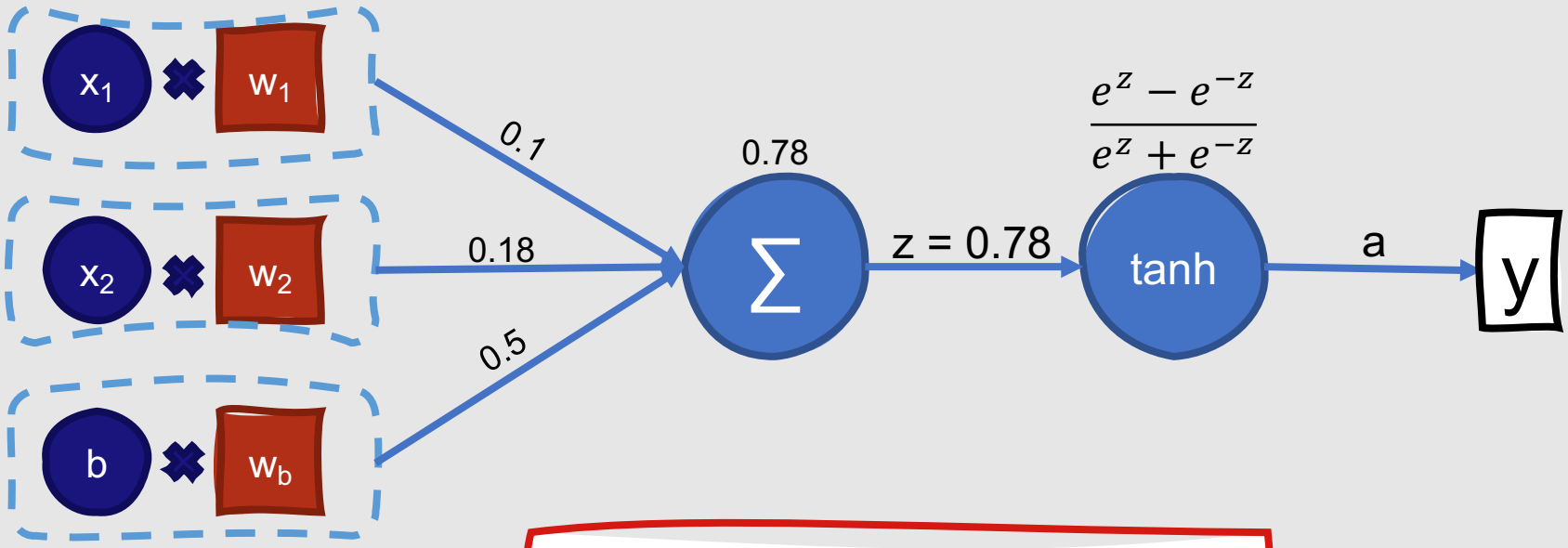
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with tanh Activation




Input: “pumpkin spice latte”

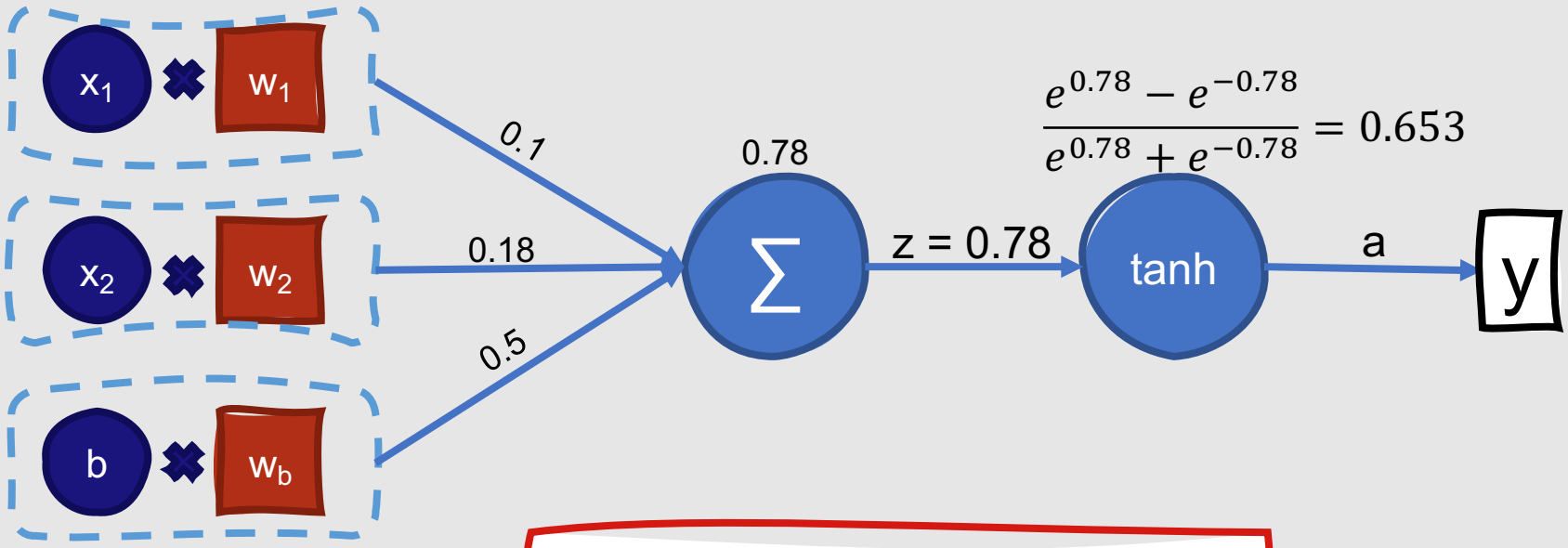
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with tanh Activation




Input: “pumpkin spice latte”

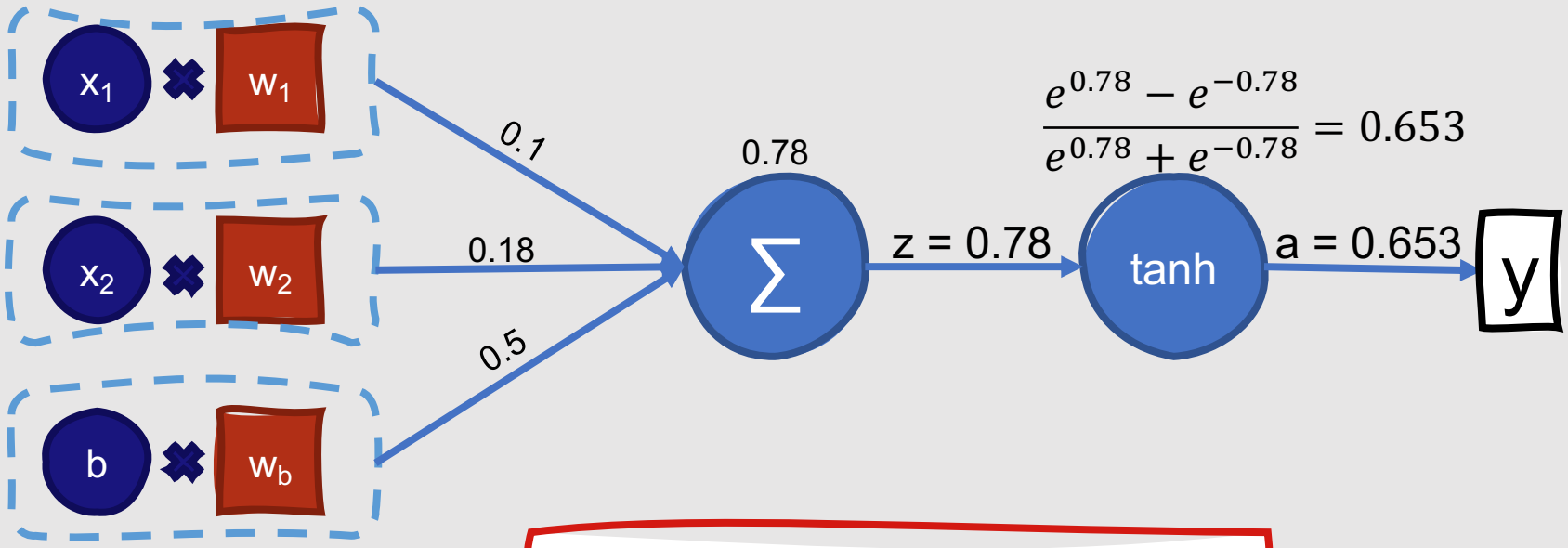
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with tanh Activation




Input: “pumpkin spice latte”

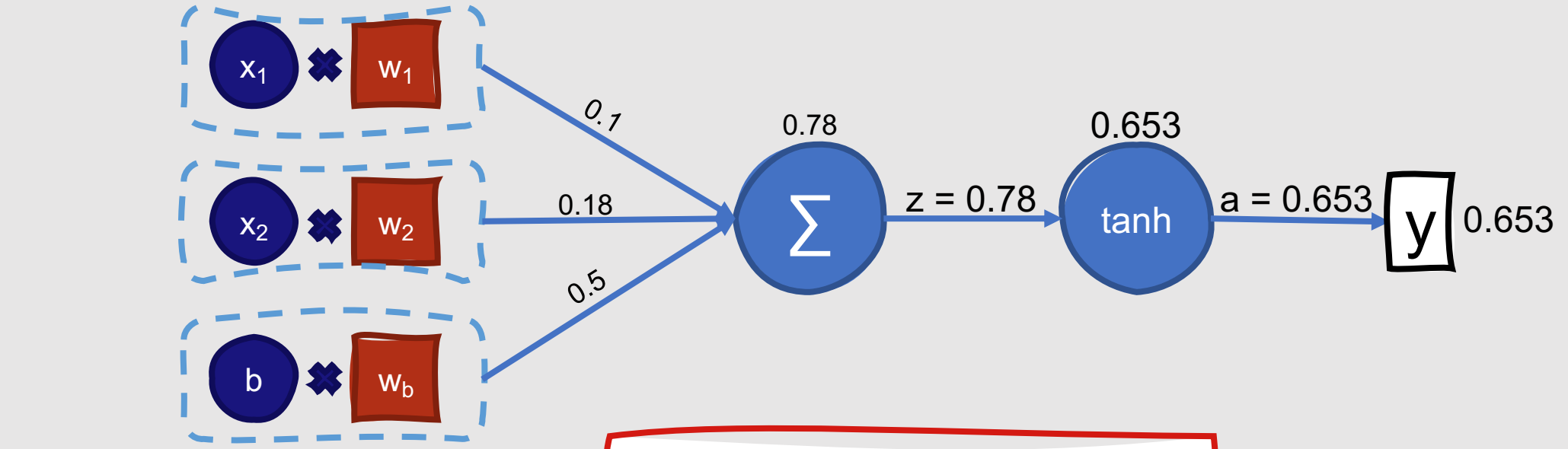
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with tanh Activation




Input: “pumpkin spice latte”

Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

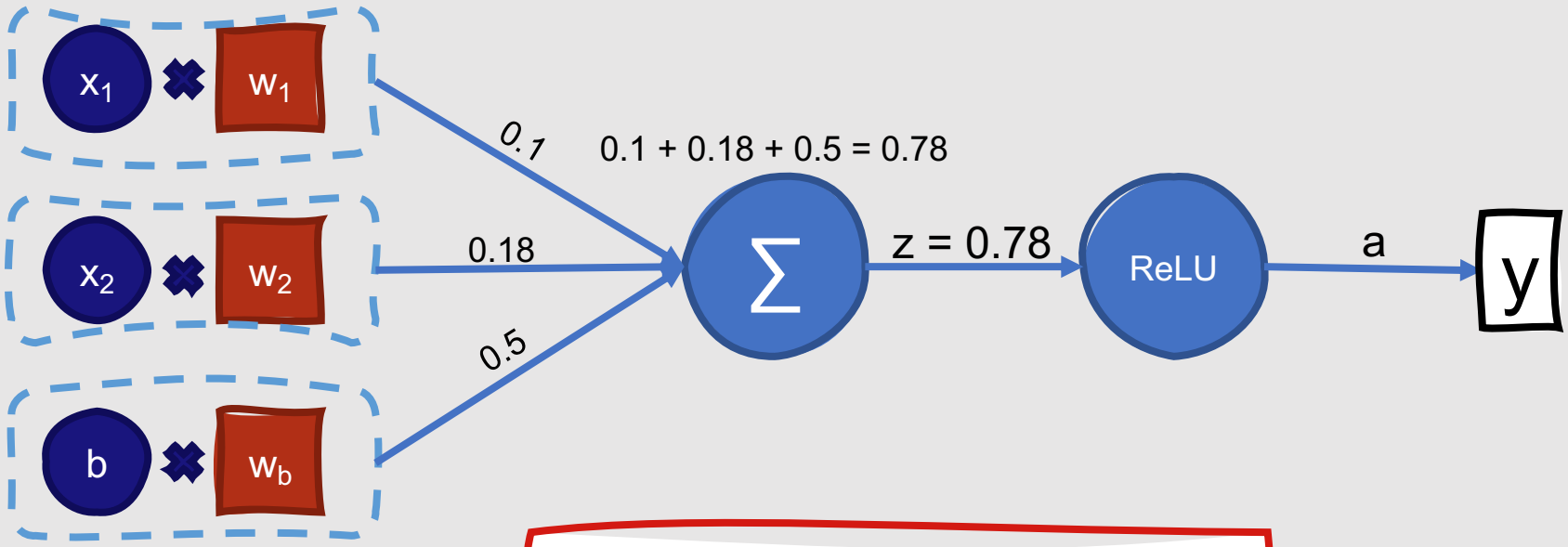
Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

# Activation: ReLU

- Ranges from 0 to $\infty$
- 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: “pumpkin spice latte”

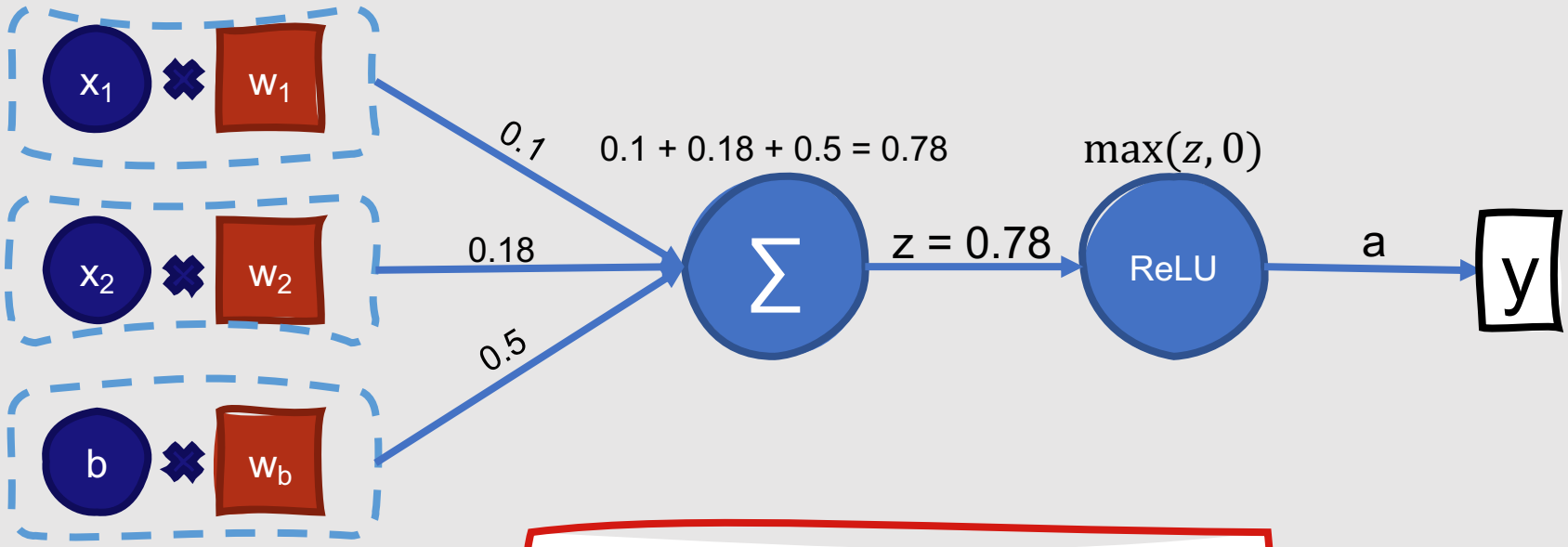
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with ReLU Activation




Input: “pumpkin spice latte”

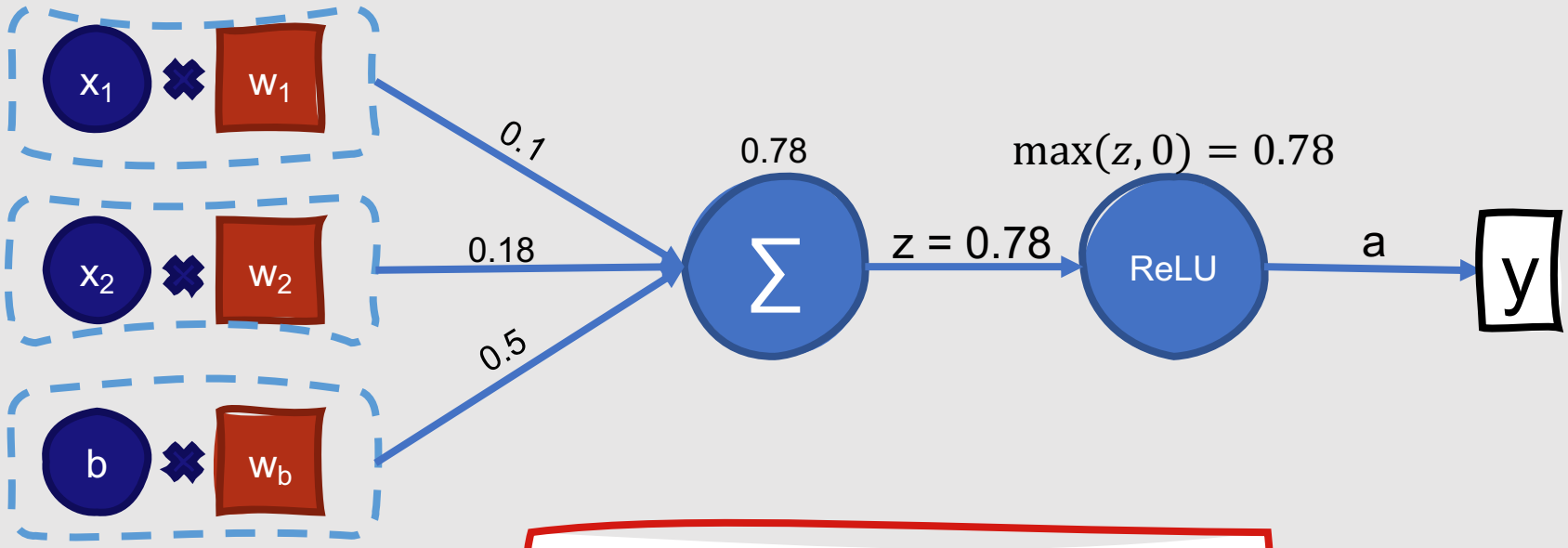
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with ReLU Activation




Input: “pumpkin spice latte”

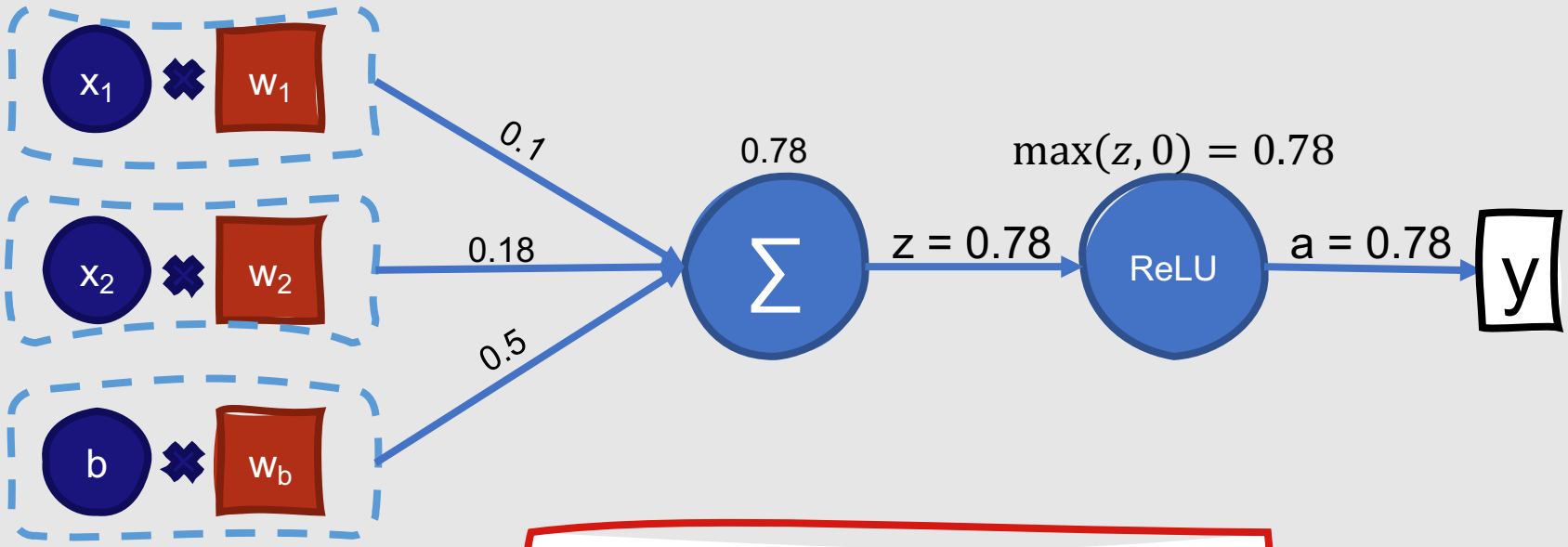
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with ReLU Activation




Input: “pumpkin spice latte”

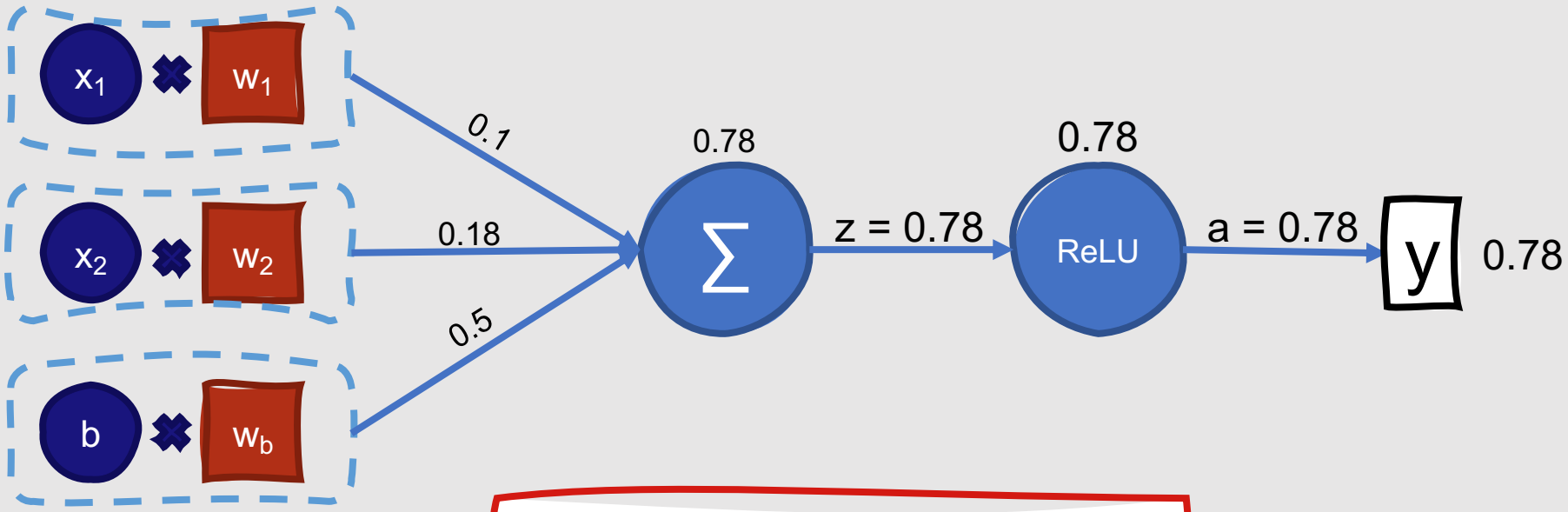
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

# Example: Computational Unit with ReLU Activation




Input: “pumpkin spice latte”

Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[0.5, 0.6]

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

Bias: 1.0

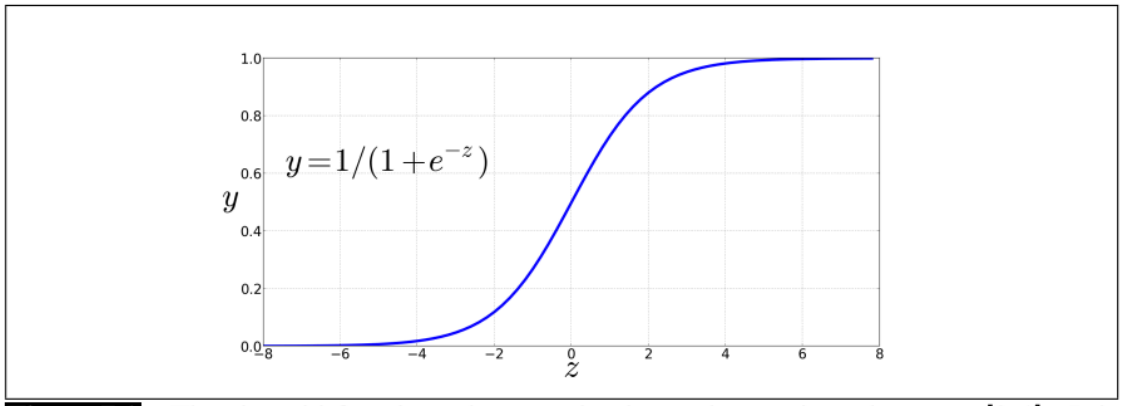



**Figure 7.1** The sigmoid function takes a real value and maps it to the range $[0,1]$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

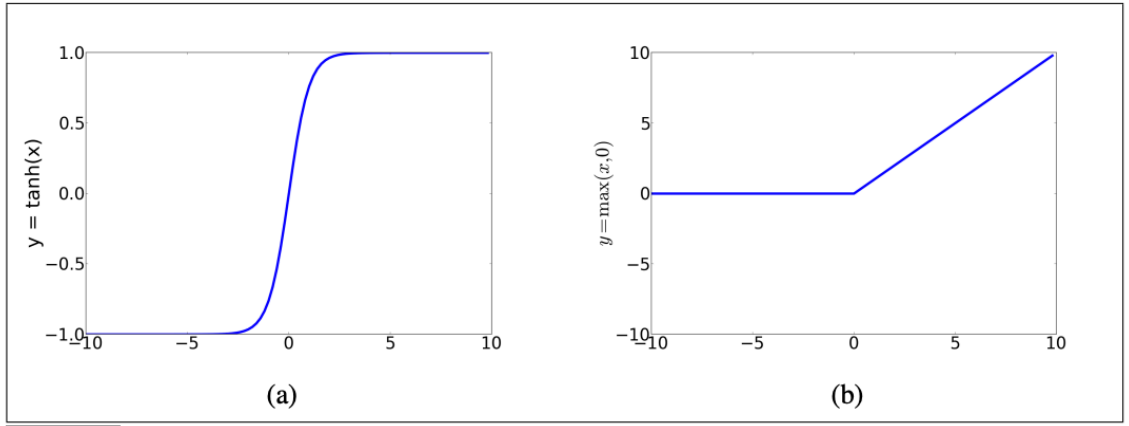



**Figure 7.3** The tanh and ReLU activation functions.

# Comparing sigmoid, tanh, and ReLU

# 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

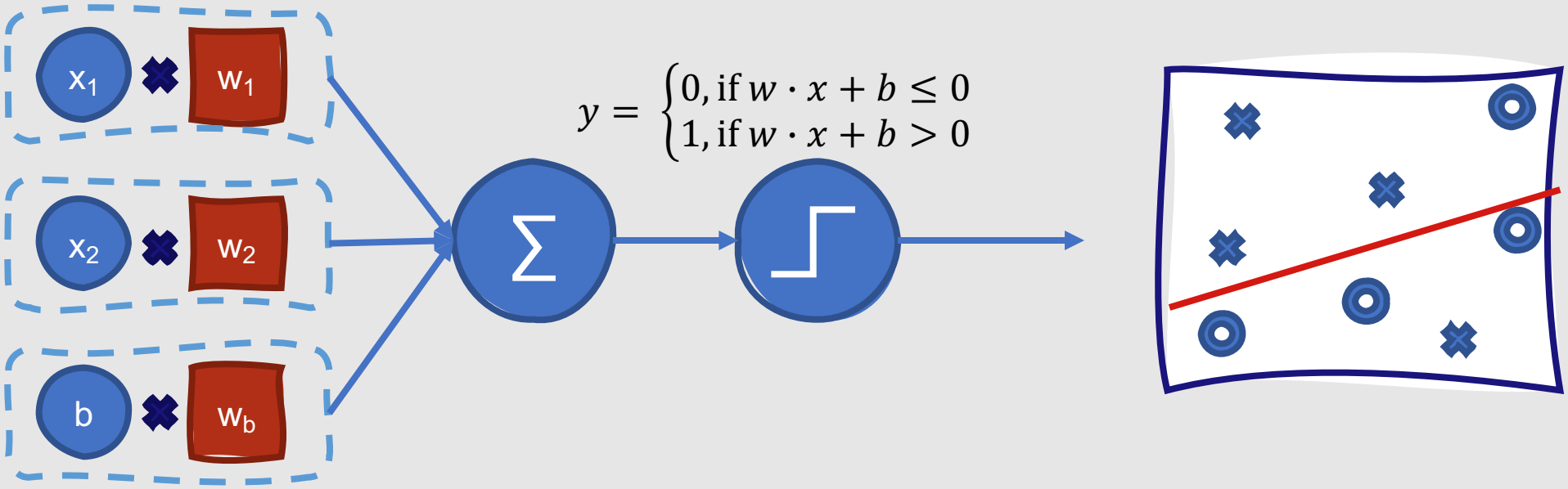
# The XOR Problem

Early example of why networks of computational units were necessary to solve some problems

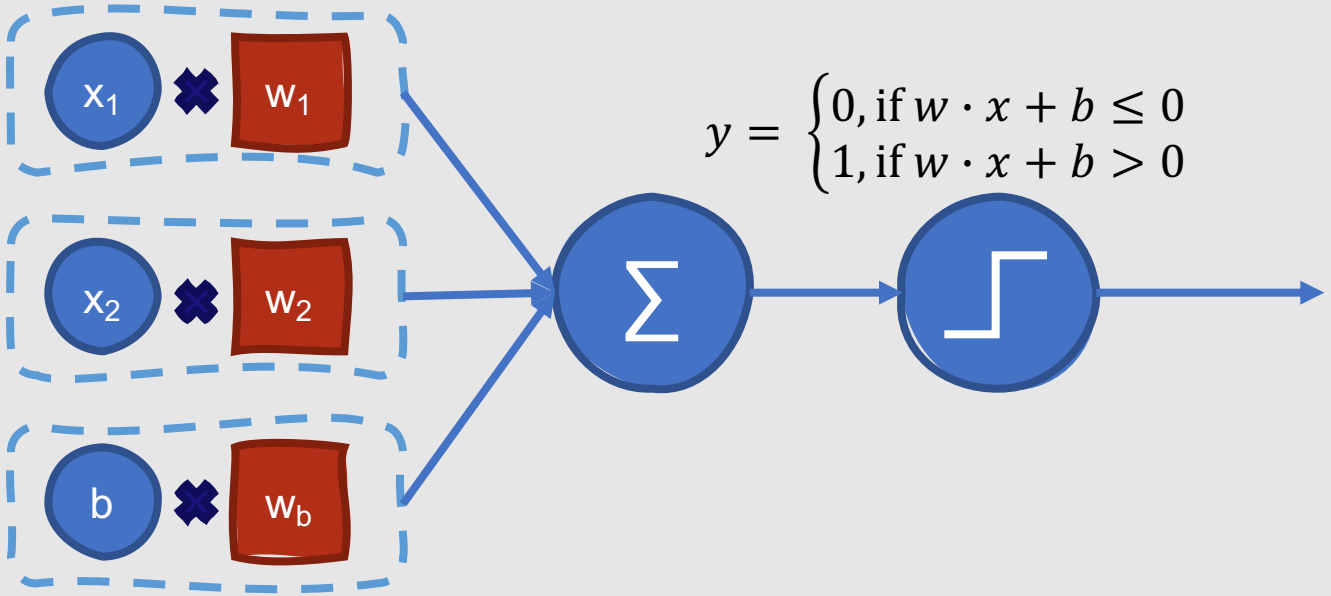
| AND | | | OR | | | XOR | | |
|---|---|---|---|---|---|---|---|---|
| x1 | x2 | y | x1 | x2 | y | x1 | x2 | y |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

# What is a perceptron?

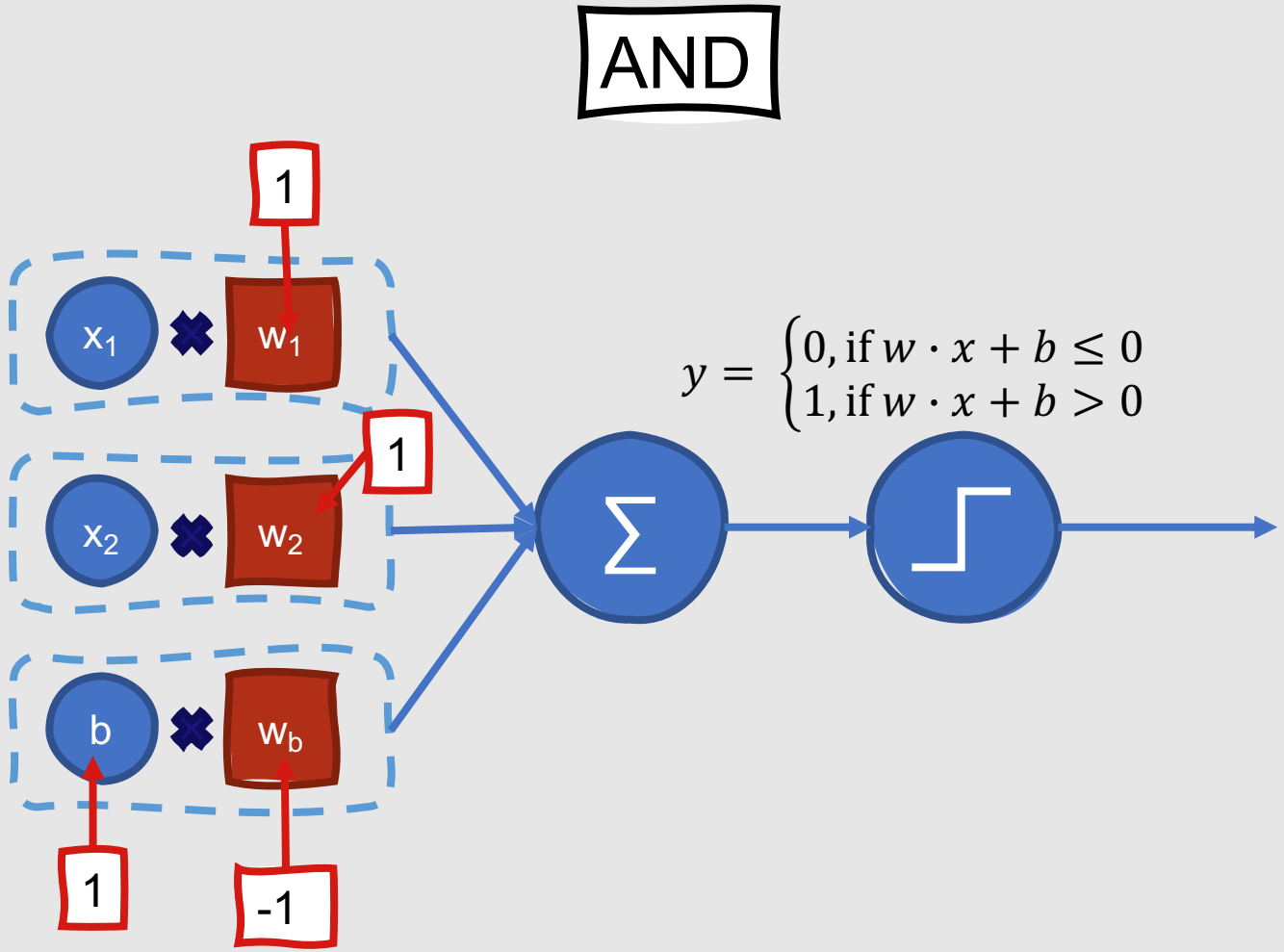
- A function that outputs a binary value based on whether or not the product of its inputs and associated weights surpasses a threshold
- Learns this threshold iteratively by trying to find the boundary that is best able to distinguish between data of different categories



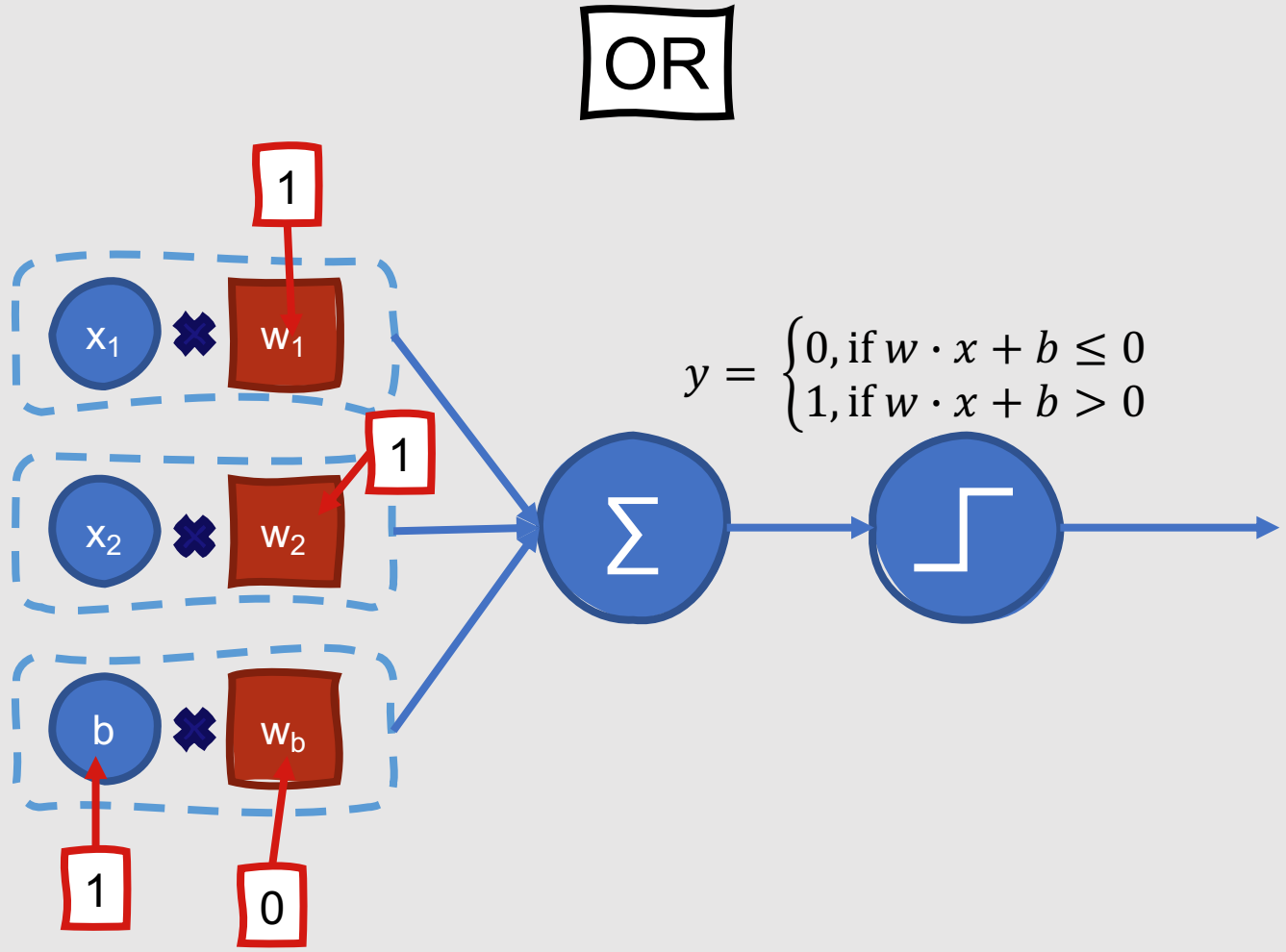
# It's easy to compute AND and OR using perceptrons.



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



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

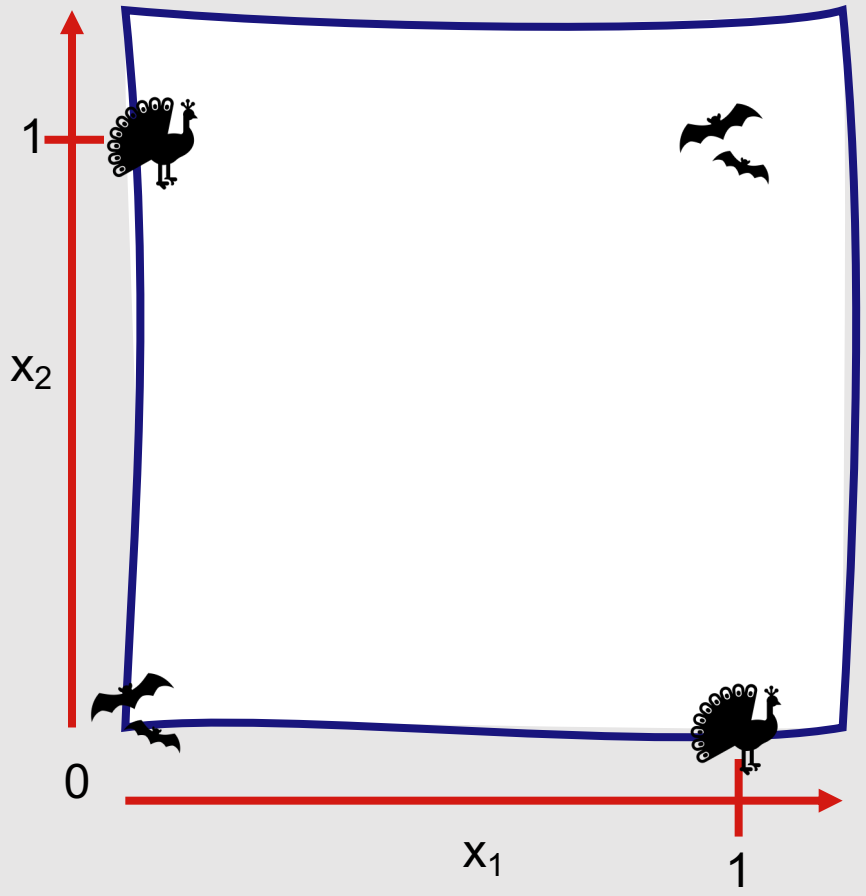


# However, it's impossible to compute XOR using a single perceptron.

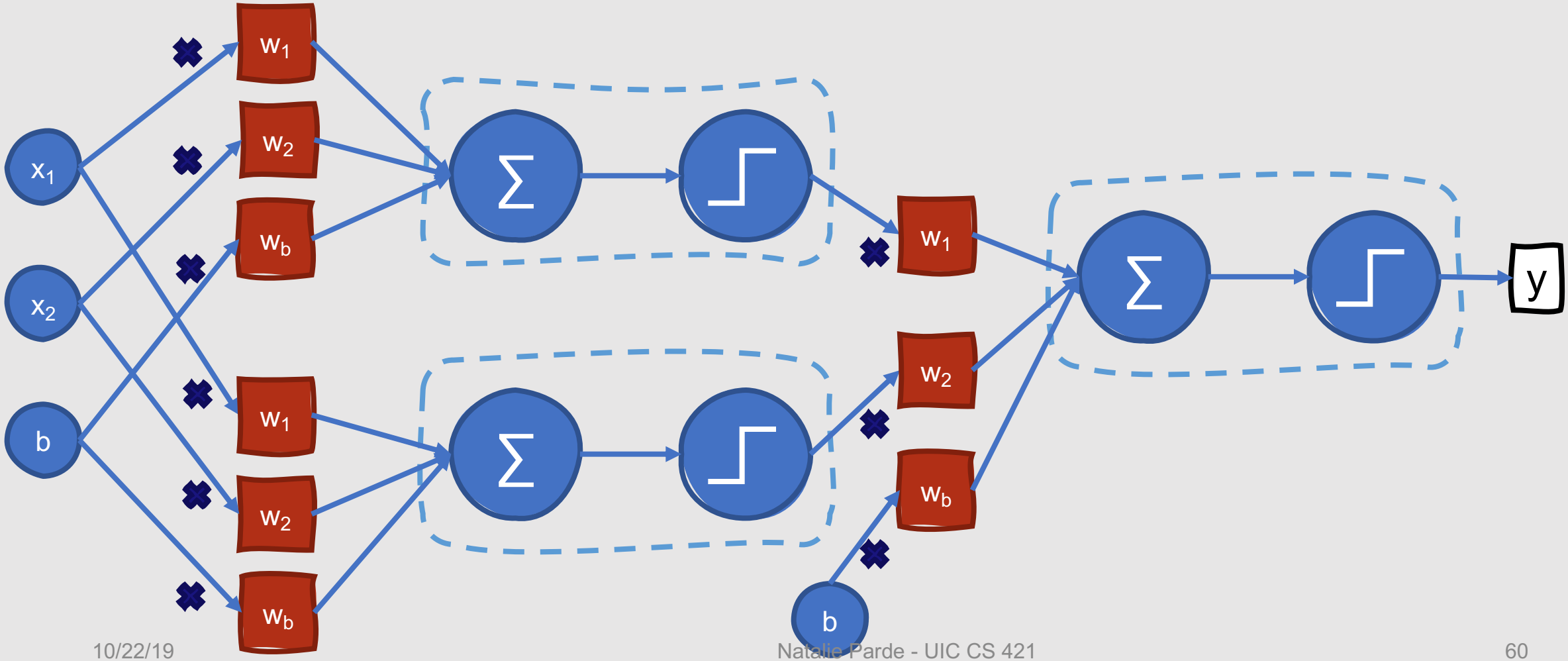
- Why?
  - Perceptrons are **linear classifiers**
- Linear classifiers learn a **decision boundary** that divides a dataset into two parts
  - All data on one side of the decision boundary is assigned a label of 0
  - All data on the other side of the decision boundary is assigned a label of 1
- However, it is impossible to draw a single line that separates the positive and negative cases of XOR
  - XOR is not a **linearly separable function**

# XOR Cases

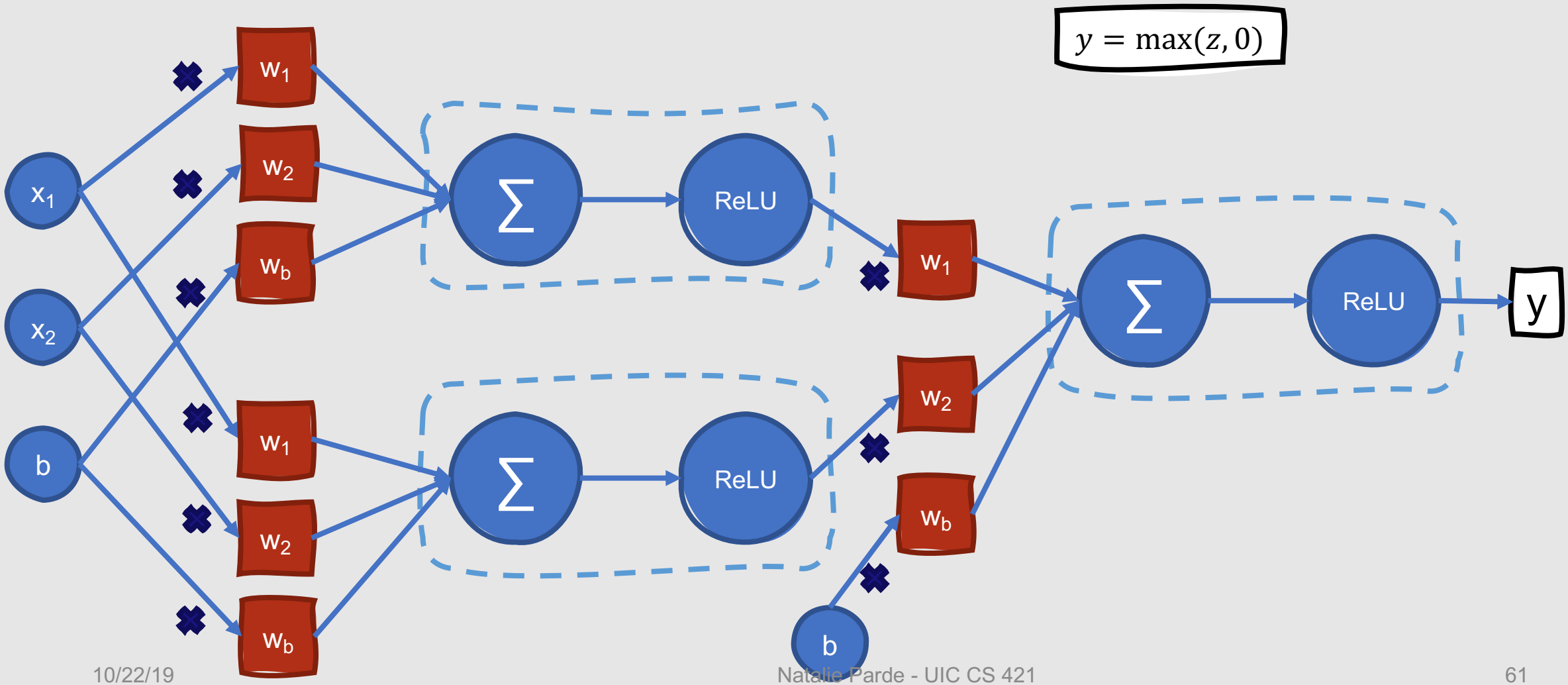
| AND | | | OR | | | XOR | | |
|---|---|---|---|---|---|---|---|---|
| x1 | x2 | y | x1 | x2 | y | x1 | x2 | y |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |



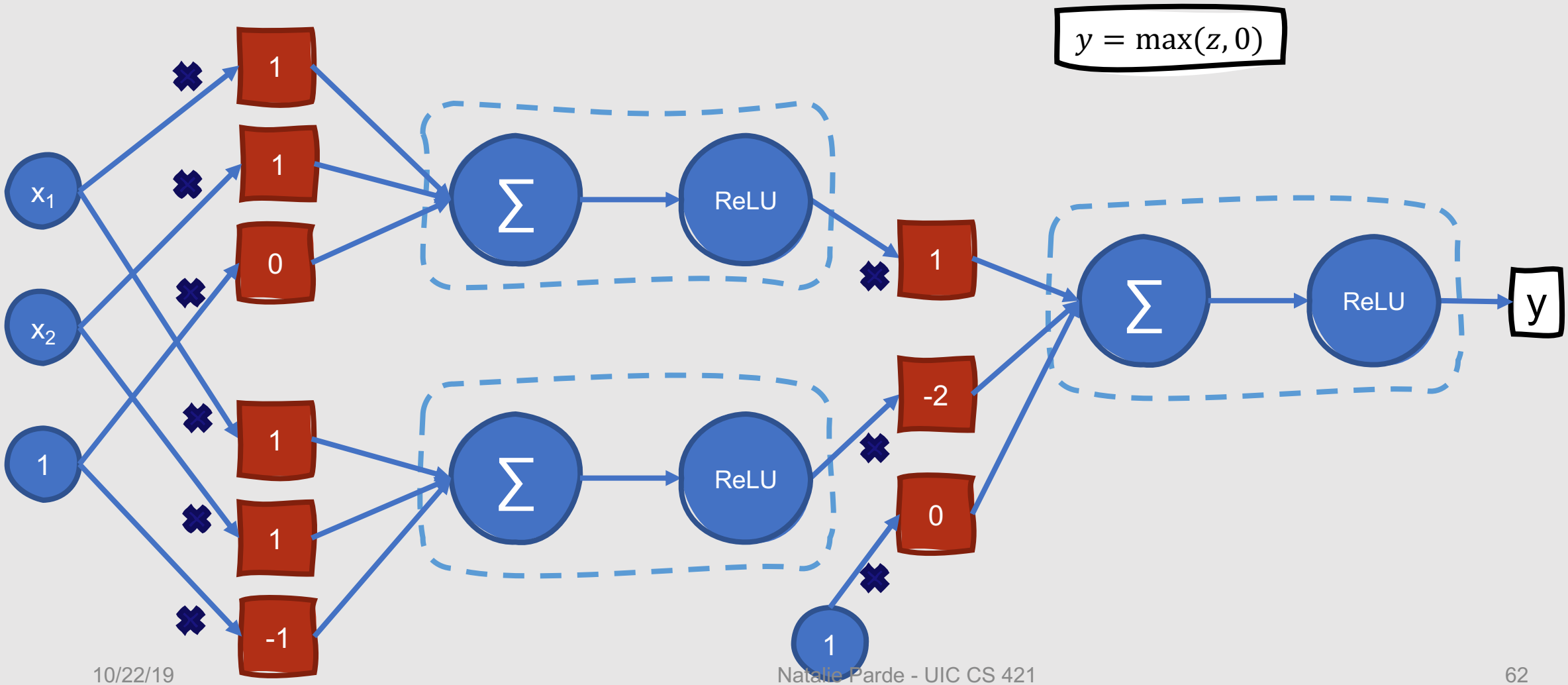
# The only way to compute XOR is by combining perceptrons!



# However, since XOR is not a linearly separable function, we'll have to use a nonlinear activation.

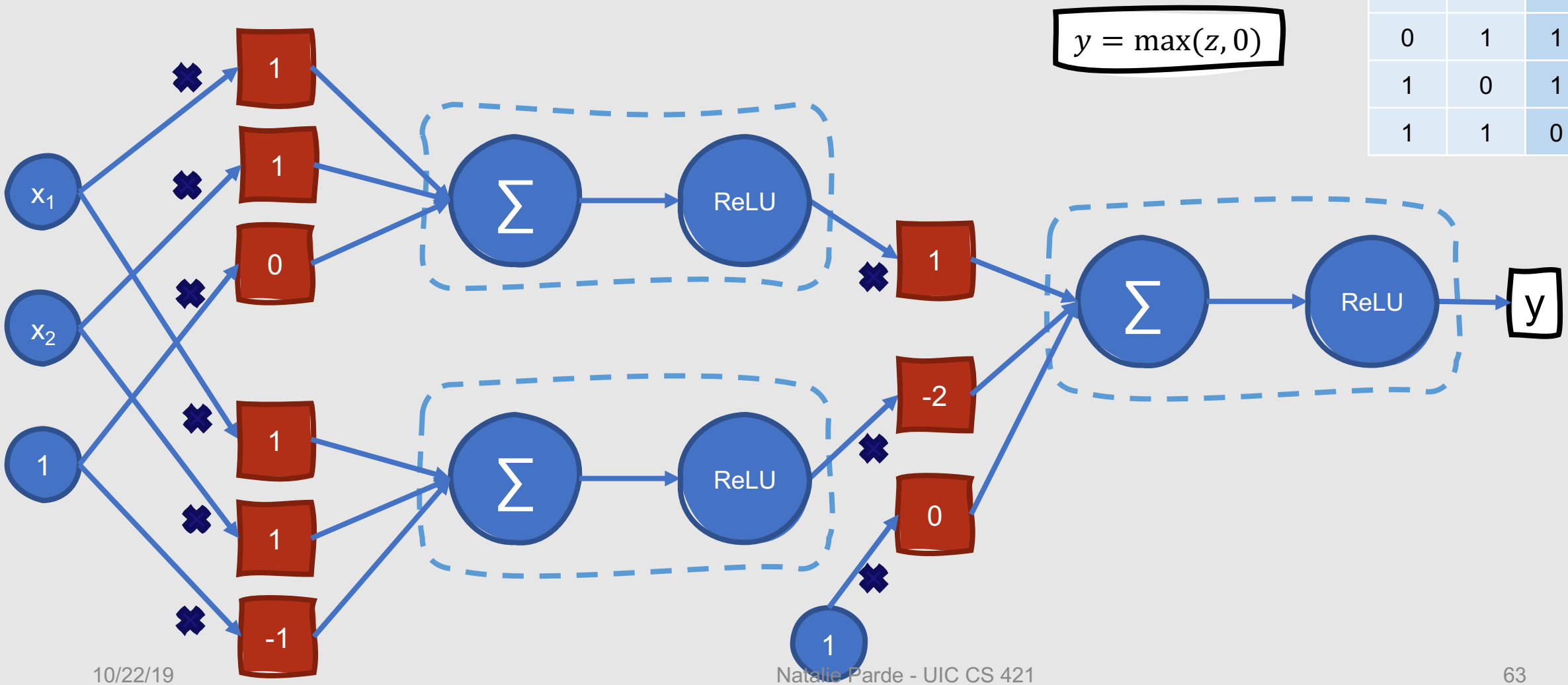


# The resulting neural network, with the weights below, can successfully solve XOR.



# Truth Table Examples: XOR

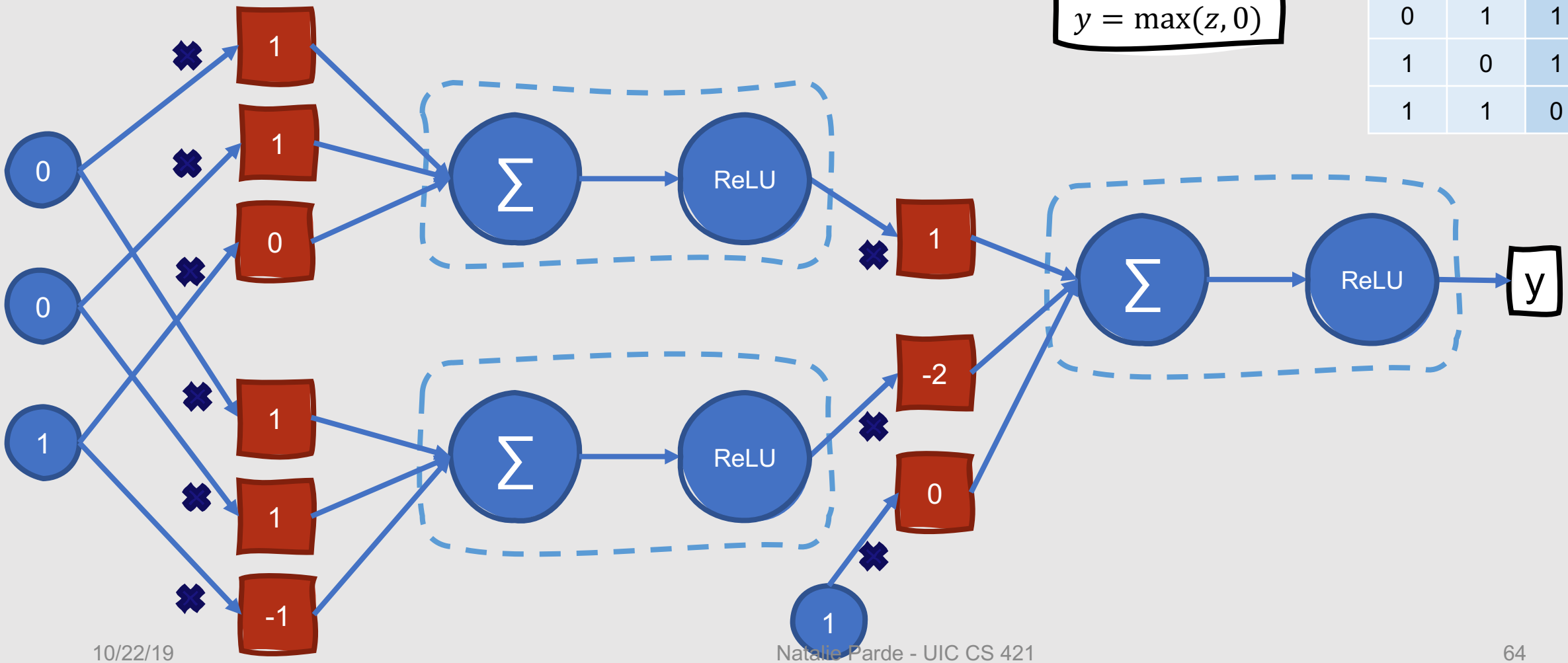
| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

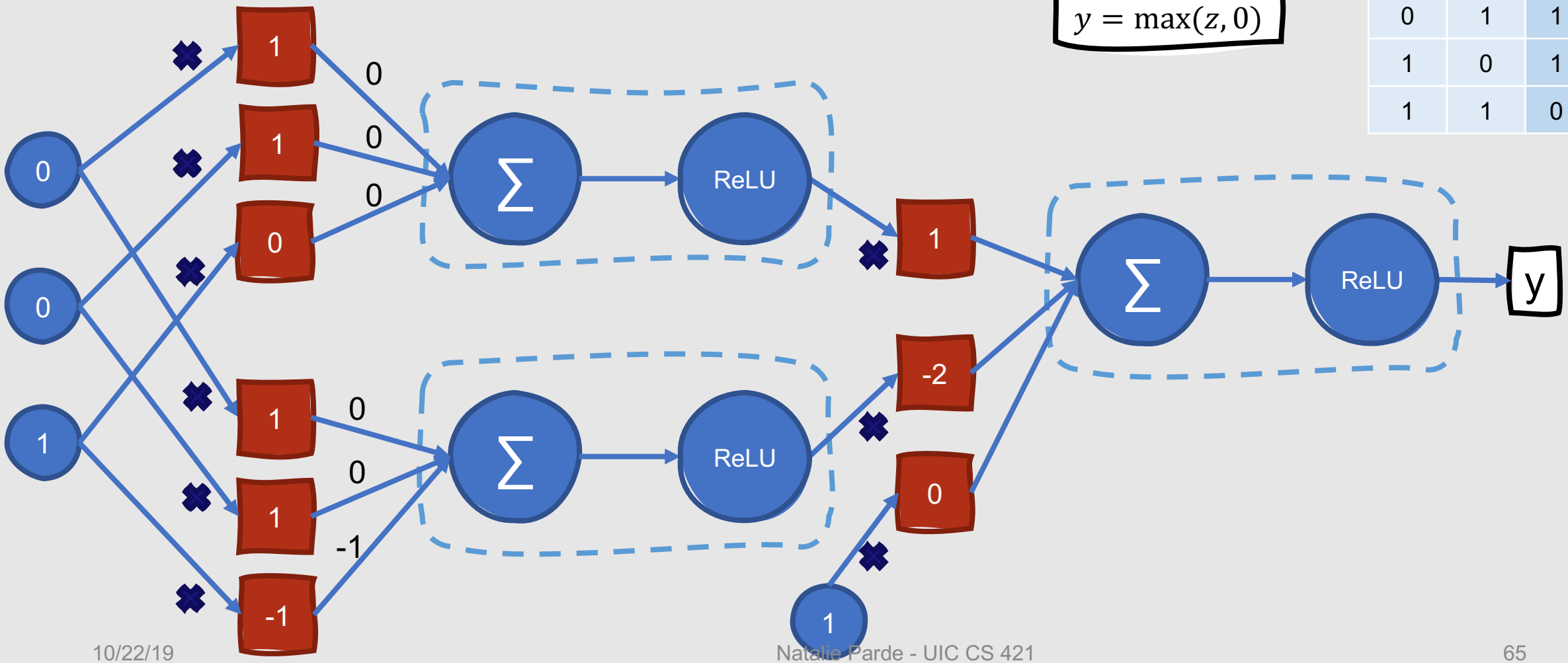
$y = \max(z, 0)$



# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

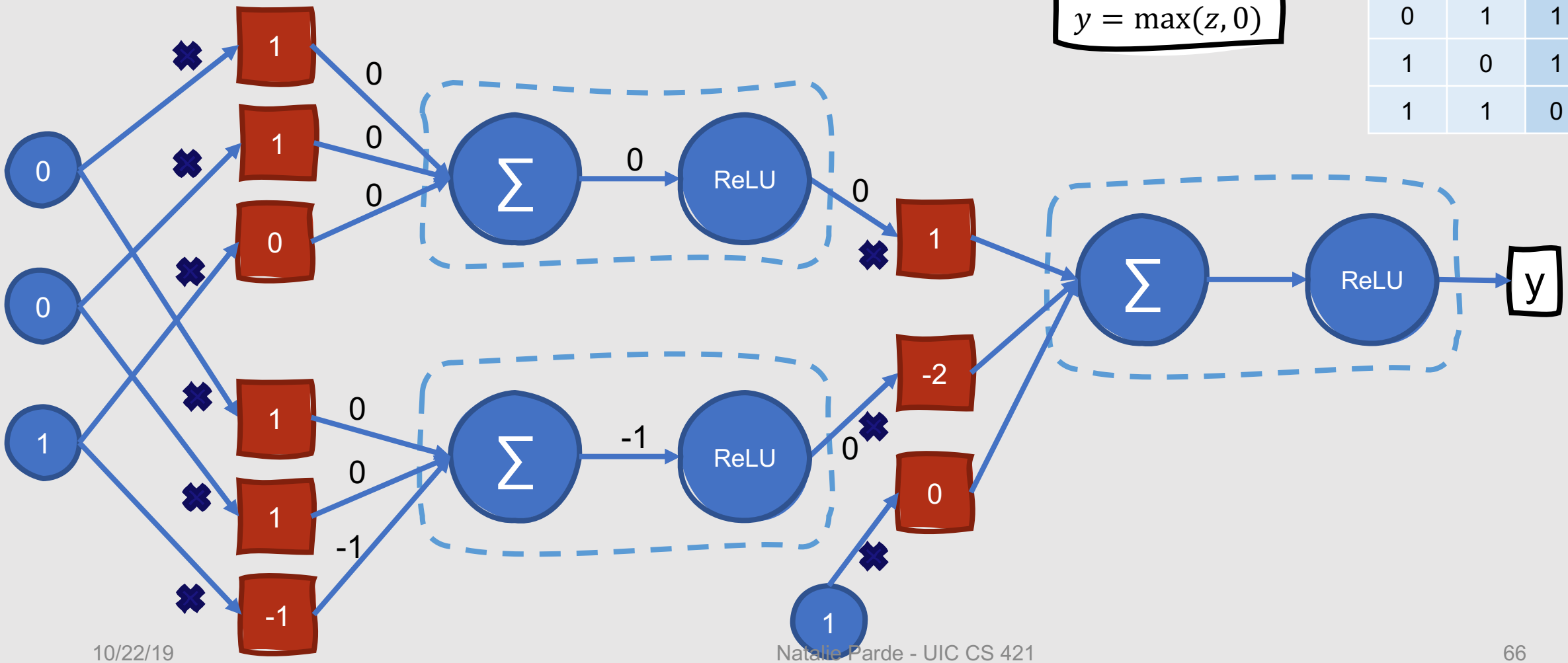
$y = \max(z, 0)$



# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

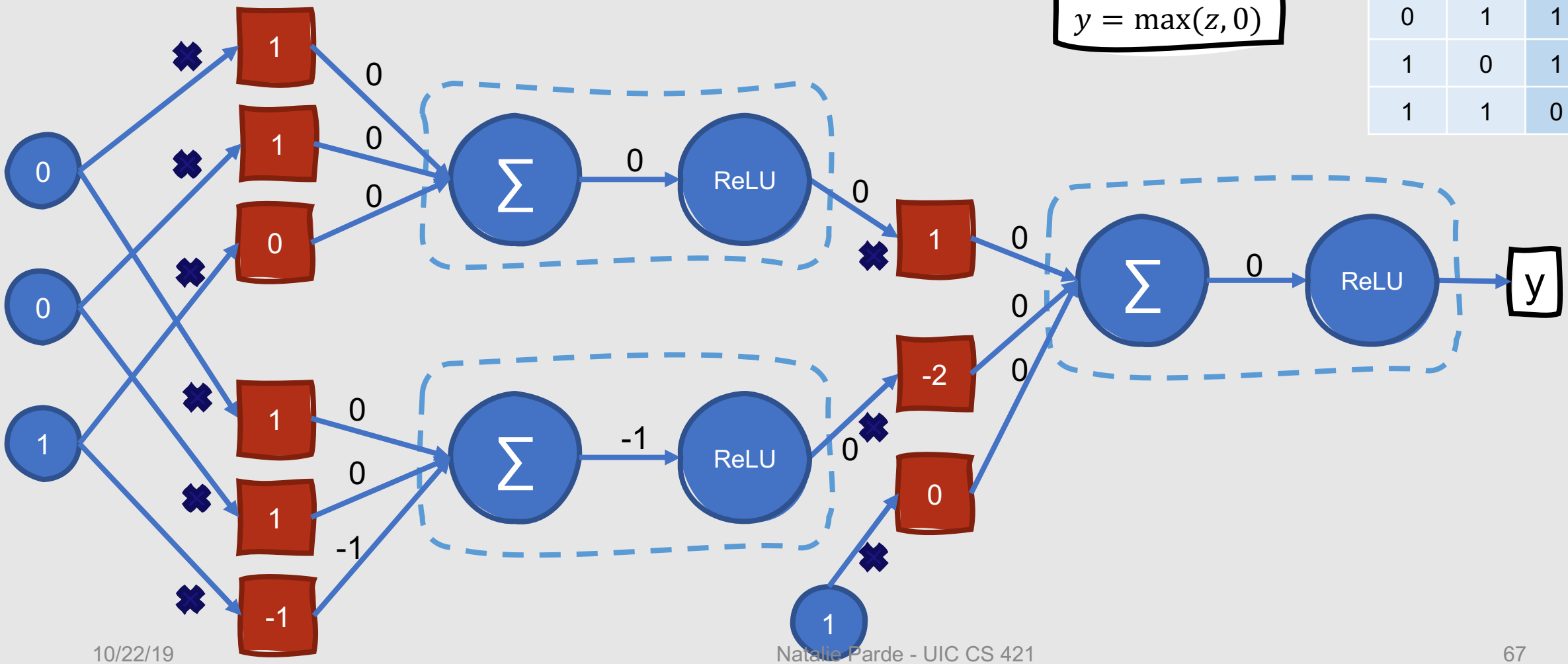
$$y = \max(z, 0)$$



# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| **0** | **0** | **0** |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

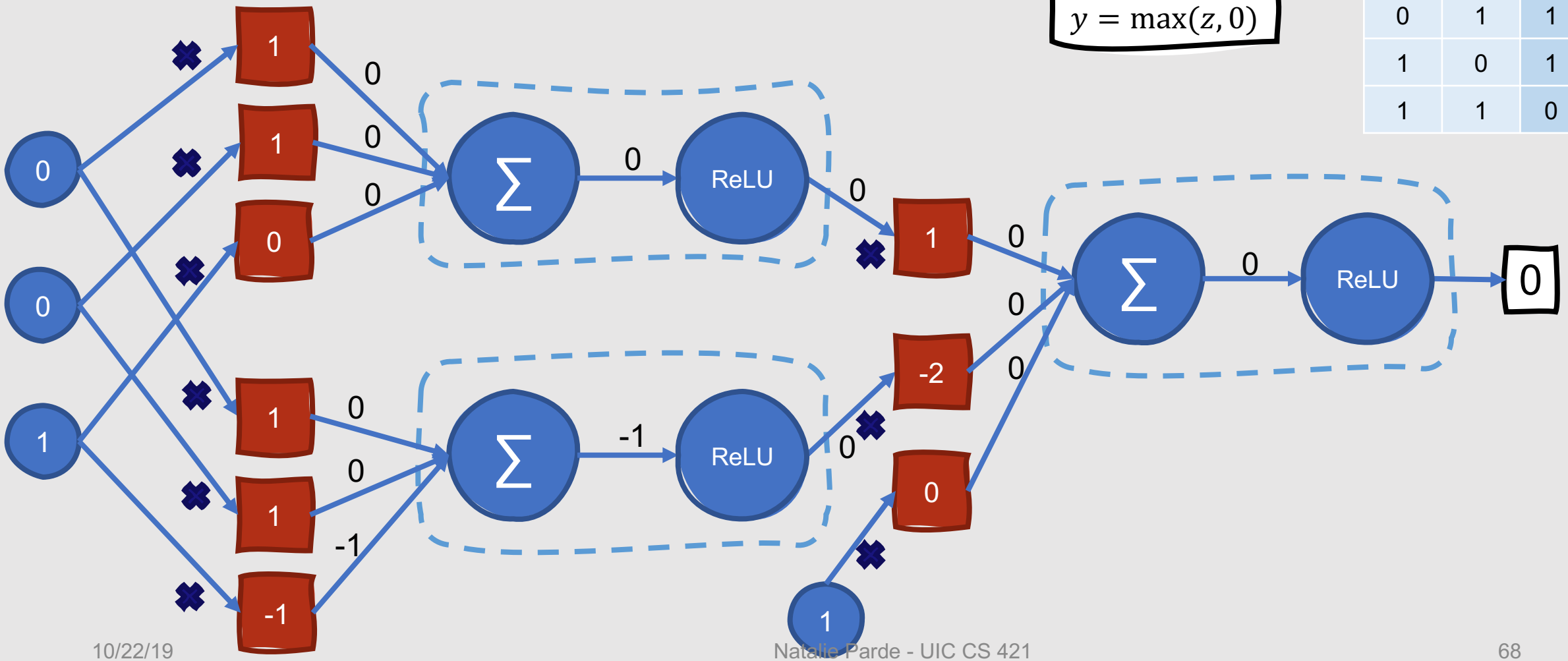
$$y = \max(z, 0)$$



# Truth Table Examples: XOR

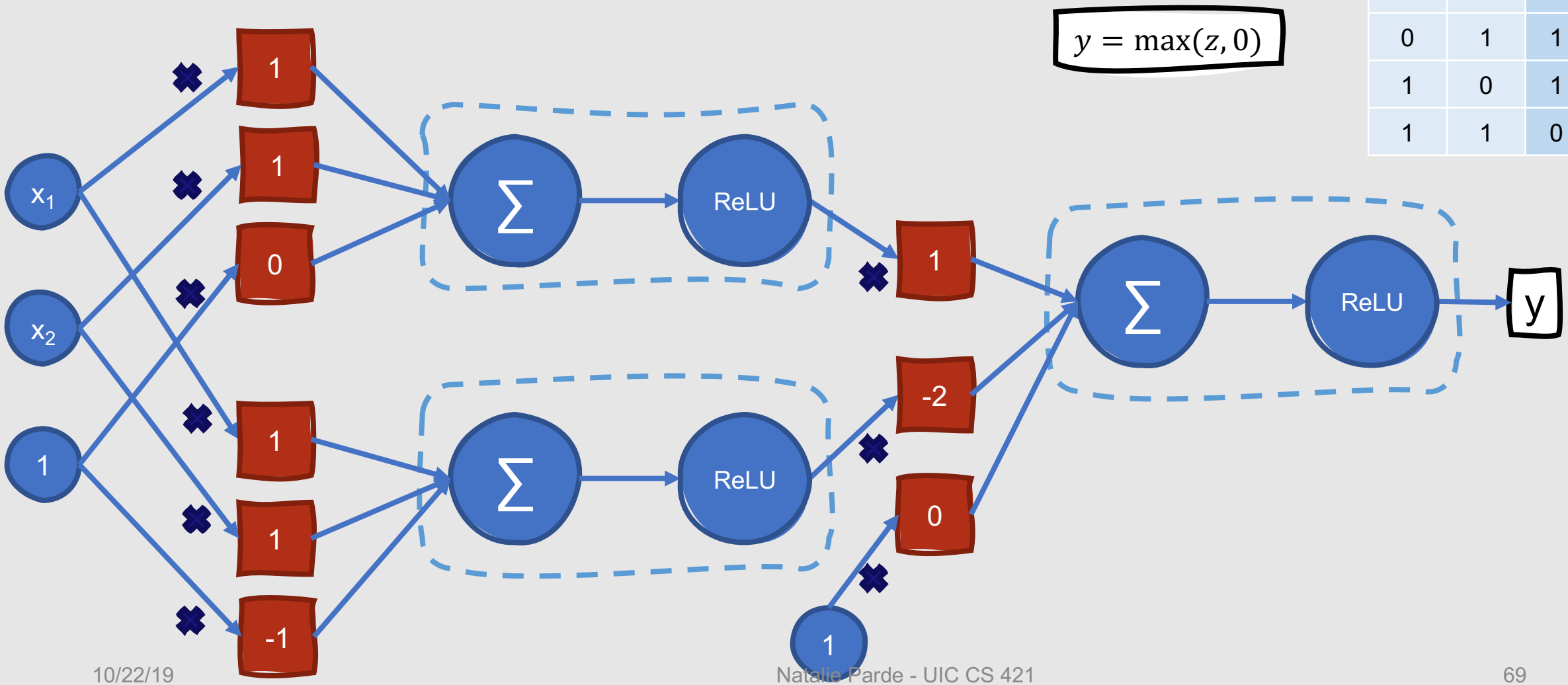
| XOR | | |
|---|---|---|
| x1 | x2 | y |
| **0** | **0** | **0** |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

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



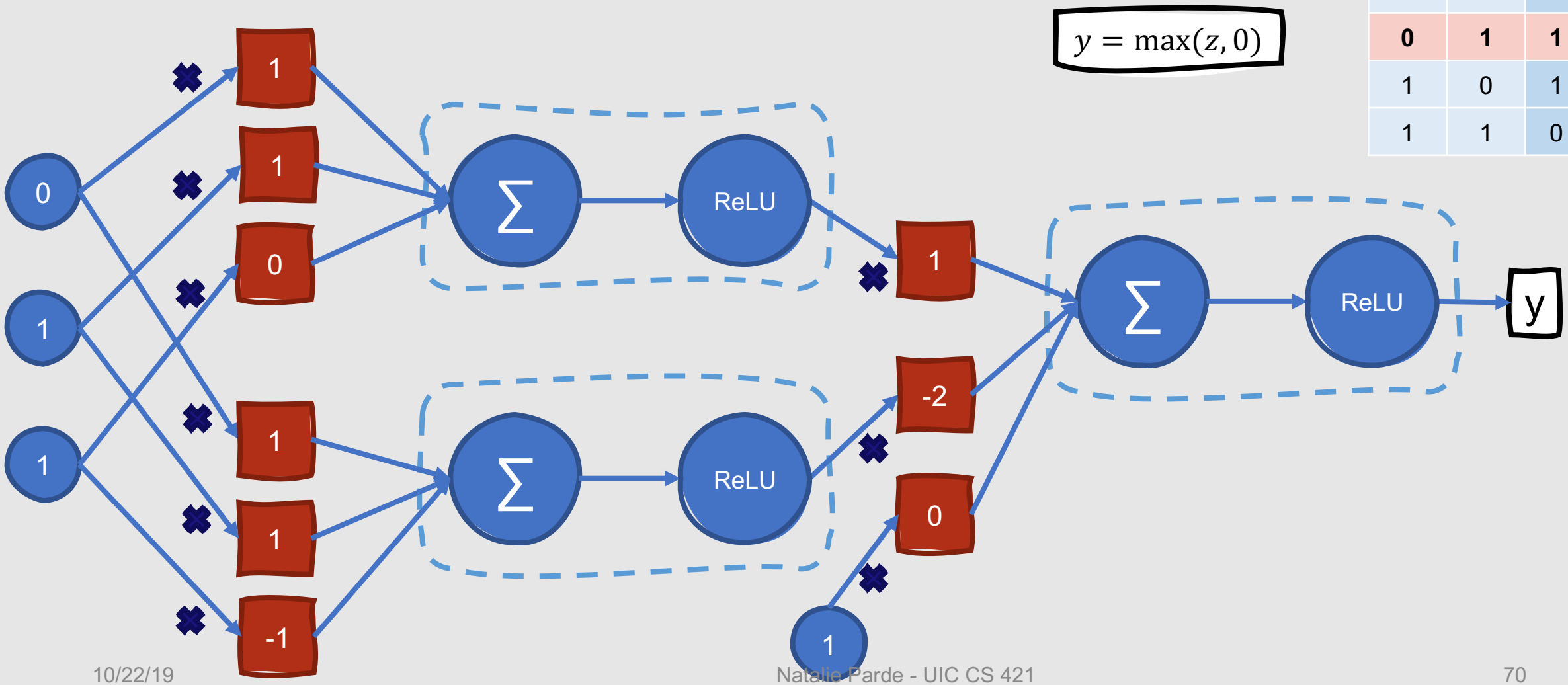
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



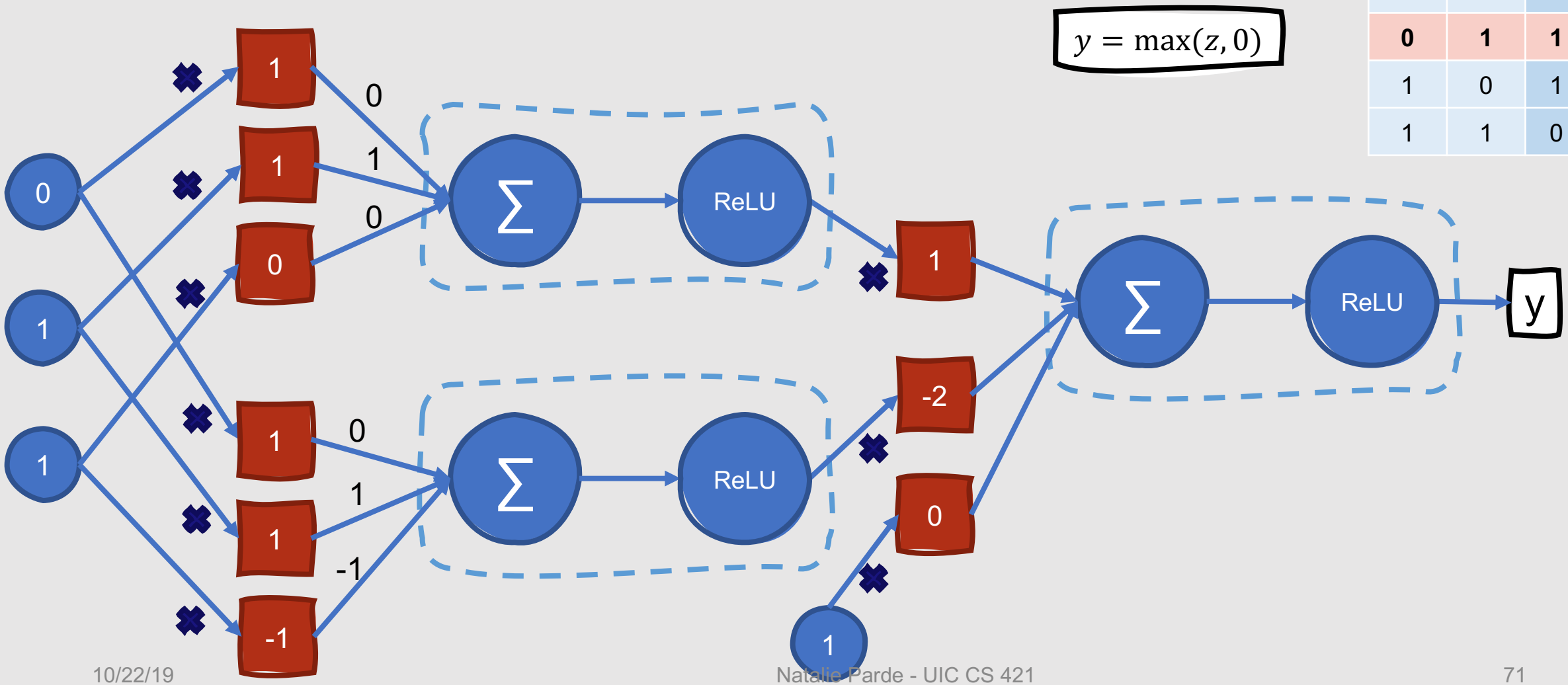
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| **0** | **1** | **1** |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



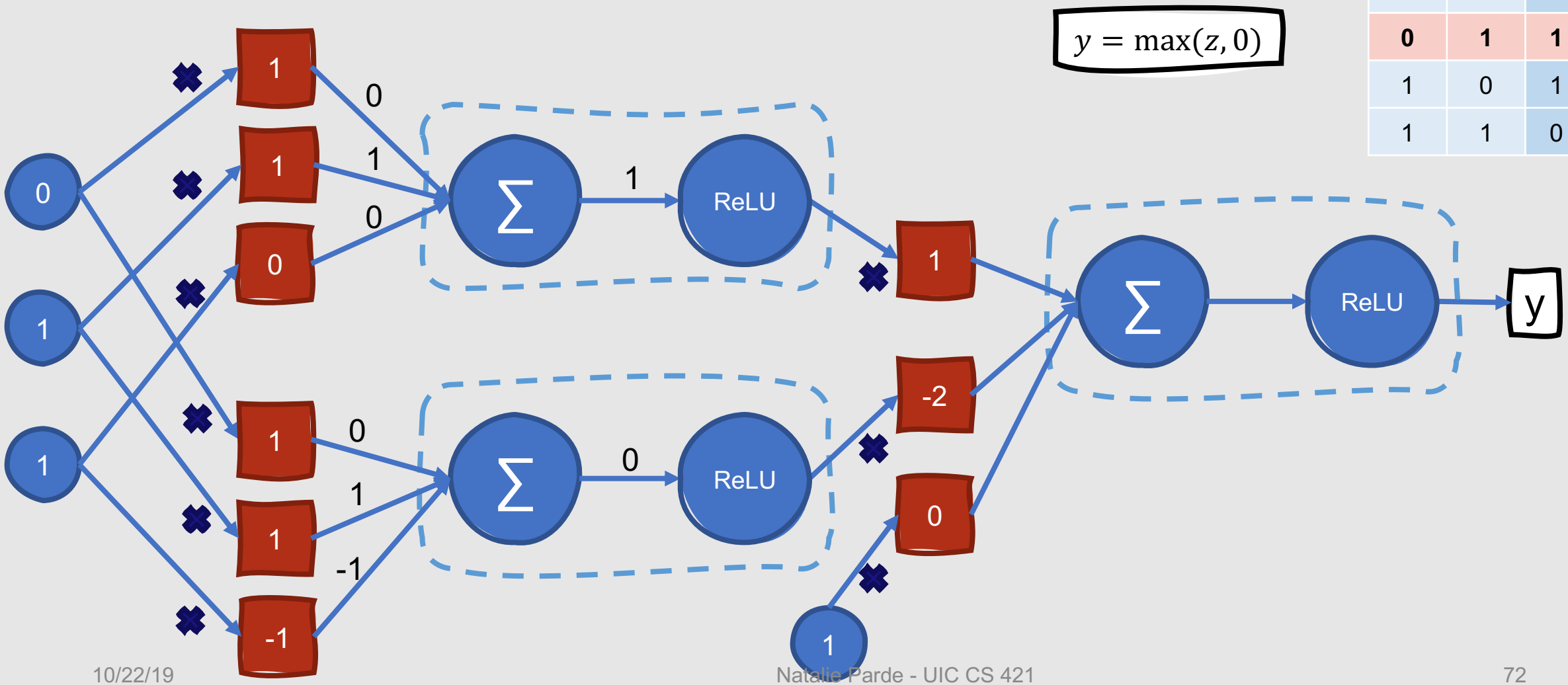
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| **0** | **1** | **1** |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



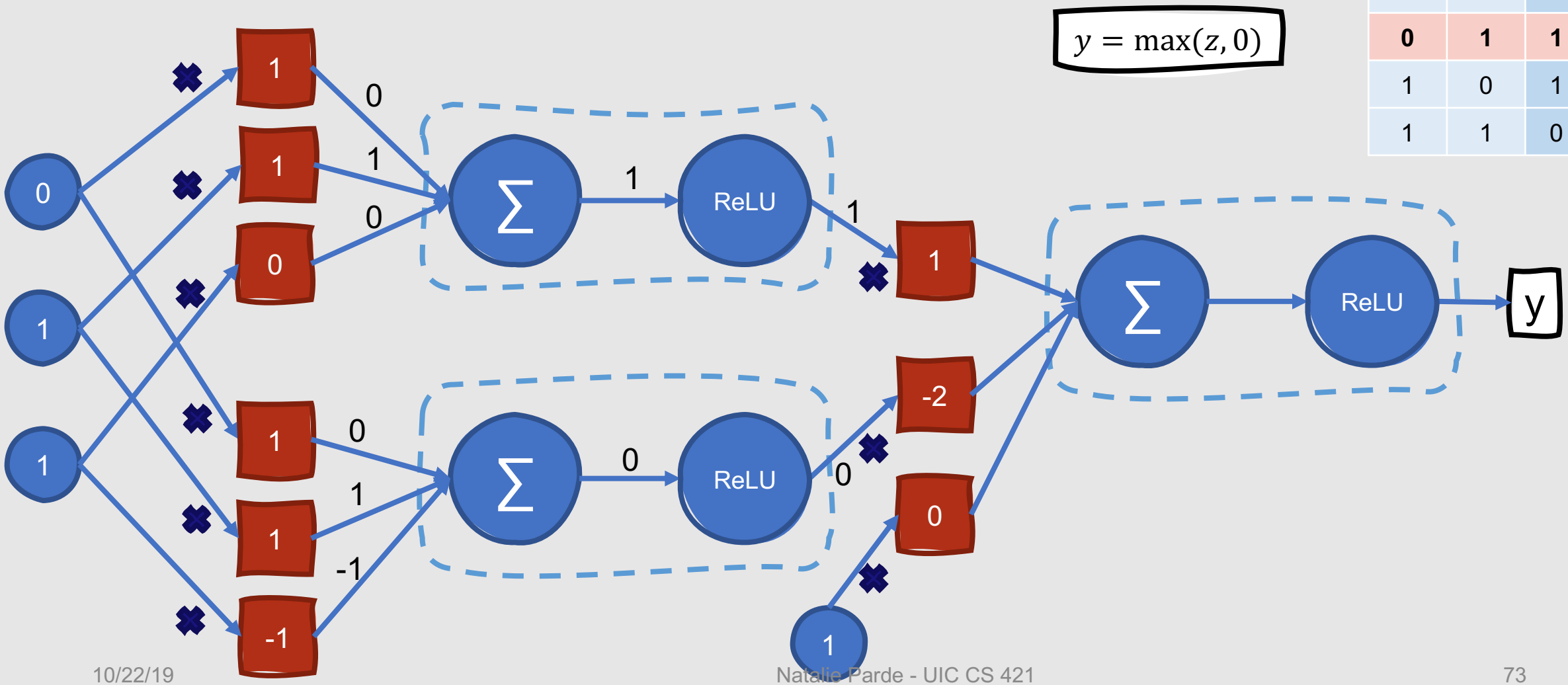
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| **0** | **1** | **1** |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



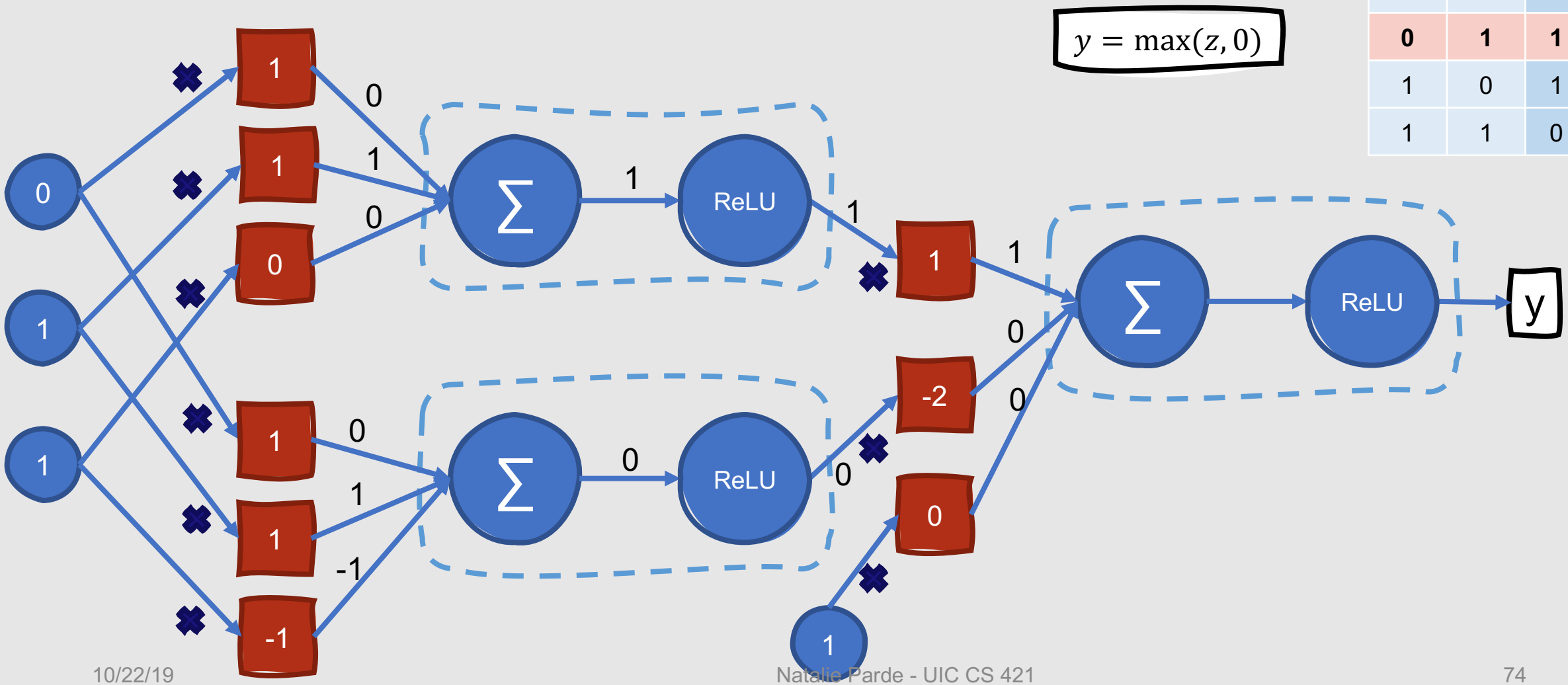
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| **0** | **1** | **1** |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



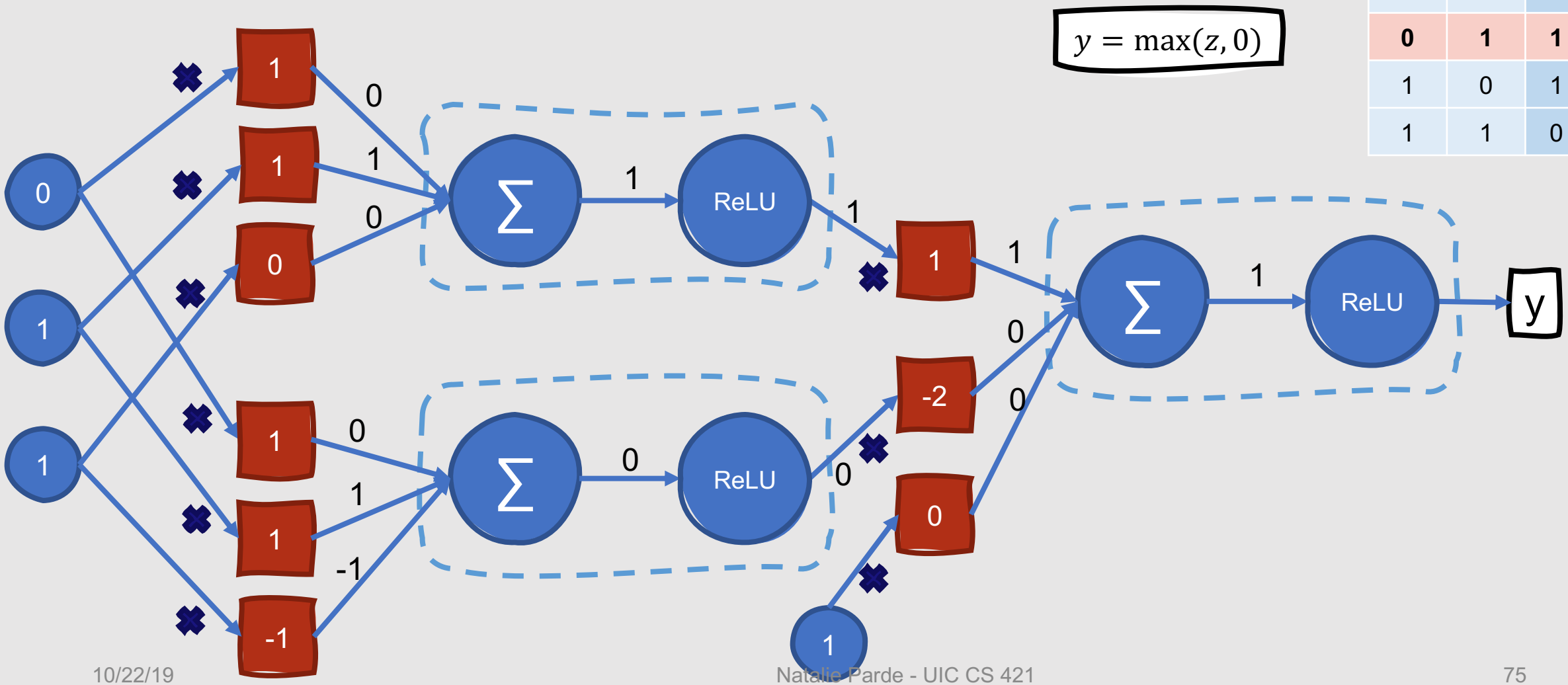
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| **0** | **1** | **1** |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



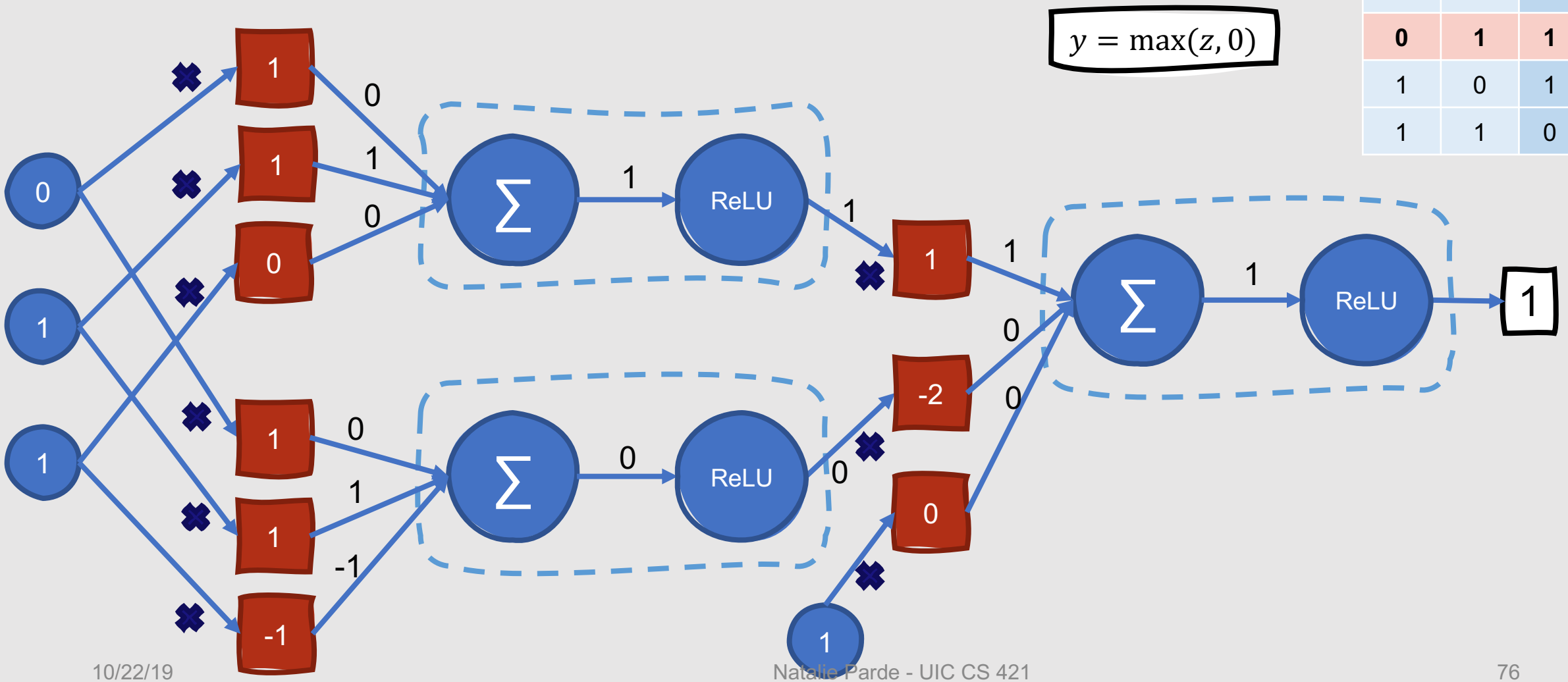
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| **0** | **1** | **1** |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



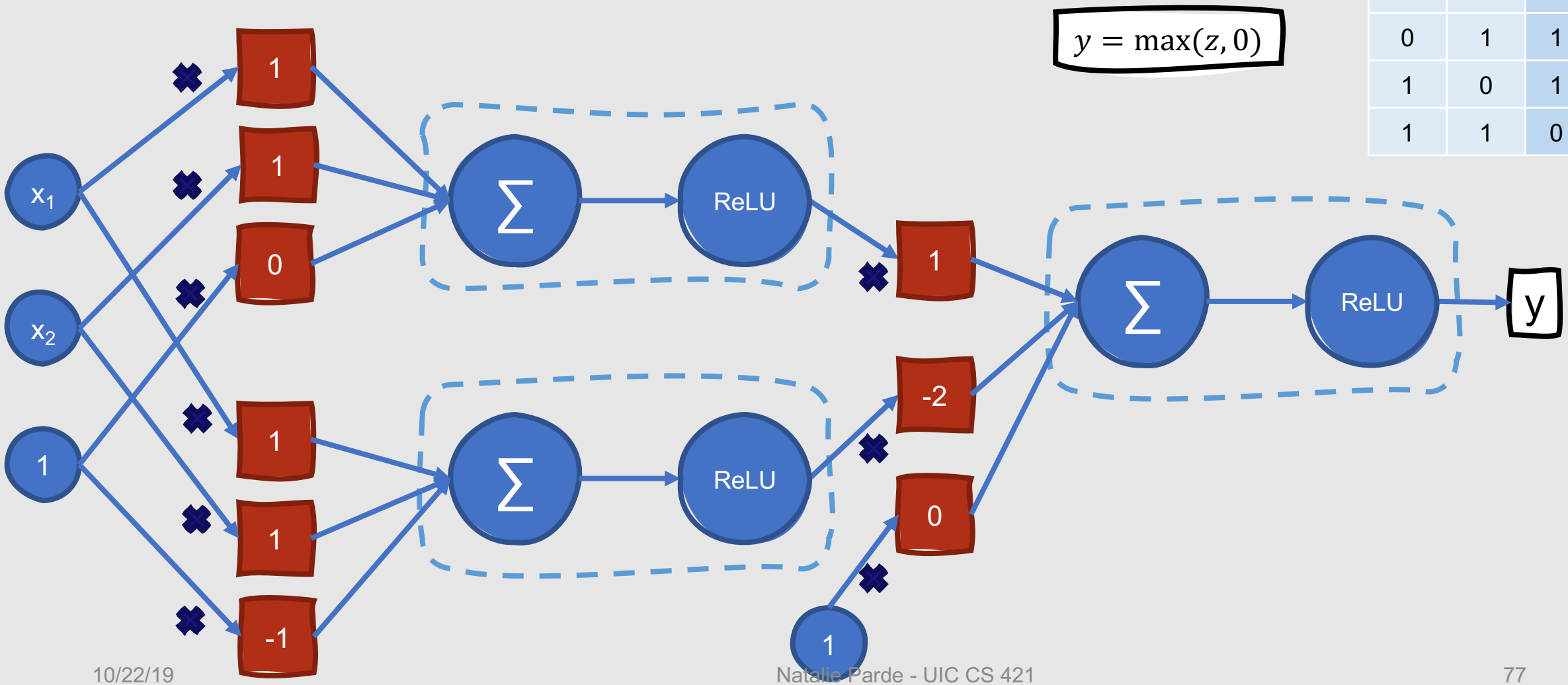
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| **0** | **1** | **1** |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



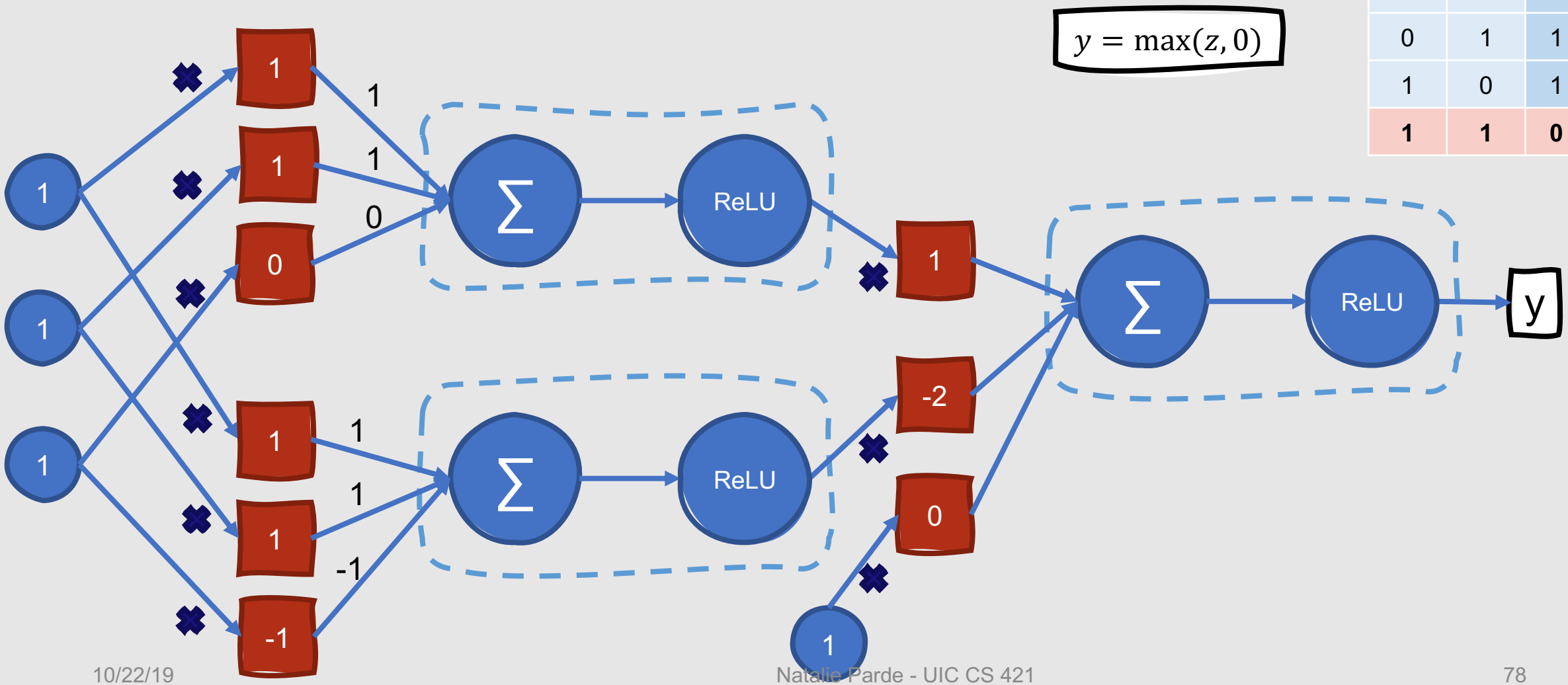
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



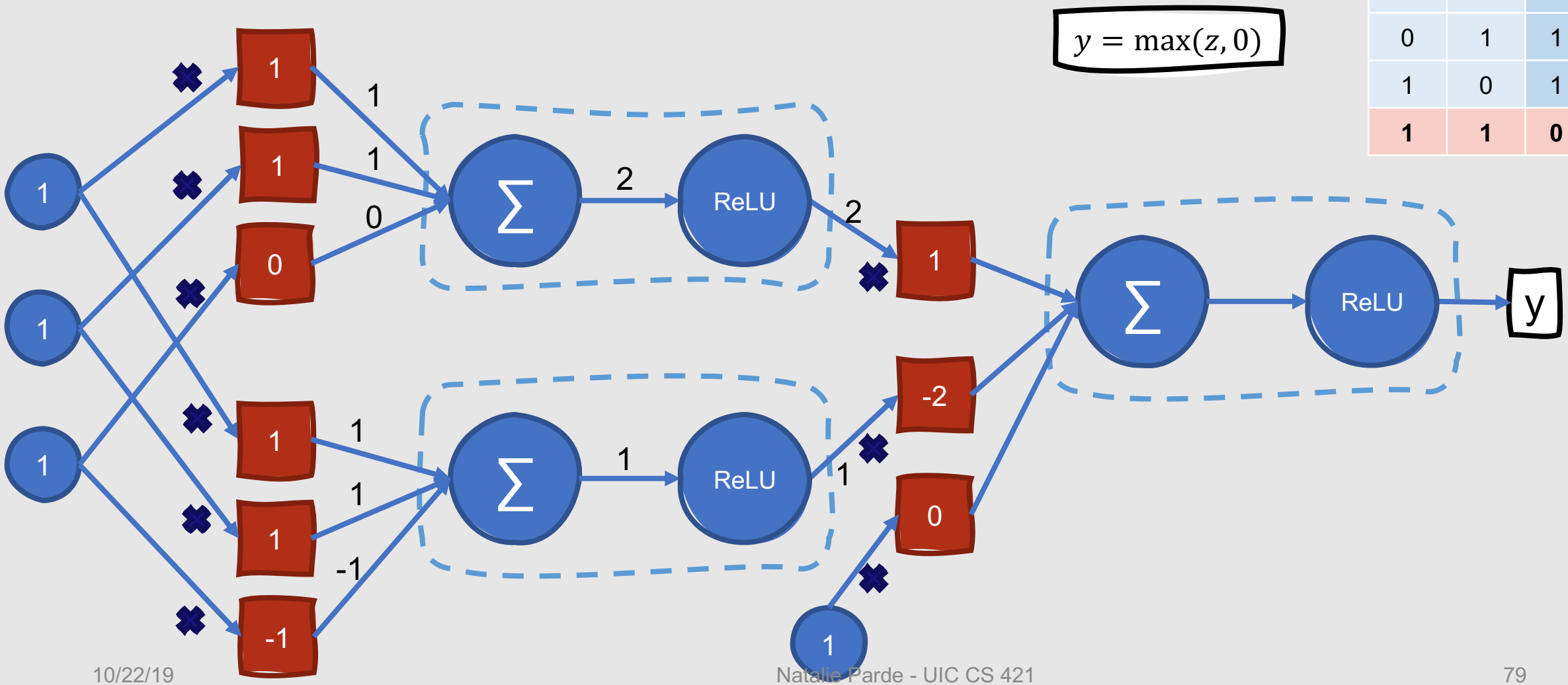
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| **1** | **1** | **0** |



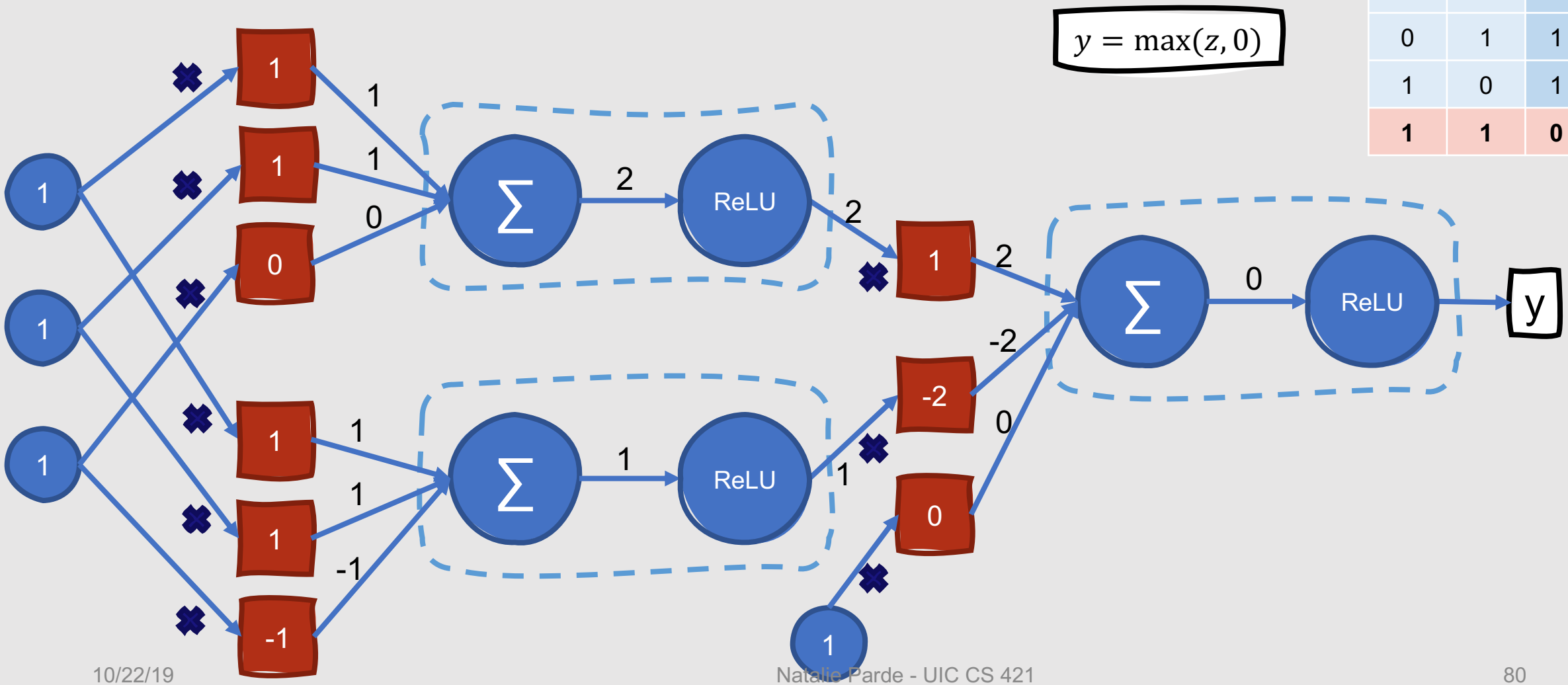
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| **1** | **1** | **0** |



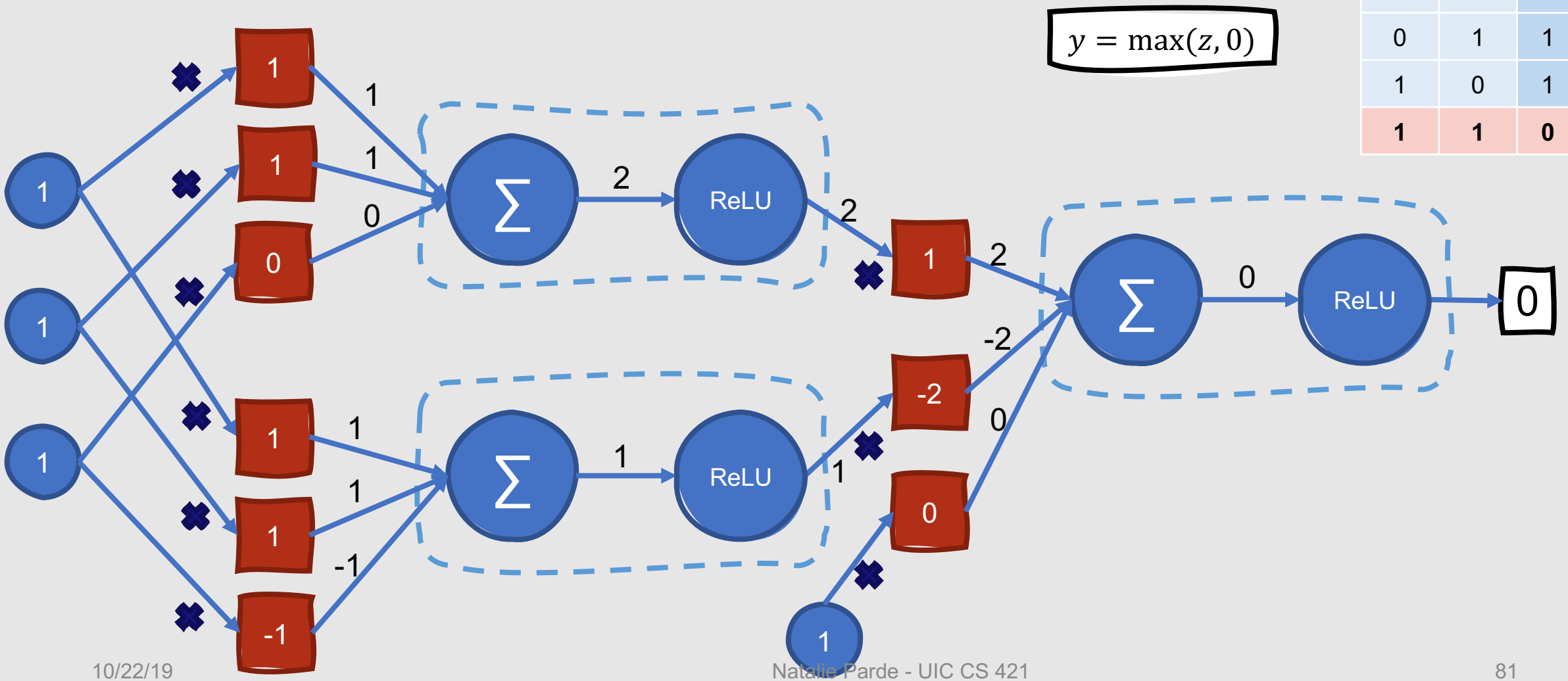
# Truth Table Examples: XOR

| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| **1** | **1** | **0** |

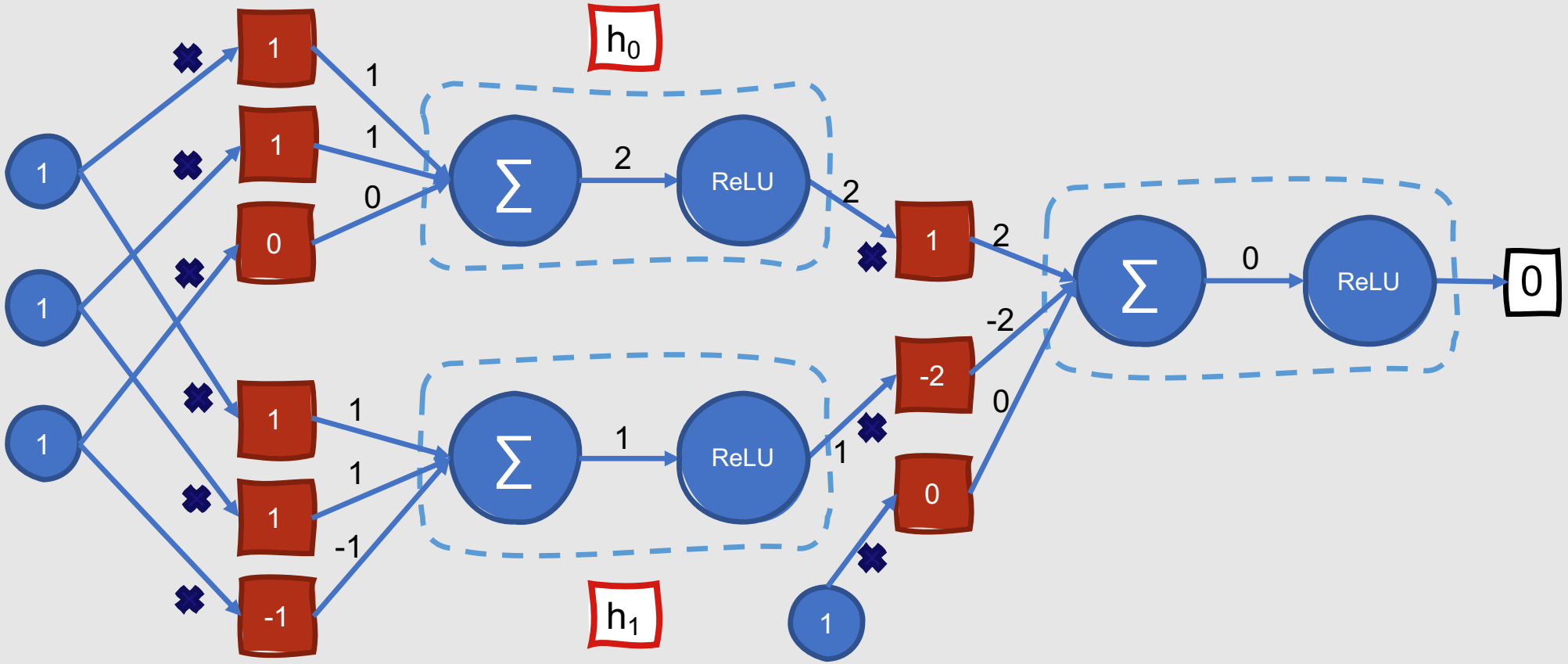


# Truth Table Examples: XOR

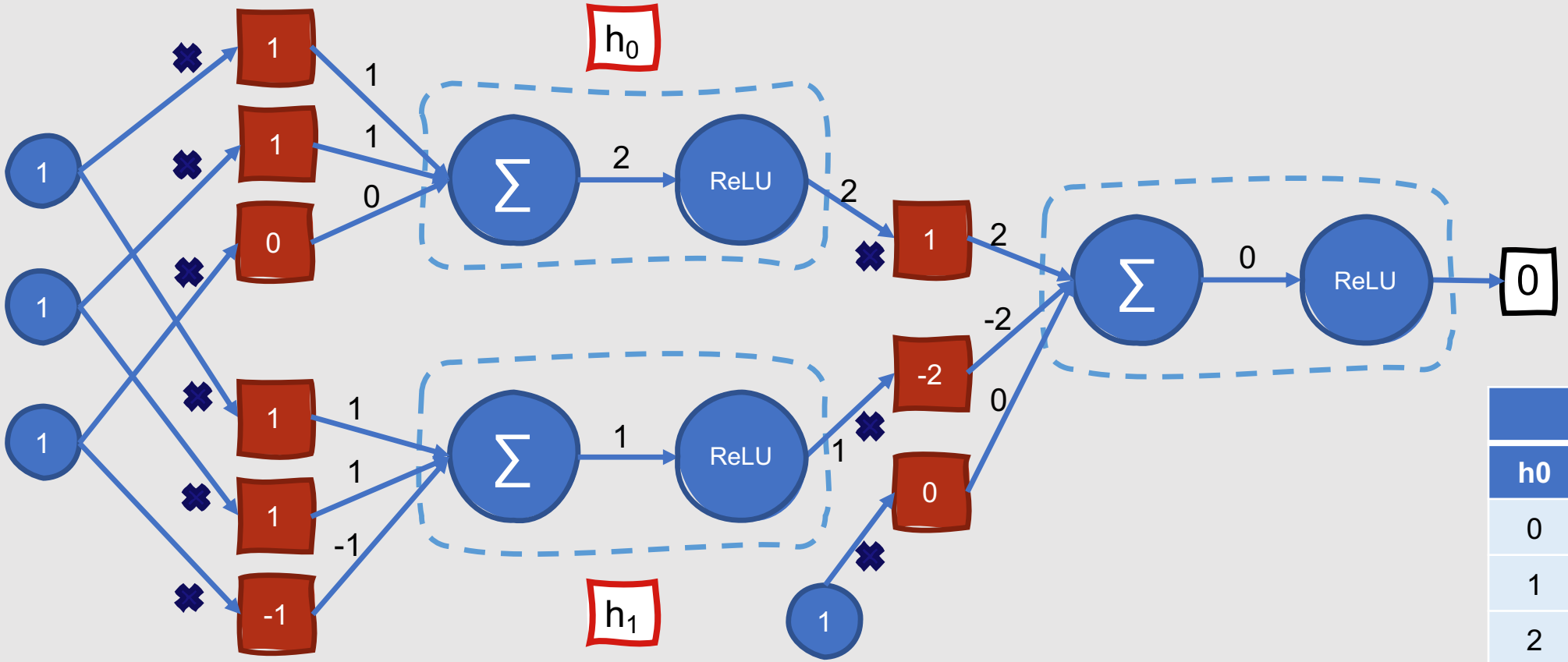
| XOR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| **1** | **1** | **0** |



# The hidden layer in the previous examples provides new representations for the input.



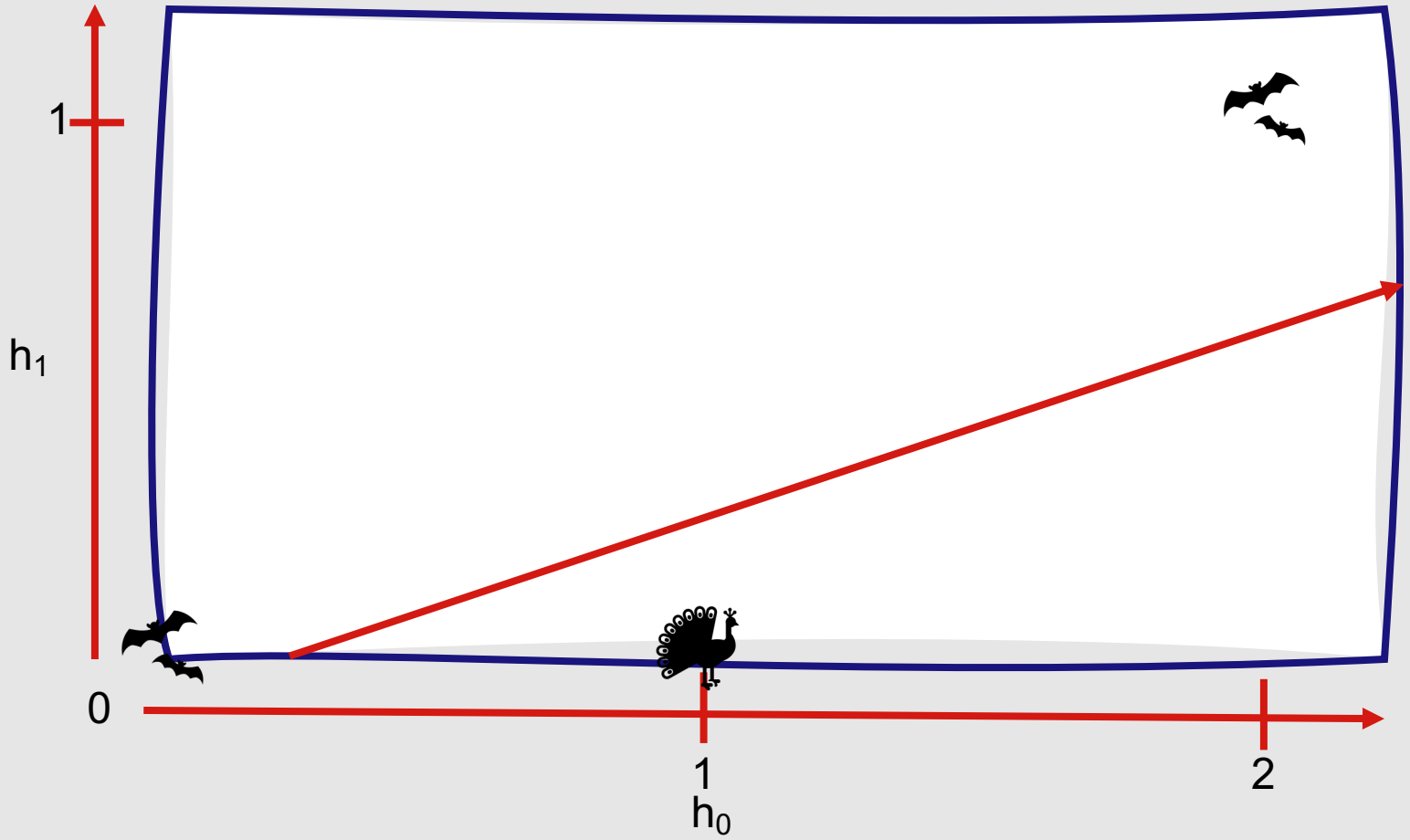
# The hidden layer in the previous examples provides new representations for the input.




| XOR | | |
|---|---|---|
| h0 | h1 | y |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |

# The new representations are linearly separable!

| 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

# Summary: Neural Networks (Basics)

- **Neural networks** are classification models comprised of interconnected computational units (**neurons**)
- They play an increasingly fundamental role in solving many NLP tasks
- There are many varieties of neural networks
  - **Feedforward**
  - **Convolutional**
  - **Recurrent**
  - Etc.…
- Neural networks with multiple layers are **deep neural networks**
- Neural networks can learn to separate data that is not **linearly separable** using **nonlinear functions**
- These **activation functions** can come in many forms
  - **sigmoid**
  - **tanh**
  - **ReLU**
- **Perceptrons** are basic linear functions that output binary values depending on whether the product of a set of inputs and weights exceeds a threshold