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/).

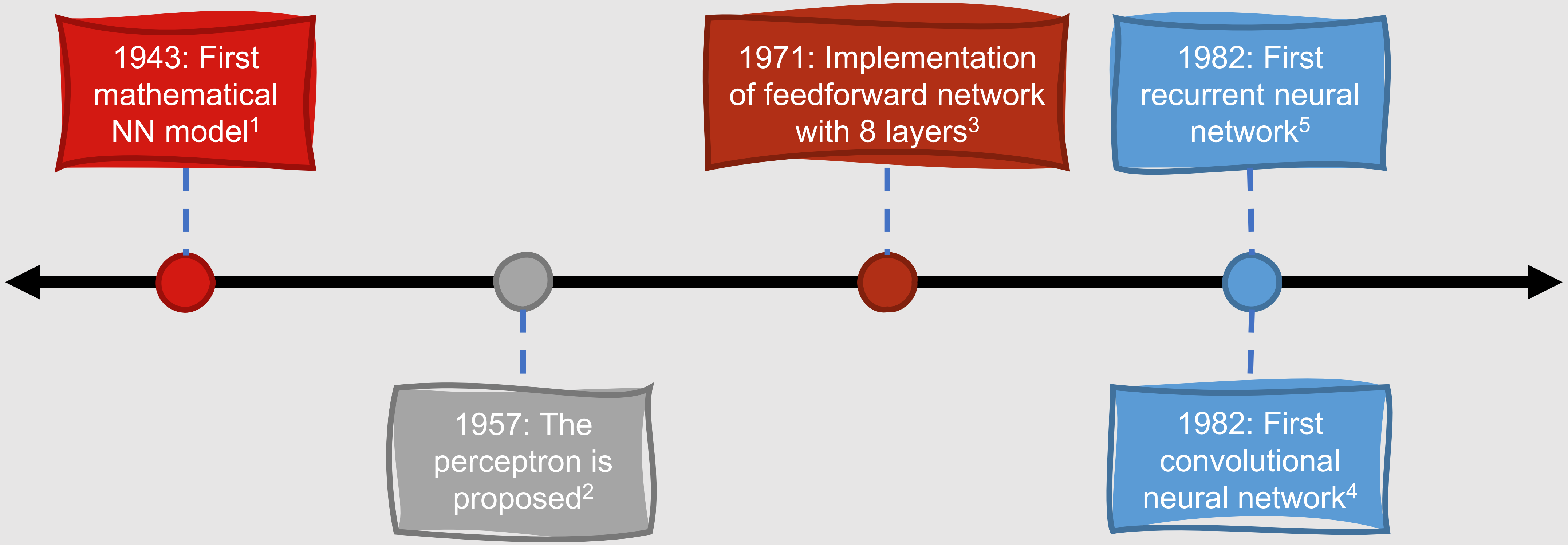
What are neural networks?

- 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?


1943: First mathematical NN model[1]
1957: The perceptron is proposed[2]
1971: Implementation of feedforward network with 8 layers[3]
1982: First recurrent neural network[5]
1982: First convolutional neural network[4]

[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.

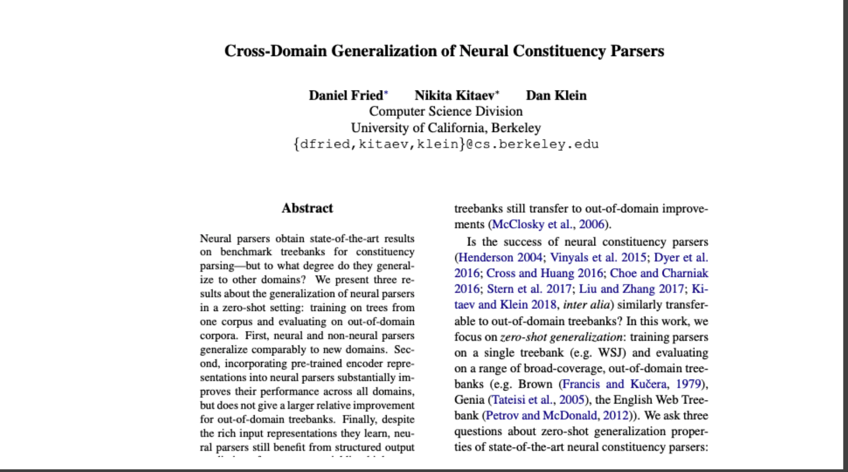
[3]Ivakhnenko, A. G. (1971). Polynomial theory of complex systems. *IEEE transactions on Systems, Man, and Cybernetics*, (4), 364-378.

[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

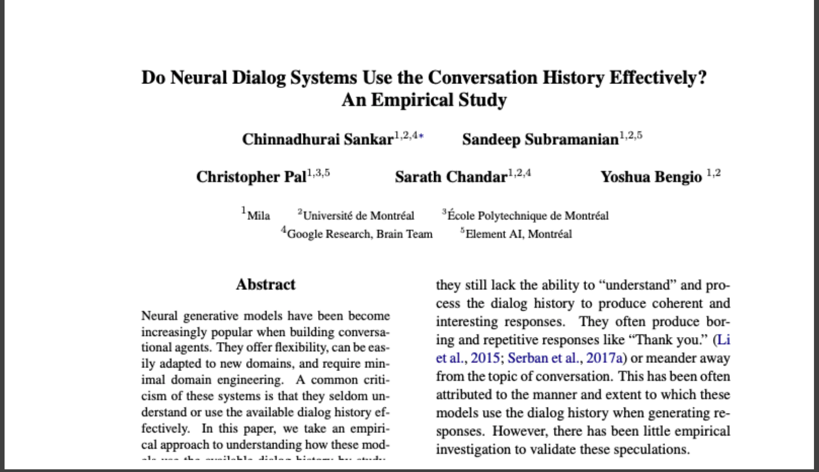
Daniel Fried* Nikita Kitaev* Dan Klein
Computer Science Division
University of California, Berkeley
{dfried,kitaev,klein}@cs.berkeley.edu

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

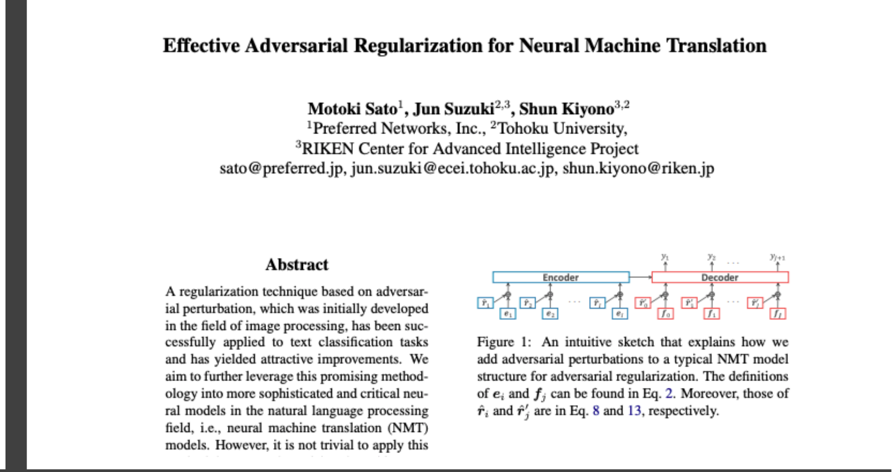
Chinnadhurai Sankar[1,2,4*] Sandeep Subramanian[1,2,5]
Christopher Pal[1,3,5] Sarath Chandar[1,2,4] Yoshua Bengio [1,2]

[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

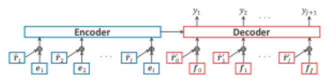
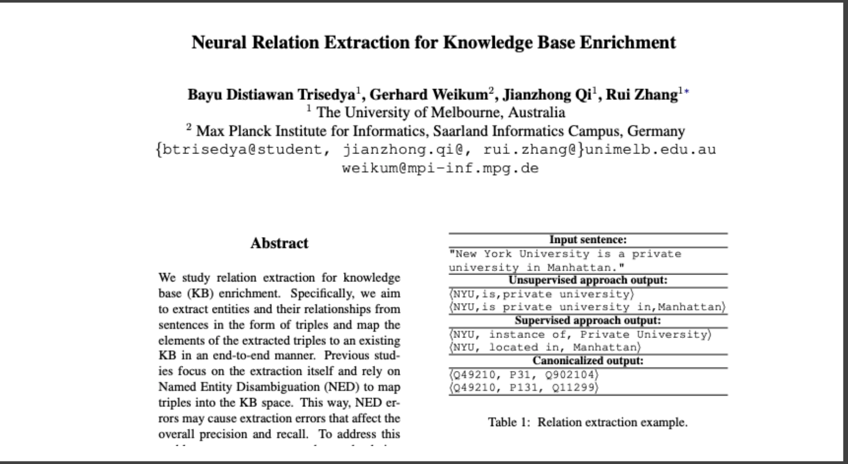


Figure 1: An intuitive sketch that explains how we add adversarial perturbations to a typical NMT model structure for adversarial regularization. The definitions of $\boldsymbol{e}_i$ and $\boldsymbol{f}_j$ can be found in Eq. 2. Moreover, those of $\hat{\boldsymbol{r}}_i$ and $\hat{\boldsymbol{r}}'_j$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



Neural Relation Extraction for Knowledge Base Enrichment

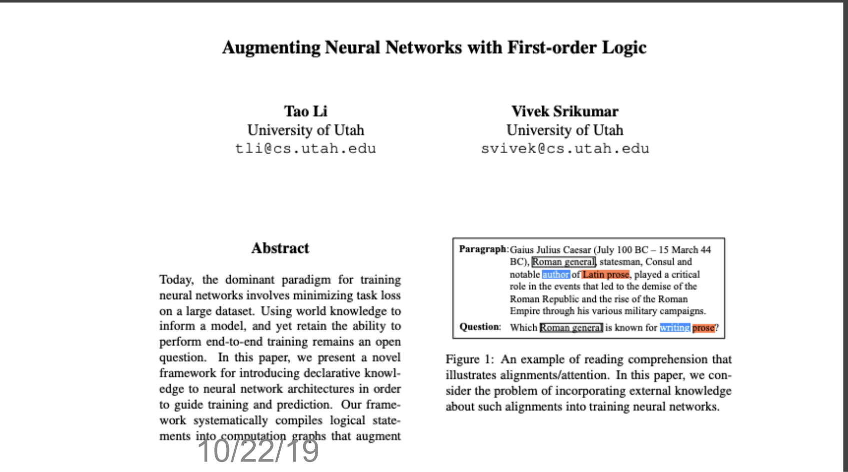
Bayu Distiawan Trisedya[1], Gerhard Weikum[2], Jianzhong Qi[1], Rui Zhang[1*]
[1] The University of Melbourne, Australia
[2] Max Planck Institute for Informatics, Saarland Informatics Campus, Germany
{btrisedya@student, jianzhong.qi@, rui.zhang@}unimelb.edu.au
weikum@mpi-inf.mpg.de

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

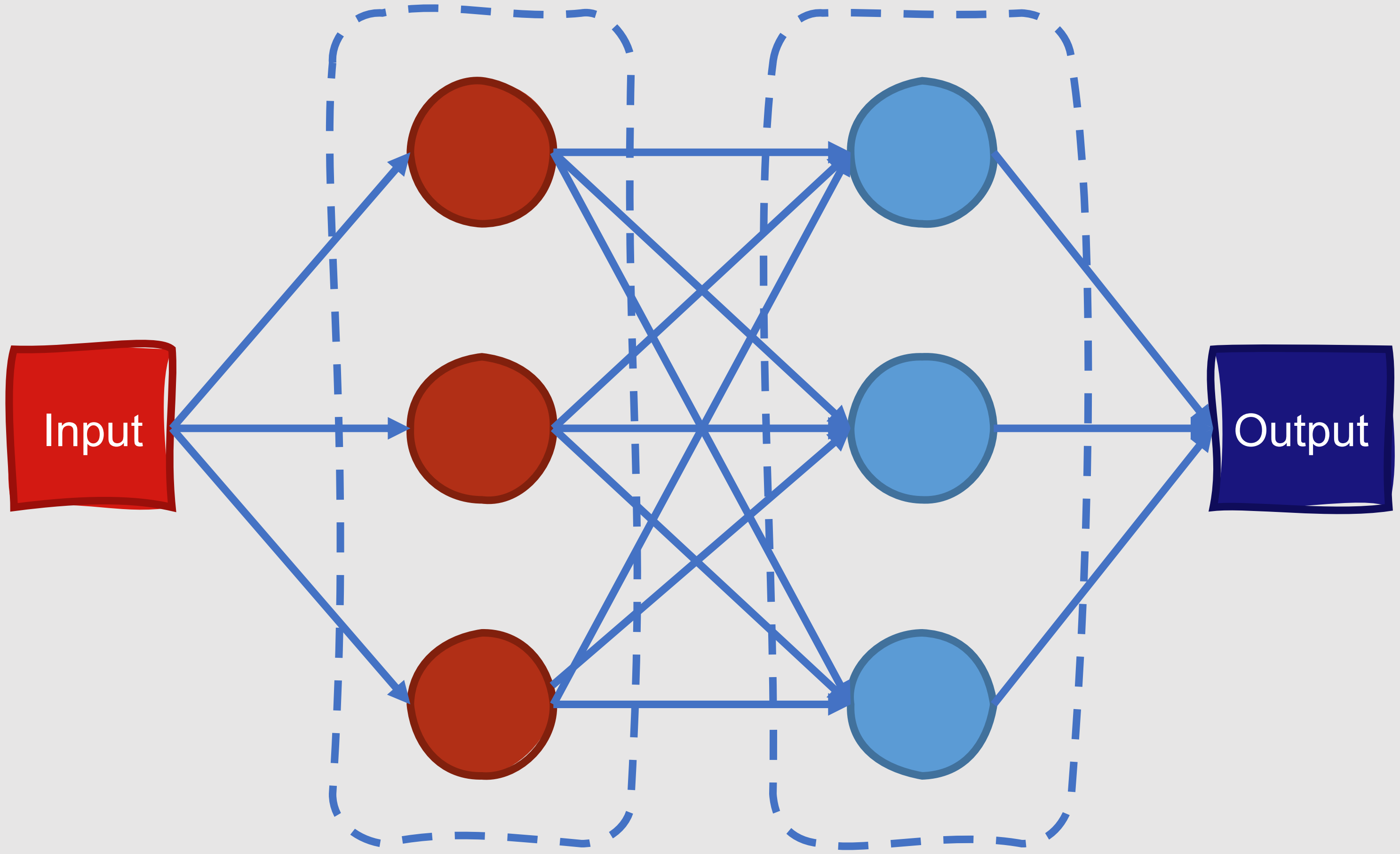
Types of Neural Networks

- 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



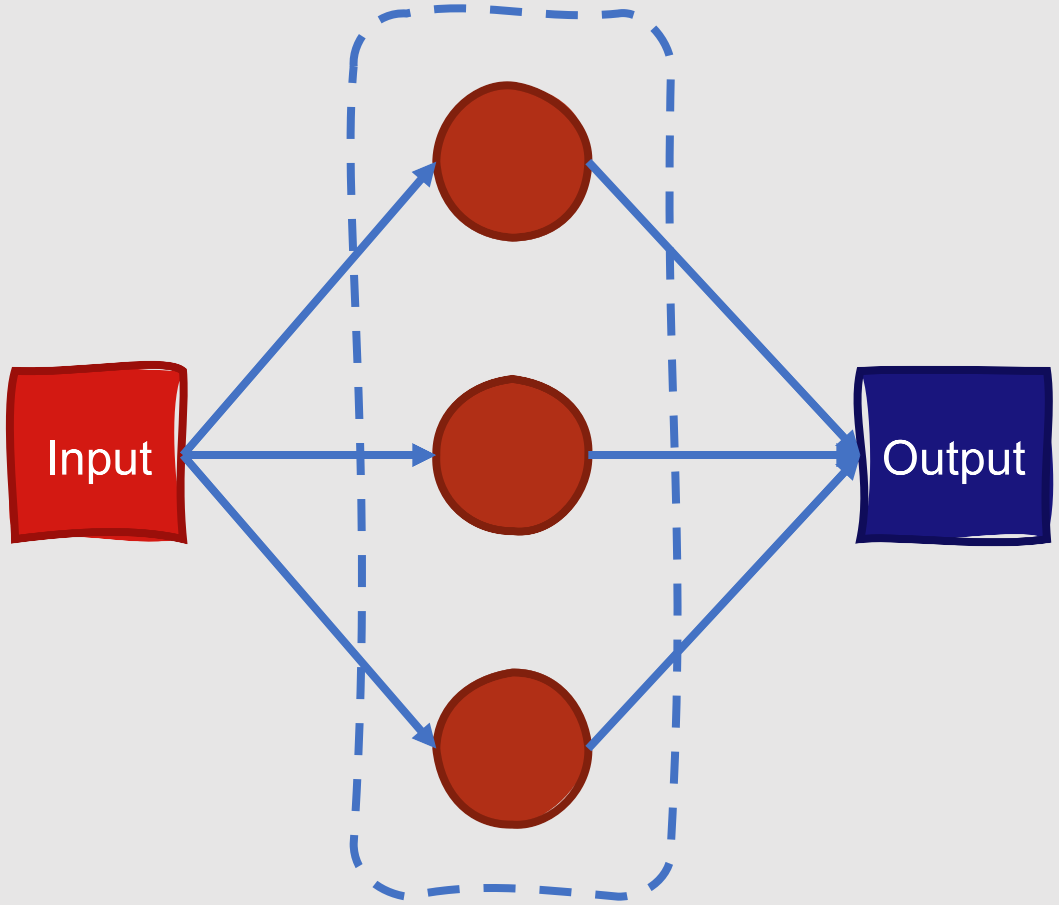
What is deep learning?

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

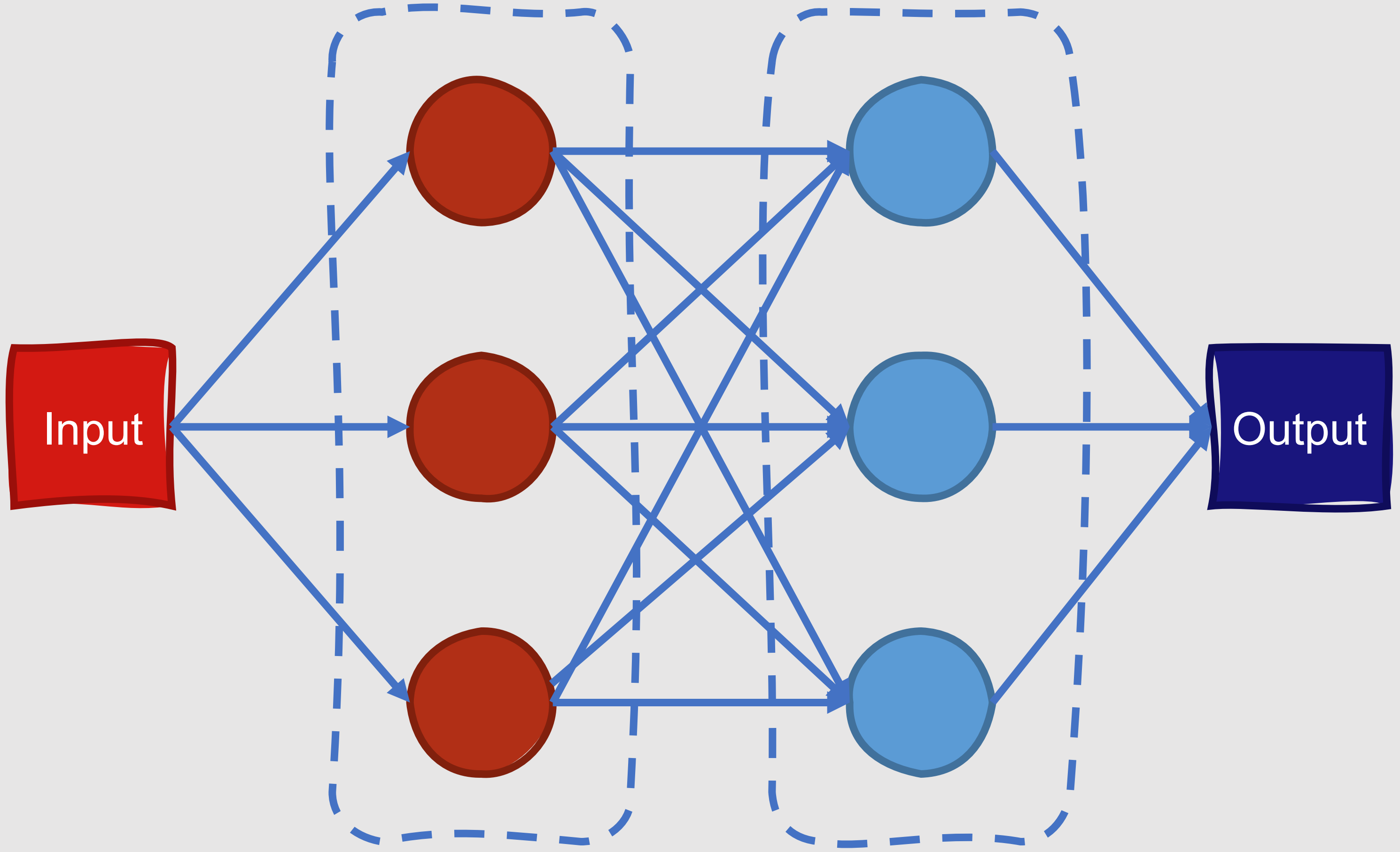
Why?

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

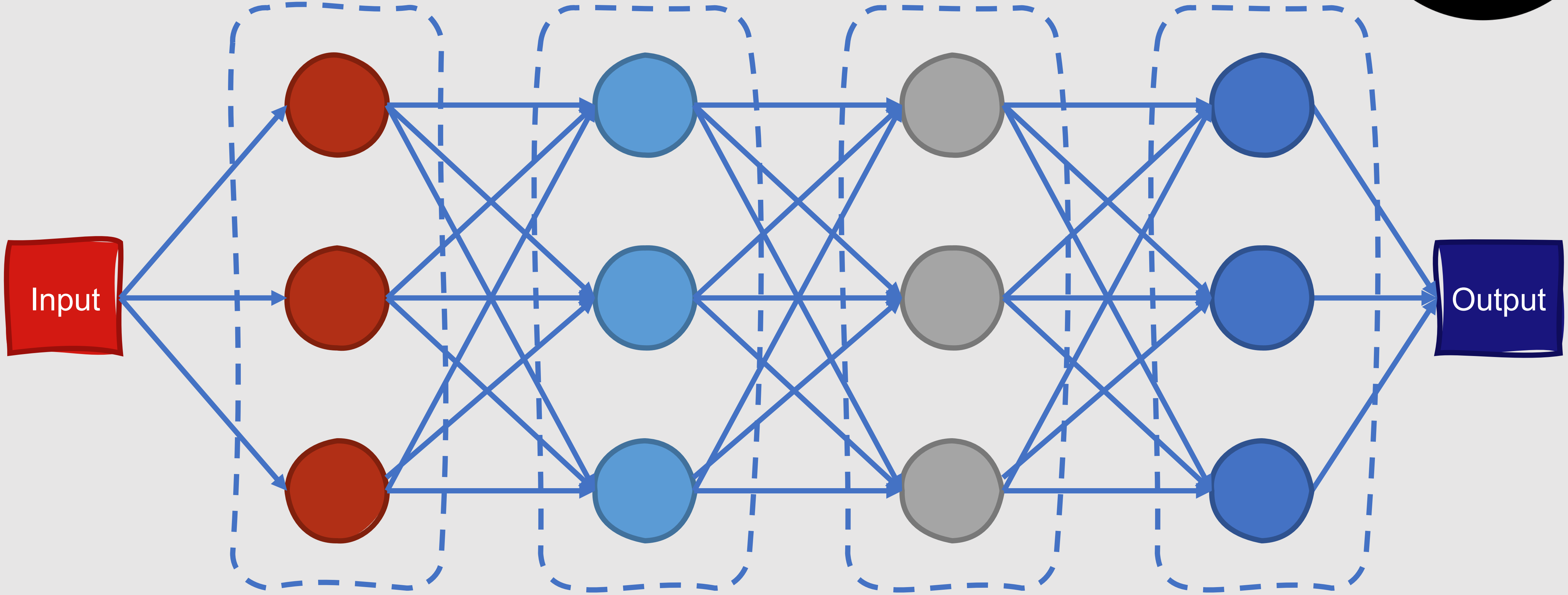
Deep Learning

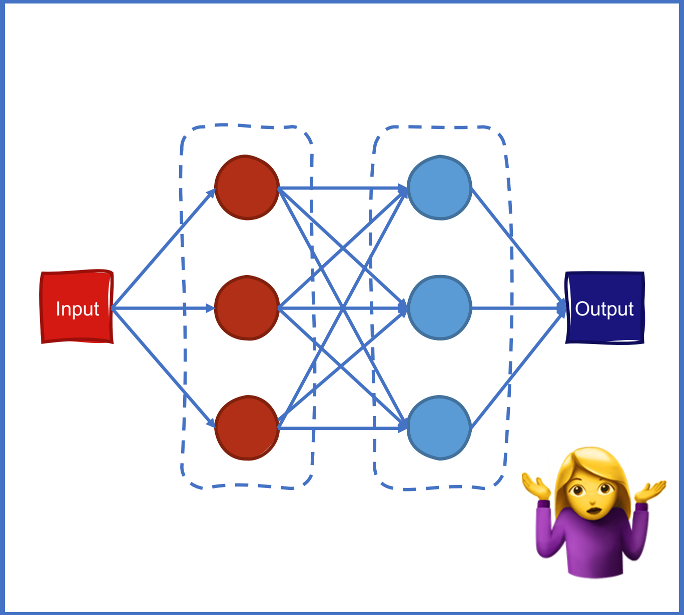


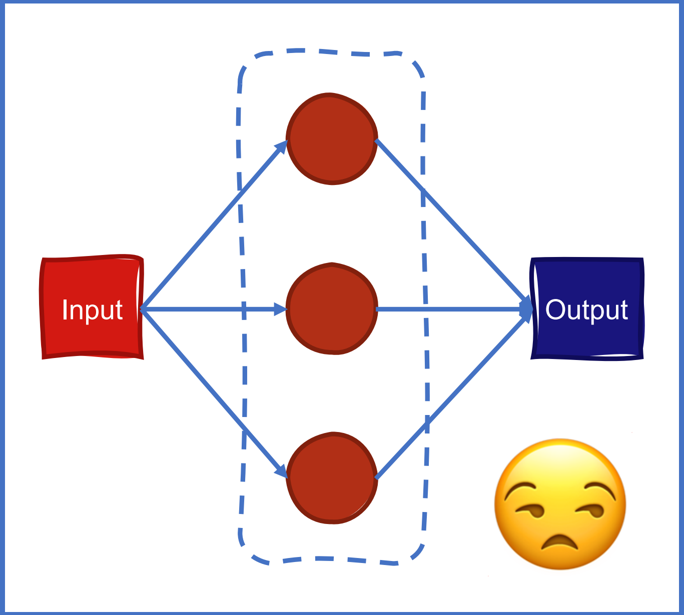
Deep Learning

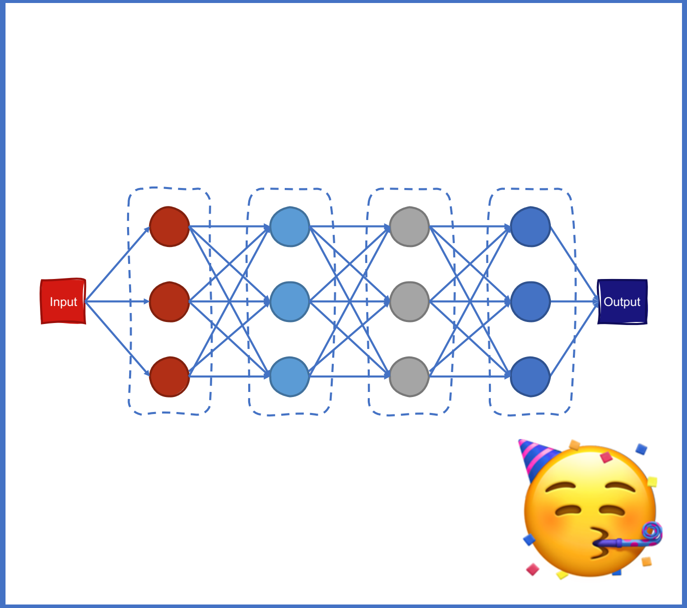


Deep Learning





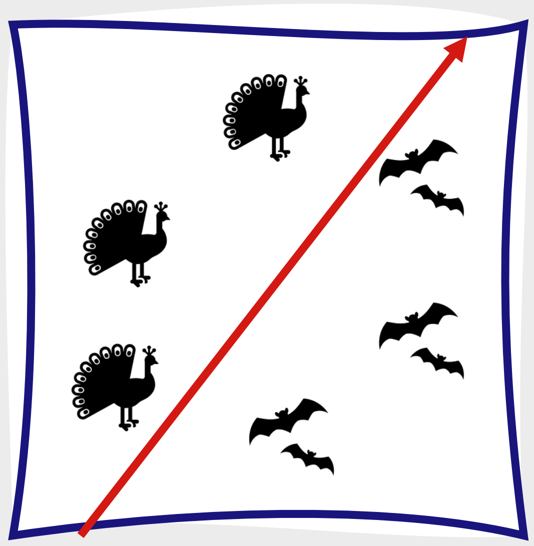


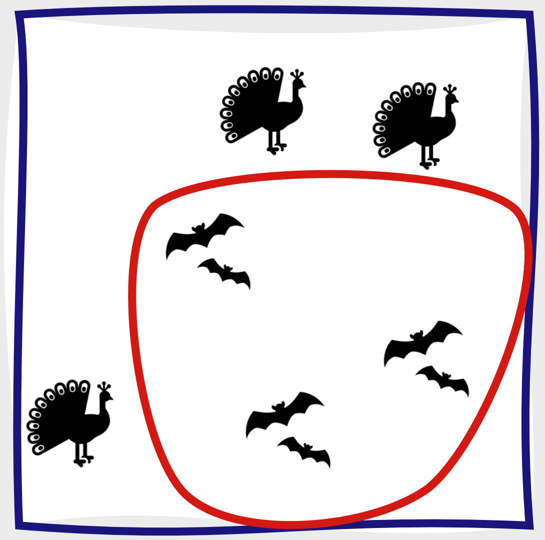


Deep Learning

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!

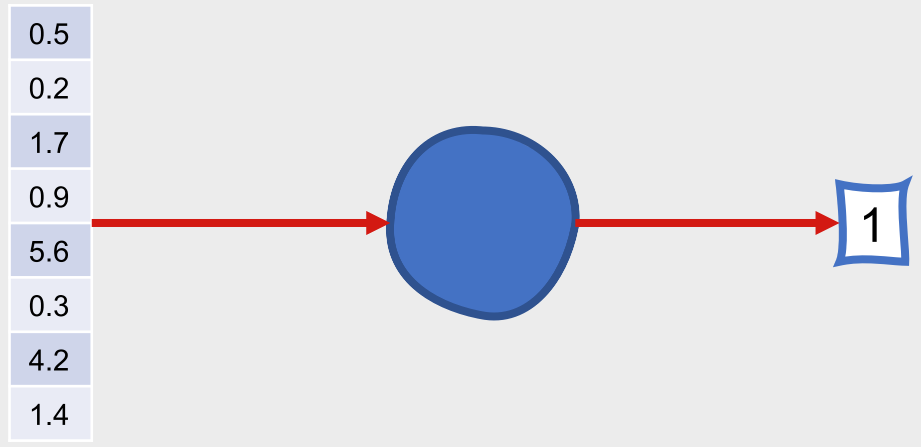
Learning features **implicitly** requires a lot of data

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!

exponential linear unit (elu)

softmax

scaled exponential linear unit (selu)

softplus

softsign

rectified linear unit (relu)

hyperbolic tangent (tanh)

sigmoid

There are many different activation functions!

exponential linear unit (elu)

softmax

scaled exponential linear unit (selu)

softplus

softsign

rectified linear unit (relu)

hyperbolic tangent (tanh)

sigmoid

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

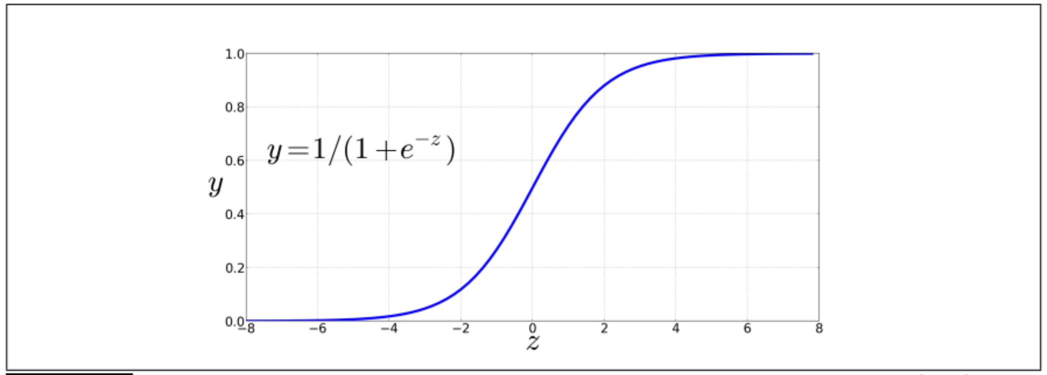
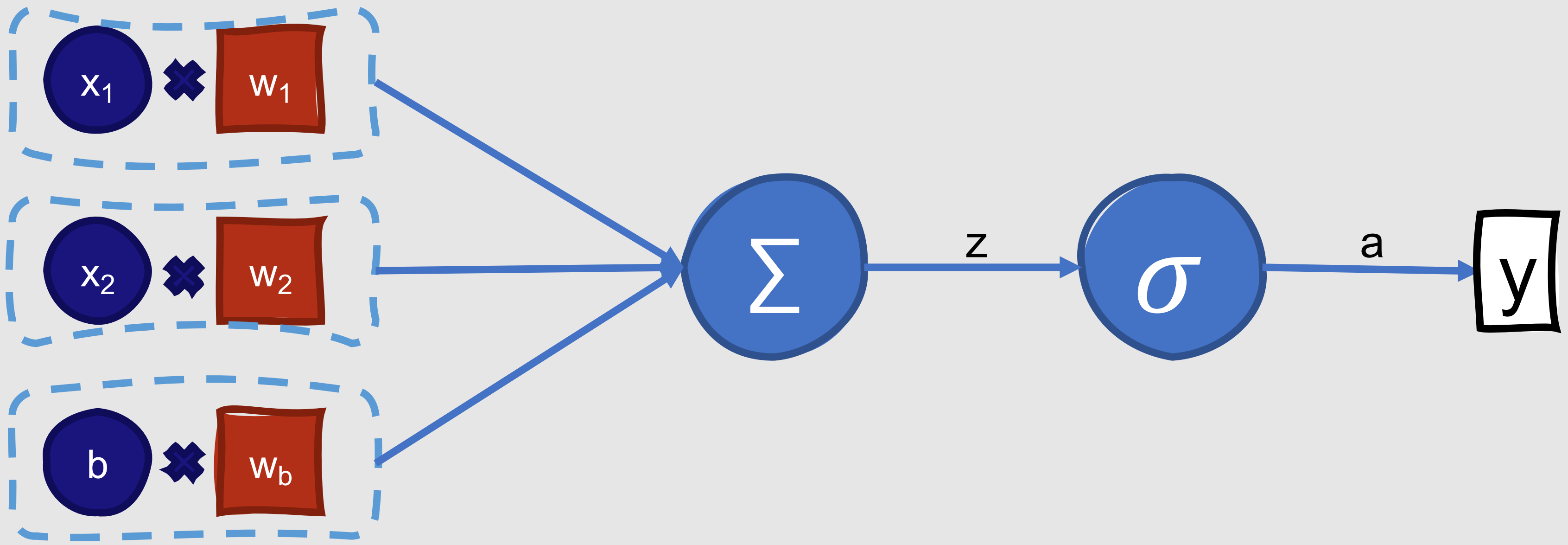


Figure 7.1 The sigmoid function takes a real value and maps it to the range [0,1]. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

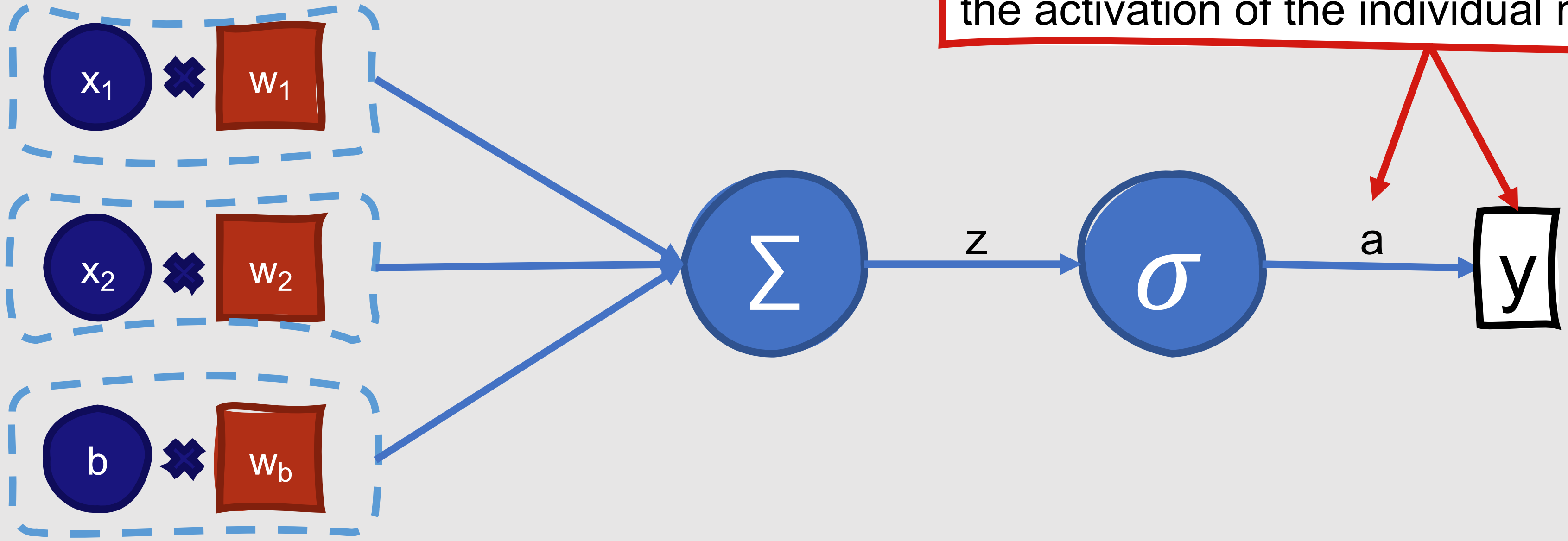
- Maps the unit's output to a [0,1] range
  - Squashes outliers
- Differentiable (useful for learning)
  - You need to be able to differentiate functions to optimize (minimize/maximize) them

Computational Unit with Sigmoid Activation

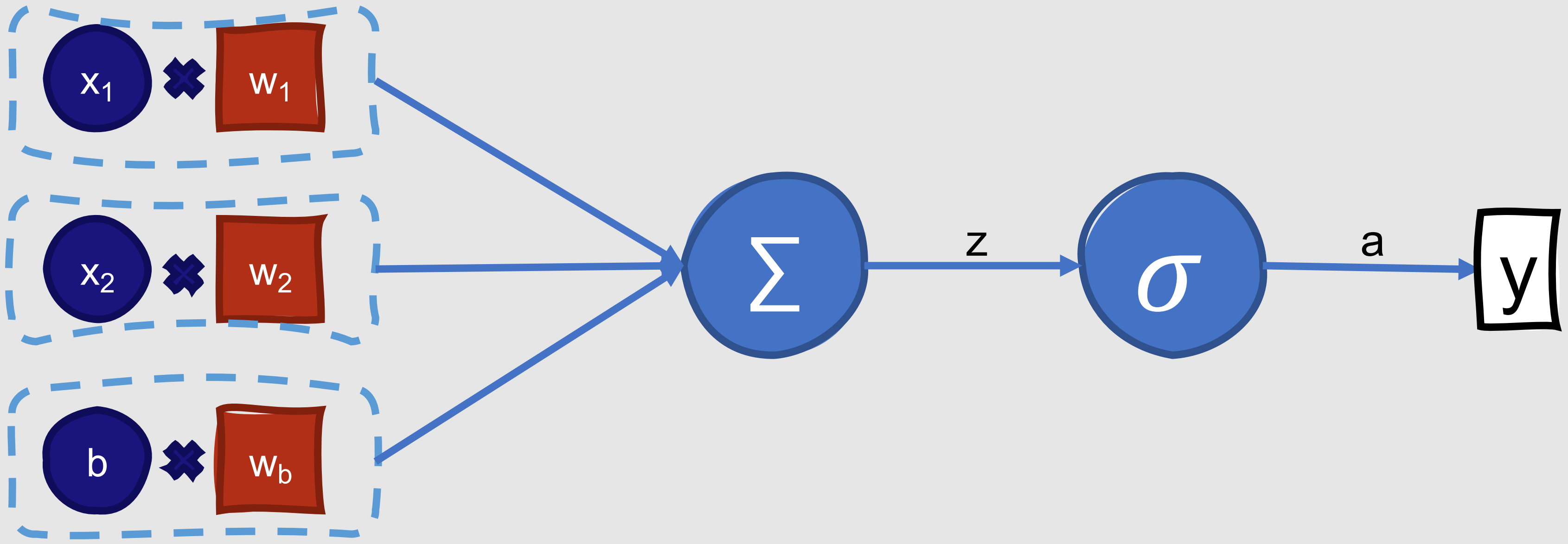


Computational Unit with Sigmoid Activation

Note that $a \neq y$ by default! Recall that y is the final output of the entire network, whereas a is the activation of the individual node.



Example: Computational Unit with Sigmoid Activation

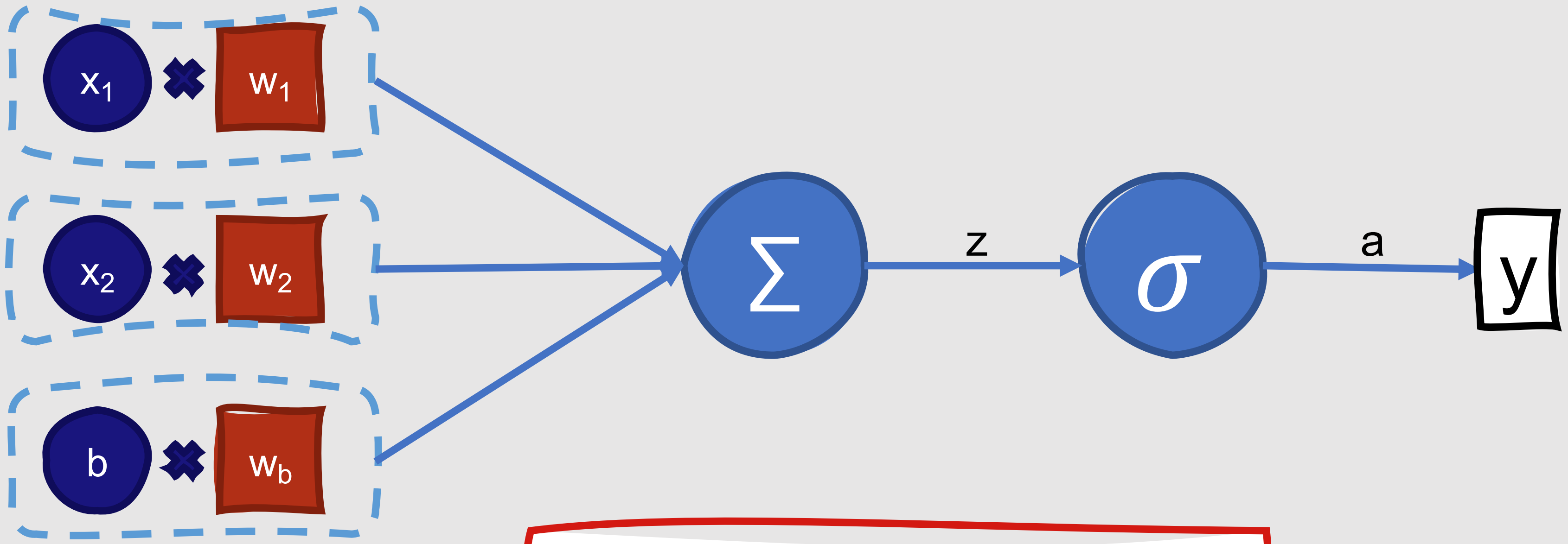


Input: “pumpkin spice latte”

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

Bias: 1.0

Example: Computational Unit with Sigmoid Activation

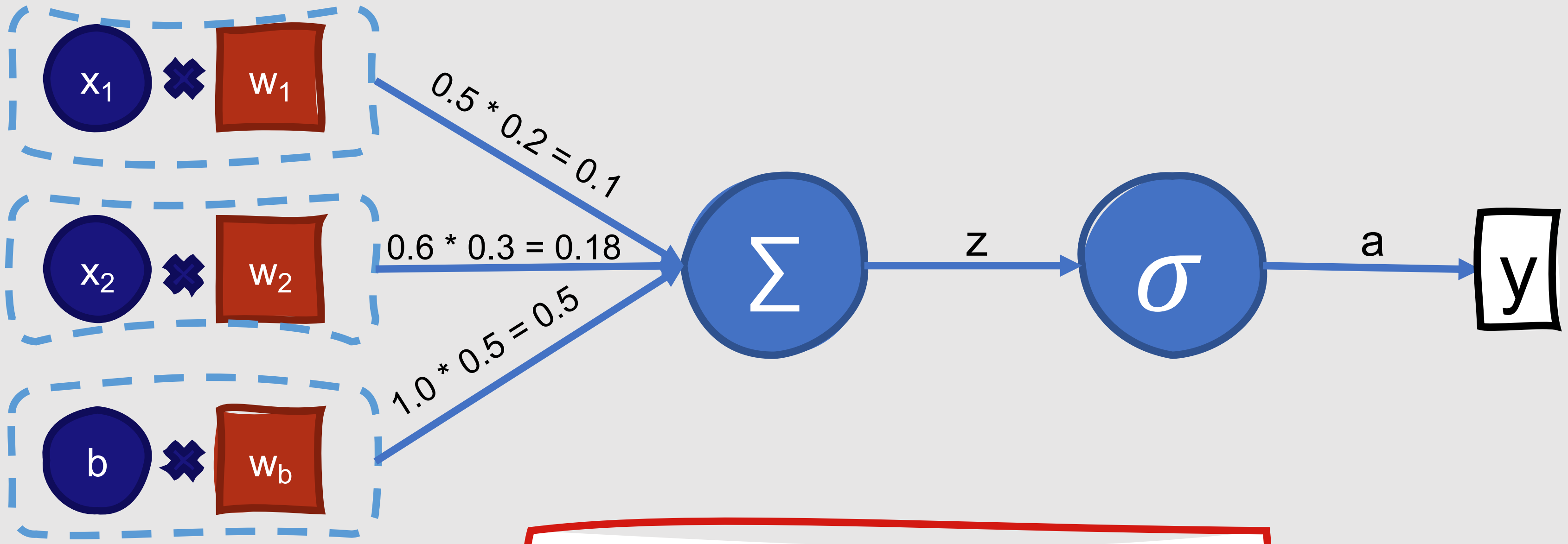


Input: “pumpkin spice latte” → Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”) → [0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with Sigmoid Activation

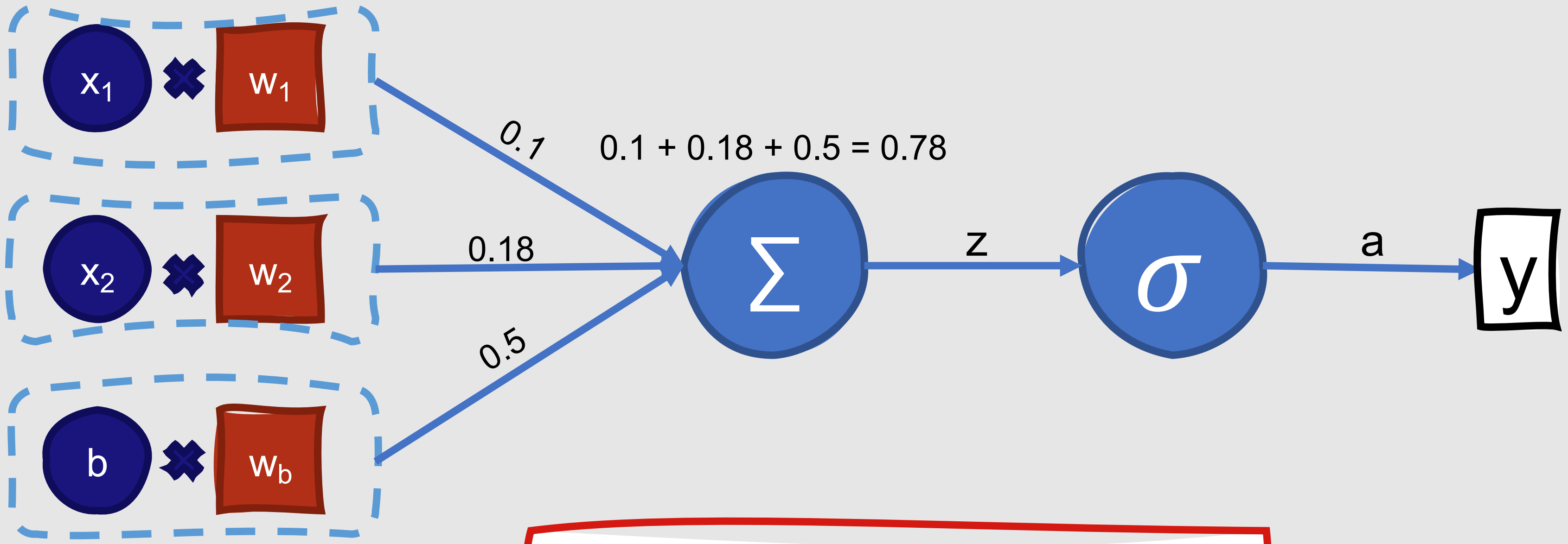


Input: "pumpkin spice latte" → Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte") → [0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with Sigmoid Activation



Input: “pumpkin spice latte”

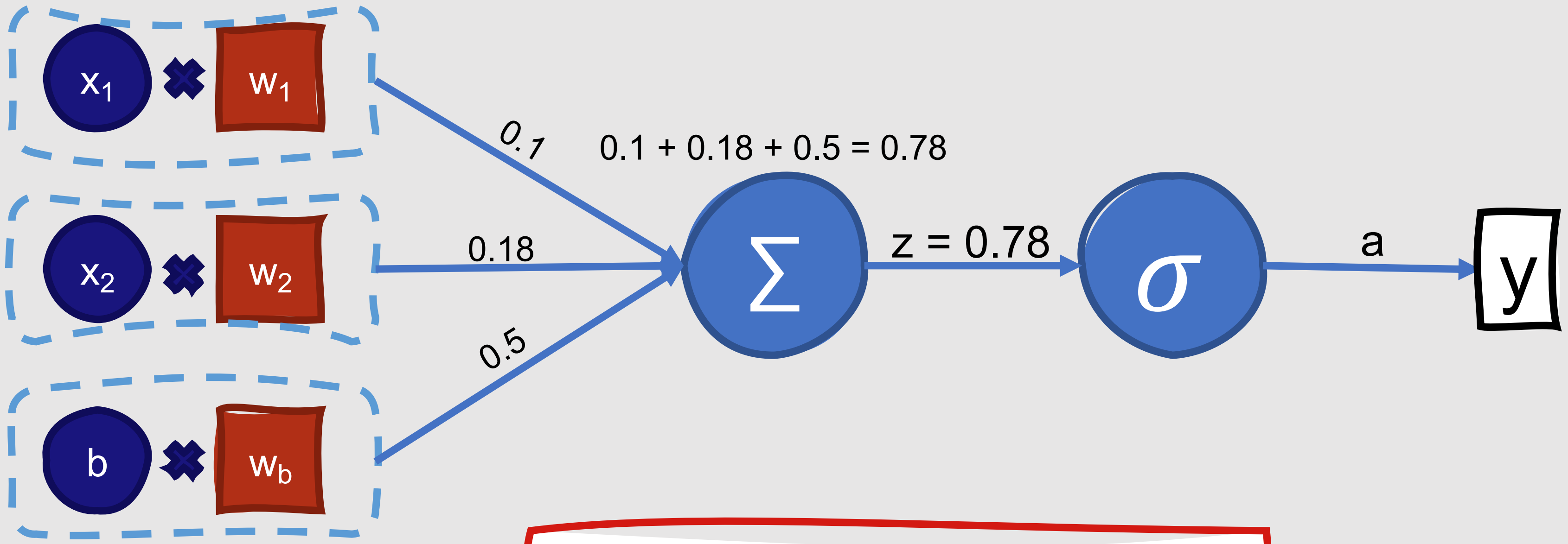
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[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"

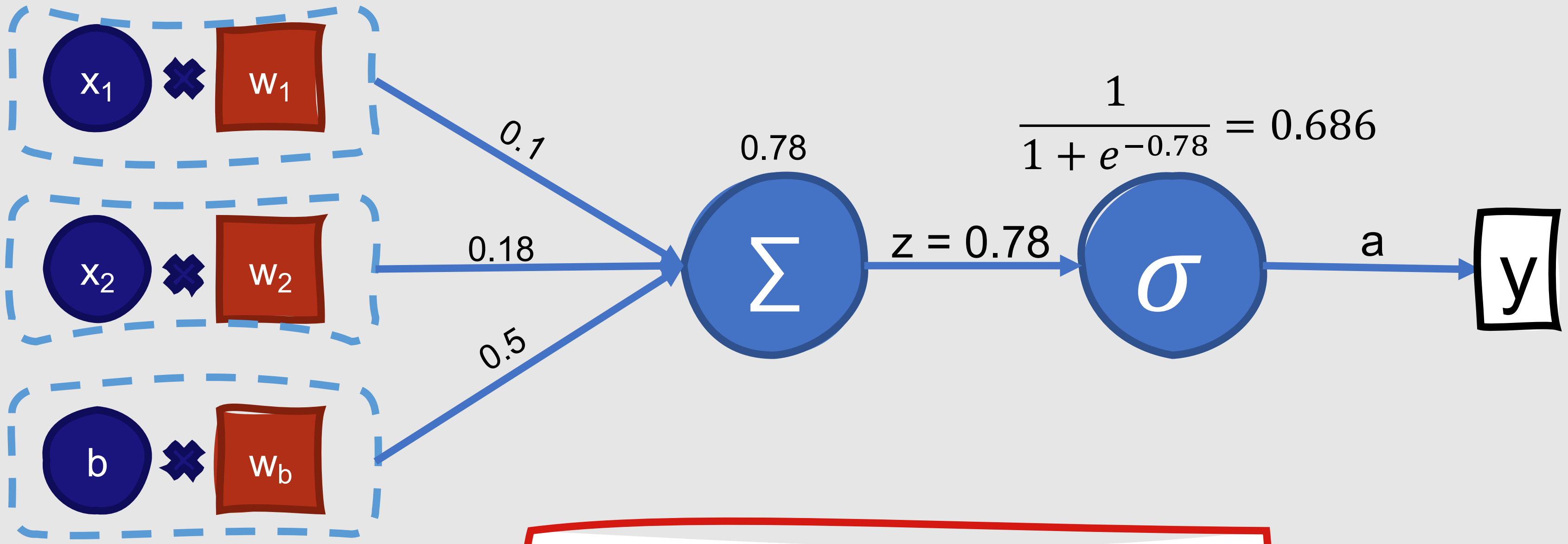
Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte")

[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”

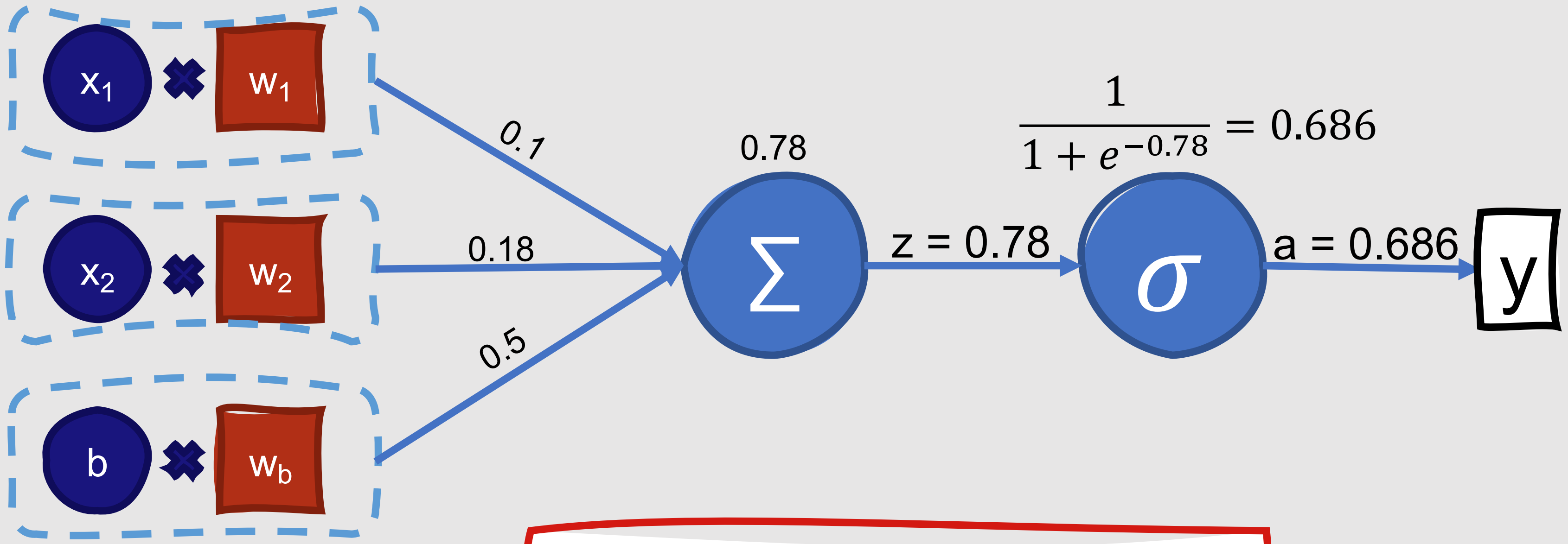
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[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"

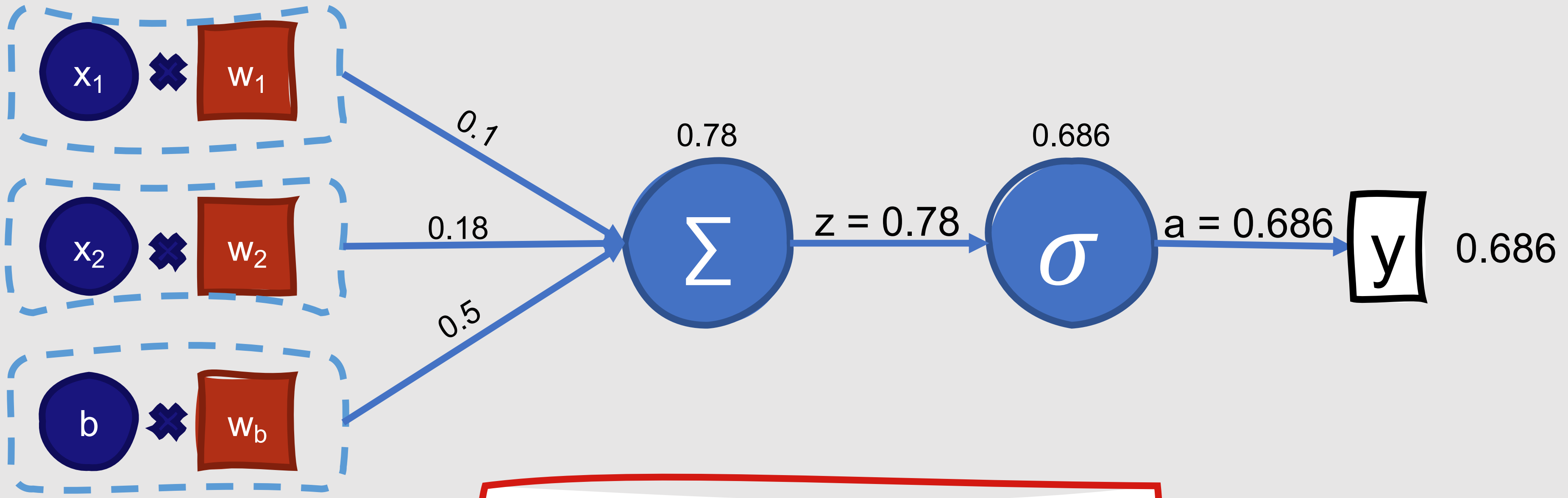
Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte")

[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!

exponential linear unit (elu)

softmax

scaled exponential linear unit (selu)

softplus

softsign

rectified linear unit (relu)

hyperbolic tangent (tanh)

sigmoid

Particularly Common Activation Functions

exponential linear unit (elu)

softmax

scaled exponential linear unit (selu)

softplus

softsign

rectified linear unit (relu)

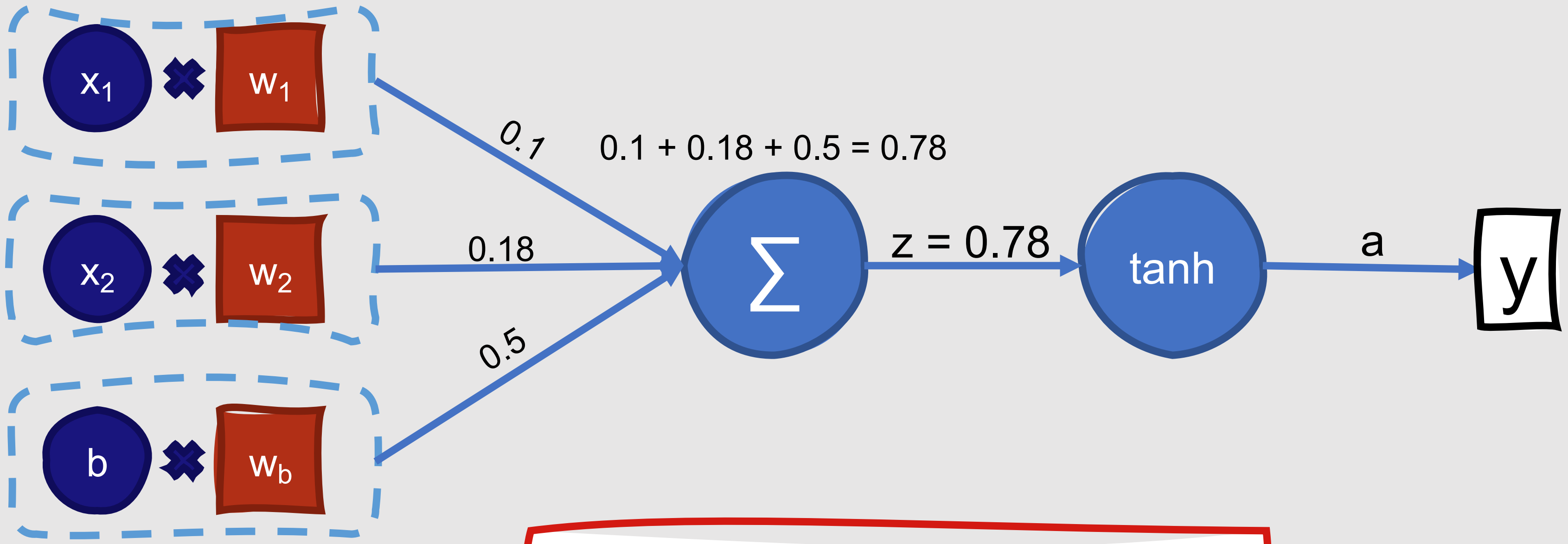
hyperbolic tangent (tanh)

sigmoid

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”

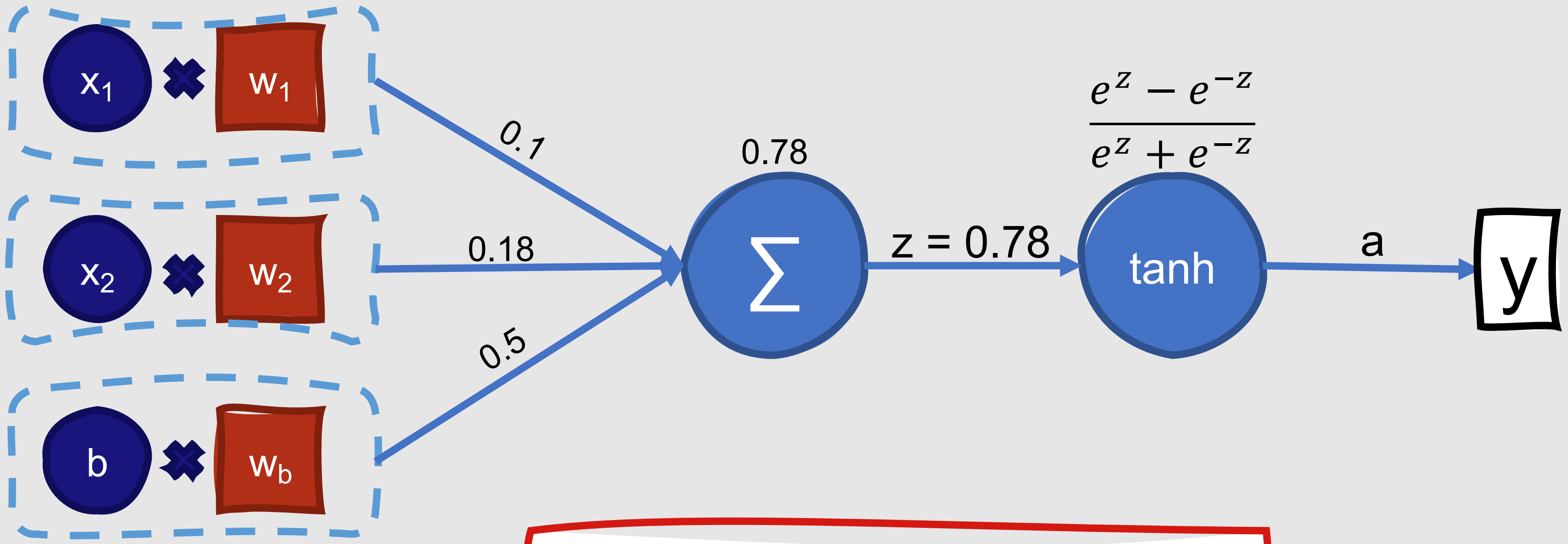
Compute vector (e.g., averaged Word2Vec embeddings for “pumpkin,” “spice,” and “latte”)

[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"

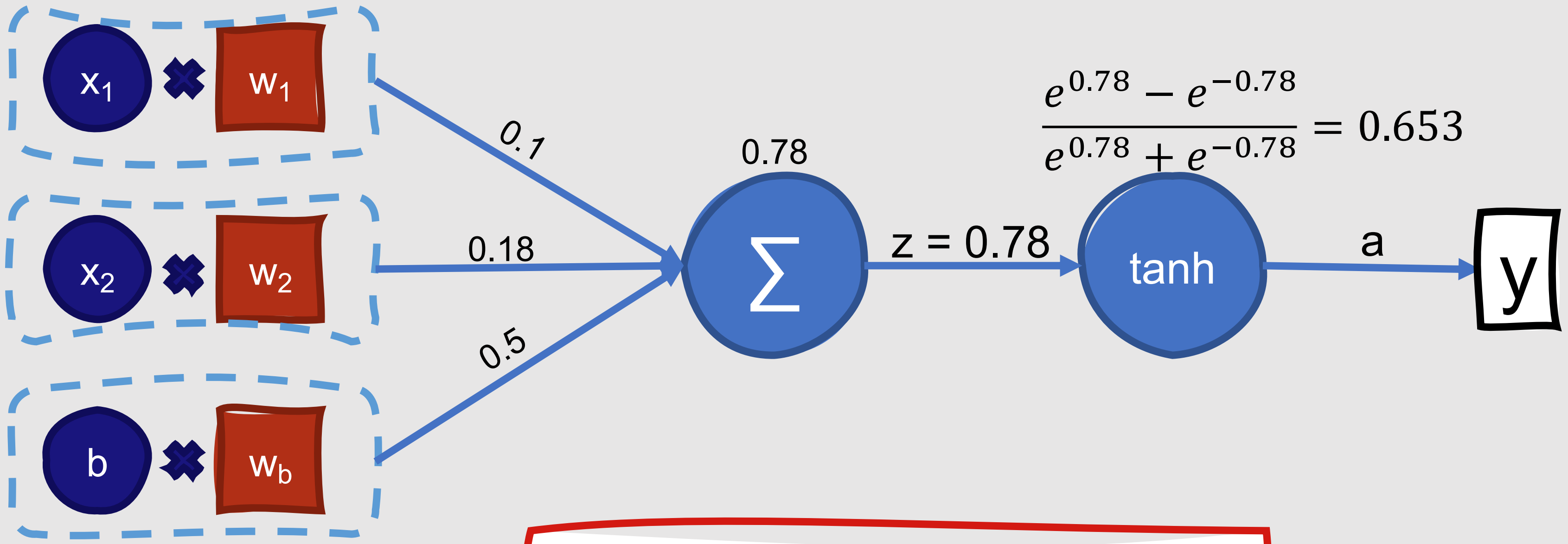
Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte")

[0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with tanh Activation

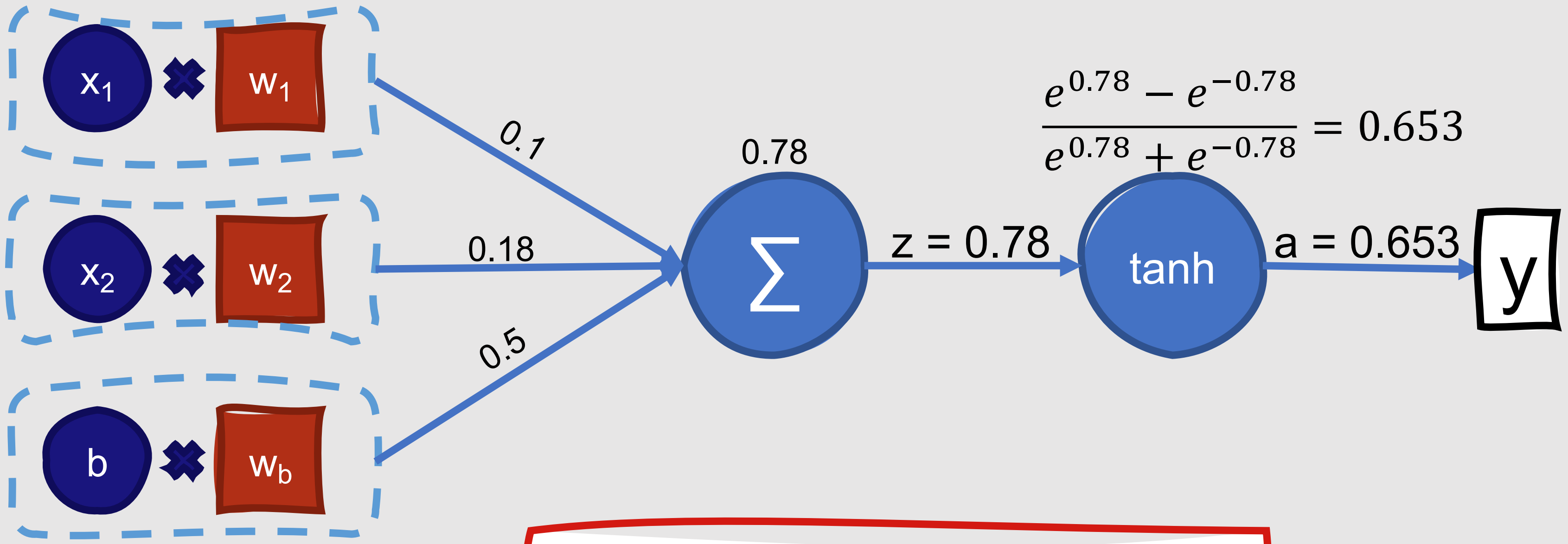


Input: "pumpkin spice latte" → Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte") → [0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with tanh Activation



$$\frac{e^{0.78} - e^{-0.78}}{e^{0.78} + e^{-0.78}} = 0.653$$

Input: "pumpkin spice latte"

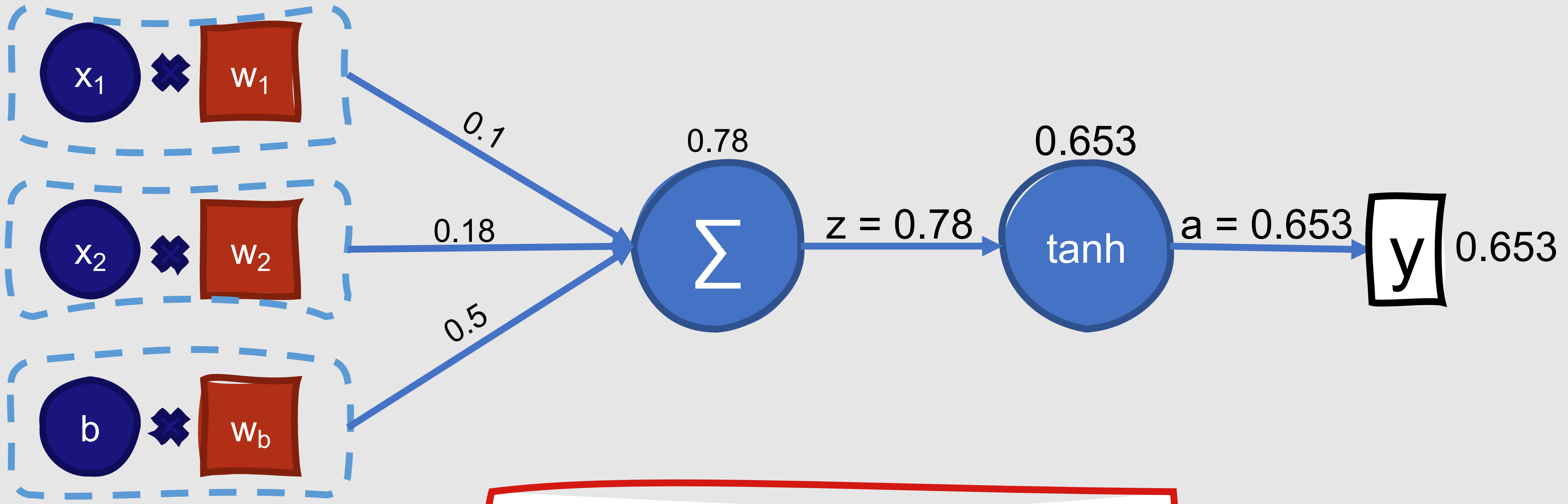
Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte")

[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]

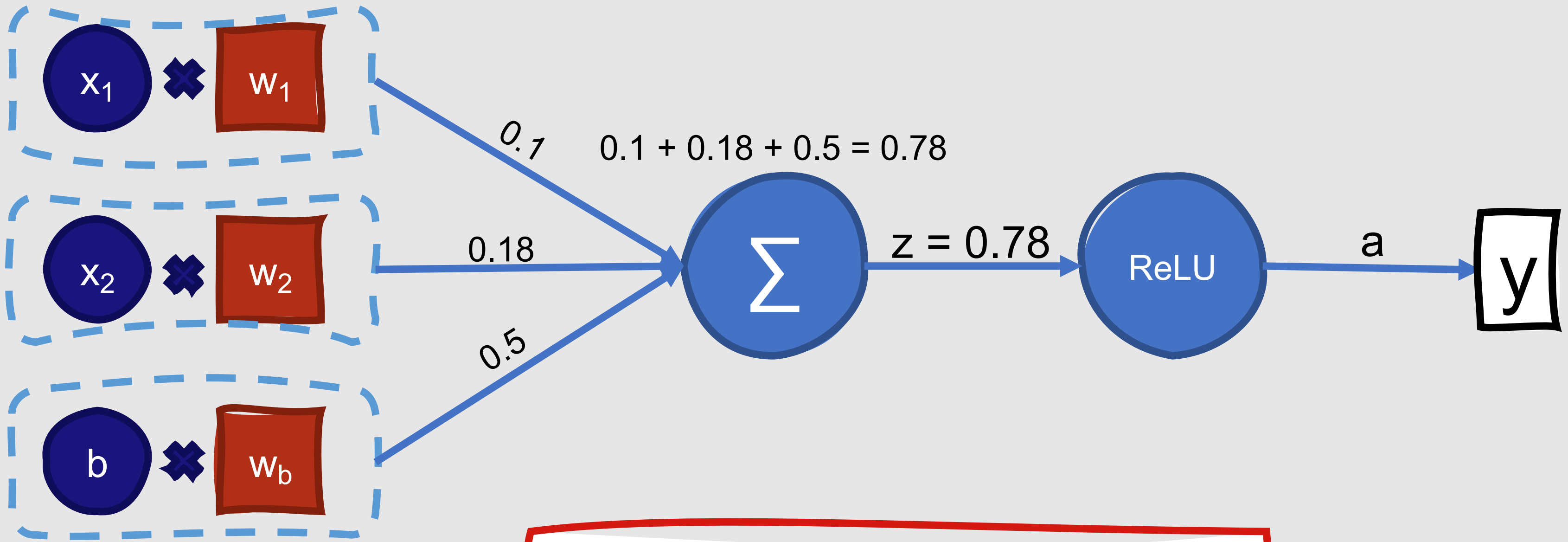
Weights (Input): [0.2, 0.3]
Weight (Bias): [0.5]

Bias: 1.0

Activation: ReLU

- Ranges from 0 to ∞
- 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"

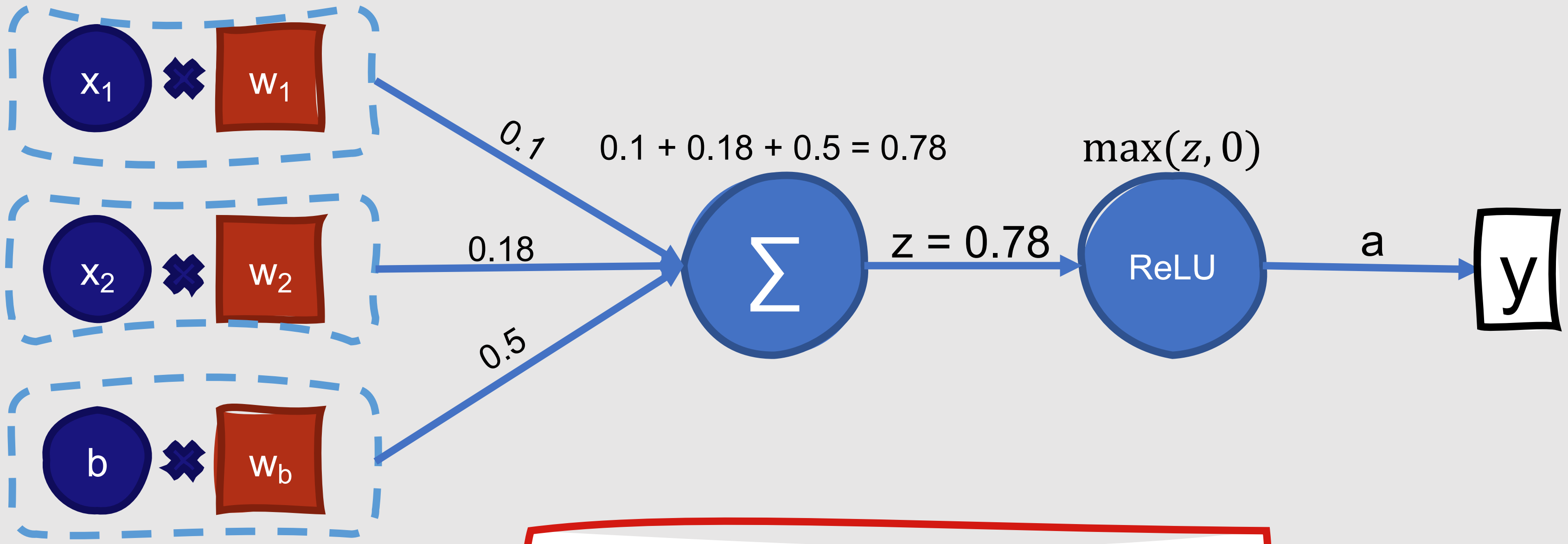
Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte")

[0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with ReLU Activation

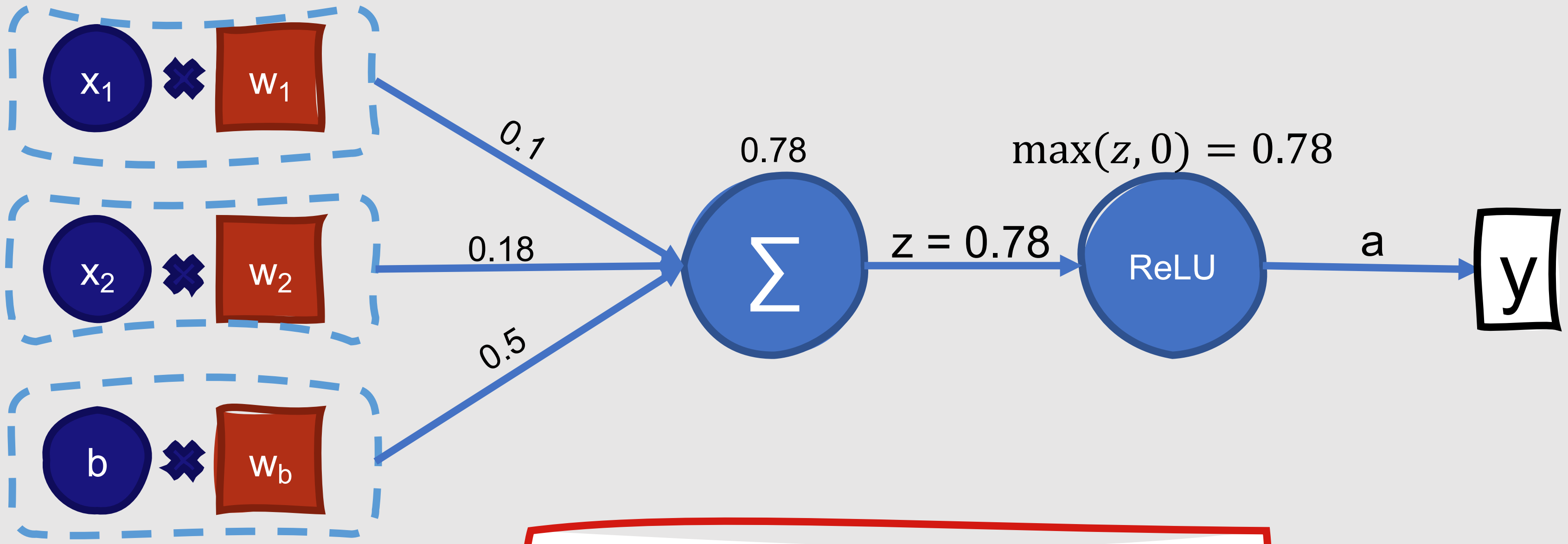


Input: "pumpkin spice latte" → Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte") → [0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with ReLU Activation



Input: "pumpkin spice latte"

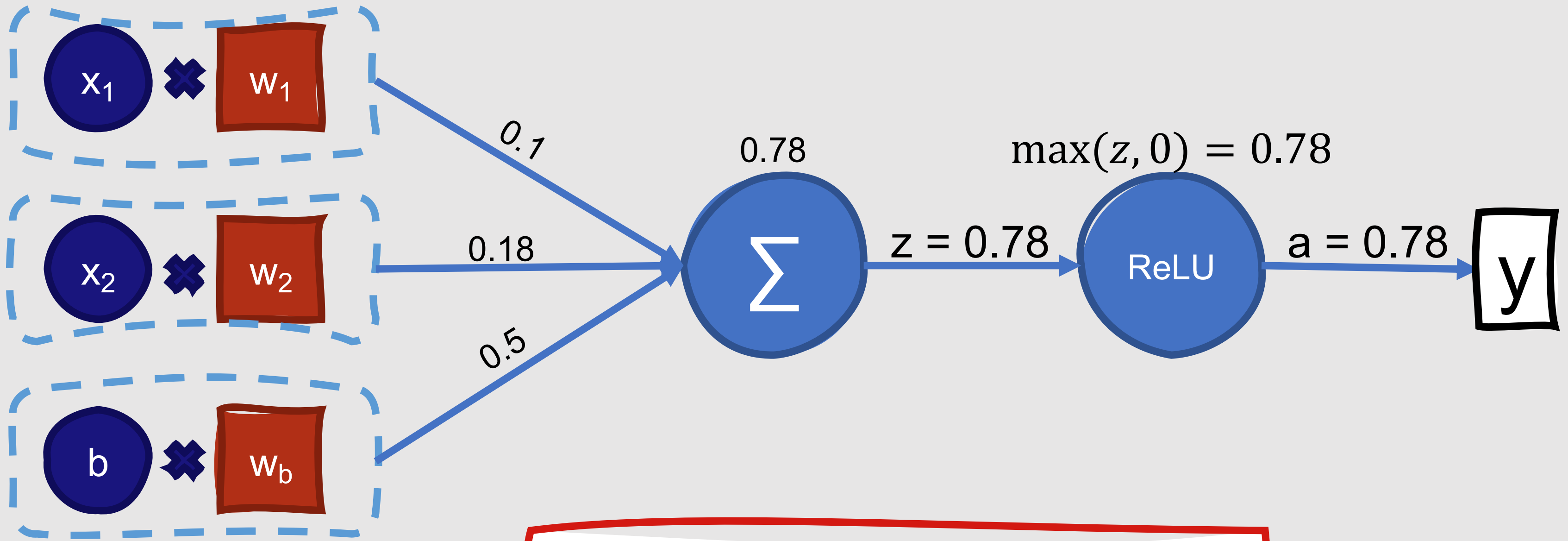
Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte")

[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"

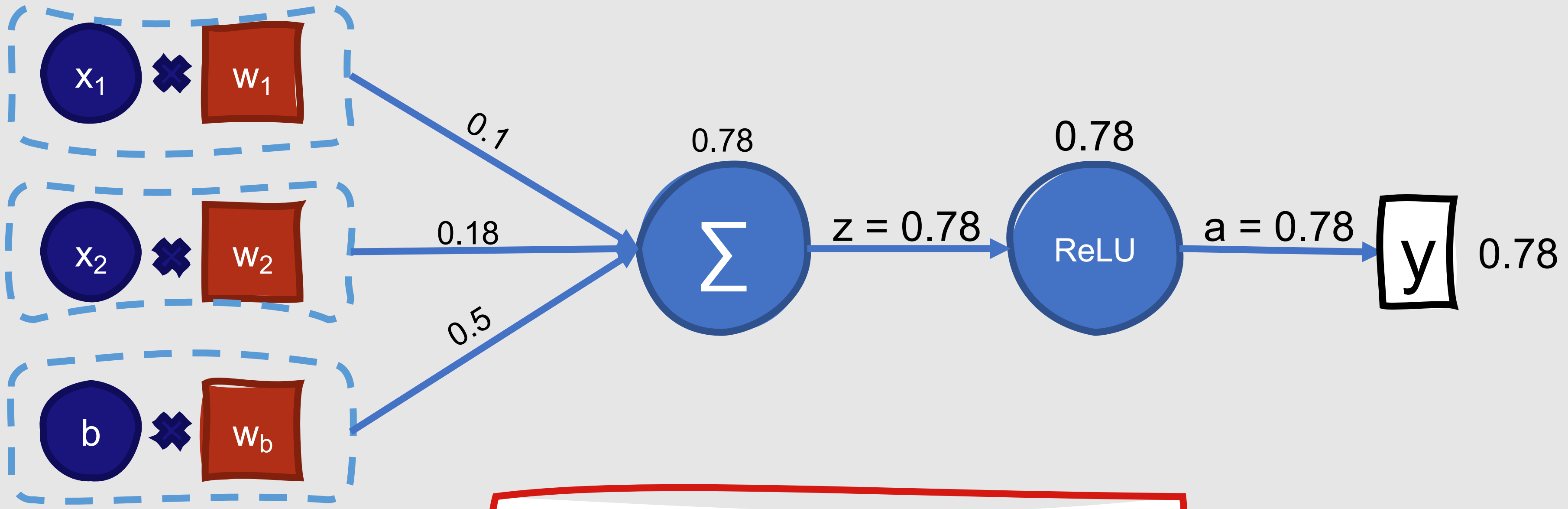
Compute vector (e.g., averaged Word2Vec embeddings for "pumpkin," "spice," and "latte")

[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

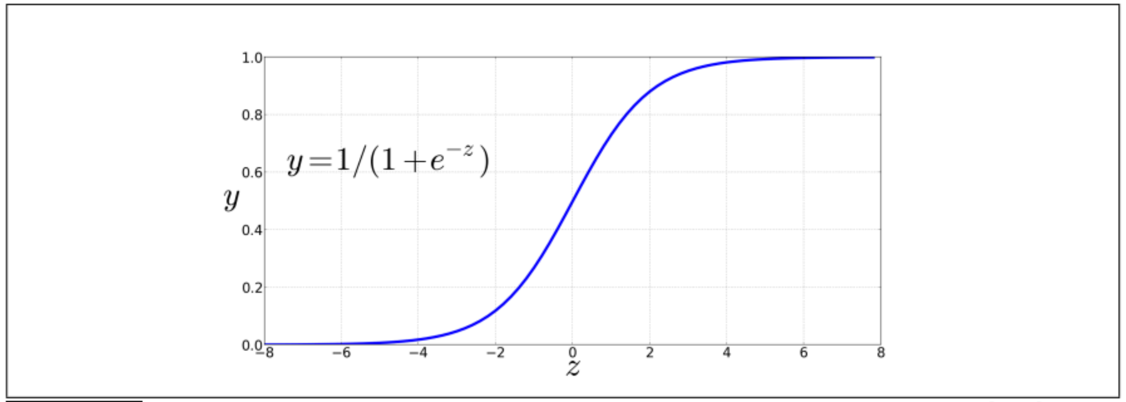


Figure 7.1 The sigmoid function takes a real value and maps it to the range [0, 1]. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

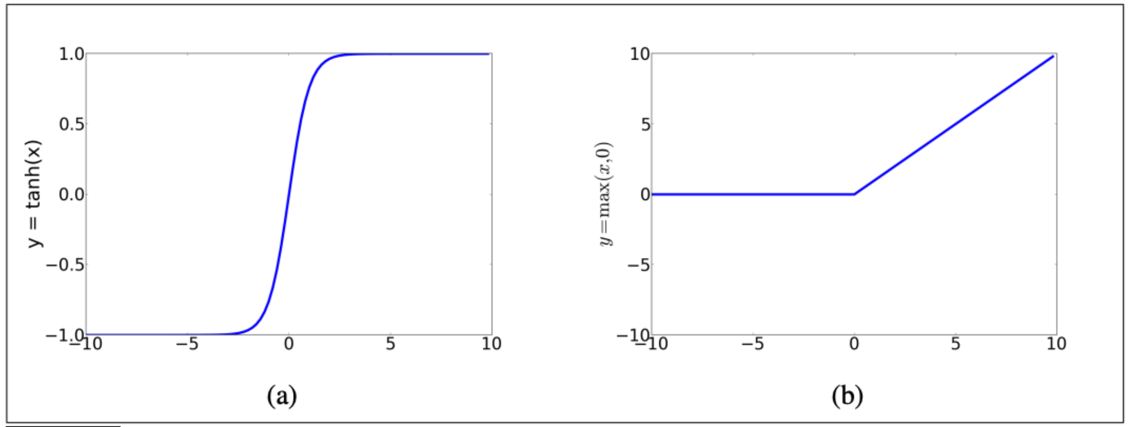


Figure 7.3 The tanh and ReLU activation functions.

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

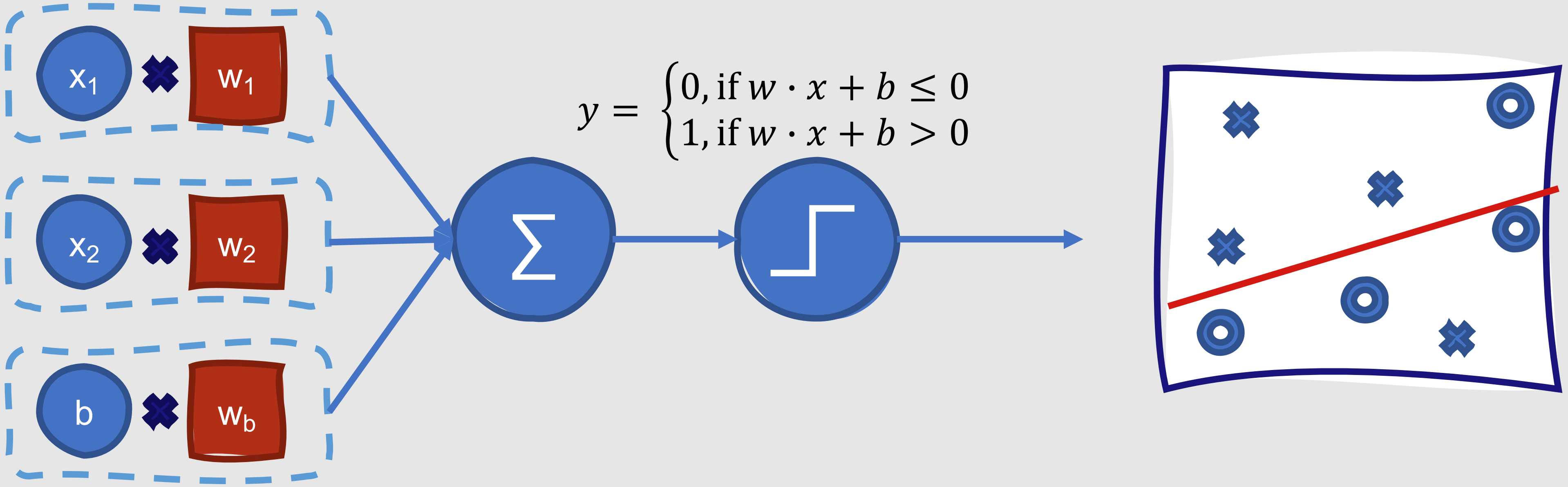
The XOR Problem

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

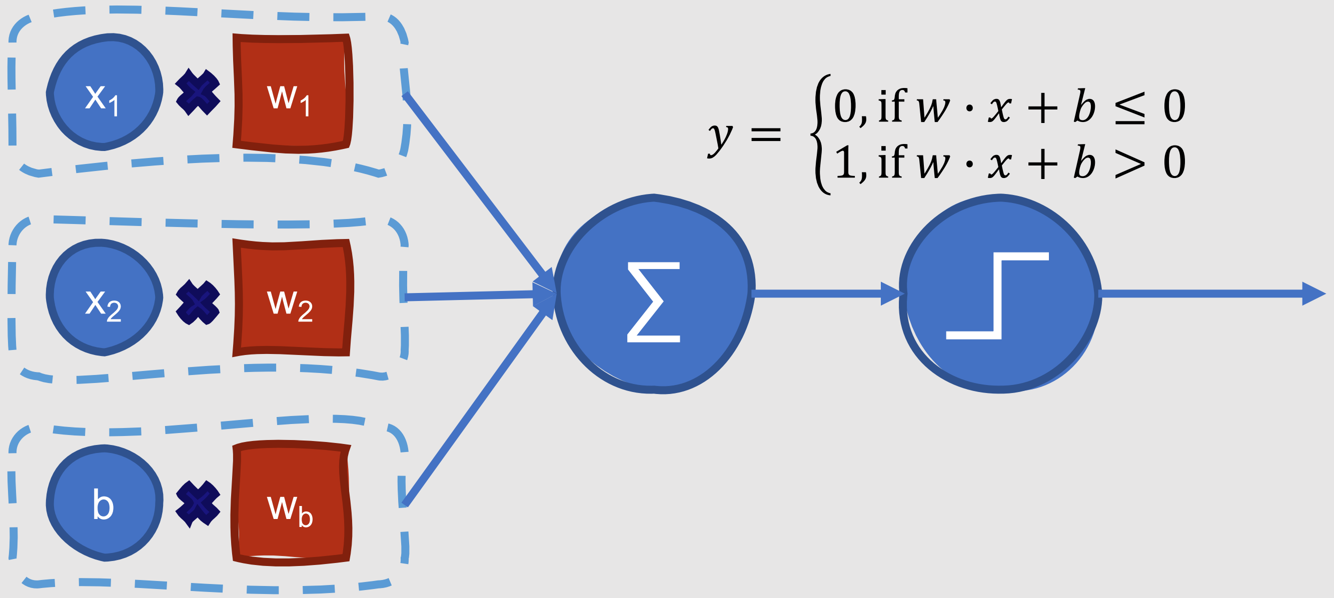
What is a perceptron?

- 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



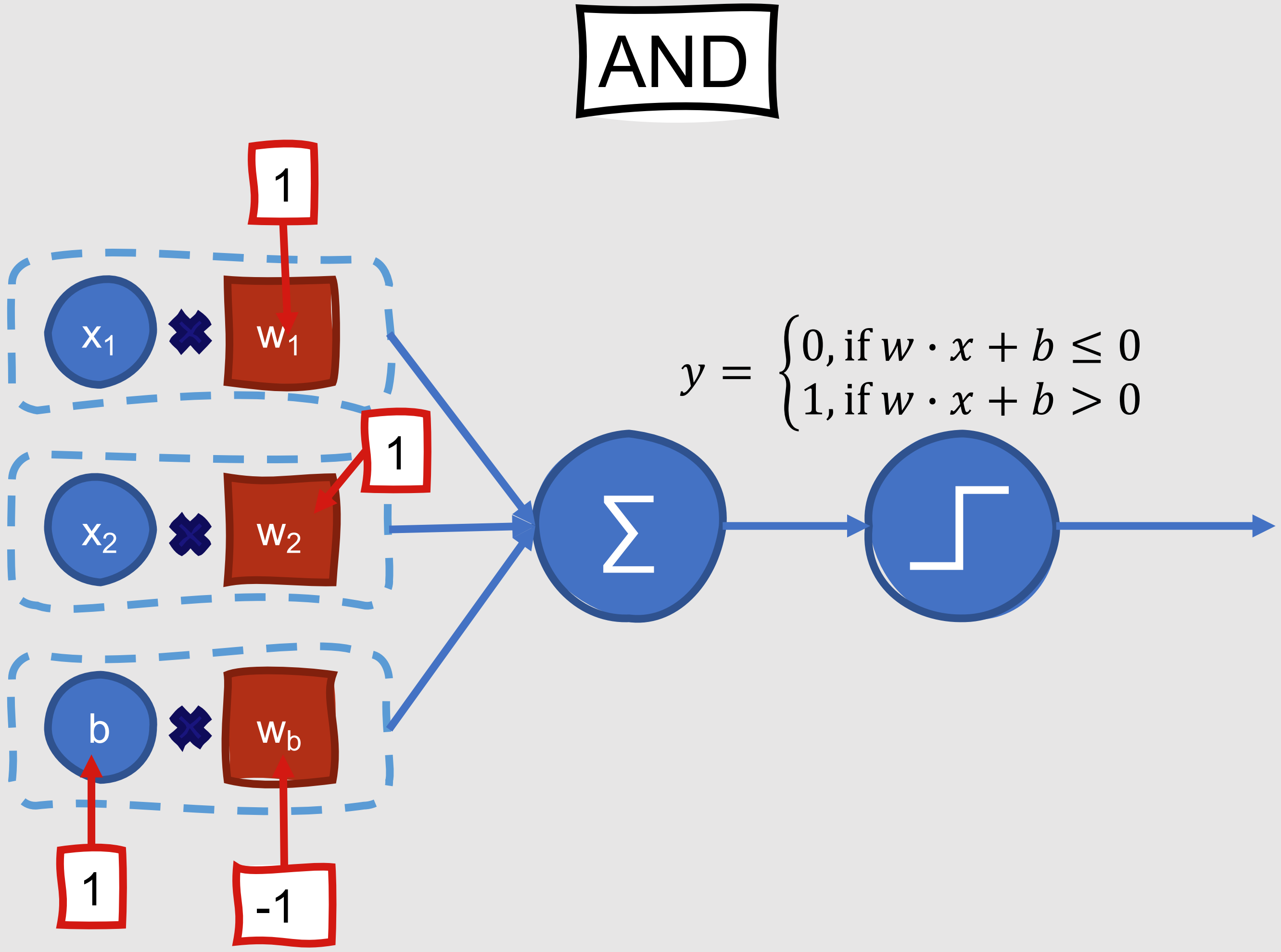
$$y = \begin{cases} 0, \text{if } w \cdot x + b \leq 0 \\ 1, \text{if } w \cdot x + b > 0 \end{cases}$$

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



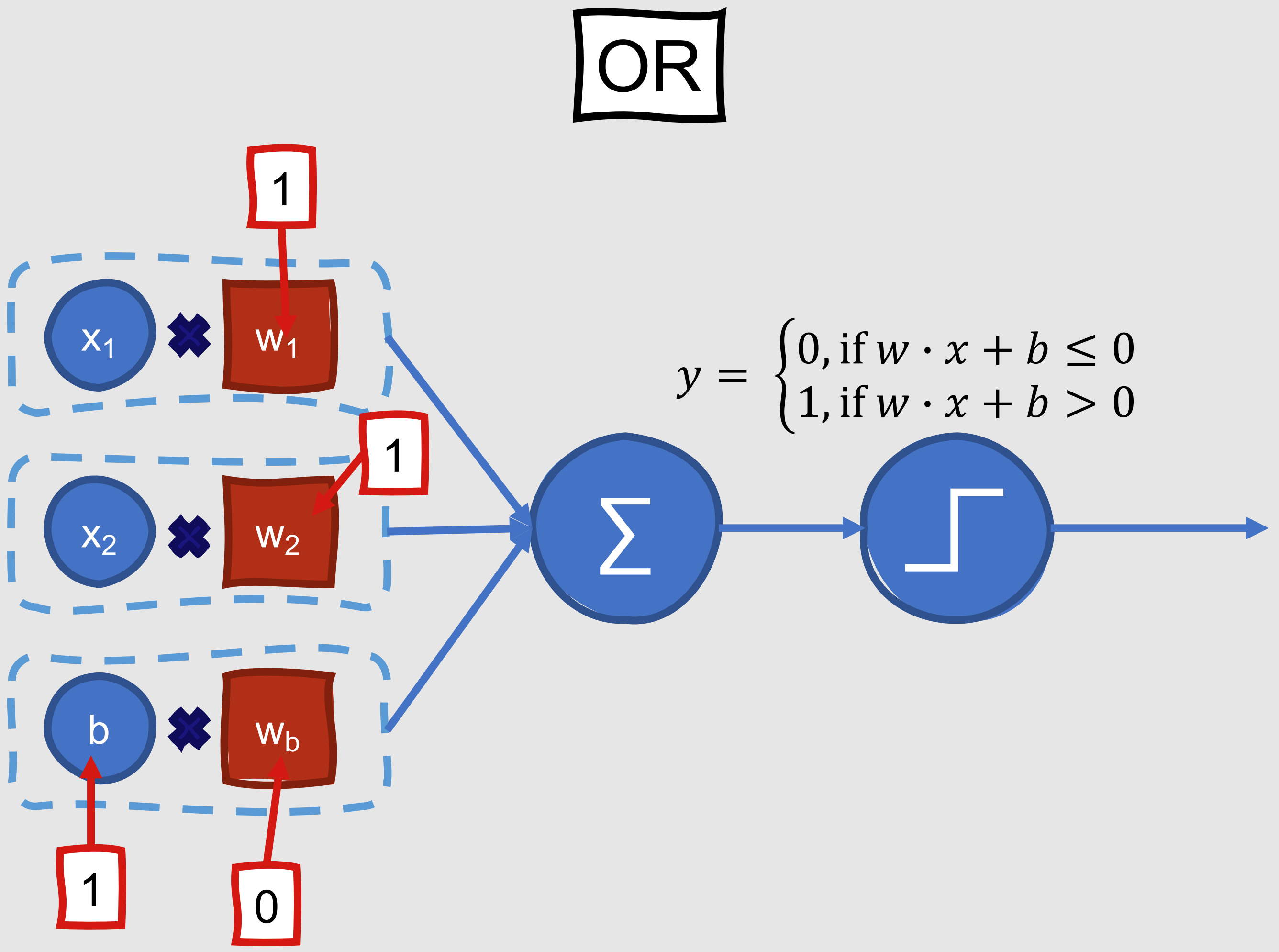
$$y = \begin{cases} 0, \text{if } w \cdot x + b \leq 0 \\ 1, \text{if } w \cdot x + b > 0 \end{cases}$$

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



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

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



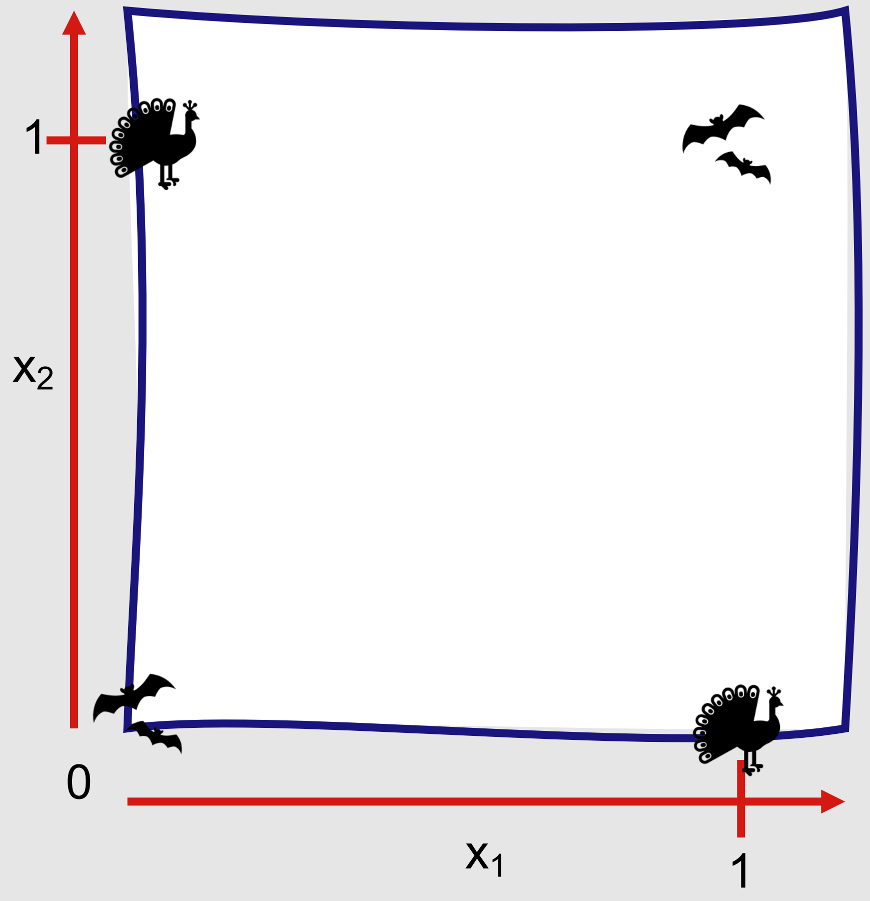
$$y = \begin{cases} 0, \text{if } w \cdot x + b \leq 0 \\ 1, \text{if } w \cdot x + b > 0 \end{cases}$$

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

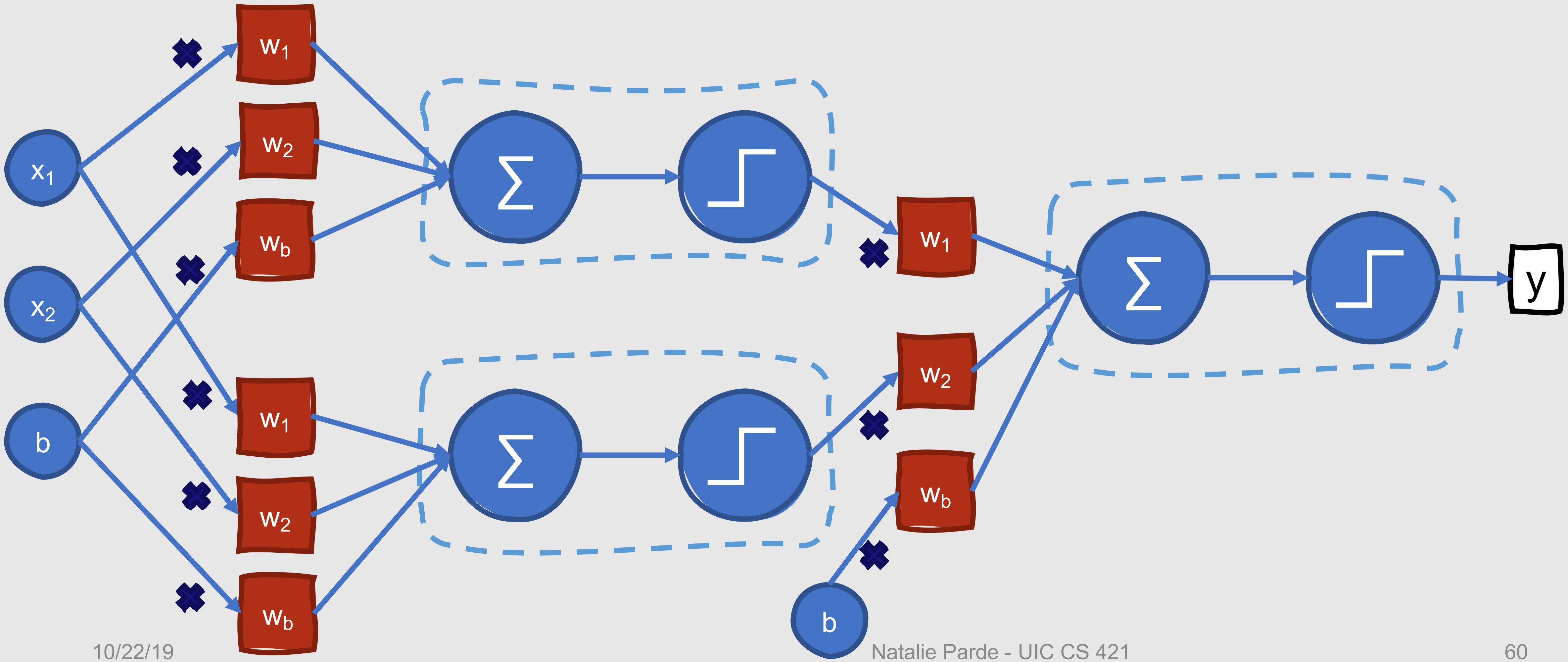
- Why?
  - Perceptrons are **linear classifiers**
- Linear classifiers learn a **decision boundary** that divides a dataset into two parts
  - All data on one side of the decision boundary is assigned a label of 0
  - All data on the other side of the decision boundary is assigned a label of 1
- However, 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

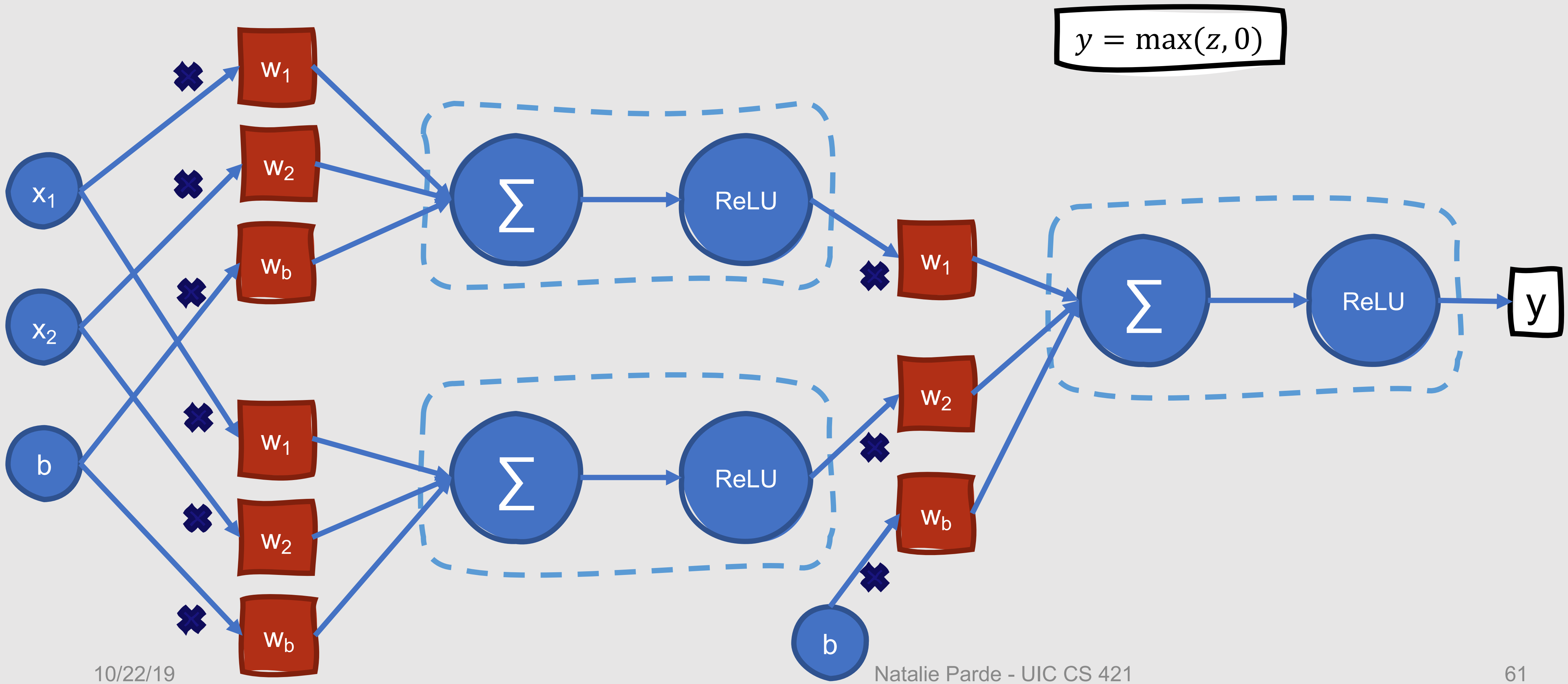
AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0



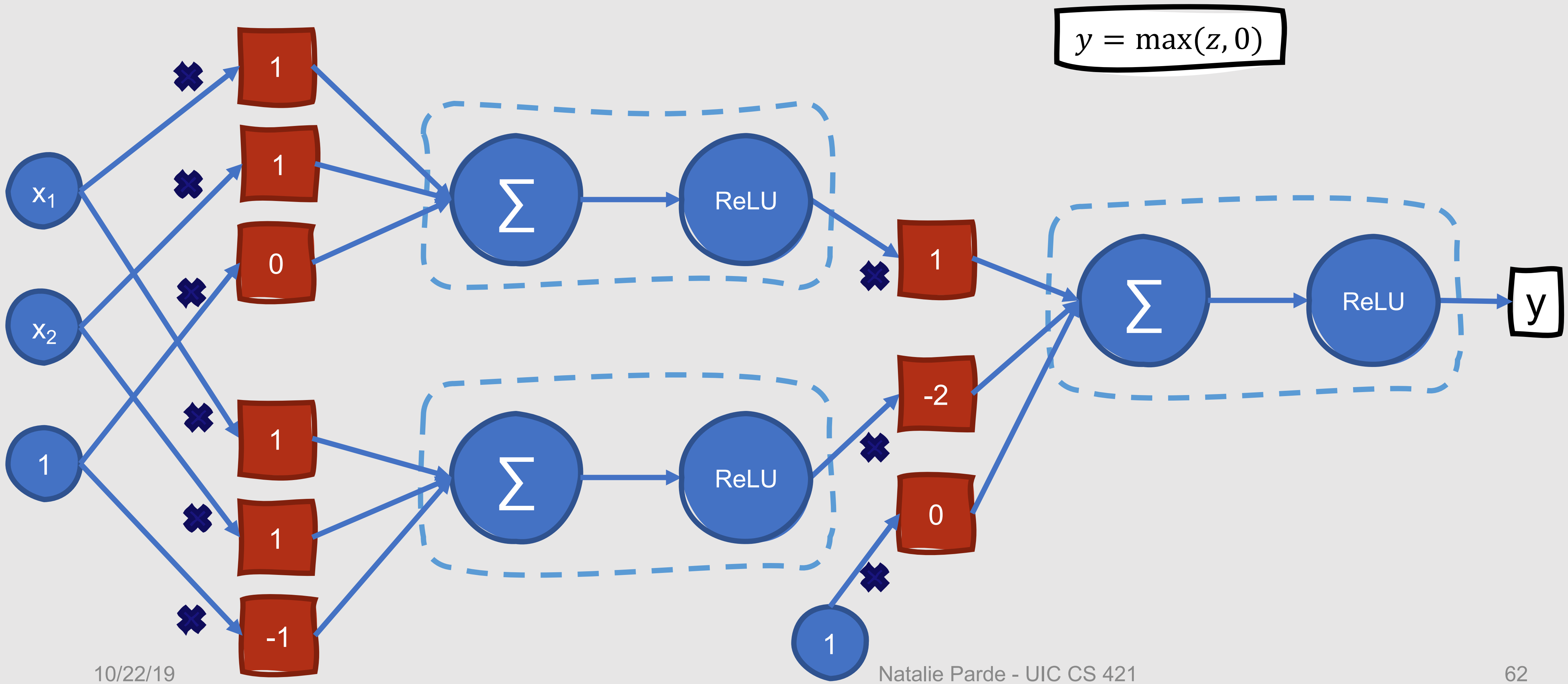
The only way to compute XOR is by combining perceptrons!



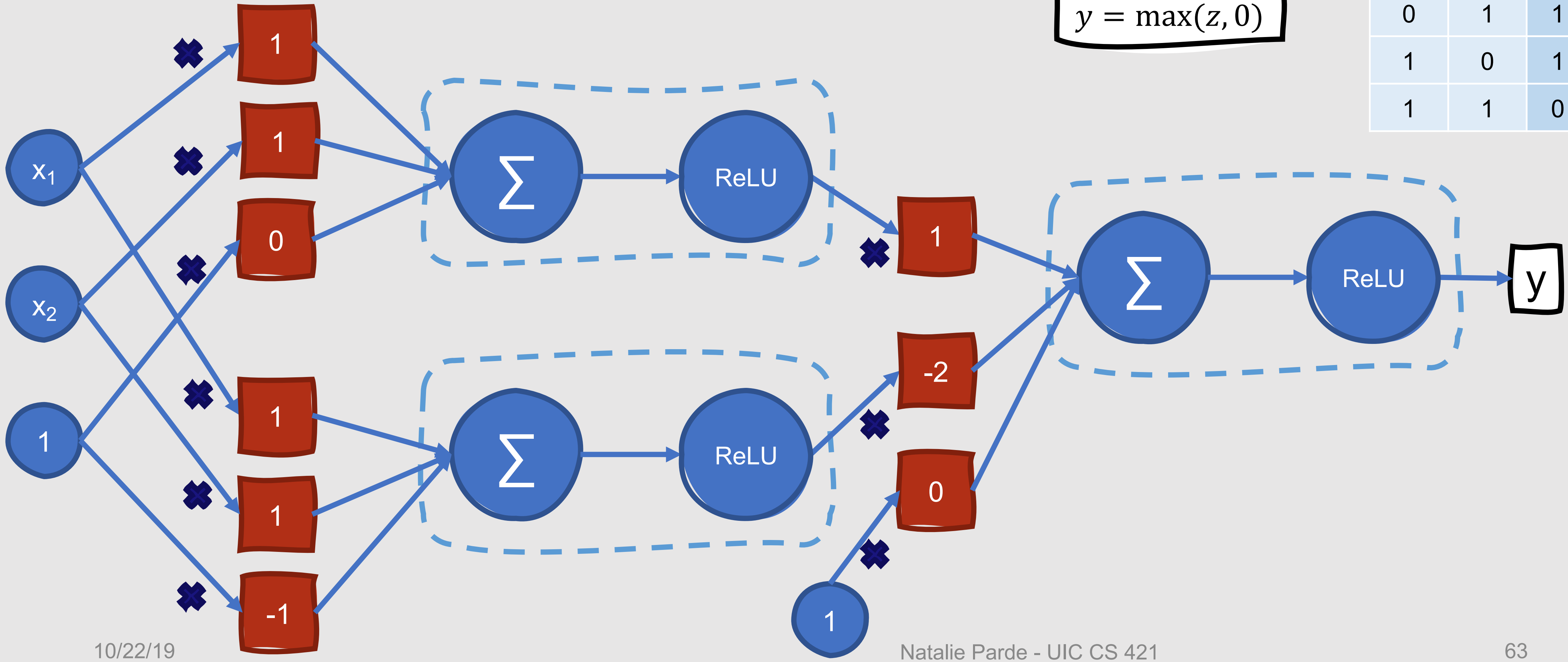
However, since XOR is not a linearly separable function, we'll have to use a nonlinear activation.



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



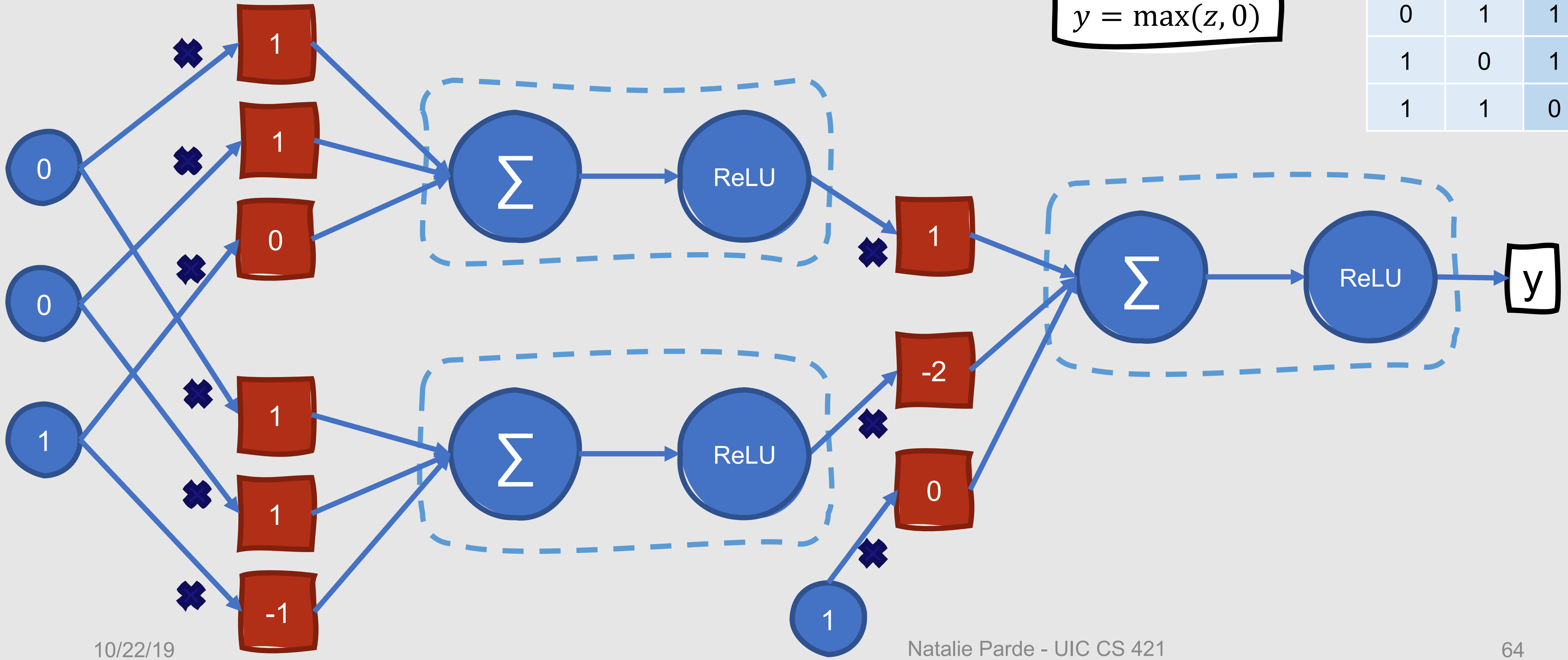
Truth Table Examples: XOR



$y = \max(z, 0)$

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

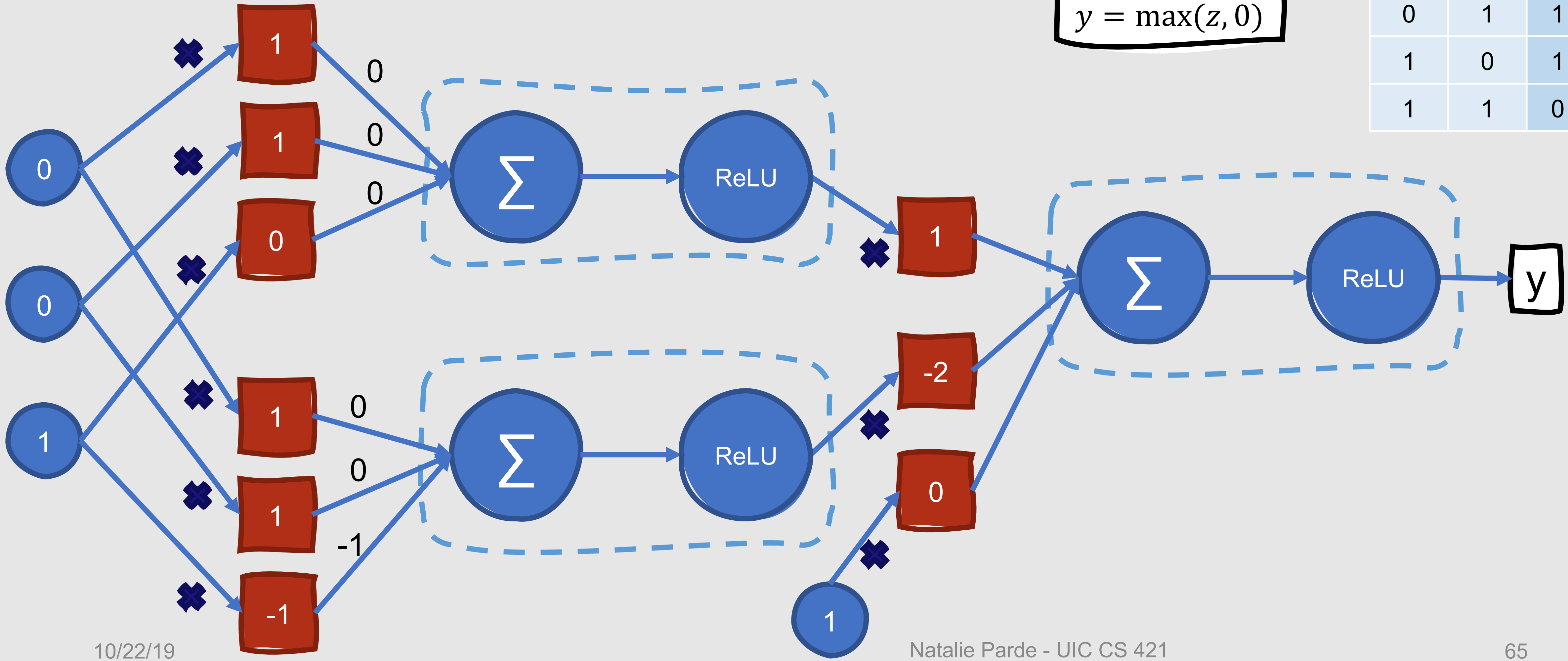
Truth Table Examples: XOR



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

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

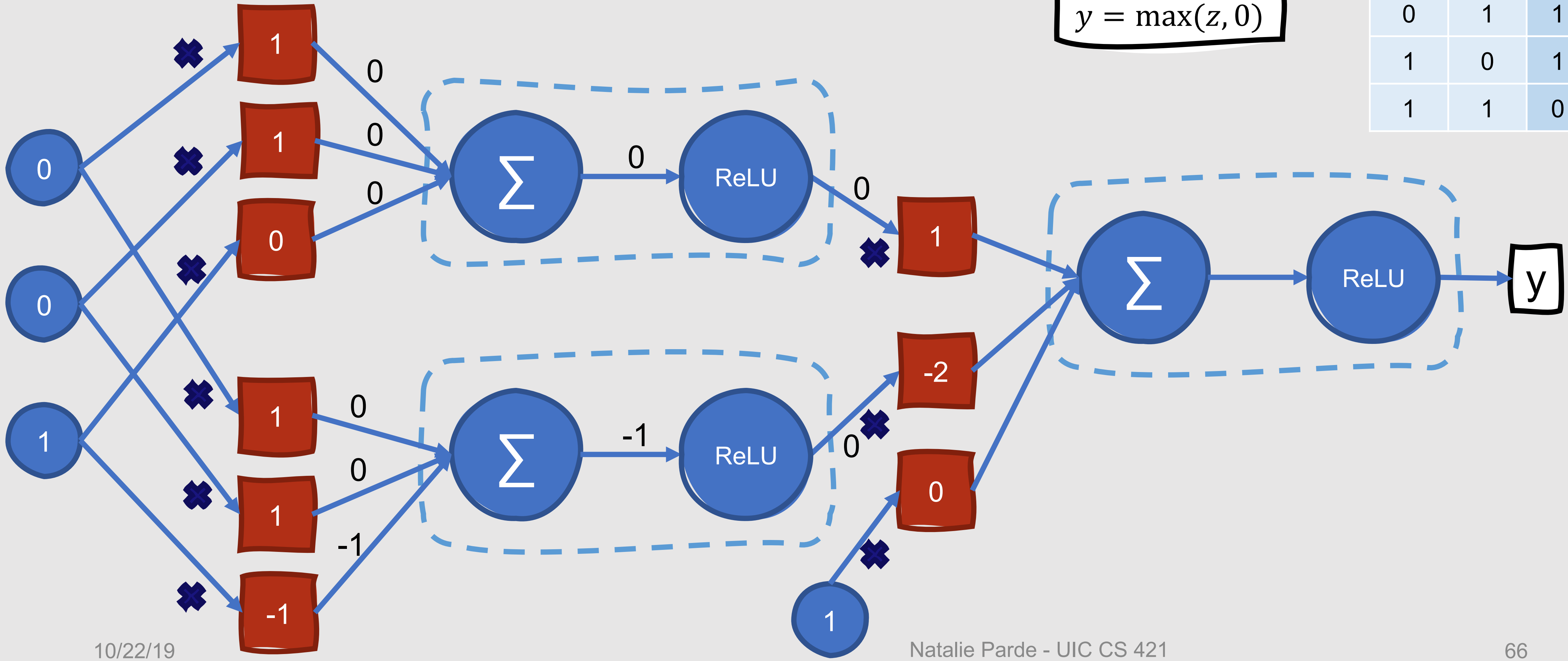
Truth Table Examples: XOR



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

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

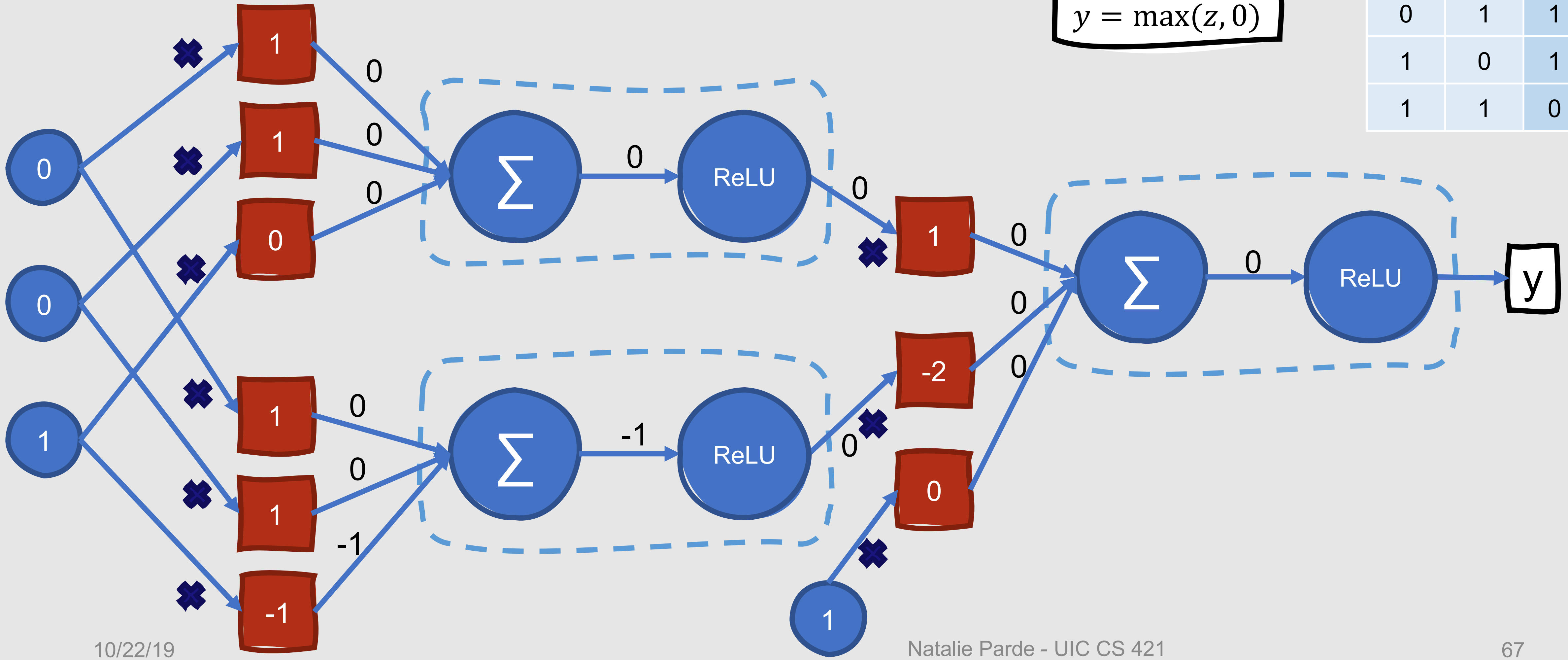
Truth Table Examples: XOR



$y = \max(z, 0)$

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

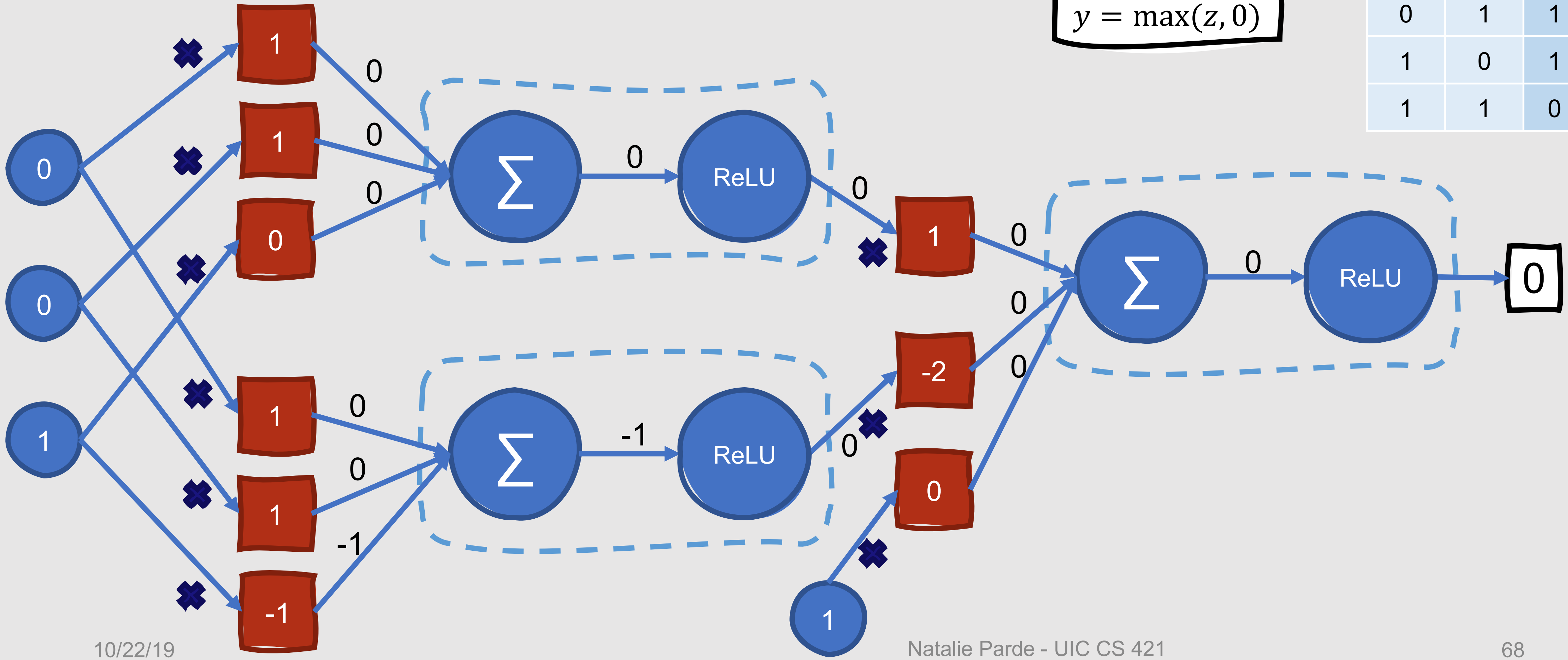
Truth Table Examples: XOR



$y = \max(z, 0)$

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

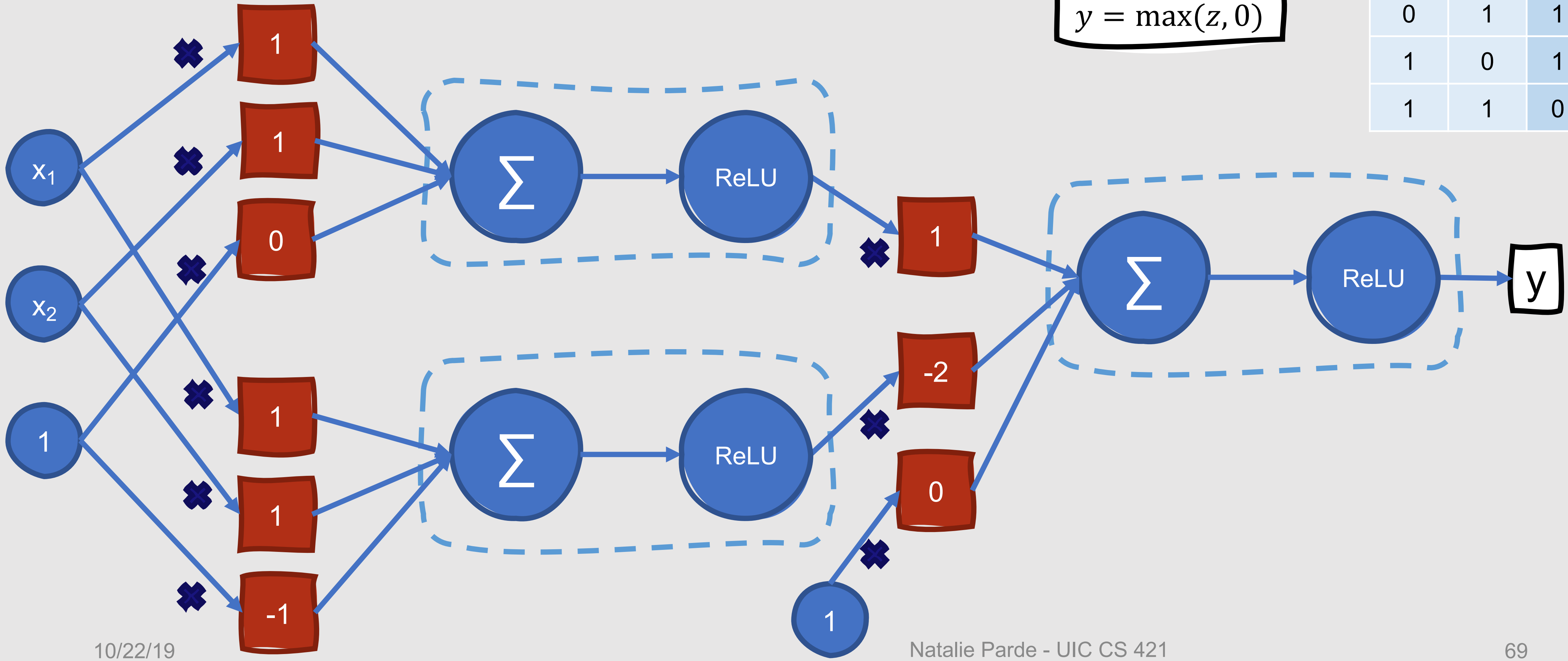
Truth Table Examples: XOR



$y = \max(z, 0)$

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

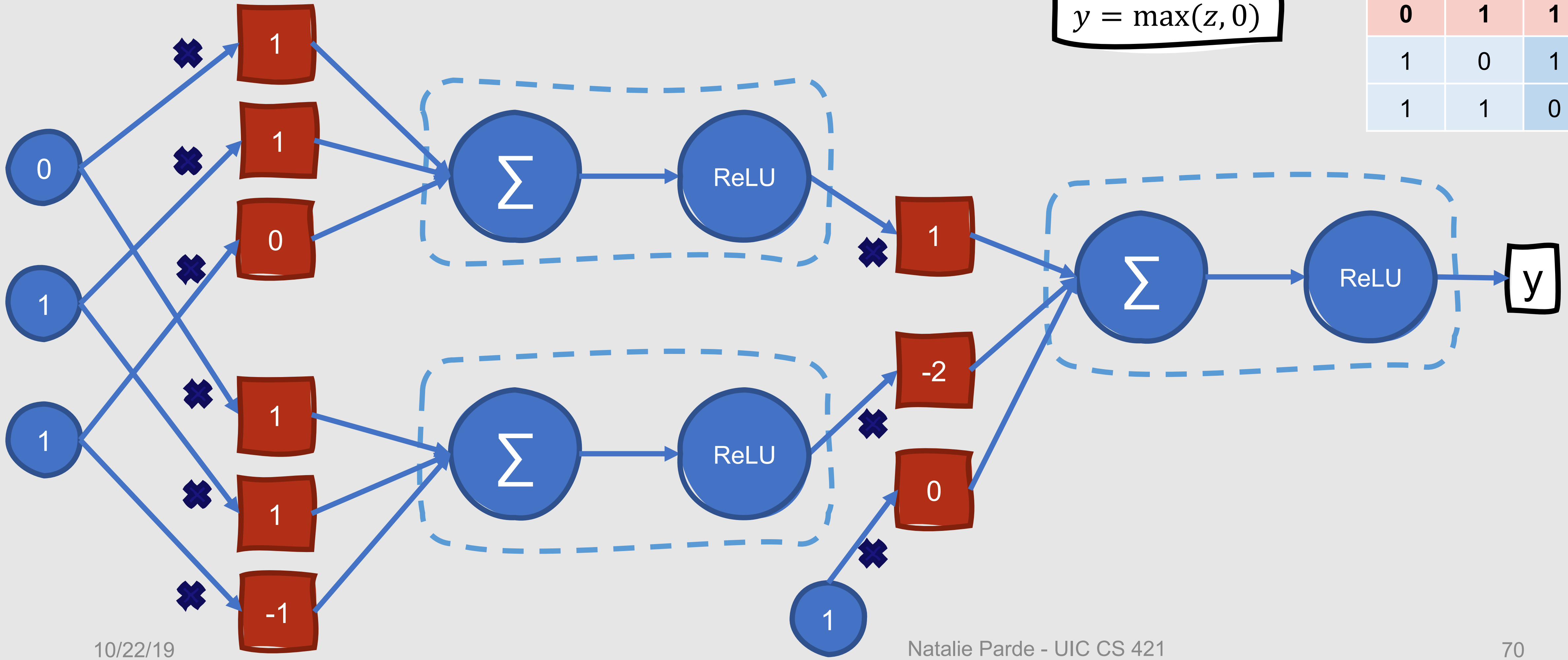
Truth Table Examples: XOR



$y = \max(z, 0)$

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

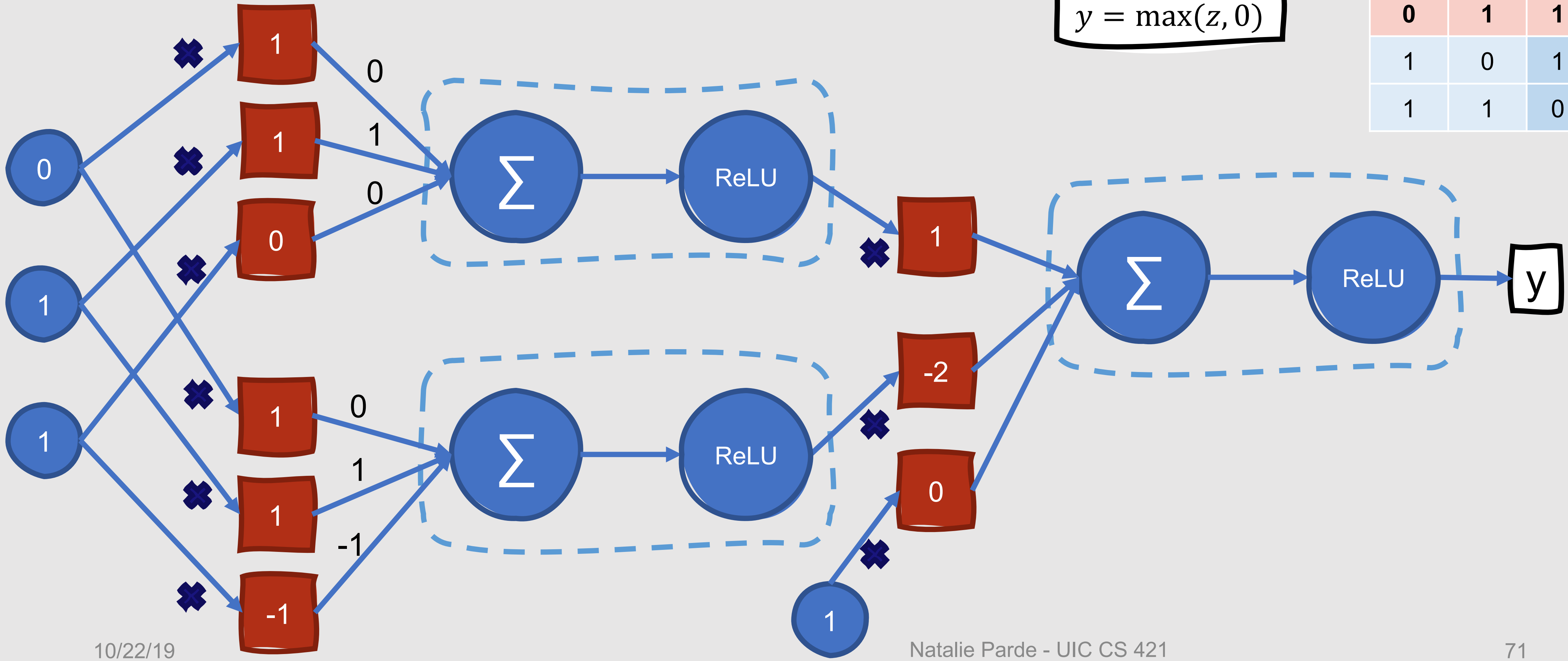
Truth Table Examples: XOR



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

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

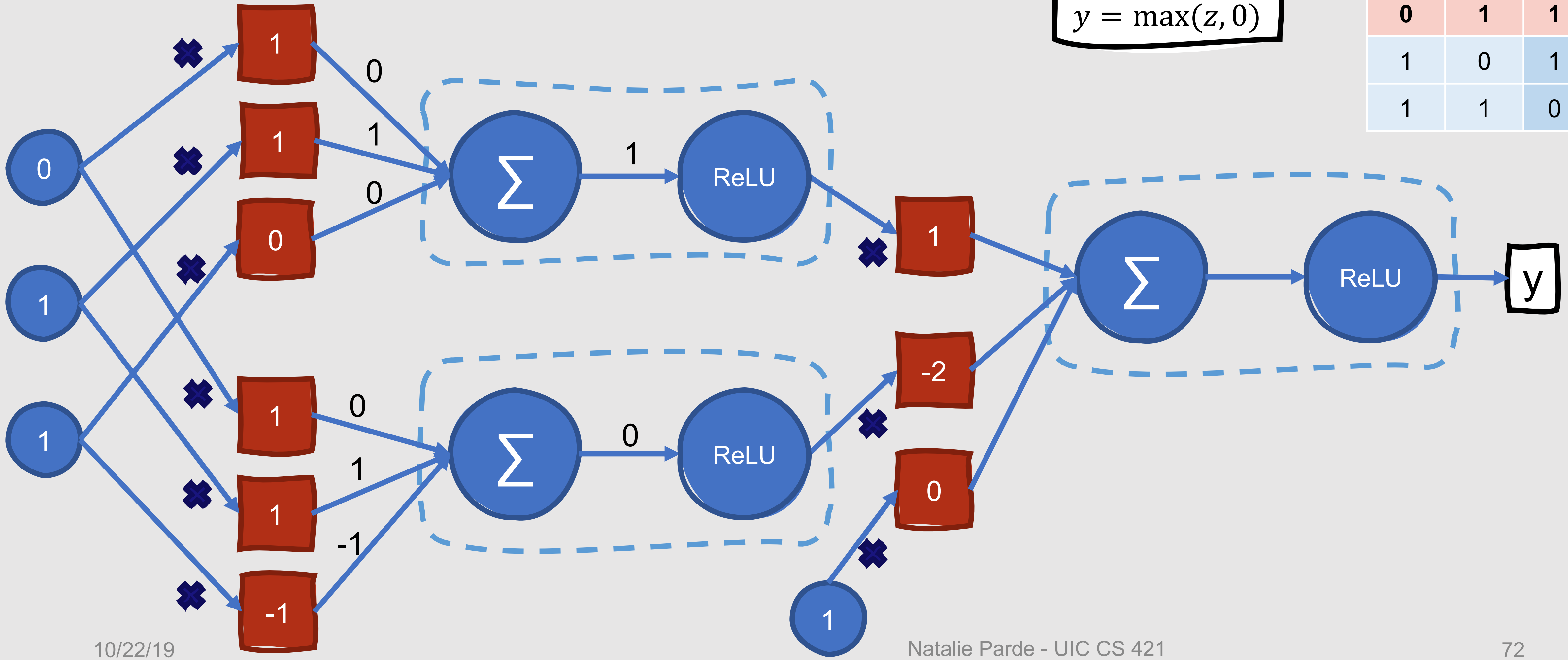
Truth Table Examples: XOR



$y = \max(z, 0)$

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

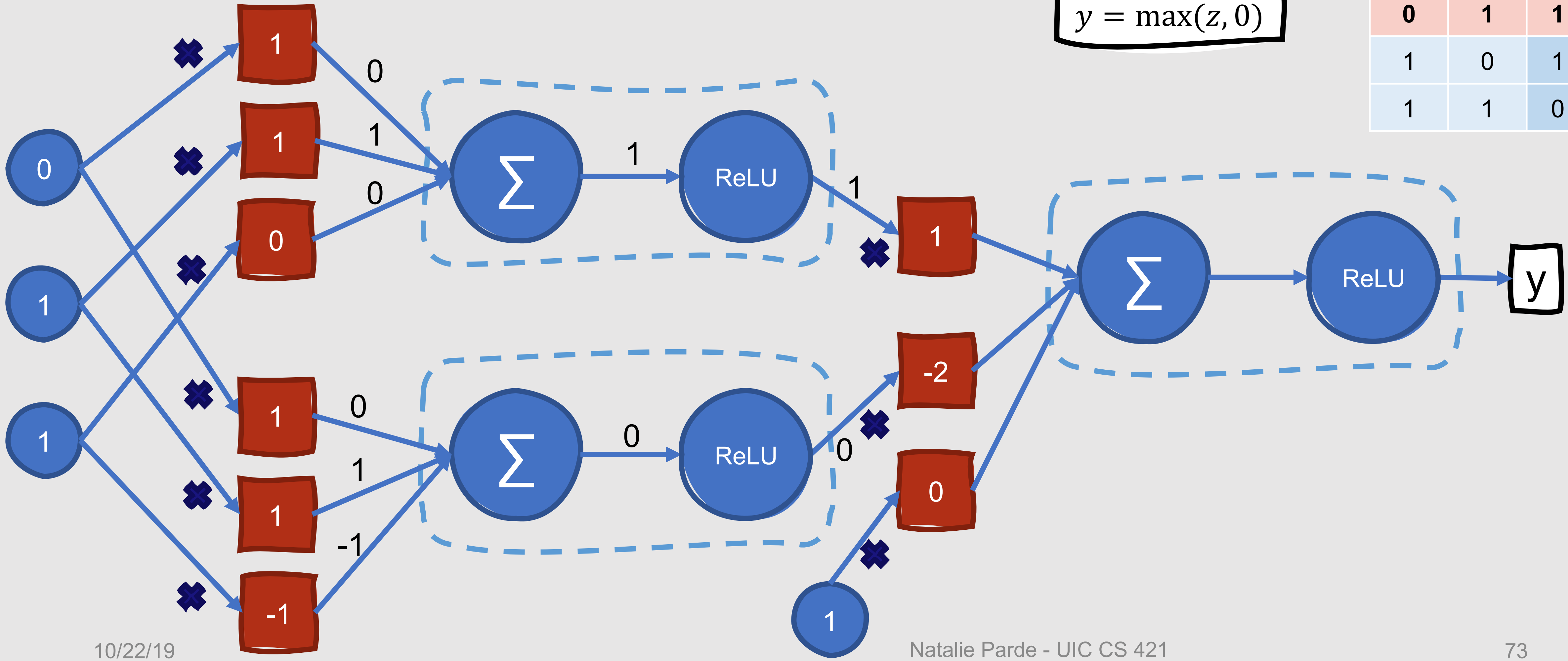
Truth Table Examples: XOR



$y = \max(z, 0)$

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

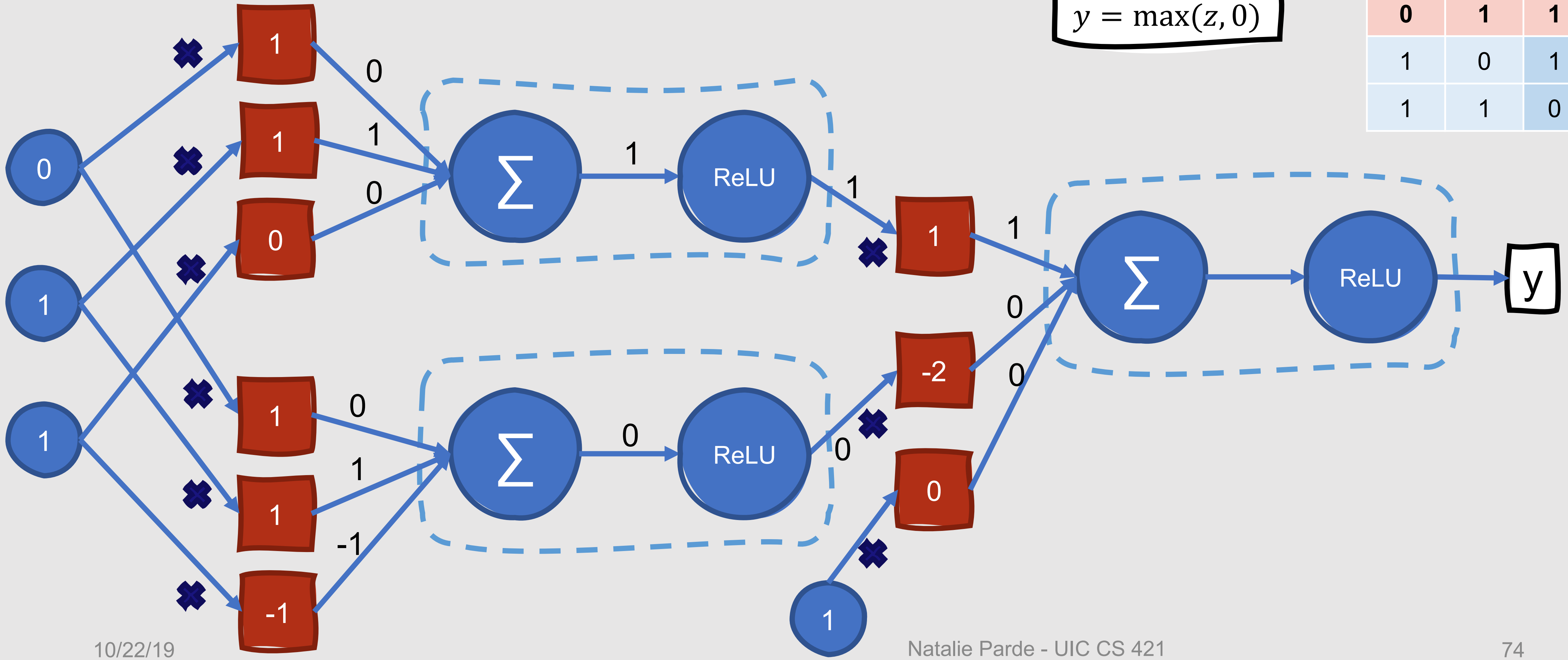
Truth Table Examples: XOR



$y = \max(z, 0)$

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

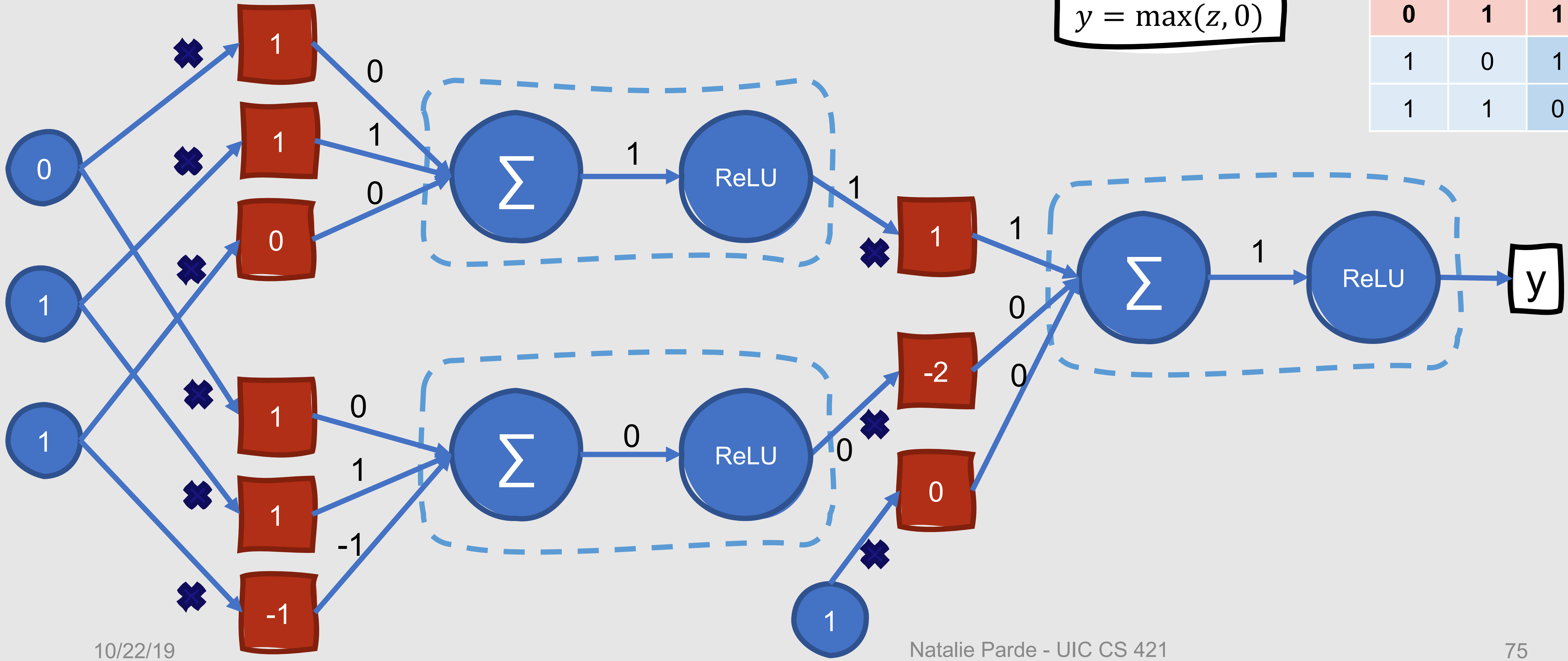
Truth Table Examples: XOR



$y = \max(z, 0)$

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

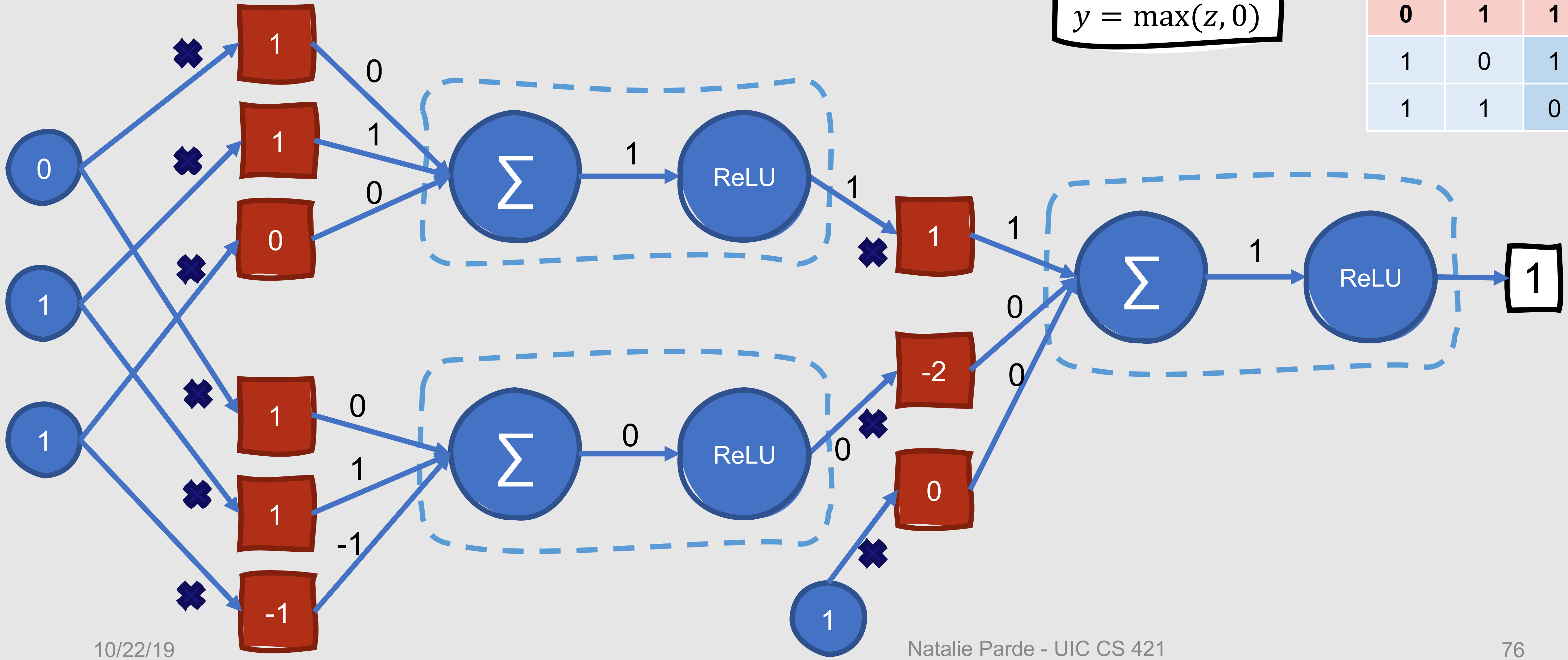
Truth Table Examples: XOR



$y = \max(z, 0)$

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

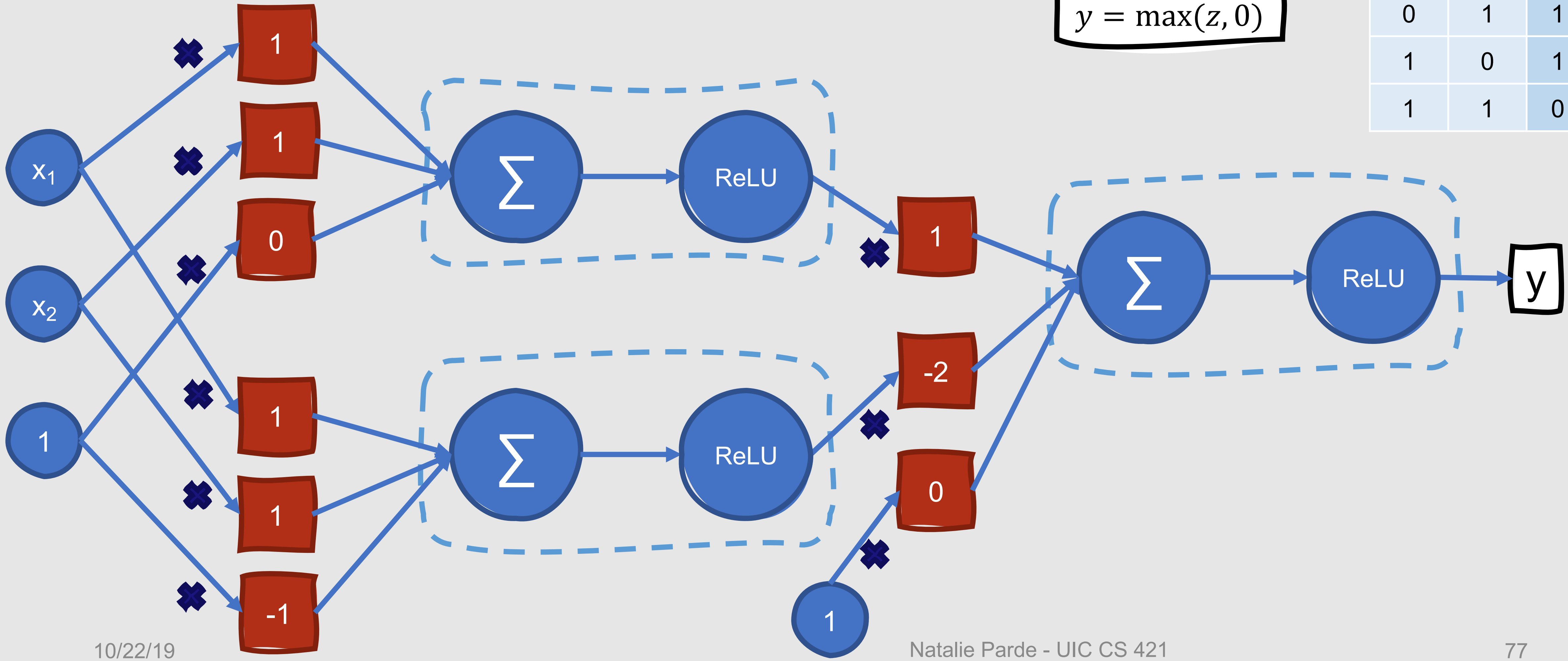
Truth Table Examples: XOR



$y = \max(z, 0)$

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

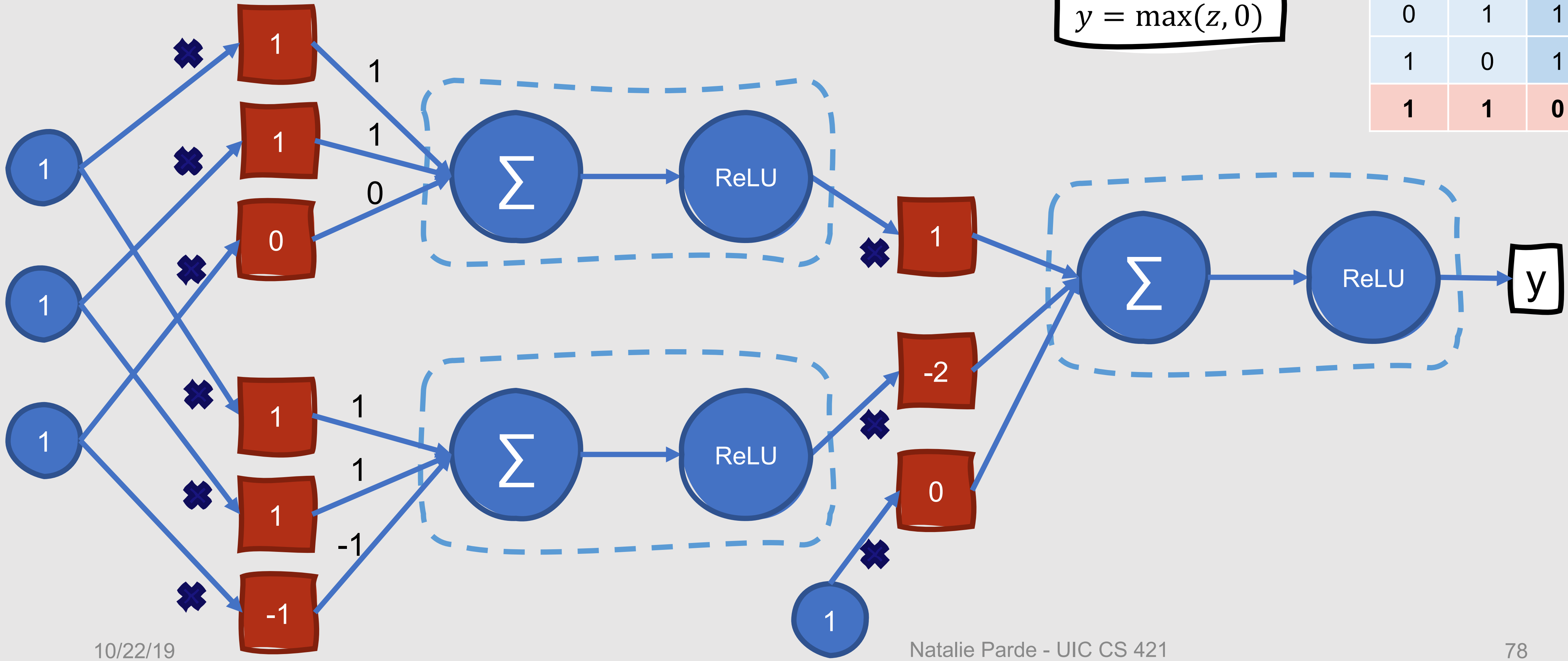
Truth Table Examples: XOR



$y = \max(z, 0)$

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

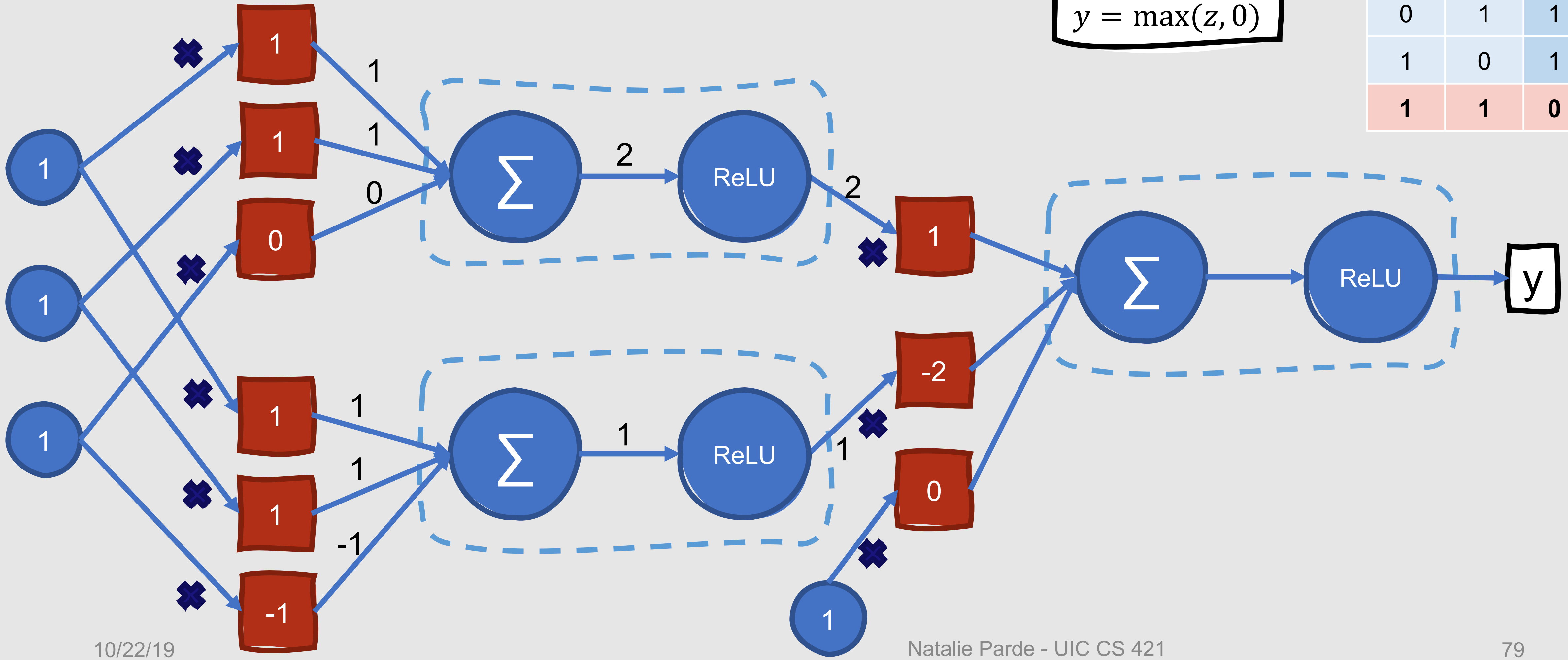
Truth Table Examples: XOR



$y = \max(z, 0)$

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

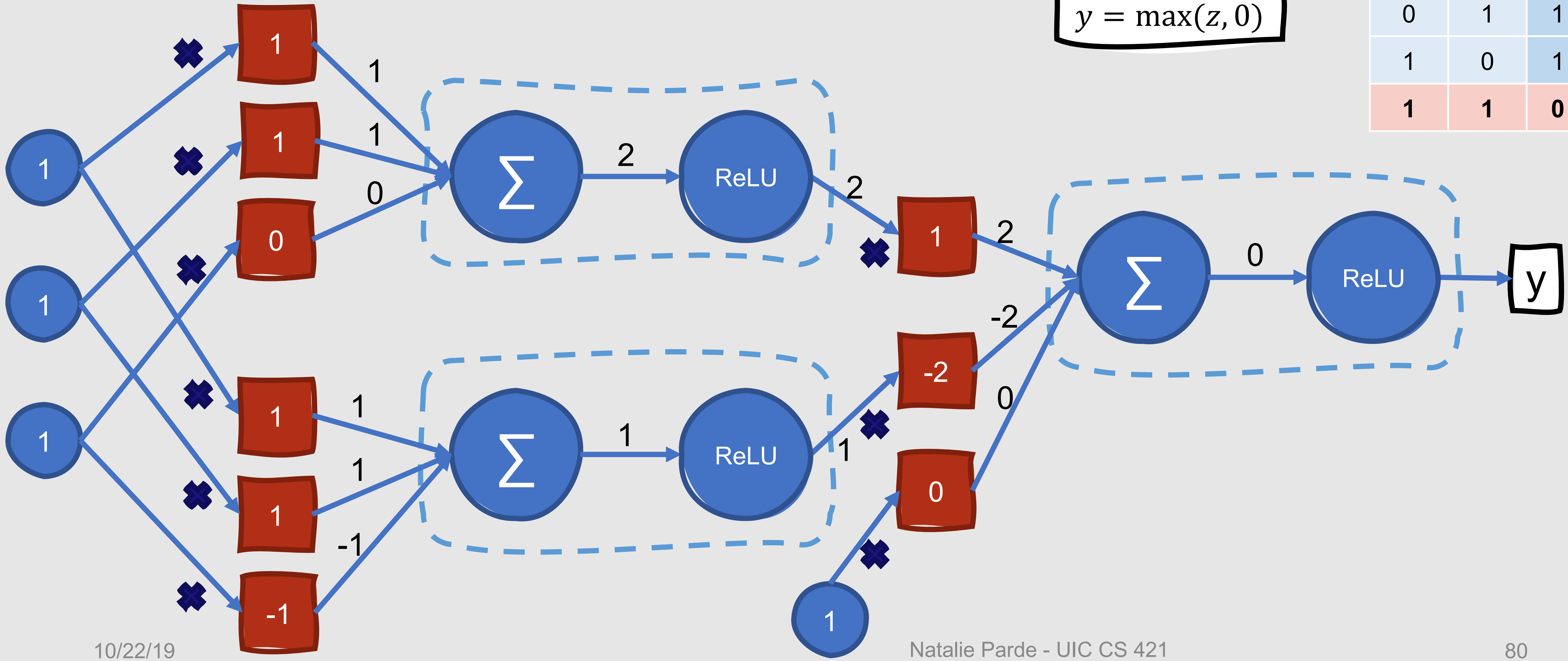
Truth Table Examples: XOR



$y = \max(z, 0)$

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

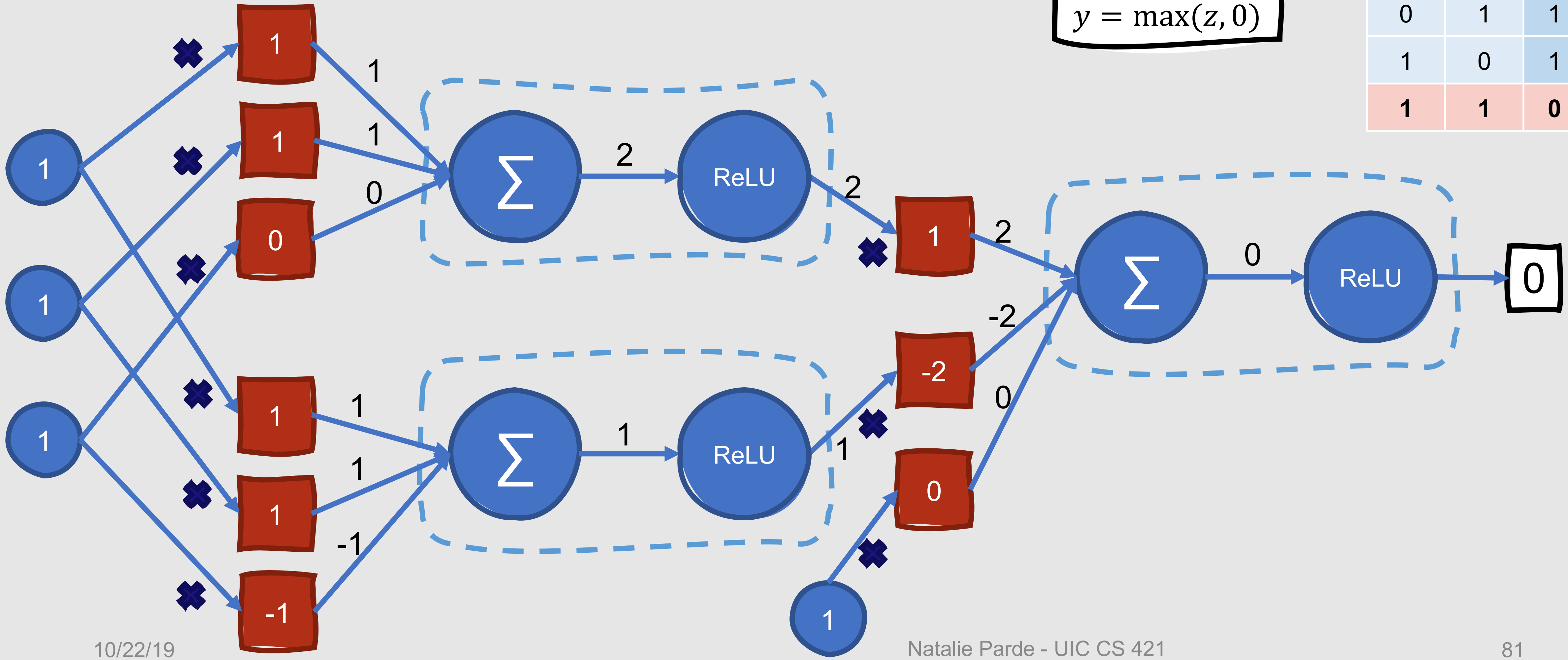
Truth Table Examples: XOR



$y = \max(z, 0)$

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

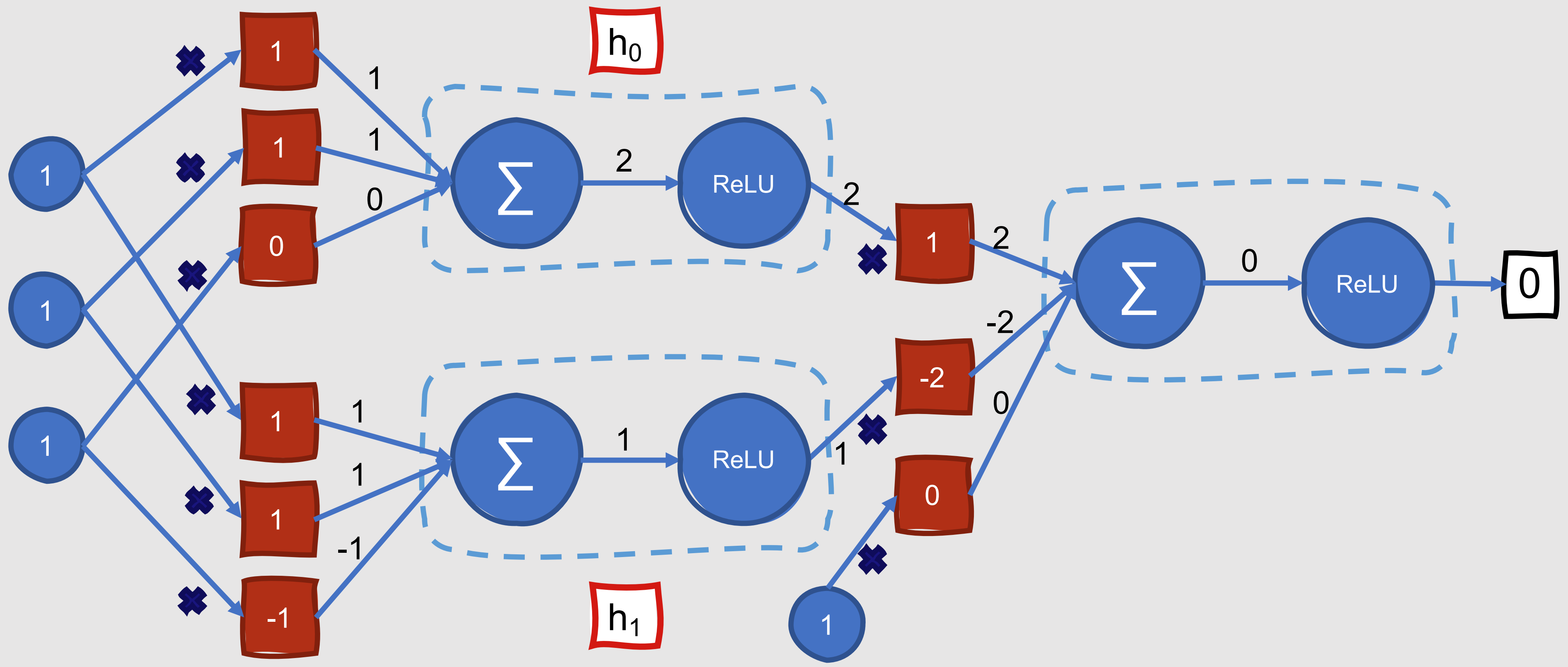
Truth Table Examples: XOR



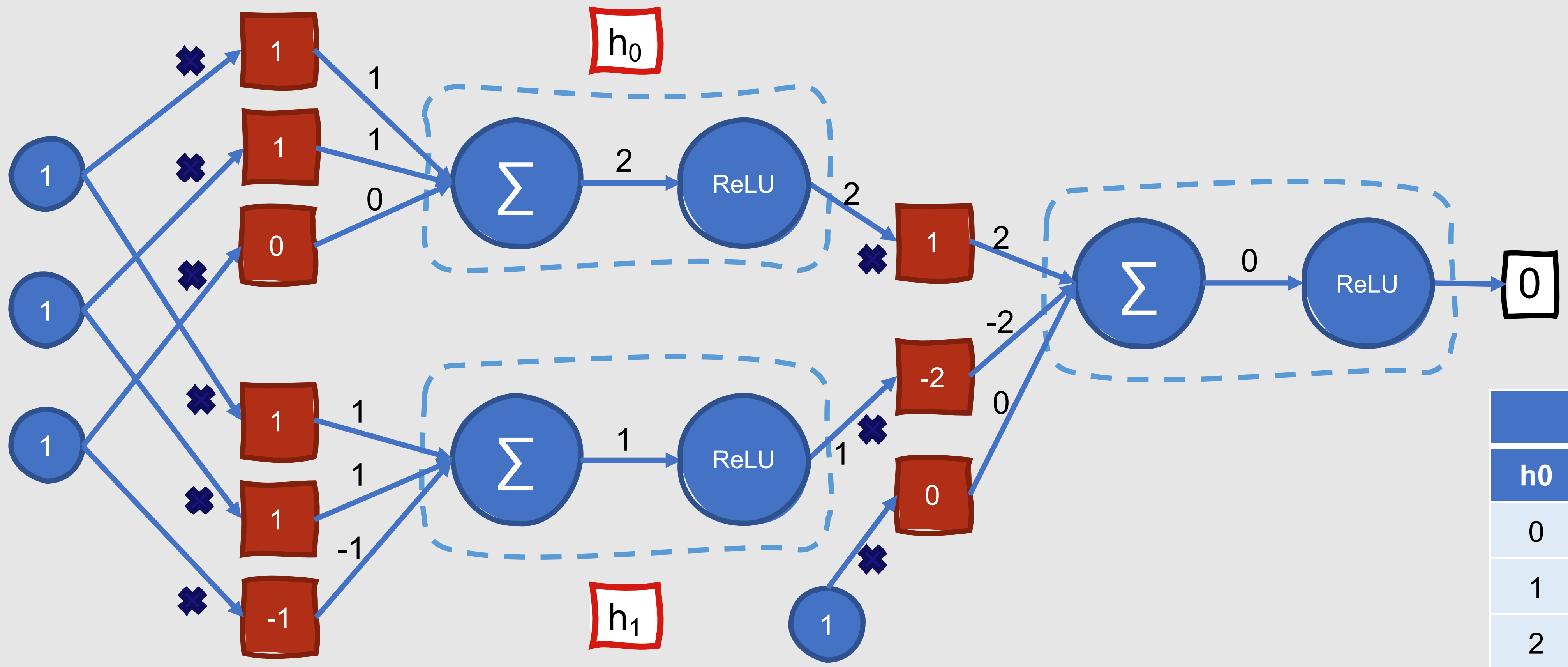
$y = \max(z, 0)$

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.



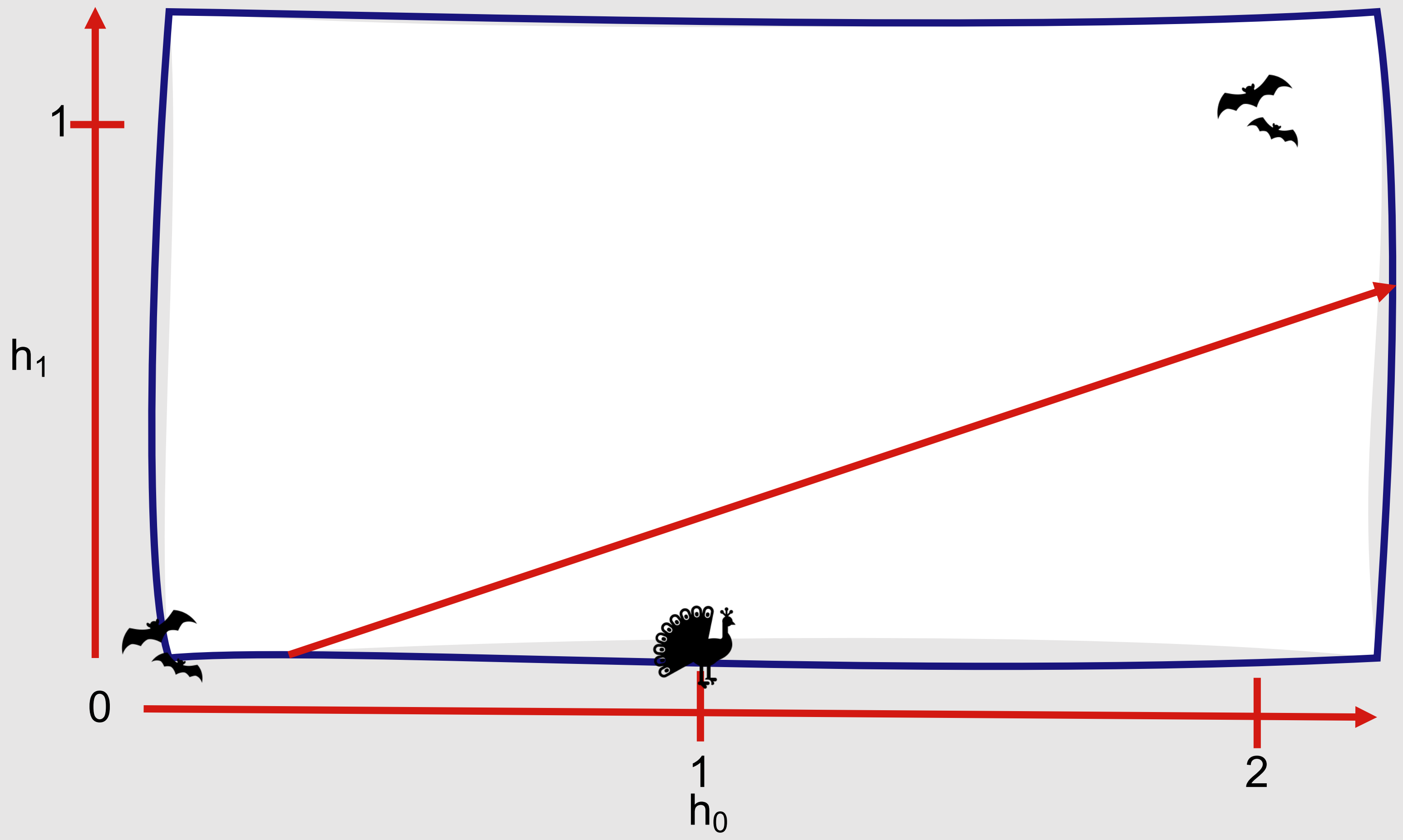
The hidden layer in the previous examples provides new representations for the input.



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

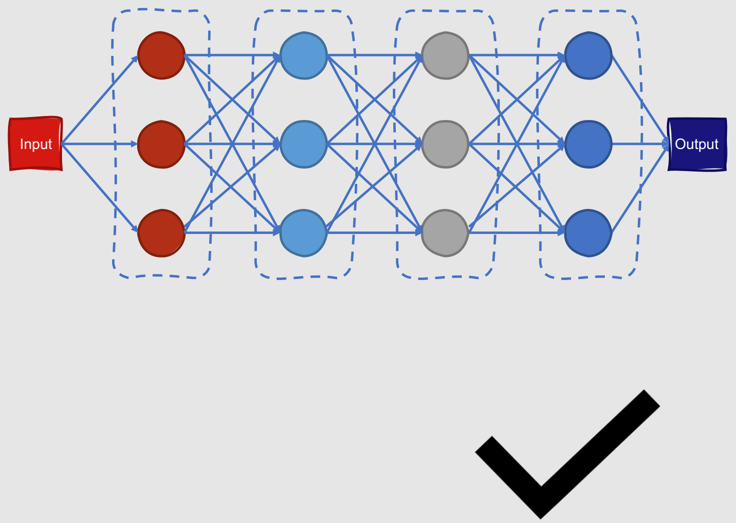
- In our XOR example, we manually assigned weights to each unit
- In real-world examples, these weights are learned automatically using a **backpropagation** algorithm
- Thus, the network is able to learn a useful representation of the input training data on its own
    - Key advantage of neural networks

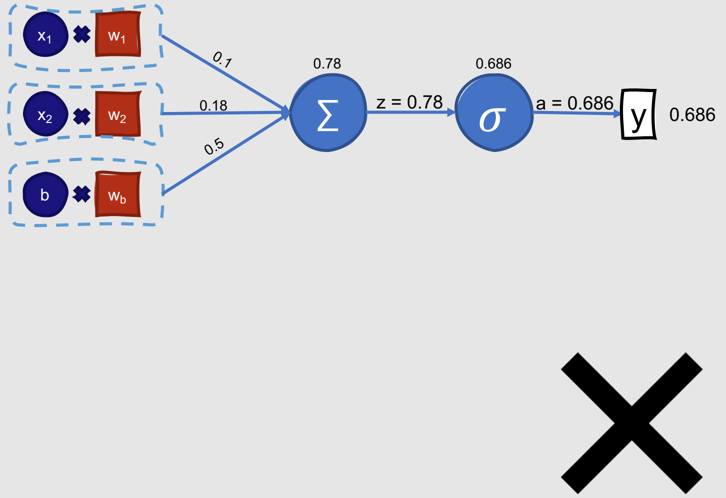
Summary: Neural Networks (Basics)

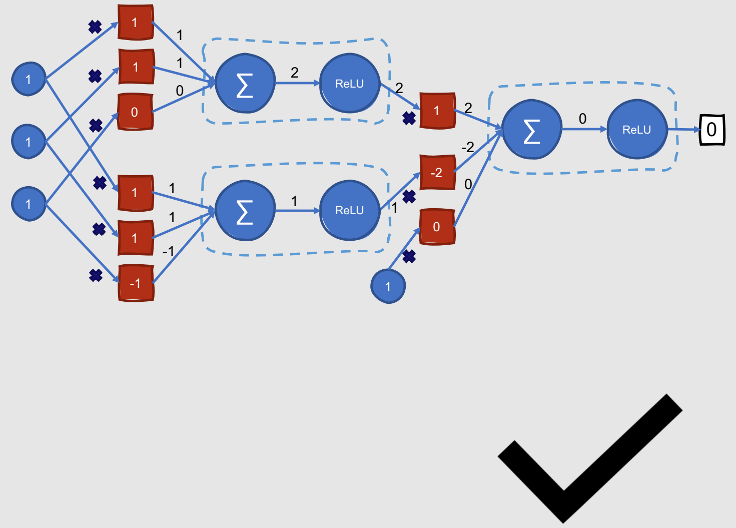
- **Neural networks** are classification models comprised of interconnected computational units (**neurons**)
- They play an increasingly fundamental role in solving many NLP tasks
- There are many varieties of neural networks
  - **Feedforward**
  - **Convolutional**
  - **Recurrent**
  - Etc.…
- Neural 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

What are feedforward networks?

- Multilayer neural network in which the units are connected with no cycles
  - Outputs from layer *n*-1 are passed to units in layer *n*
  - Outputs from layer *n* are never passed to layer *n*-1







Feedforward Networks

Feedforward Networks

Historically, sometimes called multilayer perceptrons (MLPs)

- Technically, no longer an accurate title …modern feedforward networks use units with nonlinear activations, not linear perceptrons!

Three types of units, or nodes:

- Input units
- Hidden units
- Output units

Input Units

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

0.5	0.2	0.1	0.7	0.4

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

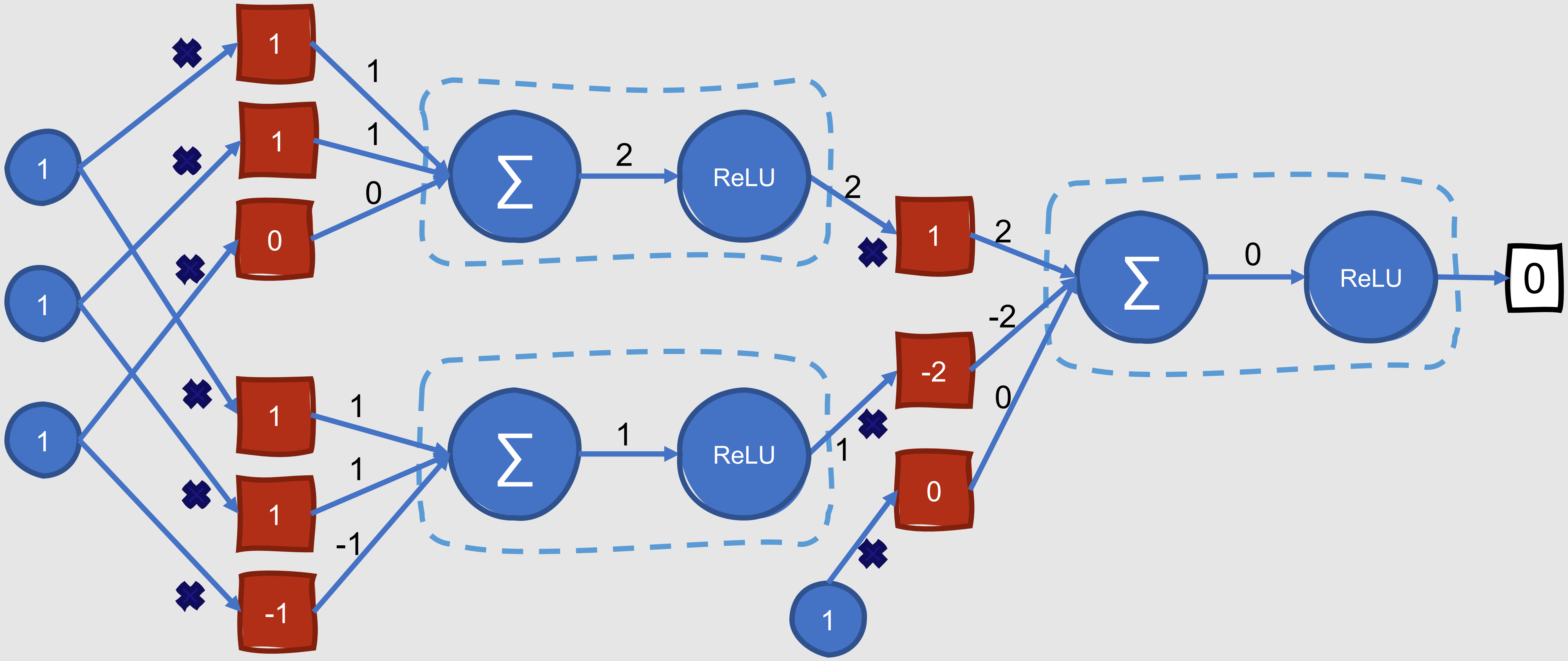
Hidden Layers

- 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

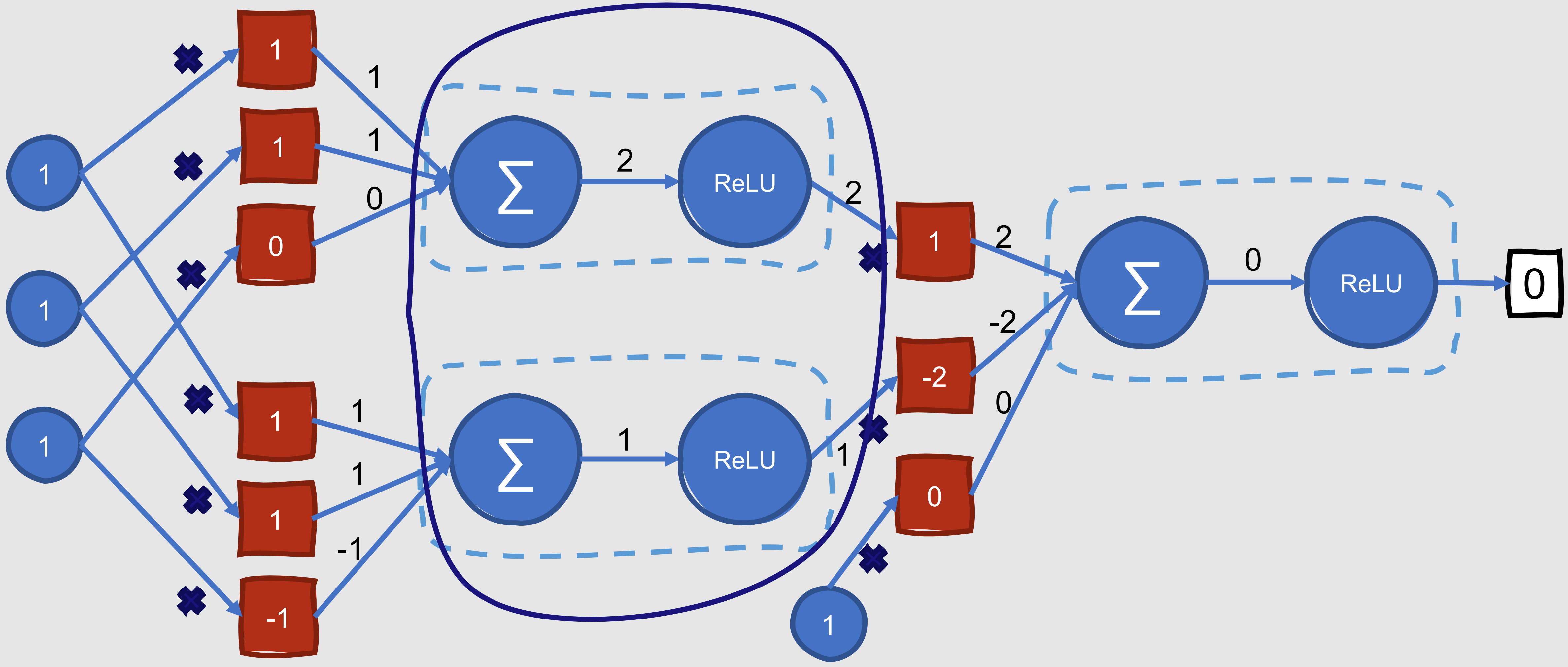
Why represent W as a single matrix?

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

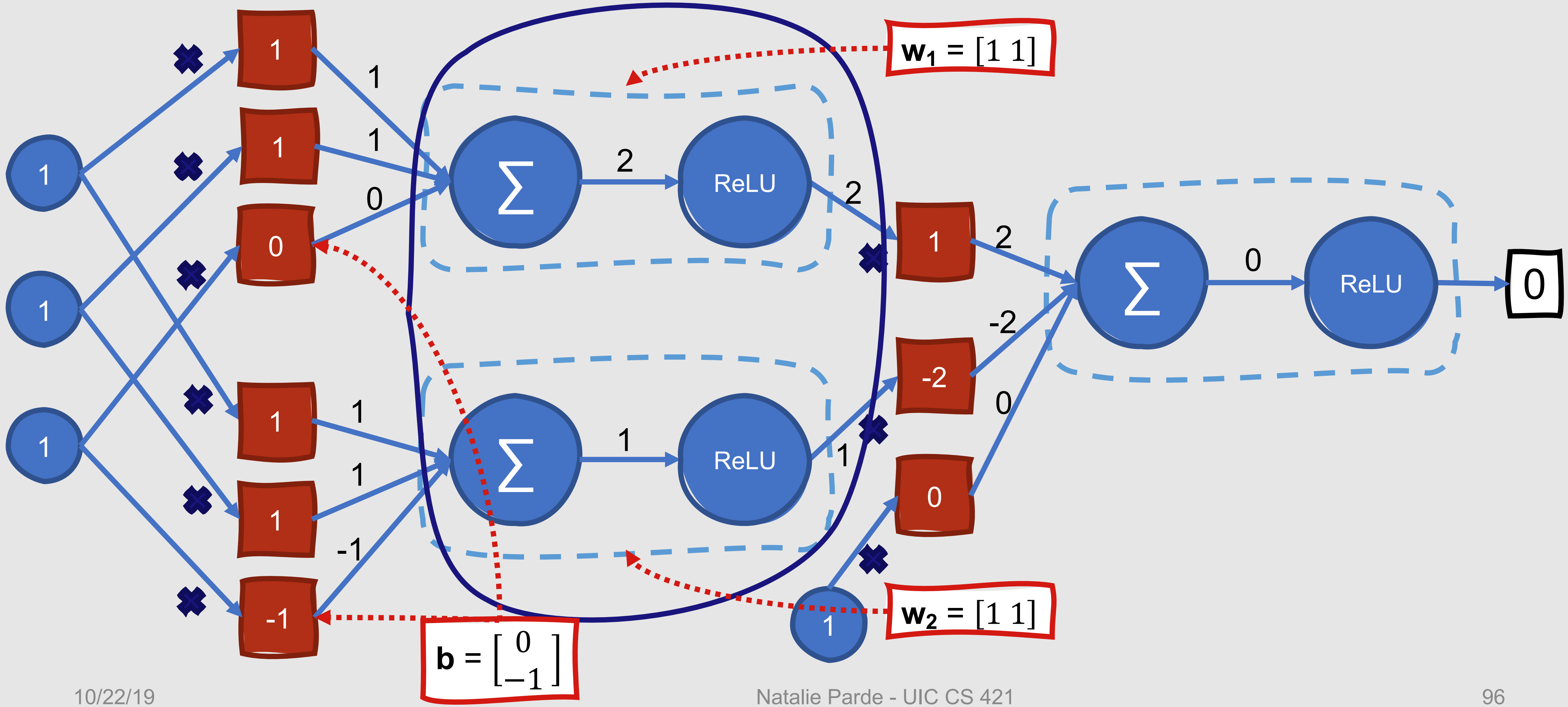
Example: Computing h across an entire hidden layer



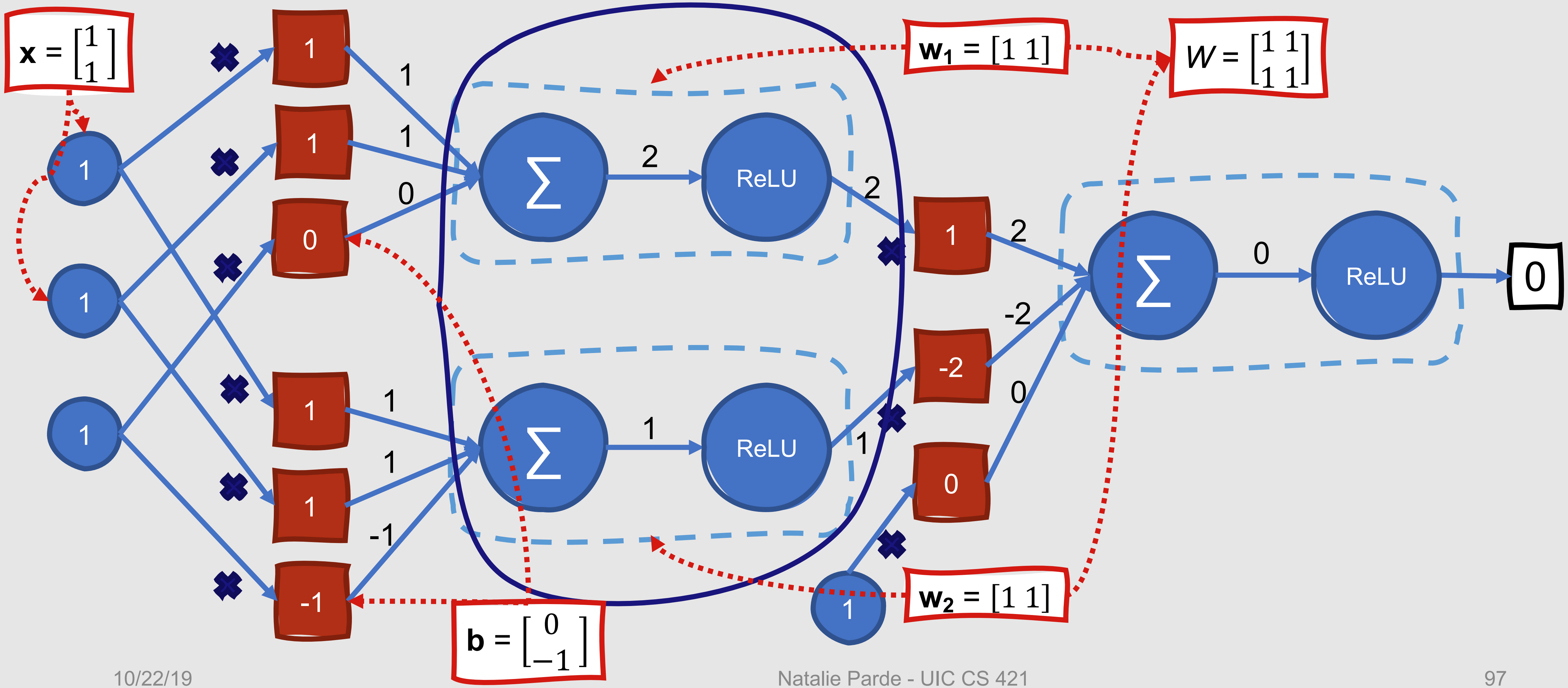
Example: Computing h across an entire hidden layer



Example: Computing h across an entire hidden layer

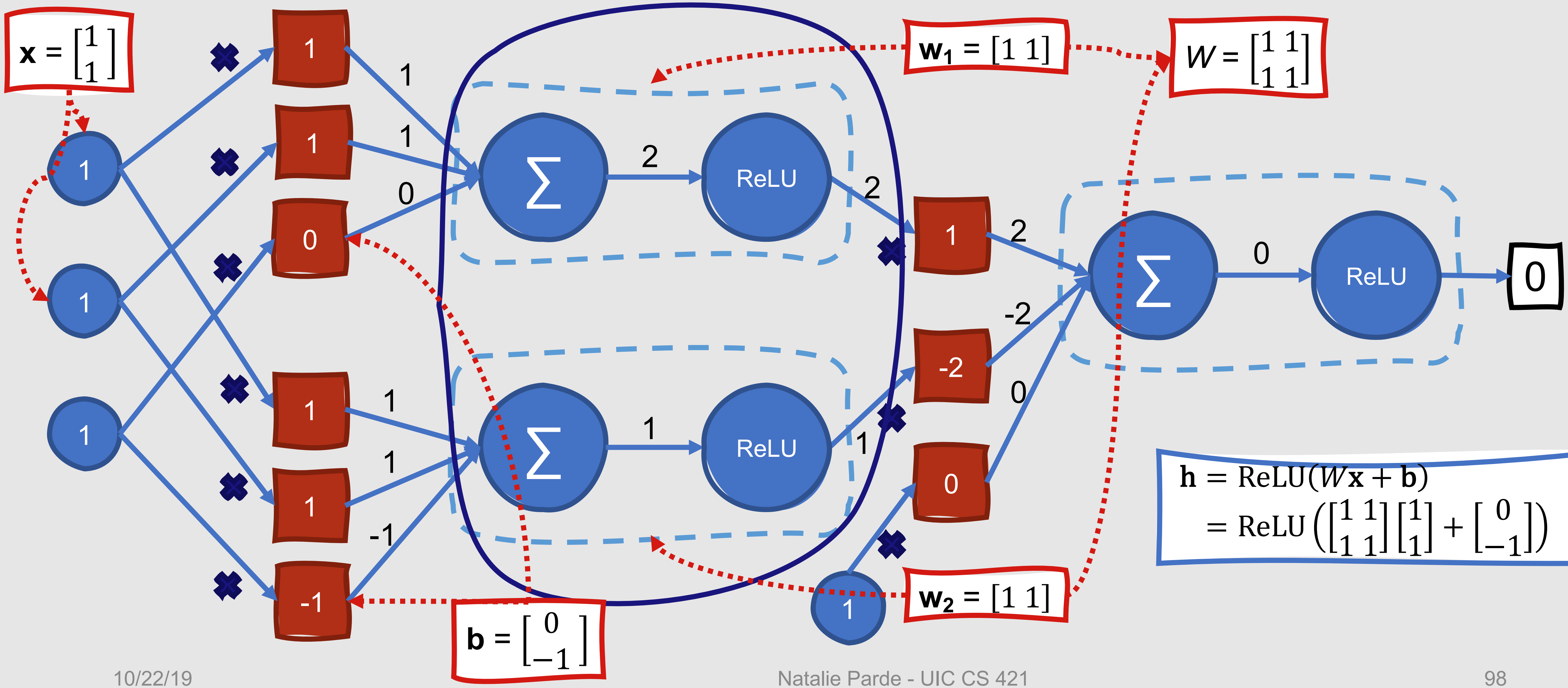


Example: Computing h across an entire hidden layer



$\mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
1
1
1
1
1
0
1
1
-1
1
1
0
1
1
-1
∑
2
ReLU
2
∑
1
ReLU
1
$\mathbf{w_1} = [1\ 1]$
$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$
1
-2
0
2
-2
0
1
∑
0
ReLU
0
$\mathbf{b} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$
$\mathbf{w_2} = [1\ 1]$

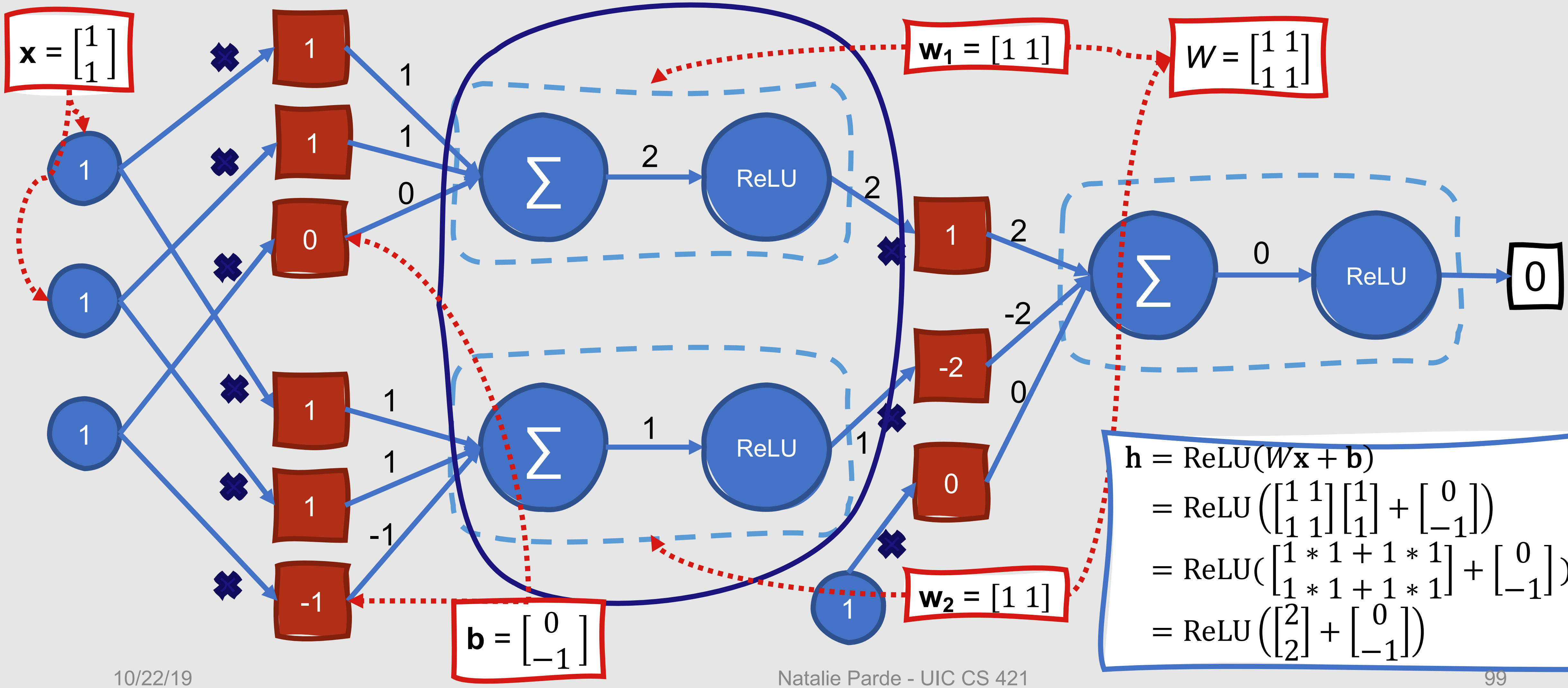
Example: Computing h across an entire hidden layer



$\mathbf{x} = \begin{bmatrix}1\\1\end{bmatrix}$
1
1
1
1
1
0
1
1
-1
1
1
0
1
1
-1
2
2
1
1
ReLU
ReLU
$\mathbf{w_1} = [1\ 1]$
$\mathbf{w_2} = [1\ 1]$
$W = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$
$\mathbf{b} = \begin{bmatrix}0\\-1\end{bmatrix}$
1
-2
0
2
-2
0
1
0
ReLU
0

$$\mathbf{h} = \mathrm{ReLU}(W\mathbf{x} + \mathbf{b})$$
$$= \mathrm{ReLU}\left(\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix}\right)$$

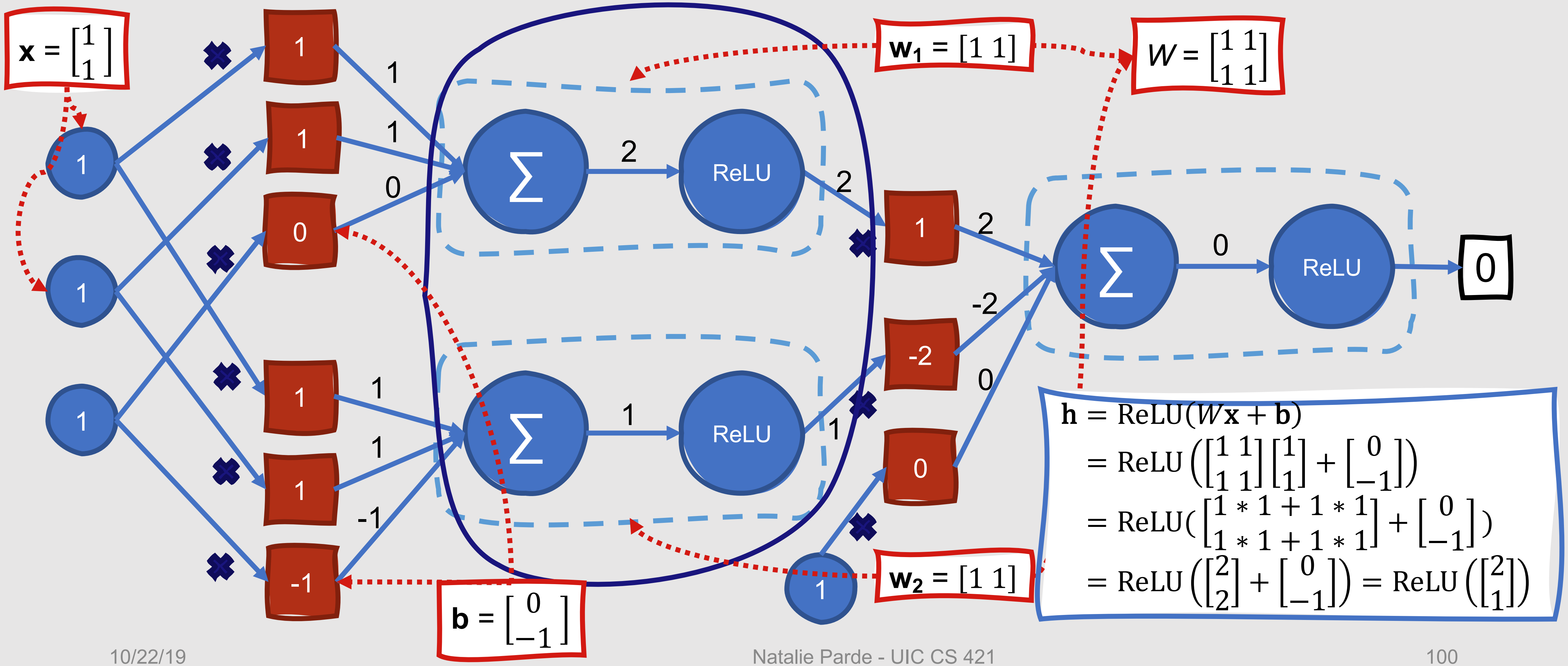
Example: Computing h across an entire hidden layer



x = $\begin{bmatrix}1\\1\end{bmatrix}$
b = $\begin{bmatrix}0\\-1\end{bmatrix}$
$\mathbf{w_1} = [1\ 1]$
$\mathbf{w_2} = [1\ 1]$
$W = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$
1
1
1
1
1
0
1
1
-1
1
1
0
1
1
-1
∑
2
ReLU
2
∑
1
ReLU
1
1
2
-2
-2
0
0
1
∑
0
ReLU
0

$$\begin{aligned}\mathbf{h} &= \text{ReLU}(W\mathbf{x}+\mathbf{b})\\ &= \text{ReLU}\left(\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}+\begin{bmatrix}0\\-1\end{bmatrix}\right)\\ &= \text{ReLU}(\begin{bmatrix}1*1+1*1\\1*1+1*1\end{bmatrix}+\begin{bmatrix}0\\-1\end{bmatrix})\\ &= \text{ReLU}\left(\begin{bmatrix}2\\2\end{bmatrix}+\begin{bmatrix}0\\-1\end{bmatrix}\right)\end{aligned}$$

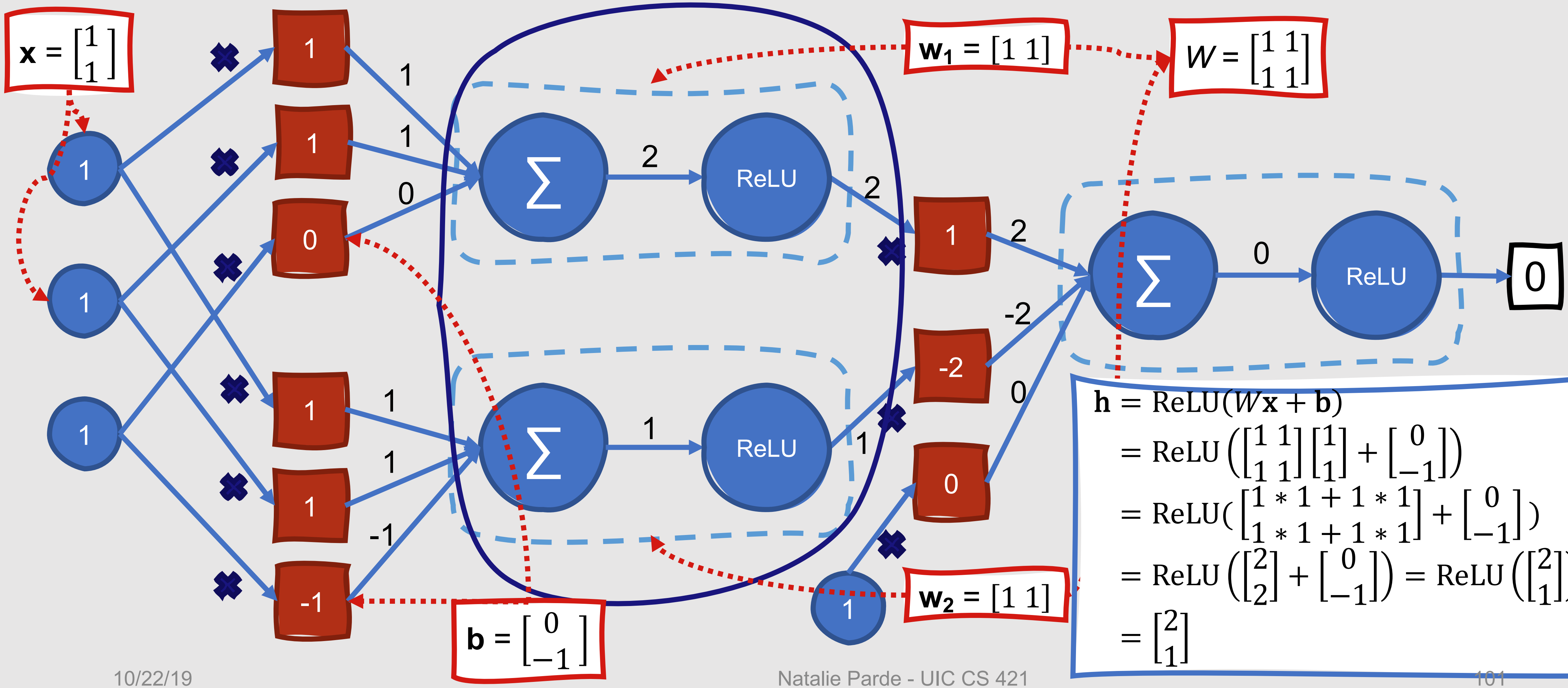
Example: Computing h across an entire hidden layer



$\mathbf{x} = \begin{bmatrix}1\\1\end{bmatrix}$
1
1
1
1
1
0
1
1
-1
1
1
0
1
1
-1
2
1
2
1
1
-2
0
2
-2
0
1
0
ReLU
ReLU
ReLU
0
$\mathbf{w_1} = [1\ 1]$
$\mathbf{w_2} = [1\ 1]$
$W = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$
$\mathbf{b} = \begin{bmatrix}0\\-1\end{bmatrix}$

$$\begin{aligned}\mathbf{h} &= \mathrm{ReLU}(W\mathbf{x} + \mathbf{b})\\ &= \mathrm{ReLU}\left(\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix}\right)\\ &= \mathrm{ReLU}(\begin{bmatrix}1*1+1*1\\1*1+1*1\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix})\\ &= \mathrm{ReLU}\left(\begin{bmatrix}2\\2\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix}\right) = \mathrm{ReLU}\left(\begin{bmatrix}2\\1\end{bmatrix}\right)\end{aligned}$$

Example: Computing h across an entire hidden layer



$\mathbf{x} = \begin{bmatrix}1\\1\end{bmatrix}$
$\mathbf{w}_1 = [1\ 1]$
$W = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$
$\mathbf{w}_2 = [1\ 1]$
$\mathbf{b} = \begin{bmatrix}0\\-1\end{bmatrix}$
1
1
1
1
1
0
1
1
-1
1
1
0
1
1
-1
Σ
2
ReLU
2
Σ
1
ReLU
1
1
2
-2
-2
0
0
1
Σ
0
ReLU
0

$$
\begin{aligned}
\mathbf{h} &= \mathrm{ReLU}(W\mathbf{x} + \mathbf{b}) \\
&= \mathrm{ReLU}\left(\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix}\right) \\
&= \mathrm{ReLU}\left(\begin{bmatrix}1*1+1*1\\1*1+1*1\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix}\right) \\
&= \mathrm{ReLU}\left(\begin{bmatrix}2\\2\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix}\right) = \mathrm{ReLU}\left(\begin{bmatrix}2\\1\end{bmatrix}\right) \\
&= \begin{bmatrix}2\\1\end{bmatrix}
\end{aligned}
$$

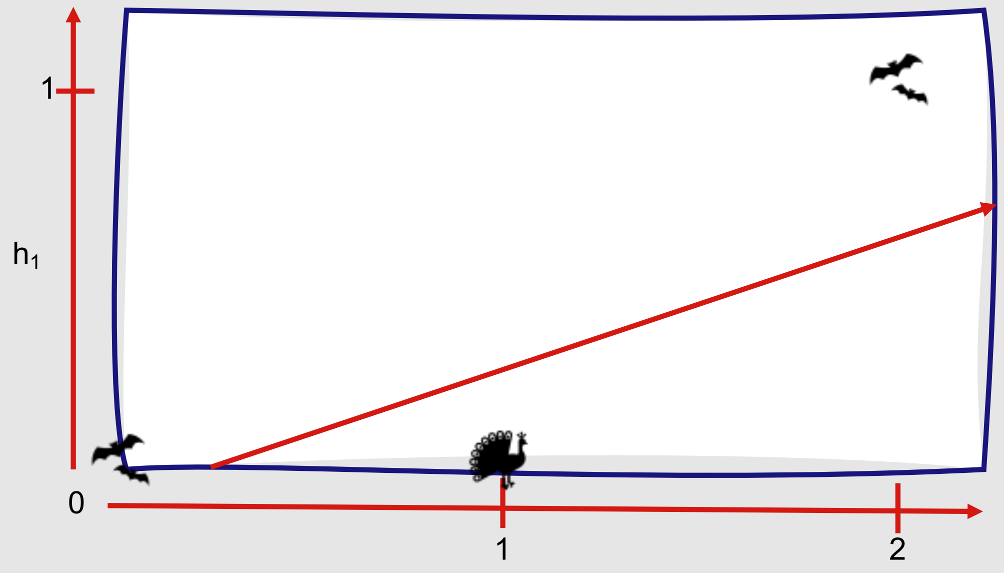
Formal Definitions

- 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 $W \in \mathbb{R}^{n_1 \times n_0}$

Hidden layers form new representations for input vectors.

- Goal of the output layer:
  - Take the new representation, *h*, and compute a final output
- What does the final output look like?
  - Real-valued number
  - Discrete label

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

However, *z* can't be the classifier output!

Remember, *z* is just a vector of real-valued numbers

For classification, a vector of **probabilities** is needed instead

How do we convert our real-valued numbers to probabilities?

- Normalization!
- Goals:
  - All numbers should lie between 0 and 1
  - All numbers should sum to 1

0.5	1.2	0.2	2.6	0.3

✗

0.3	0.1	0.1	0.2	0.3

✓

Remember, there are many different activation functions….

exponential linear unit (elu)

softmax

scaled exponential linear unit (selu)

softplus

softsign

rectified linear unit (relu)

hyperbolic tangent (tanh)

sigmoid

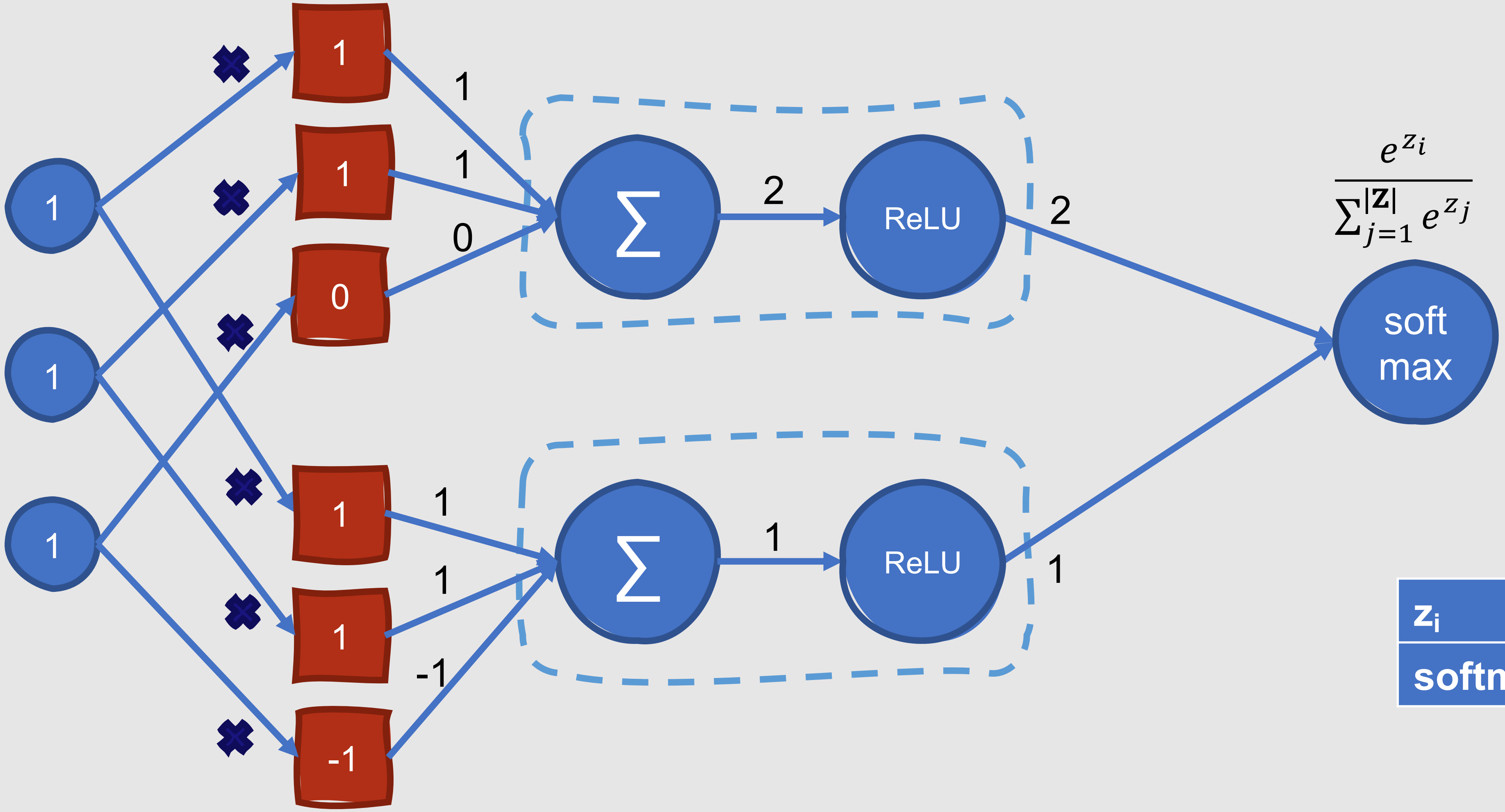
softmax

- Converts a set of real-valued inputs into a set of probabilities proportional to the exponentials of the input values
- Why exponentials instead of just the raw input values?
    - Increase the probability of the highest value in the vector
    - Decrease the probabilities of the other values
- Hence, the “soft” max is taken, rather than the “hard” max (argmax)

Formal Definition: softmax

- Letting z be a vector with values $z_i \in \mathbf{z}$,
  - $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{|\mathbf{z}|} e^{z_j}}$

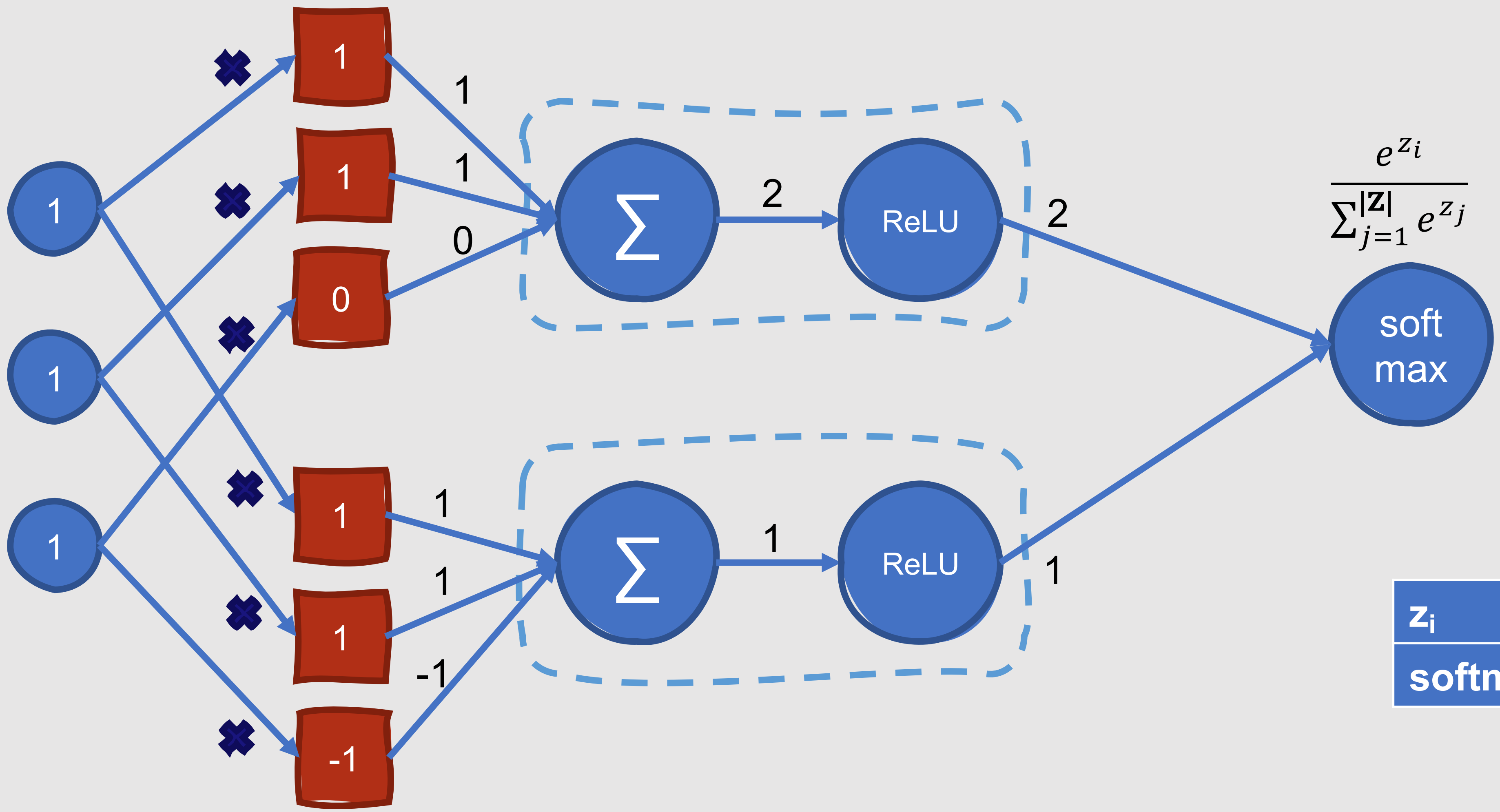
Example: softmax



$$\frac{e^{z_i}}{\sum_{j=1}^{|\mathbf{z}|} e^{z_j}}$$

z_i	2	1
softmax(z_i)		

Example: softmax

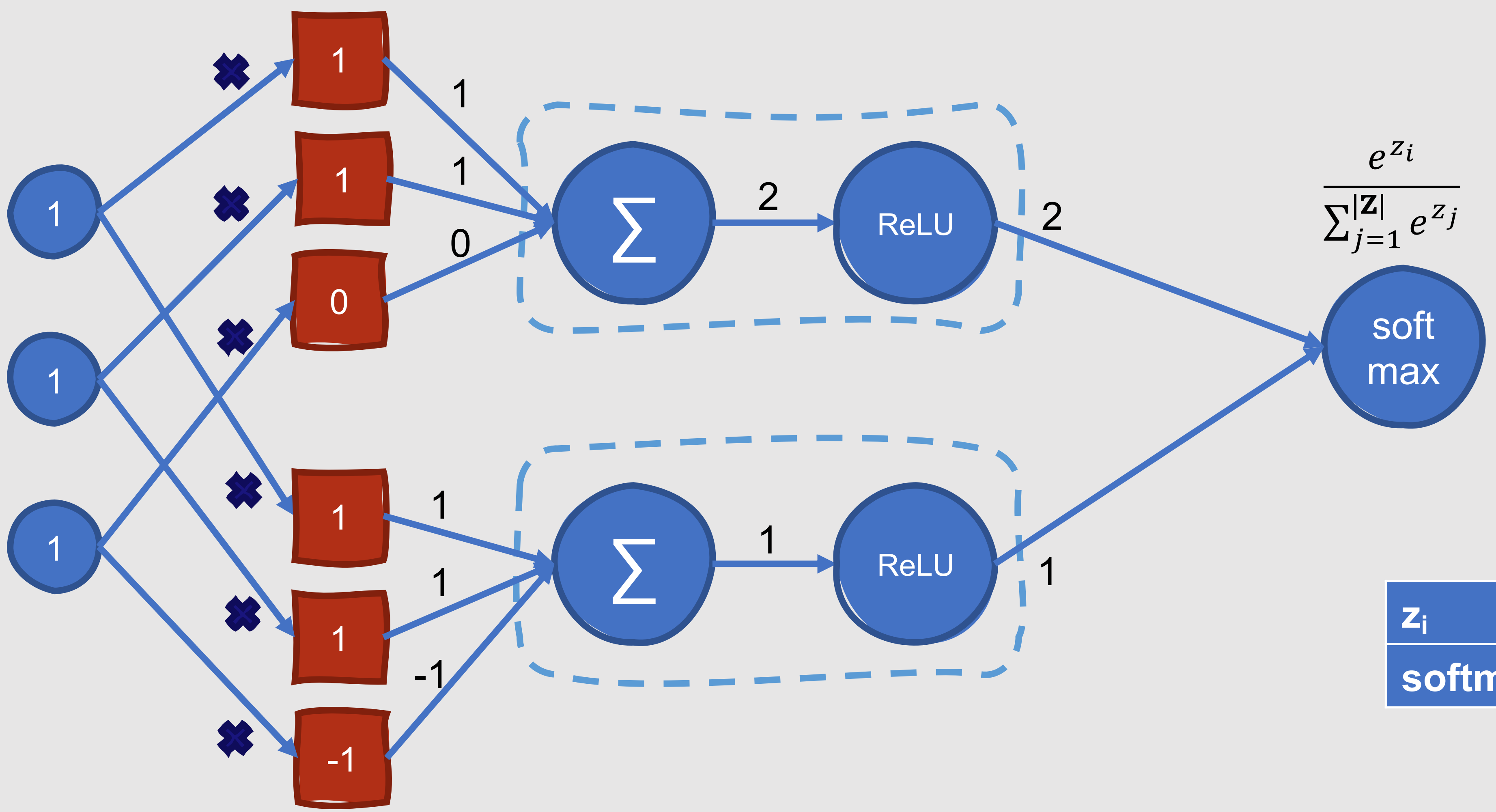


$$\boxed{\frac{e^2}{e^2 + e^1}}$$

$$\boxed{\frac{e^1}{e^2 + e^1}}$$

z_i	2	1
softmax(z_i)		

Example: softmax



$$\frac{e^{z_i}}{\sum_{j=1}^{|\mathbf{z}|} e^{z_j}}$$

$$\frac{e^2}{e^2 + e^1} = 0.73$$

$$\frac{e^1}{e^2 + e^1} = 0.27$$

$\mathbf{z_i}$	2	1
softmax($\mathbf{z_i}$)	0.73	0.27

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)

Training Neural Networks

- Feedforward neural networks are a type of supervised classification model
- Thus, the model learns to predict the correct output *y* for an observation *x* using labeled training data
- Feedforward neural networks do this by learning parameters $W^{[n]}$ and $\mathbf{b}^{[n]}$ for each layer *n* such that the predicted value *y'* for a training observation is as close as possible to the actual *y* value for that observation

How do we learn these parameters?

- We need two elements:
  - A **loss function** that models the distance between the predicted and actual labels
  - An **optimization algorithm**
- We also need to use a technique called **backpropagation**

Loss Functions

- Loss functions let neural networks know how well they're doing at modeling their training data
  - If they're predicting values that are pretty close to those in the validation set, they're doing well (minimal loss)
  - If they're predicting values that are pretty different from those in the validation set, they're not doing very well (high loss)
- Just like with activation functions, many loss functions exist!

☹

Predicted	Actual
.6	0
.7	0
.2	1

Predicted	Actual
.2	0
.1	0
.3	1

☺

Loss Functions

mean squared error

mean absolute error

hinge loss

KL divergence

cross-entropy

Cross-Entropy Loss

- Most common loss function for many classification tasks
- Measures the distance between the probability distributions of predicted and actual values
  - $loss(y_i, y_i') = -\sum_{c=1}^{|C|} p_{i,c} \log p_{i,c}'$
    - C is the set of all possible classes
    - $p_{i,c}$ is the actual probability that instance *i* should be labeled with class *c*
    - $p_{i,c}'$ is the predicted probability that instance *i* should be labeled with class *c*
- Ranges from 0 (best) to 1 (worst)
- Observations with a big distance between the predicted and actual values have much higher cross-entropy loss than observations with only a small distance between the two values

Example: Cross-Entropy Loss

Oh yay time for another midterm.

[✓ Sarcastic]

Not Sarcastic

Example: Cross-Entropy Loss

Oh yay time for another midterm.

[✓ Sarcastic] Not Sarcastic

Instance	Predicted Probability: Sarcastic	Predicted Probability: Not Sarcastic	Actual Probability: Sarcastic	Actual Probability: Not Sarcastic
Oh yay time for another midterm.			1	0

Example: Cross-Entropy Loss

Oh yay time for another midterm.

[✓ Sarcastic]

Not Sarcastic

Instance	Predicted Probability: Sarcastic	Predicted Probability: Not Sarcastic	Actual Probability: Sarcastic	Actual Probability: Not Sarcastic
Oh yay time for another midterm.	0.7	0.3	1	0

Example: Cross-Entropy Loss

Oh yay time for another midterm.

[✓ Sarcastic]

Not Sarcastic

Instance	Predicted Probability: Sarcastic	Predicted Probability: Not Sarcastic	Actual Probability: Sarcastic	Actual Probability: Not Sarcastic
Oh yay time for another midterm.	0.7	0.3	1	0

$$loss(y_i, y_i') = -\sum_{c=1}^{|C|} p_{i,c} \log p_{i,c}' = -p_{i,sarcastic} \log p_{i,sarcastic}' - p_{i,not\ sarcastic} \log p_{i,not\ sarcastic}'$$

Example: Cross-Entropy Loss

Oh yay time for another midterm.

[✓ Sarcastic]

Not Sarcastic

Instance	Predicted Probability: Sarcastic	Predicted Probability: Not Sarcastic	Actual Probability: Sarcastic	Actual Probability: Not Sarcastic
Oh yay time for another midterm.	0.7	0.3	1	0

$$loss(y_i, y_i') = -\sum_{c=1}^{|C|} p_{i,c} \log p_{i,c}' = -p_{i,sarcastic} \log p_{i,sarcastic}' - p_{i,not\ sarcastic} \log p_{i,not\ sarcastic}'$$

$$loss(y_i, y_i') = -1 * \log 0.7 - 0 * \log 0.3$$

Example: Cross-Entropy Loss

Oh yay time for another midterm.

[✓ Sarcastic]

Not Sarcastic

Instance	Predicted Probability: Sarcastic	Predicted Probability: Not Sarcastic	Actual Probability: Sarcastic	Actual Probability: Not Sarcastic
Oh yay time for another midterm.	0.7	0.3	1	0

$$loss(y_i, y_i') = -\sum_{c=1}^{|C|} p_{i,c} \log p_{i,c}' = -p_{i,sarcastic} \log p_{i,sarcastic}' - p_{i,not\ sarcastic} \log p_{i,not\ sarcastic}'$$

$$loss(y_i, y_i') = -1 * \log 0.7 - 0 * \log 0.3 = -\log 0.7 = 0.15$$

Loss functions measure how well your model works for a single observation.

- What if you want to know how well your model works in general (across all observations)?
  - **Cost Function:** Average the loss over all examples

Example: Cost Function

Instance	Predicted Probability: Sarcastic	Predicted Probability: Not Sarcastic	Actual Probability: Sarcastic	Actual Probability: Not Sarcastic
Oh yay time for another midterm.	0.7	0.3	1	0
I love learning about neural networks!	0.5	0.5	0	1
I hope the next assignment is way harder than the other assignments.	0.2	0.8	1	0

$$loss(y_i, y_i') = -\sum_{c=1}^{|C|} p_{i,c} \log p_{i,c}' = -p_{i,sarcastic} \log p_{i,sarcastic}' - p_{i,not\ sarcastic} \log p_{i,not\ sarcastic}'$$

$$loss(y_i, y_i') = -1 * \log 0.7 - 0 * \log 0.3 = -\log 0.7 = 0.15$$

Example: Cost Function

Instance	Predicted Probability: Sarcastic	Predicted Probability: Not Sarcastic	Actual Probability: Sarcastic	Actual Probability: Not Sarcastic
Oh yay time for another midterm.	0.7	0.3	1	0
I love learning about neural networks!	0.5	0.5	0	1
I hope the next assignment is way harder than the other assignments.	0.2	0.8	1	0

$$loss(y_i, y_i') = -\sum_{c=1}^{|C|} p_{i,c} \log p_{i,c}' = -p_{i,sarcastic} \log p_{i,sarcastic}' - p_{i,not\ sarcastic} \log p_{i,not\ sarcastic}'$$

$$loss(y_i, y_i') = -1 * \log 0.7 - 0 * \log 0.3 = -\log 0.7 = 0.15$$

$$loss(y_i, y_i') = -0 * \log 0.5 - 1 * \log 0.5 = -\log 0.5 = 0.30$$

$$loss(y_i, y_i') = -1 * \log 0.2 - 0 * \log 0.8 = -\log 0.2 = 0.70$$

Example: Cost Function

Instance	Predicted Probability: Sarcastic	Predicted Probability: Not Sarcastic	Actual Probability: Sarcastic	Actual Probability: Not Sarcastic
Oh yay time for another midterm.	0.7	0.3	1	0
I love learning about neural networks!	0.5	0.5	0	1
I hope the next assignment is way harder than the other assignments.	0.2	0.8	1	0

$$loss(y_i, y_i') = -\sum_{c=1}^{|C|} p_{i,c} \log p_{i,c}' = -p_{i,sarcastic} \log p_{i,sarcastic}' - p_{i,not\ sarcastic} \log p_{i,not\ sarcastic}'$$

$$loss(y_i, y_i') = -1 * \log 0.7 - 0 * \log 0.3 = -\log 0.7 = 0.15$$
$$loss(y_i, y_i') = -0 * \log 0.5 - 1 * \log 0.5 = -\log 0.5 = 0.30$$
$$loss(y_i, y_i') = -1 * \log 0.2 - 0 * \log 0.8 = -\log 0.2 = 0.70$$

$$\boxed{\frac{0.15 + 0.30 + 0.70}{3} = 0.38}$$

How do we make sure the overall cost is as small as possible?

- In other words, how do we minimize our cost function?
  - Optimization algorithms!
- As with activation functions and loss functions, there are many different optimization algorithms

Optimization Algorithms

stochastic gradient descent

RMSprop

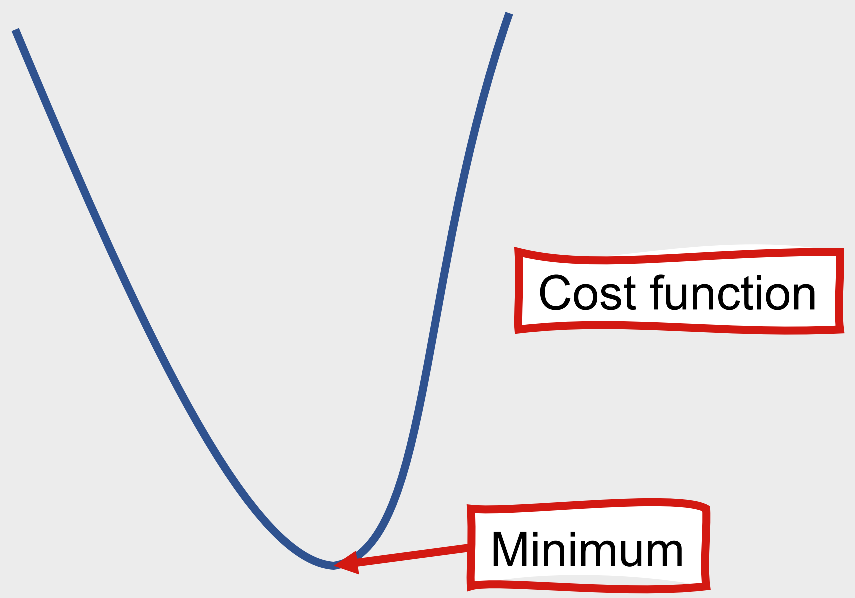
adam

adamax

adagrad

Gradient Descent

- Views a cost function as a gradient
- Goal: Find the lowest point of that gradient

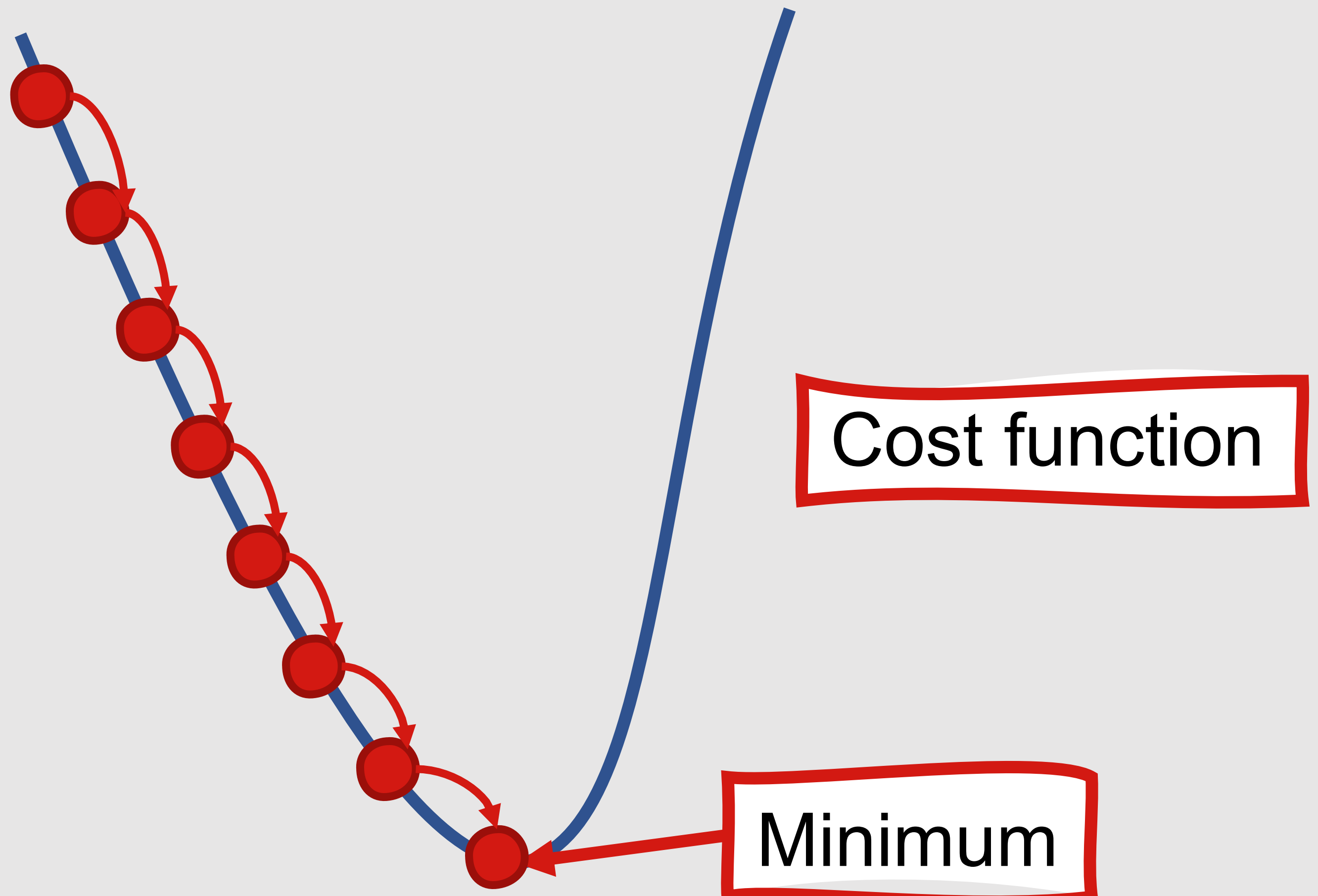


How do we find the gradient of a function?

- Take its derivative!
  - Letting $f(\cdot)$ be our cost function:
    - $\text{gradient}(f(x)) = \frac{\partial f(x)}{\partial x_i}$

Gradient Descent

- In gradient descent, we move along the curve specified by our cost function and find the gradient at each new point that we encounter
    - The distance between the points we encounter is specified by a **learning rate** along with weight and bias parameters
- When we reach a point at which the gradient does not decrease (or even increases) from one point to the next, we stop
    - This is the point at which the optimization algorithm converges



The problem with gradient descent….

- In more complex cost functions, gradient descent algorithms can get stuck in **local minima**
- This is why choosing a good **learning rate** is important!
- The learning rate for gradient descent is one of many **hyperparameters** that you can **tune** when building your neural network model

Stochastic Gradient Descent

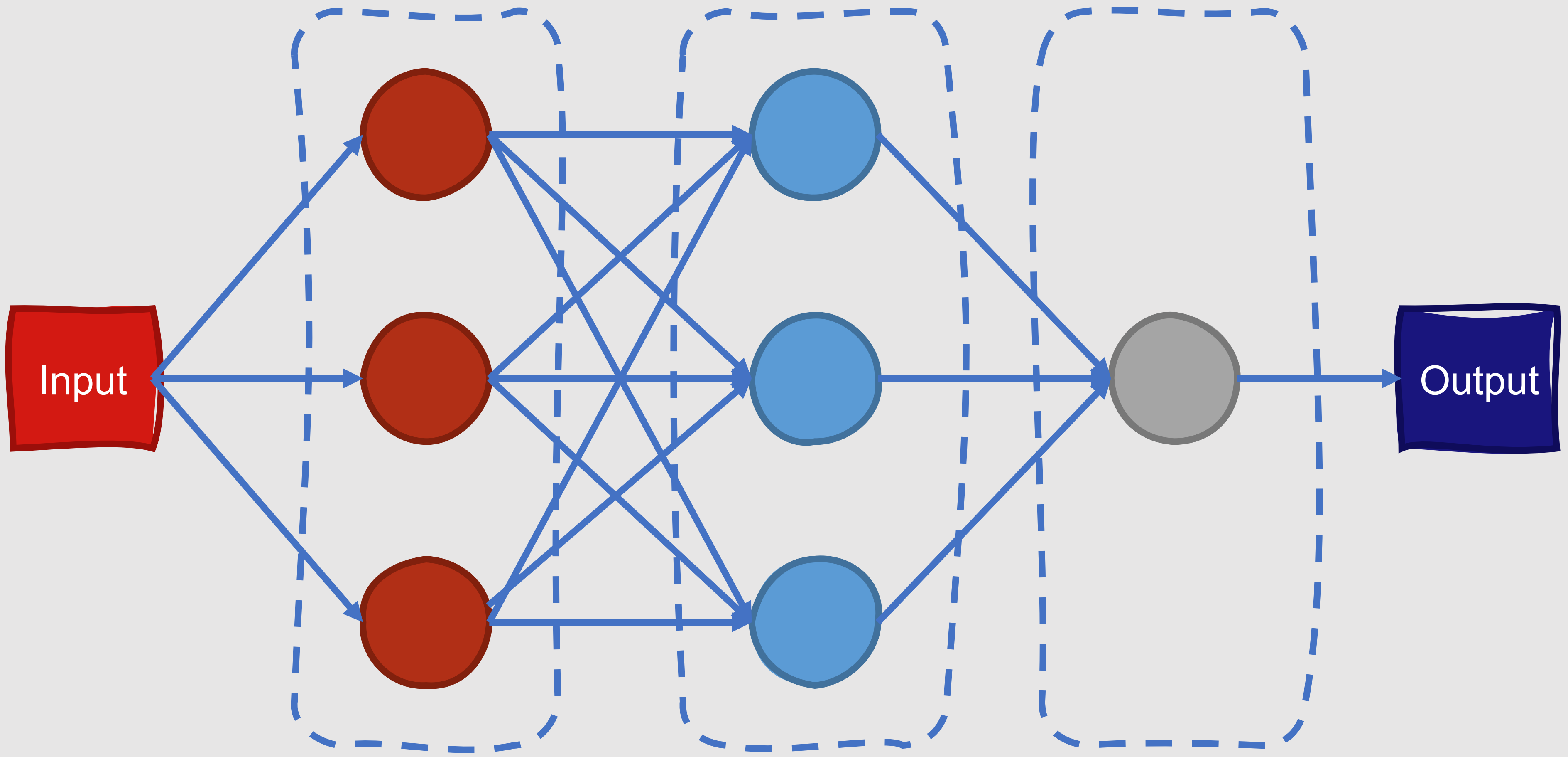
- **Gradient Descent:** To update the weight and bias parameters for the gradient descent algorithm, you run through every sample in your dataset
- **Stochastic Gradient Descent:** To update the weight and bias parameters, you run through one randomly selected sample from your dataset
- Stochastic gradient descent → much quicker!

For deep neural networks, we need to compute gradients with respect to weight parameters that appear early on in the network …even though loss is only computed at the end of the network once we have our predicted values.

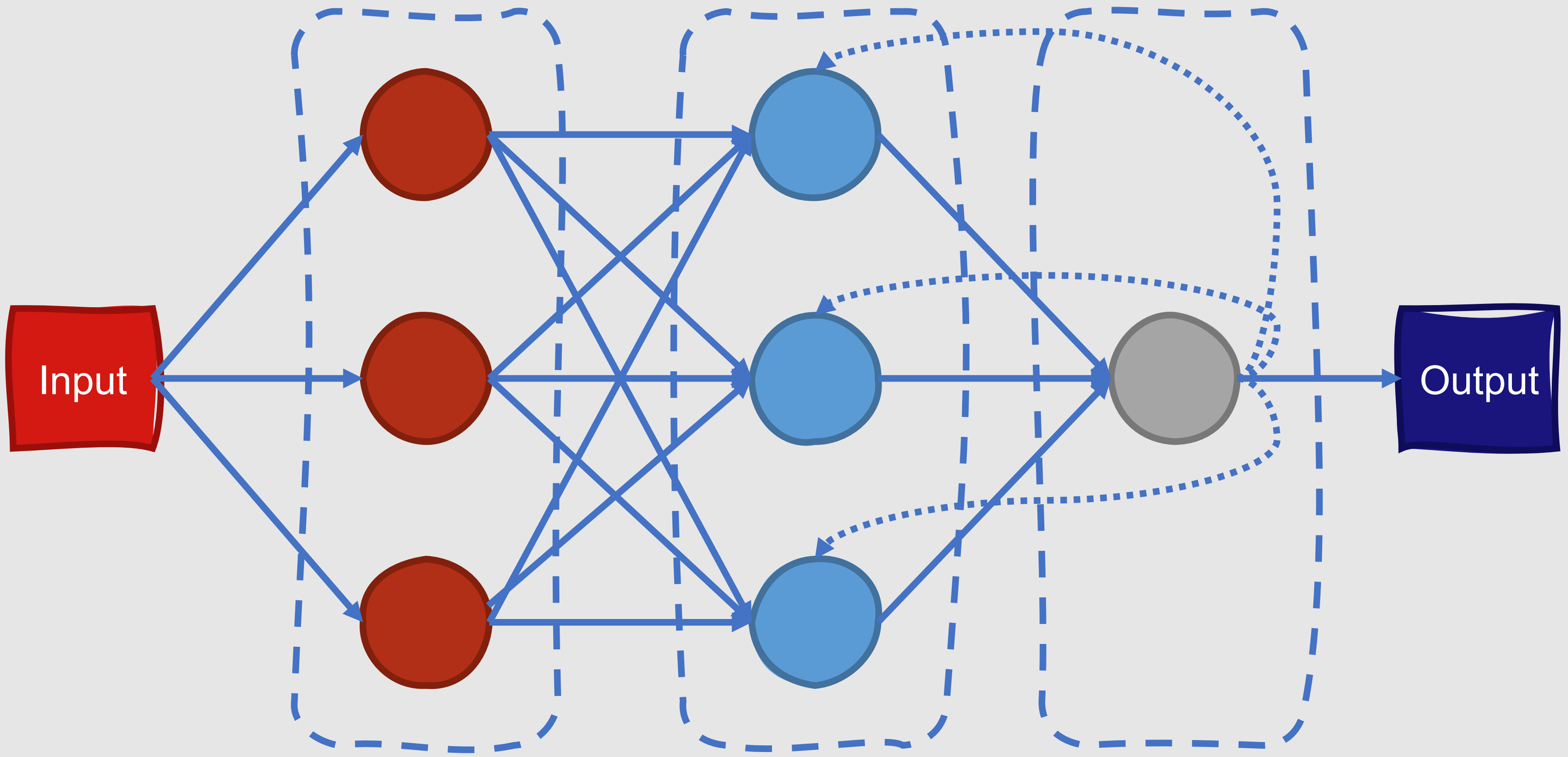
Backpropagation

- Allows us to propagate our error backward all the way to the beginning of the network
- How?
  - Compute partial derivatives using the chain rule, starting with the output node(s) and ending with the input units
  - At each node:
    1. Compute the local partial derivative with respect to the parent
    2. Multiply that by the partial derivative being passed down from the parent
    3. Pass the result along to its child(ren)

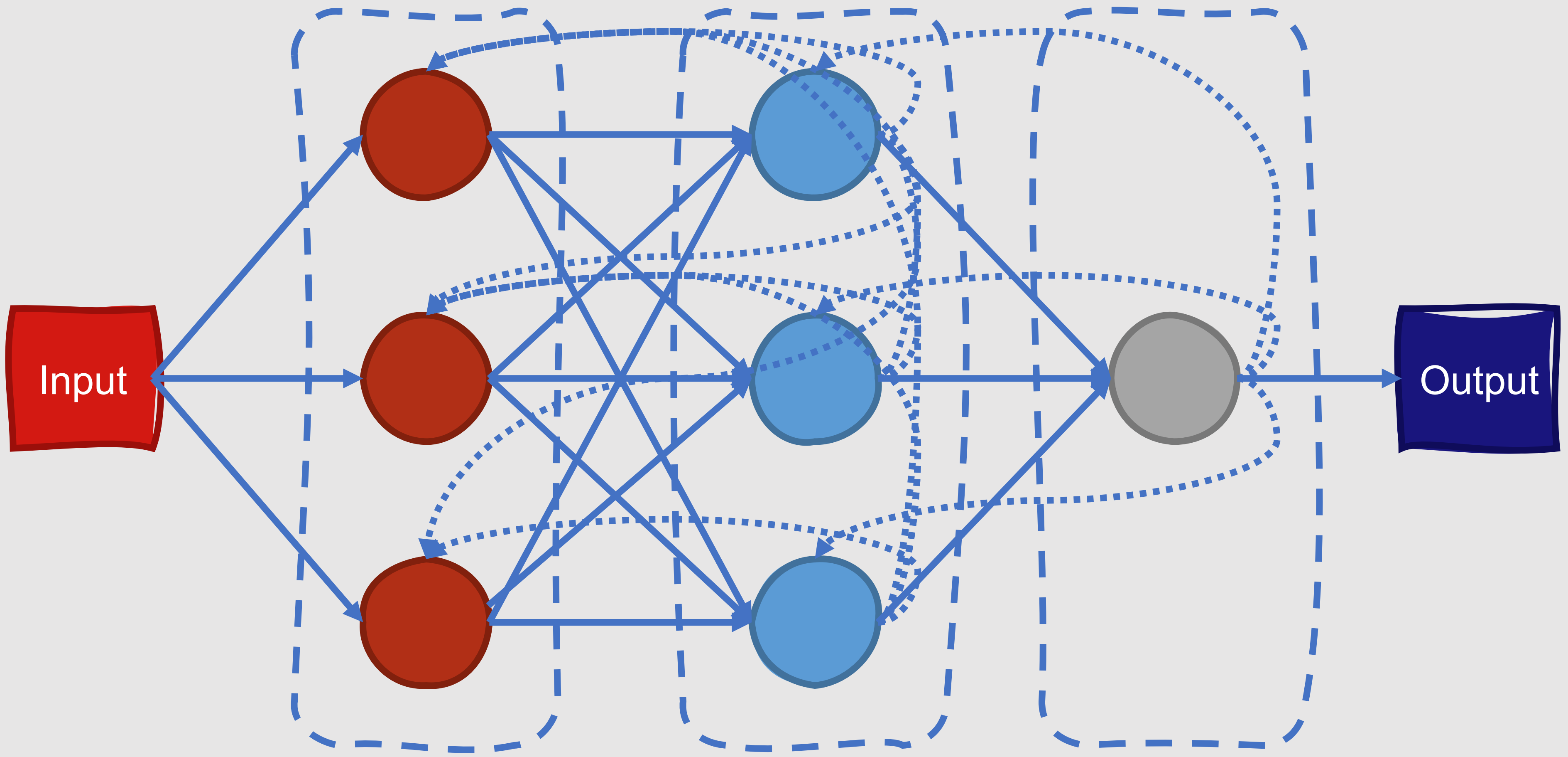
Backpropagation



Backpropagation



Backpropagation



What are the derivatives of the sigmoid, tanh, and ReLU activation functions?

- sigmoid
  - $\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z))$
- tanh
  - $\frac{\partial \tanh(z)}{\partial z} = 1 - \tanh^2(z)$
- ReLU
  - $\frac{\partial \text{ReLU}(z)}{\partial z} = \begin{cases} 0 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{cases}$

What are some ways to improve the performance of a neural network?

- **Normalize input values** to have a mean of 0
- Randomly drop some units and their connections from the network during training (**dropout**)
- **Tune hyperparameters**
  - Learning rate
  - Number of layers
  - Number of units per layer
  - Type of activation function
  - Type of optimization function

Tool for Building Neural Networks

TensorFlow

- https://www.tensorflow.org/

Keras

- https://keras.io/

PyTorch

- https://pytorch.org/

DL4J

- https://deeplearning4j.org/

Neural Language Models

- Extremely popular application of neural networks
- Advantages over *n*-gram language models:
  - Don’t need smoothing
  - Can handle much longer histories
  - Can generalize over contexts of similar words
- Disadvantage:
  - *Much* slower to train!
- Neural language models have much higher predictive accuracy than *n*-gram language models trained on datasets of similar sizes

Neural Language Models

- Neural language models are used to boost performance for many modern NLP tasks
  - Machine translation
  - Dialogue systems
  - Language generation

Sample Generated by a Neural Language Model (GPT-2)

- Link to article: https://openai.com/blog/better-language-models/

System Prompt (Human-Written): *In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.*

Sample Generated by a Neural Language Model (GPT-2)

Model Completion (Machine-Written, 10 Tries): The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez.

Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them – they were so close they could touch their horns.

While examining these bizarre creatures the scientists discovered that the creatures also spoke some fairly regular English. Pérez stated, "We can see, for example, that they have a common 'language,' something like a dialect or dialectic."

Dr. Pérez believes that the unicorns may have originated in Argentina, where the animals were believed to be descendants of a lost race of people who lived there before the arrival of humans in those parts of South America.

While their origins are still unclear, some believe that perhaps the creatures were created when a human and a unicorn met each other in a time before human civilization. According to Pérez, "In South America, such incidents seem to be quite common."

However, Pérez also pointed out that it is likely that the only way of knowing for sure if unicorns are indeed the descendants of a lost alien race is through DNA. "But they seem to be able to communicate in English quite well, which I believe is a sign of evolution, or at least a change in social organization," said the scientist.

Feedforward Neural Language Model

- Input: Representation of some number of previous words
    - w_{t-1}, w_{t-2}, etc.
- Output: Probability distribution over possible next words
- Goal: Approximate the probability of a word given the entire prior context $P(w_t|w_1^{t-1})$ based on the *n* previous words
    - $P(w_t|w_1^{t-1}) \approx P(w_t|w_{t-n+1}^{t-1})$

Neural language models represent prior context using embeddings of the previous words.

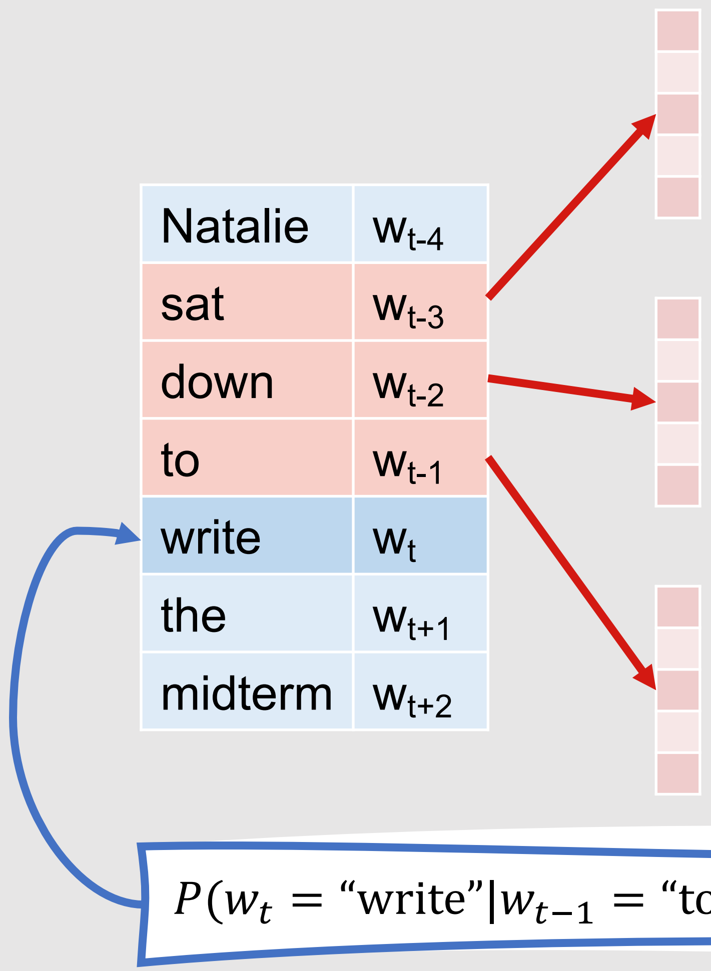
- Allows neural language models to generalize to unseen data better than *n*-gram models
- Embeddings can come from various sources
  - E.g., pretrained Word2Vec embedding dictionary

Neural Language Model

Natalie	w_{t-4}
sat	w_{t-3}
down	w_{t-2}
to	w_{t-1}
write	w_t
the	w_{t+1}
midterm	w_{t+2}

$$P(w_t = \text{“write”} | w_{t-1} = \text{“to”}, w_{t-2} = \text{“down”}, w_{t-3} = \text{“sat”})$$

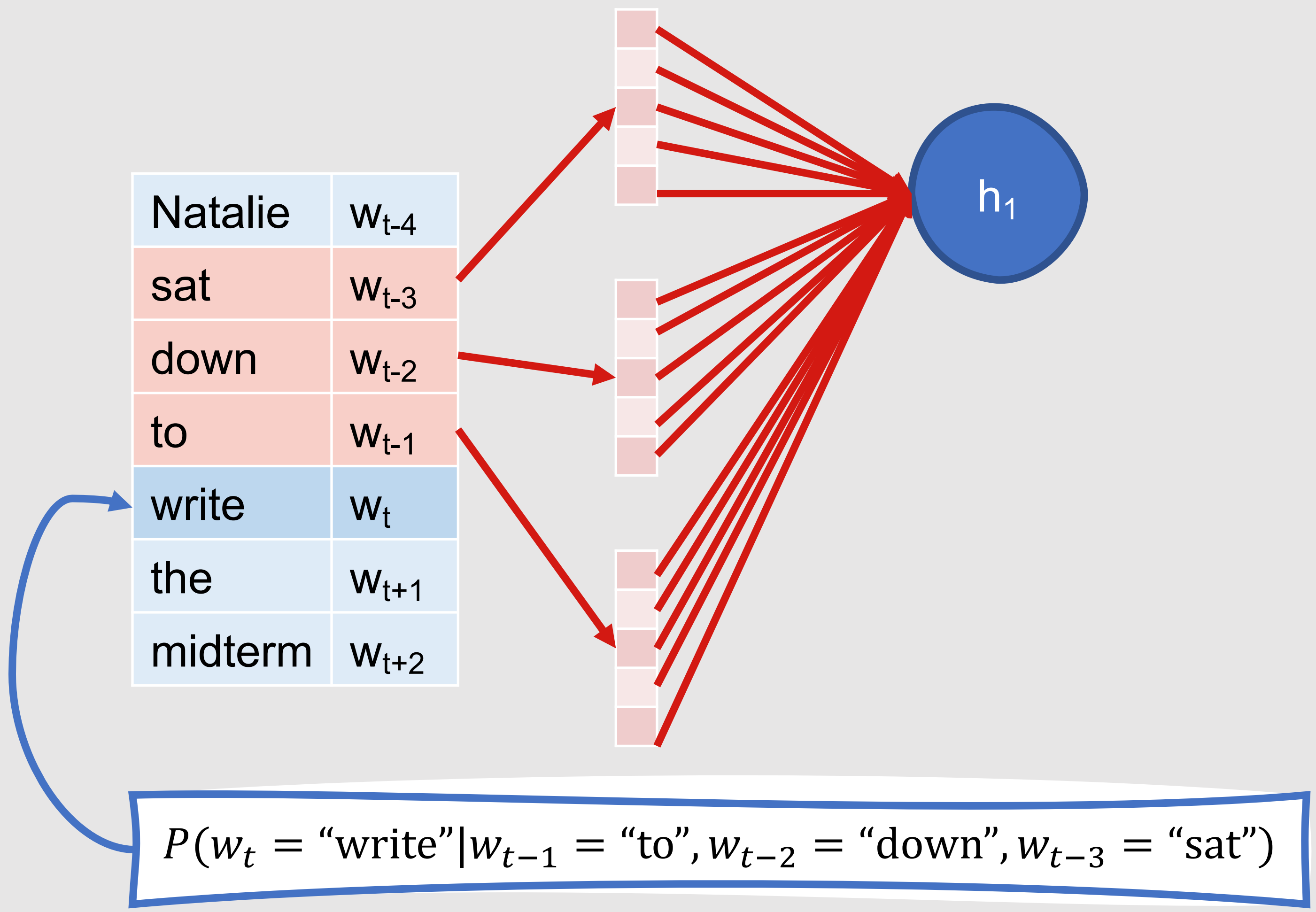
Neural Language Model



Natalie	w_{t-4}
sat	w_{t-3}
down	w_{t-2}
to	w_{t-1}
write	w_t
the	w_{t+1}
midterm	w_{t+2}

$$P(w_t = \text{“write”} | w_{t-1} = \text{“to”}, w_{t-2} = \text{“down”}, w_{t-3} = \text{“sat”})$$

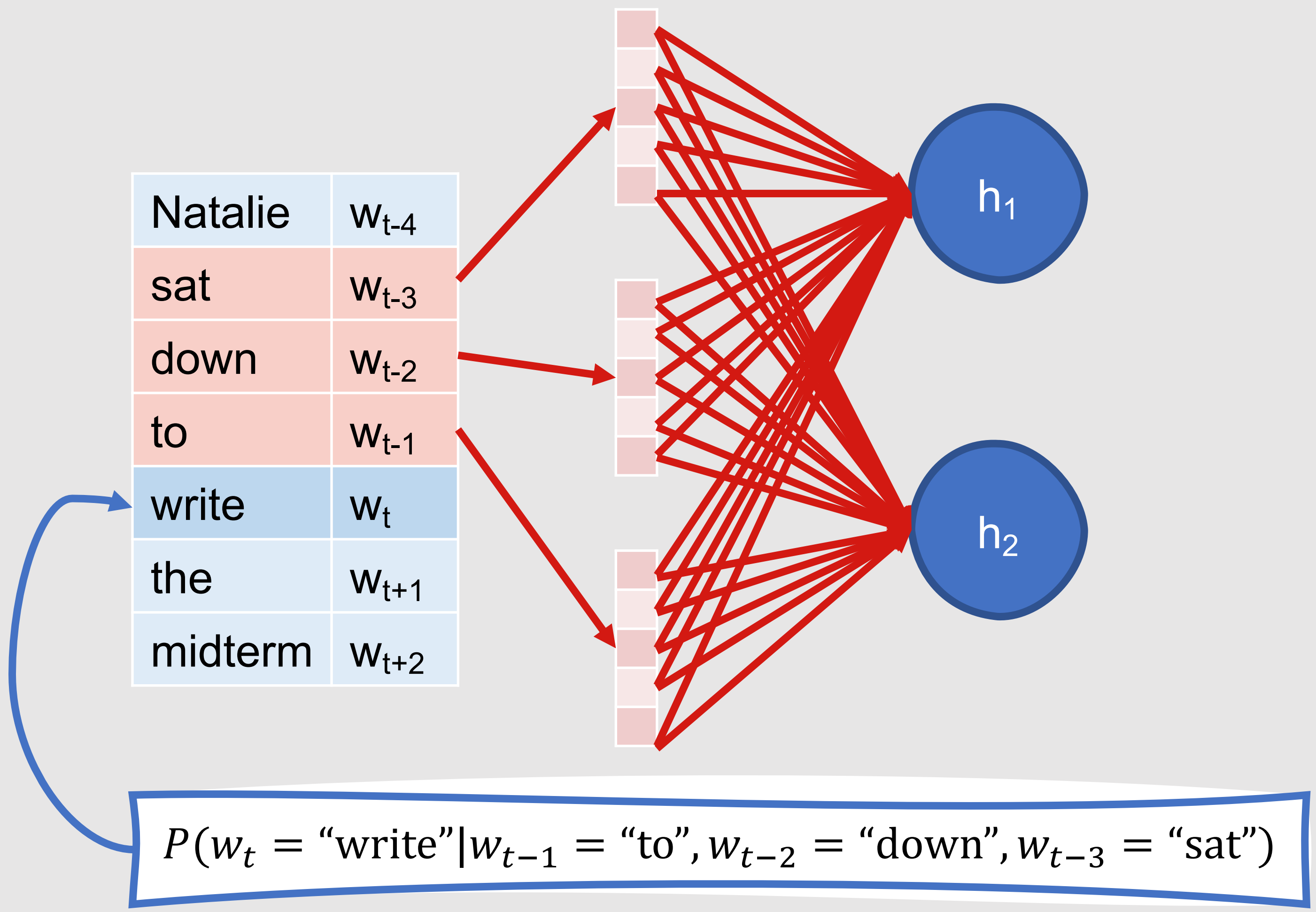
Neural Language Model



Natalie	w_{t-4}
sat	w_{t-3}
down	w_{t-2}
to	w_{t-1}
write	w_t
the	w_{t+1}
midterm	w_{t+2}

$$P(w_t = \text{“write”} | w_{t-1} = \text{“to”}, w_{t-2} = \text{“down”}, w_{t-3} = \text{“sat”})$$

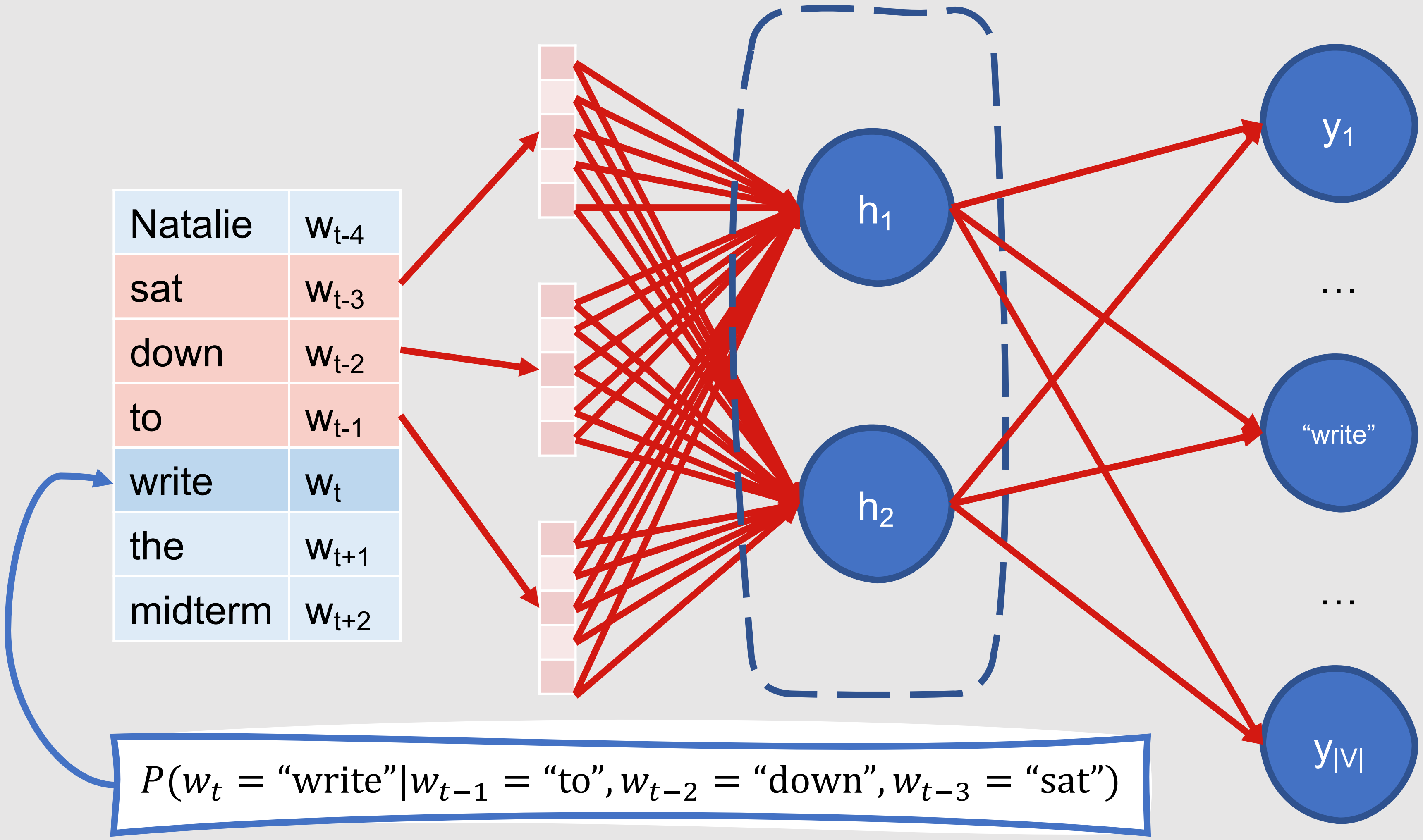
Neural Language Model



Natalie	w_{t-4}
sat	w_{t-3}
down	w_{t-2}
to	w_{t-1}
write	w_t
the	w_{t+1}
midterm	w_{t+2}

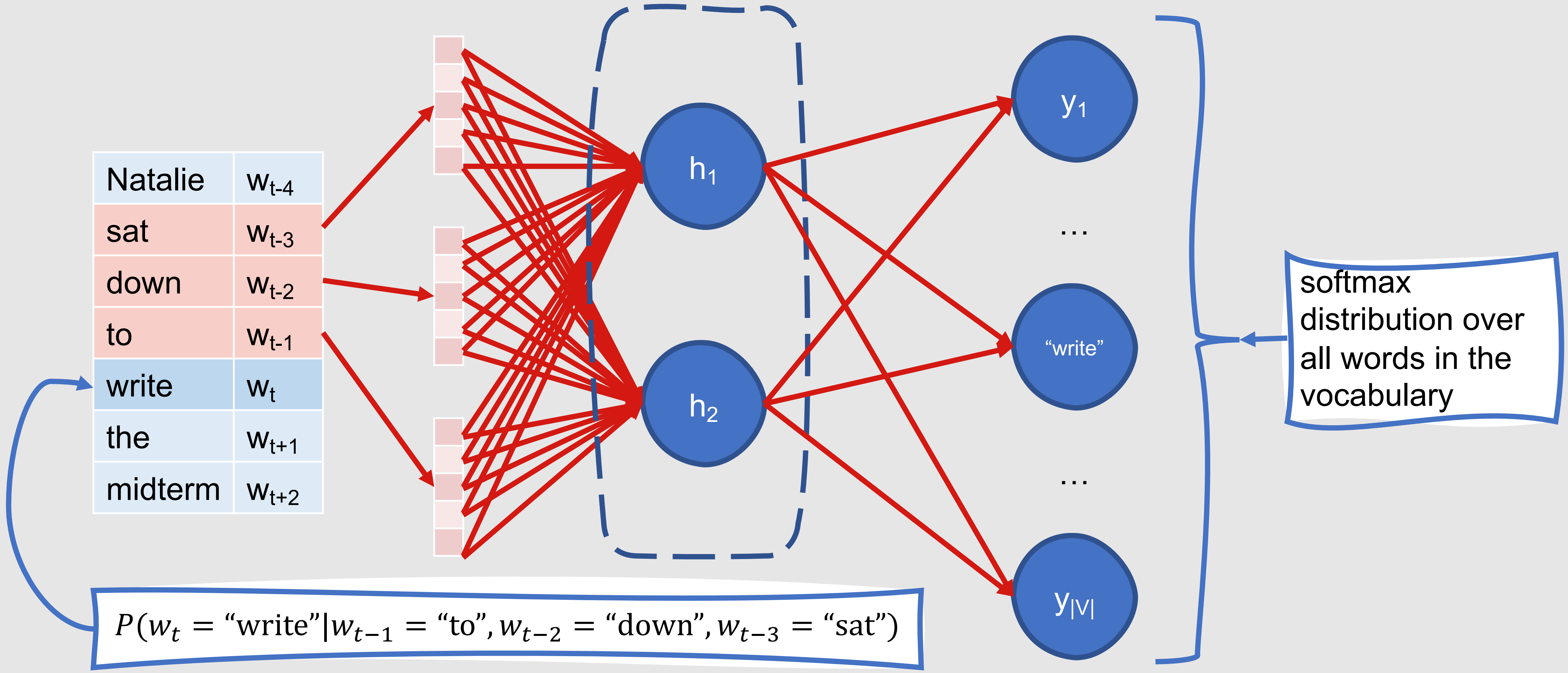
$$P(w_t = \text{“write”} | w_{t-1} = \text{“to”}, w_{t-2} = \text{“down”}, w_{t-3} = \text{“sat”})$$

Neural Language Model



$$P(w_t = \text{“write”} | w_{t-1} = \text{“to”}, w_{t-2} = \text{“down”}, w_{t-3} = \text{“sat”})$$

Neural Language Model



Natalie	w_{t-4}
sat	w_{t-3}
down	w_{t-2}
to	w_{t-1}
write	w_t
the	w_{t+1}
midterm	w_{t+2}

$$P(w_t = \text{“write”} | w_{t-1} = \text{“to”}, w_{t-2} = \text{“down”}, w_{t-3} = \text{“sat”})$$

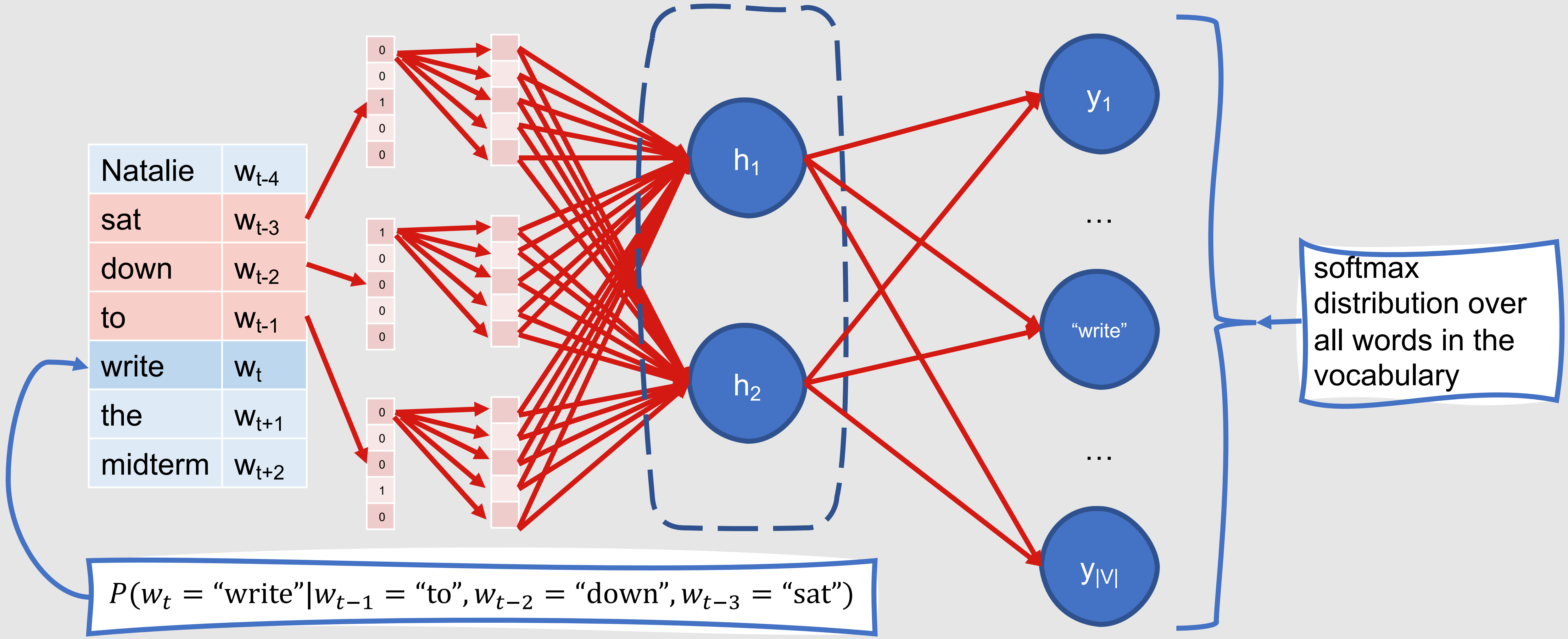
What if we don't already have dense word embeddings?

- When we use another algorithm to learn the embeddings for our input words, this is called **pretraining**
- However, sometimes it's preferable to learn embeddings while training the network, rather than using **pretrained embeddings**
  - E.g., if the desired application places strong constraints on what makes a good representation

Learning New Embeddings

- Start with a **one-hot vector** for each word in the vocabulary
  - Modified bag-of-words representation
  - Element for a given word is set to 1
  - All other elements are set to 0
- Randomly initialize the embedding layer
- Maintain a separate vector of weights for the embedding layer, for each vocabulary word

Neural Language Model



$P(w_t = \text{“write”} | w_{t-1} = \text{“to”}, w_{t-2} = \text{“down”}, w_{t-3} = \text{“sat”})$

Formal Definition: Learning New Embeddings

- Letting *E* be an embedding matrix of dimensionality *d*, with one row for each word in the vocabulary:
  - $\mathbf{e} = (E_{x_1}, E_{x_2}, \ldots, E_{x_n})$
  - $\mathbf{h} = \sigma(W\mathbf{e} + \mathbf{b})$
  - $\mathbf{z} = U\mathbf{h}$
  - $y = \text{softmax}(\mathbf{z})$
- Optimizing this network using the same techniques discussed for other neural networks will result in both (a) a model that predicts words, and (b) a new set of word embeddings that can be used for other tasks

Summary: Neural Networks and Neural Language Models

- **Feedforward neural networks** are multilayer networks in which all units in a layer are connected to all units in the previous and next layers (but no units in the current layer)
- Neural networks have **input**, **hidden**, and **output** units
- Values output by the output layer need to be converted to probabilities using specific **activation functions**
  - **softmax** converts output values into a set of probabilities proportional to the exponentials of the input values
- Training neural networks requires a **loss function** and an **optimization algorithm**
- With multilayer networks, losses need to be **backpropagated** all the way to the input layer when optimizing the model's weights
- **Neural language models** represent context using dense embeddings rather than *n*-grams
- Neural language models can either use **pretrained embeddings** for this purpose, or they can learn their own