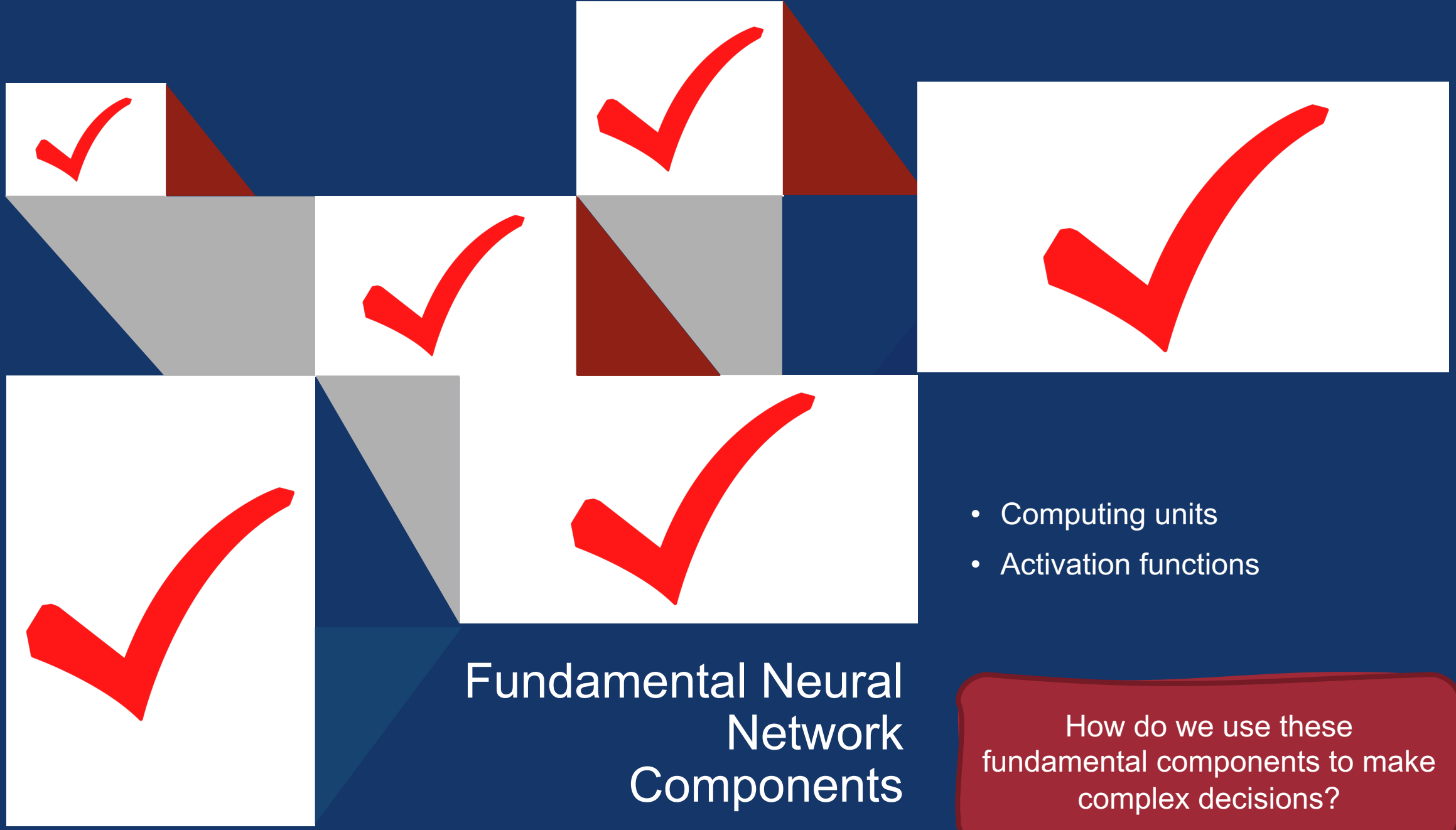
Neural Networks and Neural Language Models

Natalie Parde
UIC CS 421



Fundamental Neural Network Components

- Computing units
- Activation functions

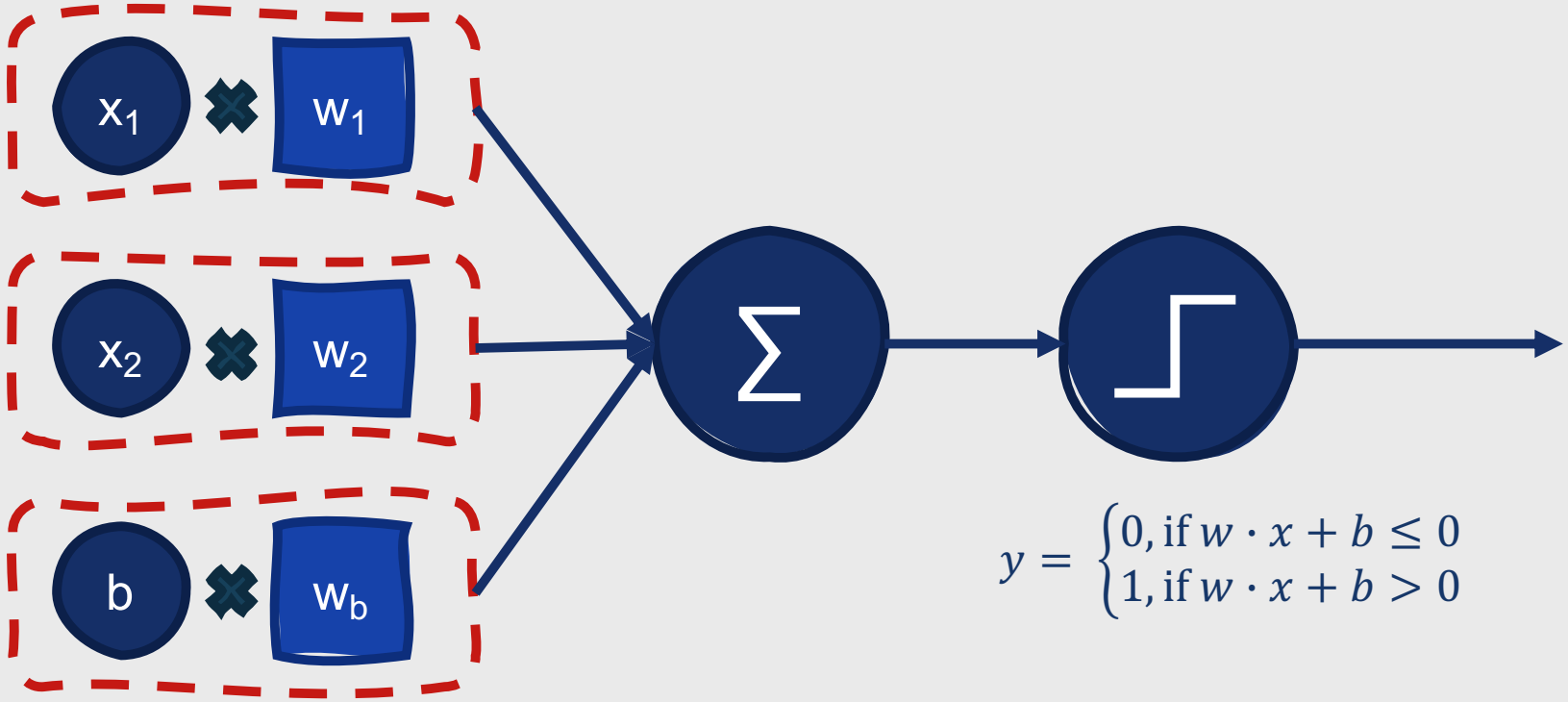
How do we use these fundamental components to make complex decisions?

Combining Computational Units

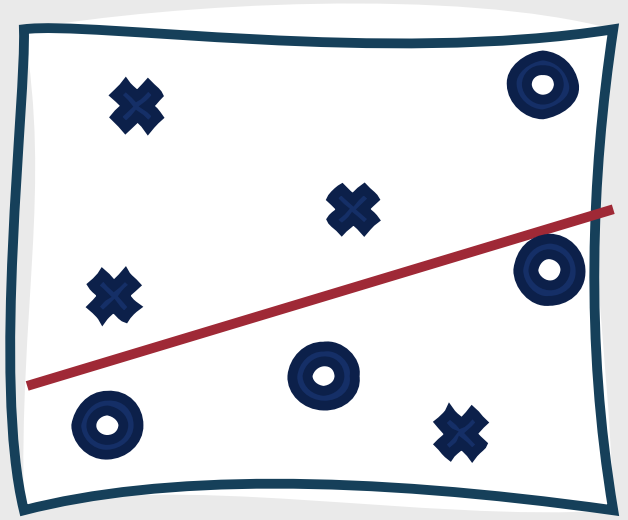
- Neural networks are powerful primarily because they are able to **combine multiple computational units into larger networks**
- Many problems cannot be solved using a single computational unit

Early example of this: The XOR problem

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



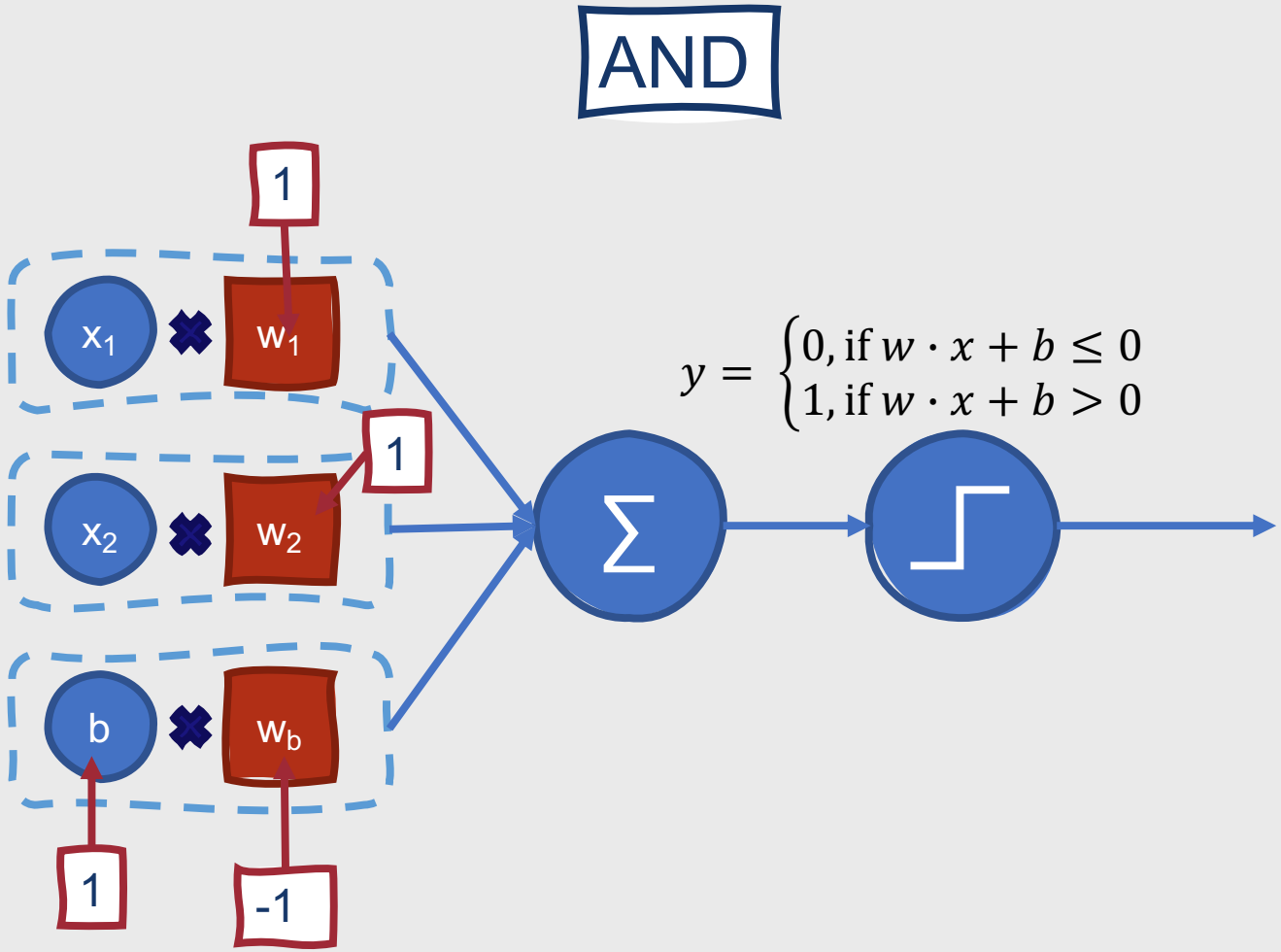
$$y = \begin{cases} 0, \text{if } w \cdot x + b \leq 0 \\ 1, \text{if } w \cdot x + b > 0 \end{cases}$$



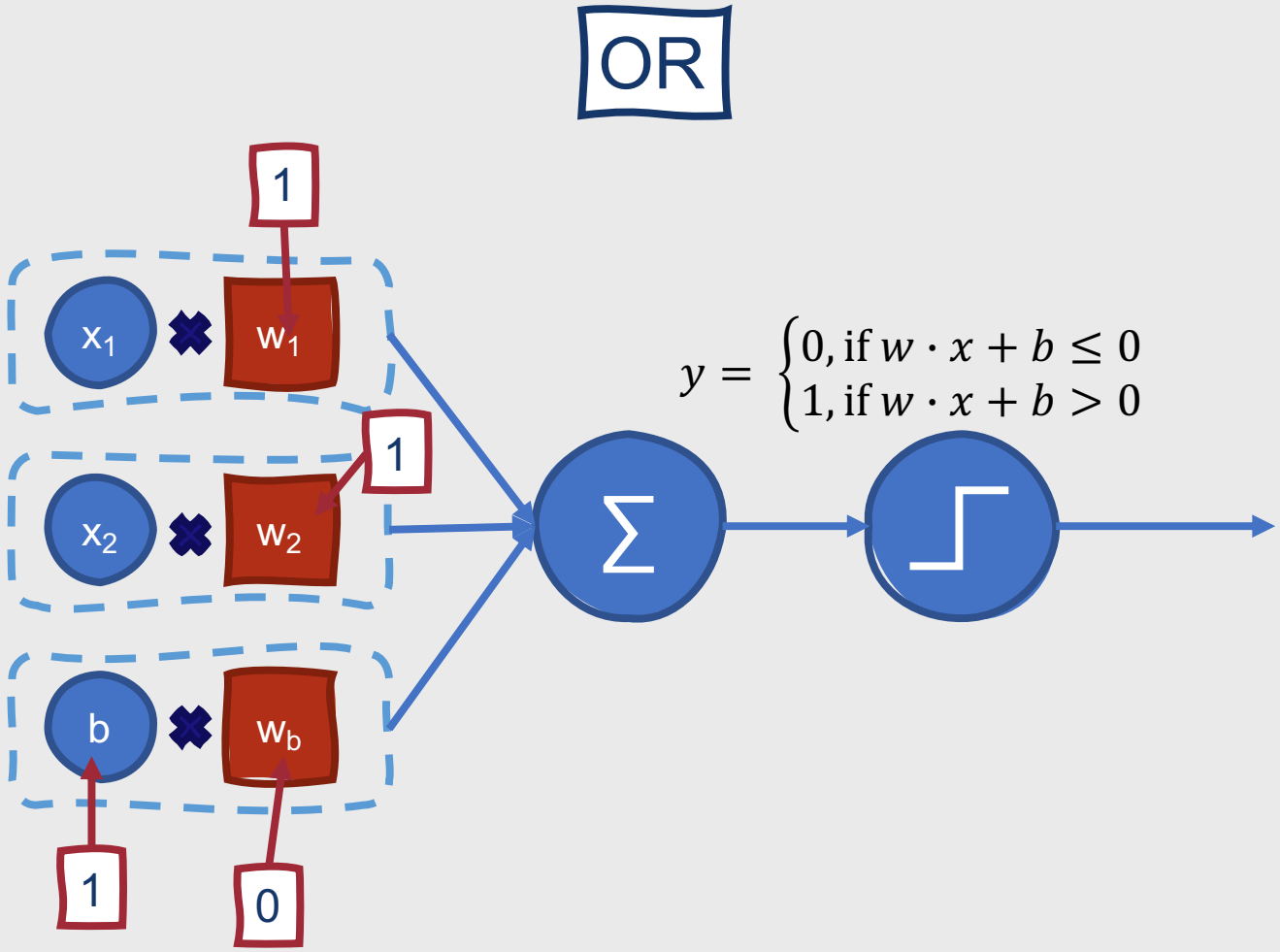
AND and OR can both be solved using a single perceptron.

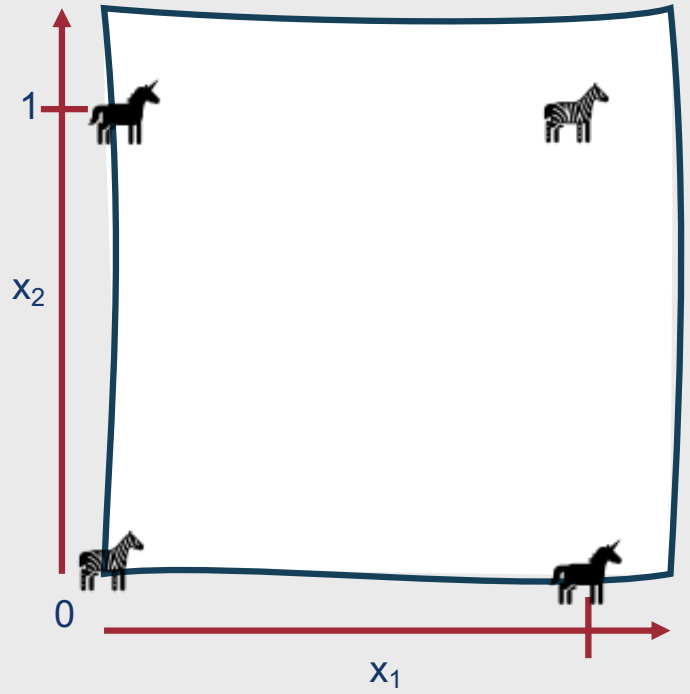
- **Perceptron:** A function that outputs a binary value based on whether the product of its inputs and associated weights surpasses a threshold
  - Learns this threshold iteratively by trying to find the boundary that is best able to distinguish between data of different categories

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



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



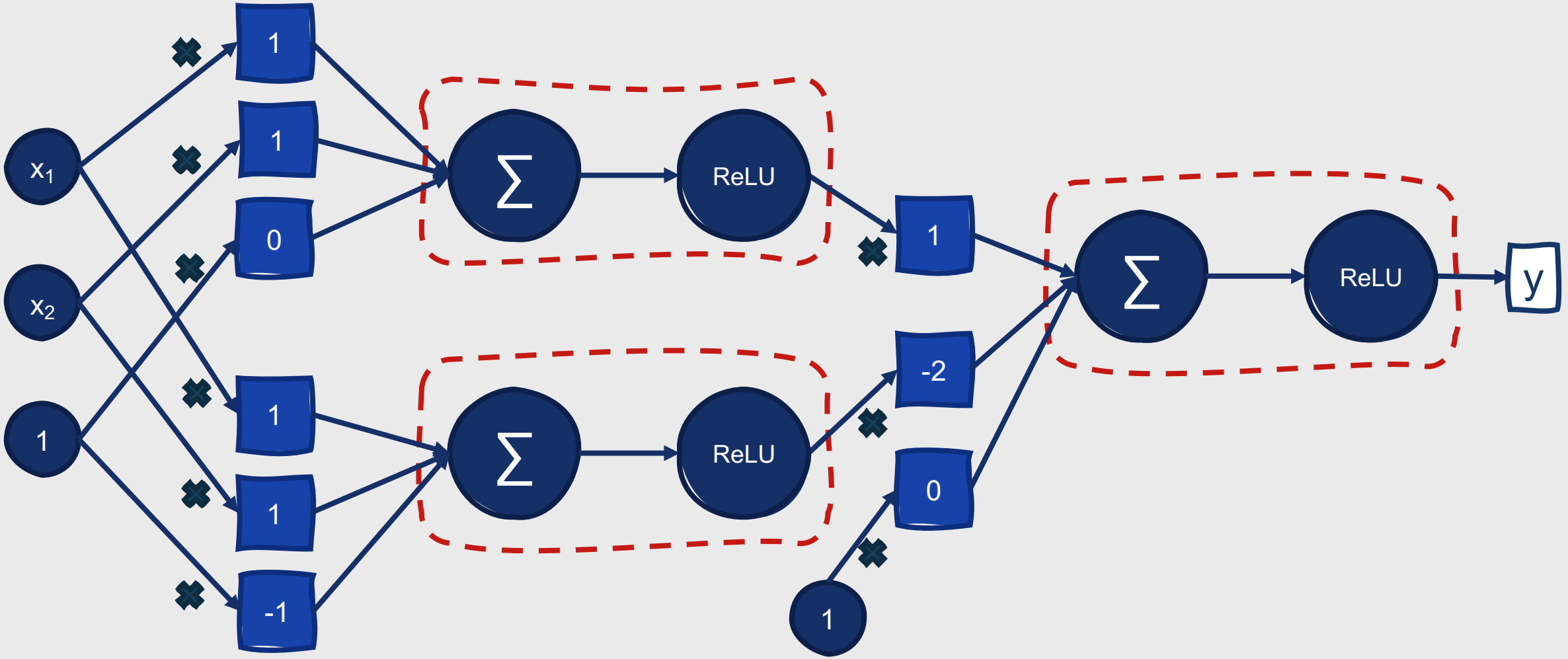


AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

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

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

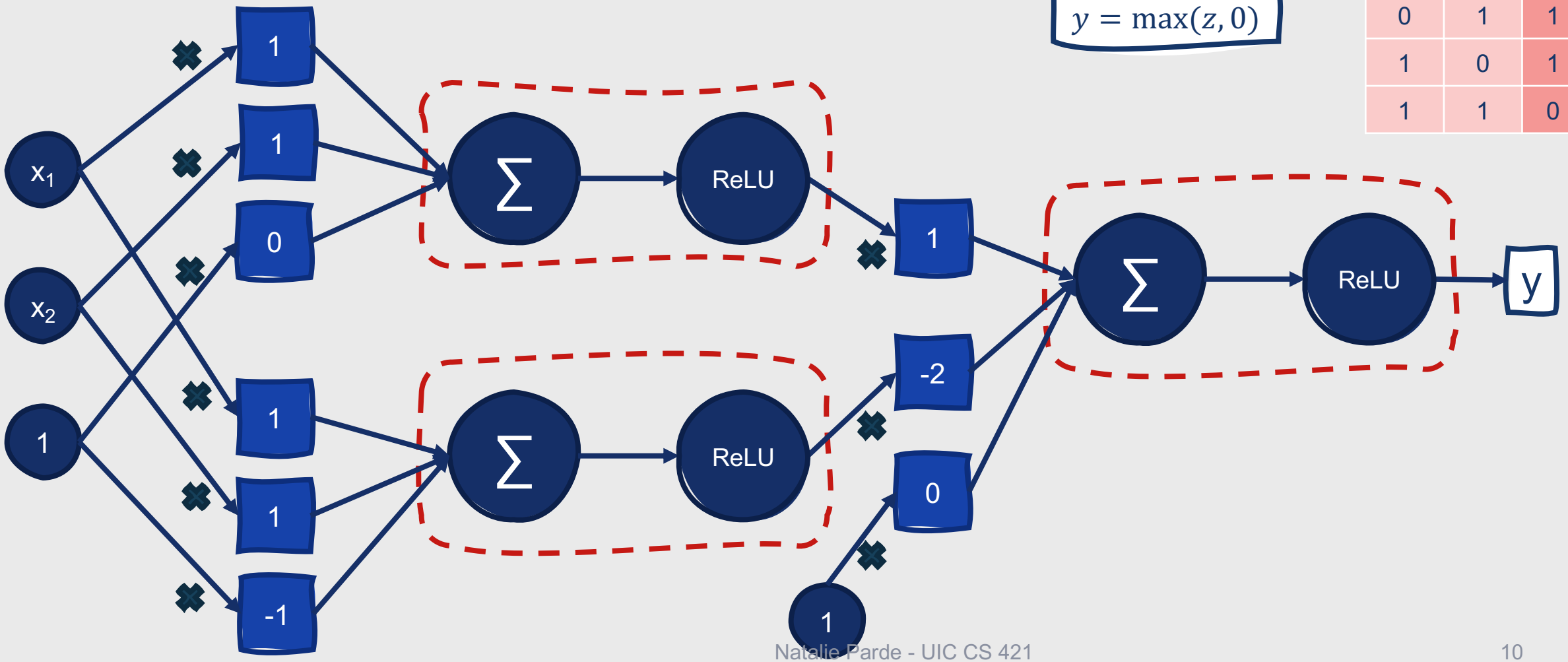
The only successful way to compute XOR is by combining these smaller units into a larger network.



Truth Table Examples: XOR

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

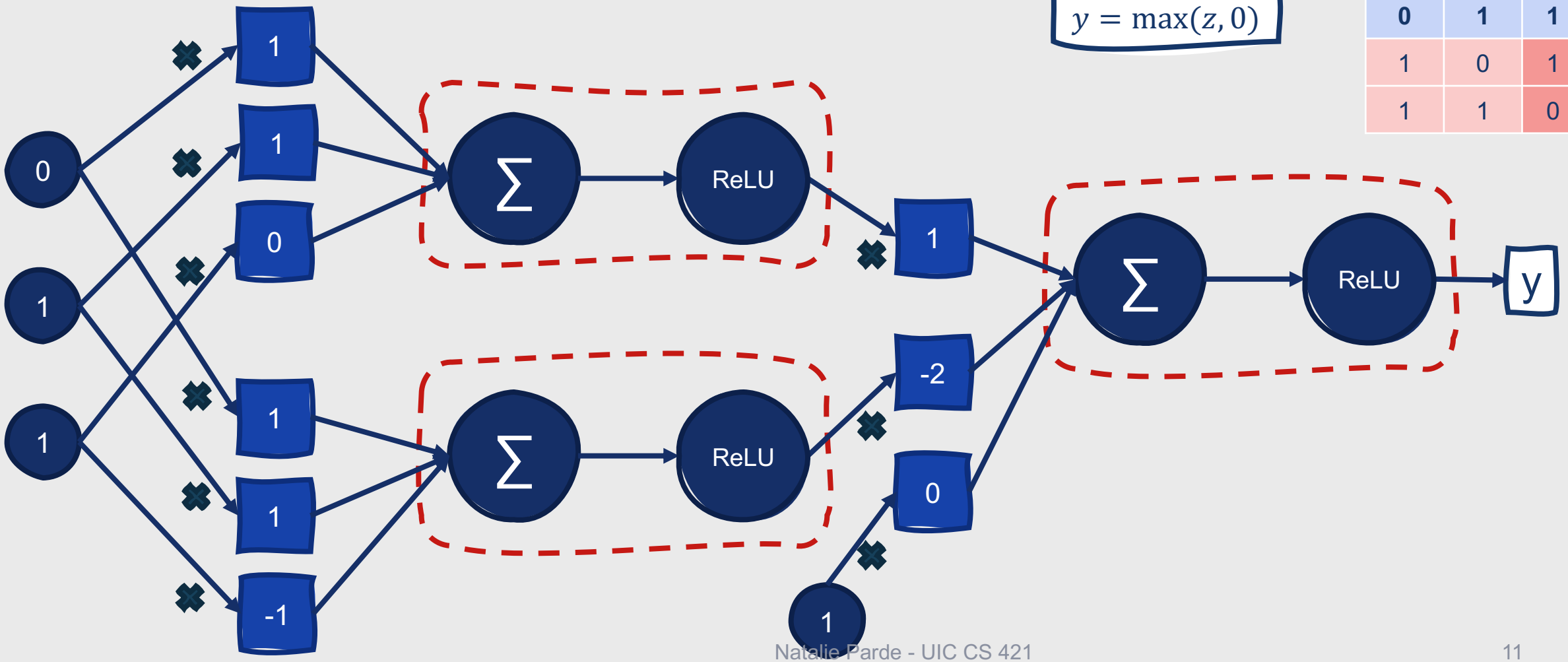
$y = \max(z, 0)$



Truth Table Examples: XOR

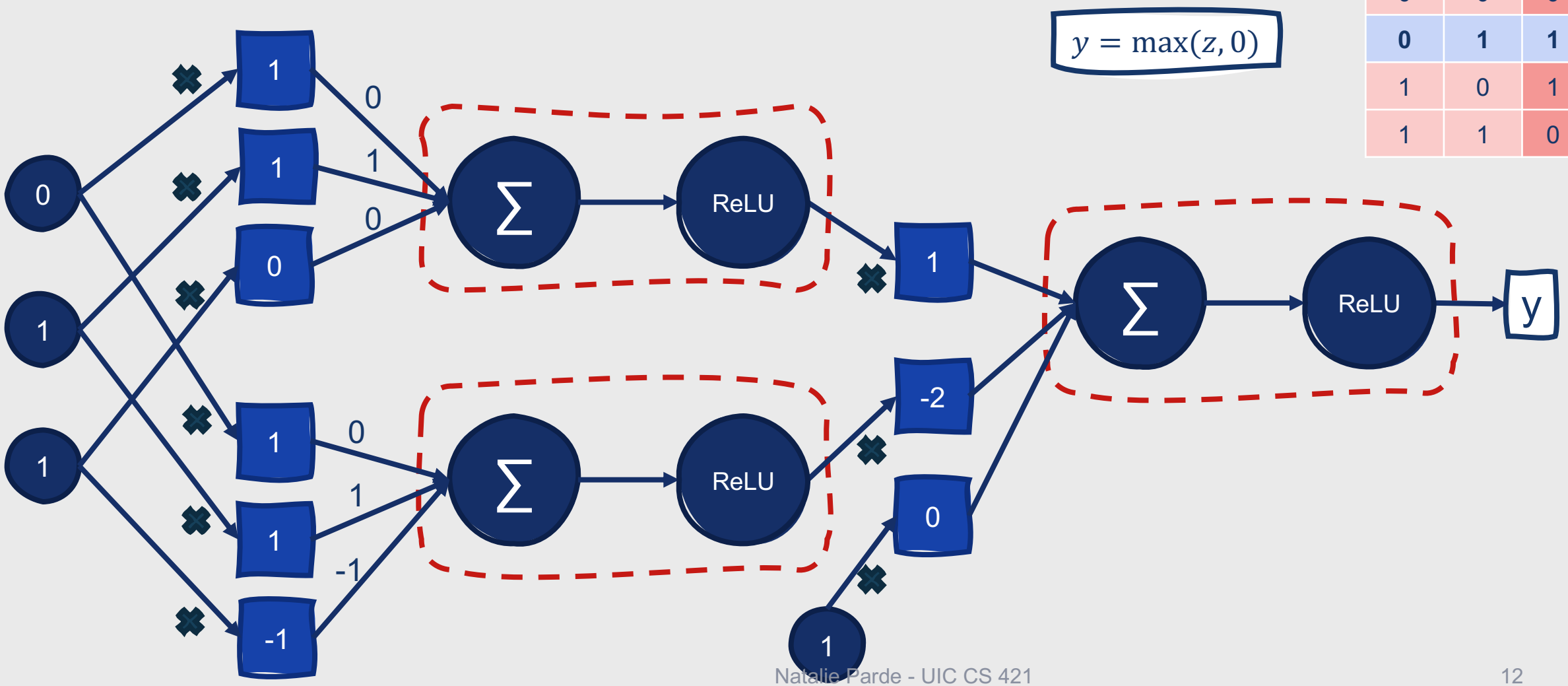
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

$y = \max(z, 0)$



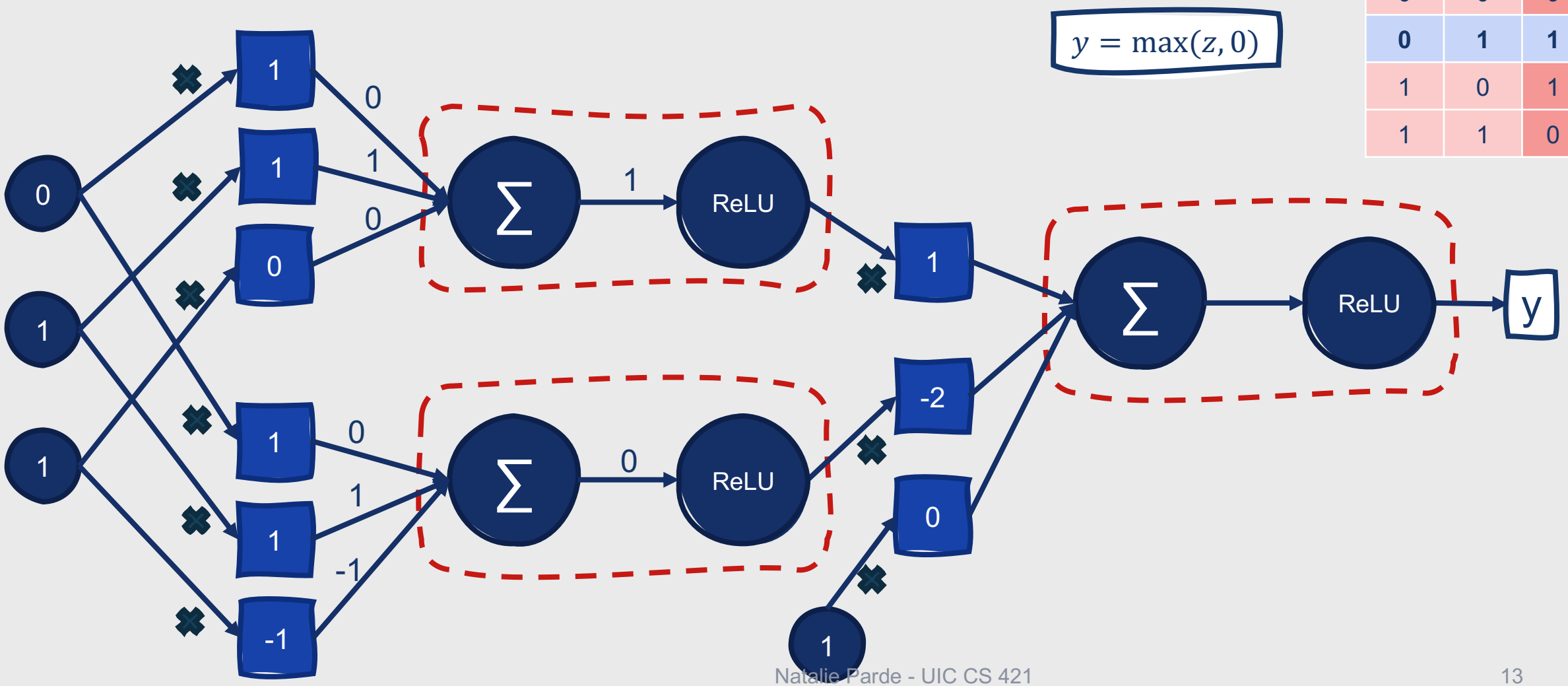
Truth Table Examples: XOR

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



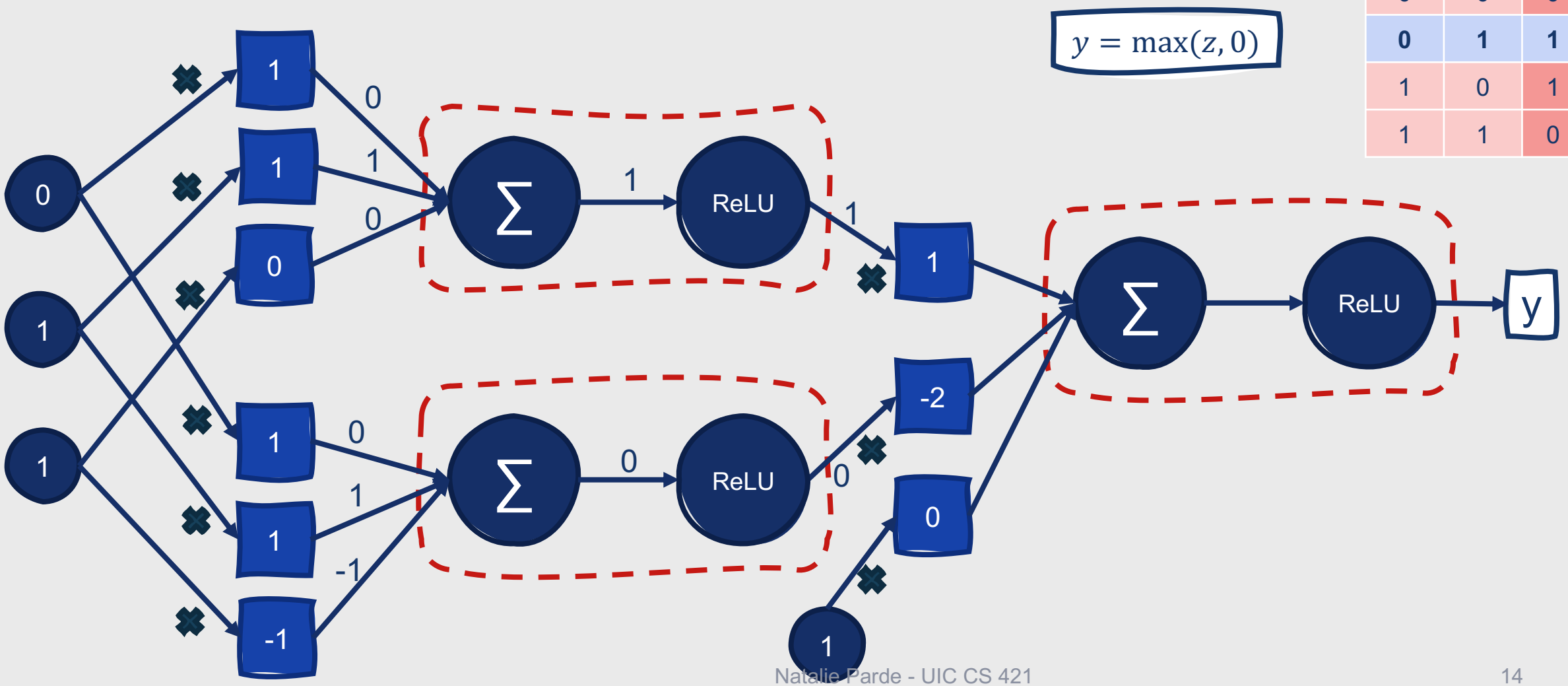
Truth Table Examples: XOR

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



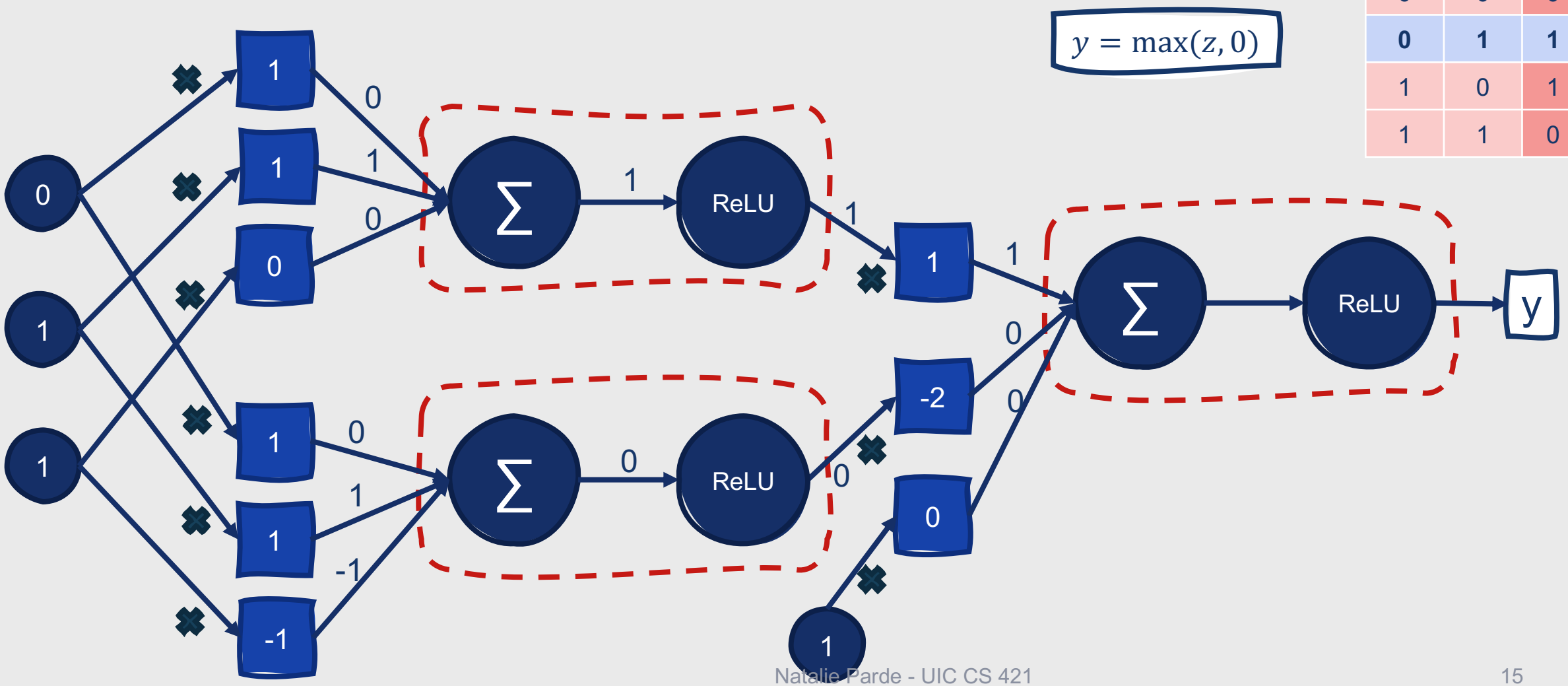
Truth Table Examples: XOR

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



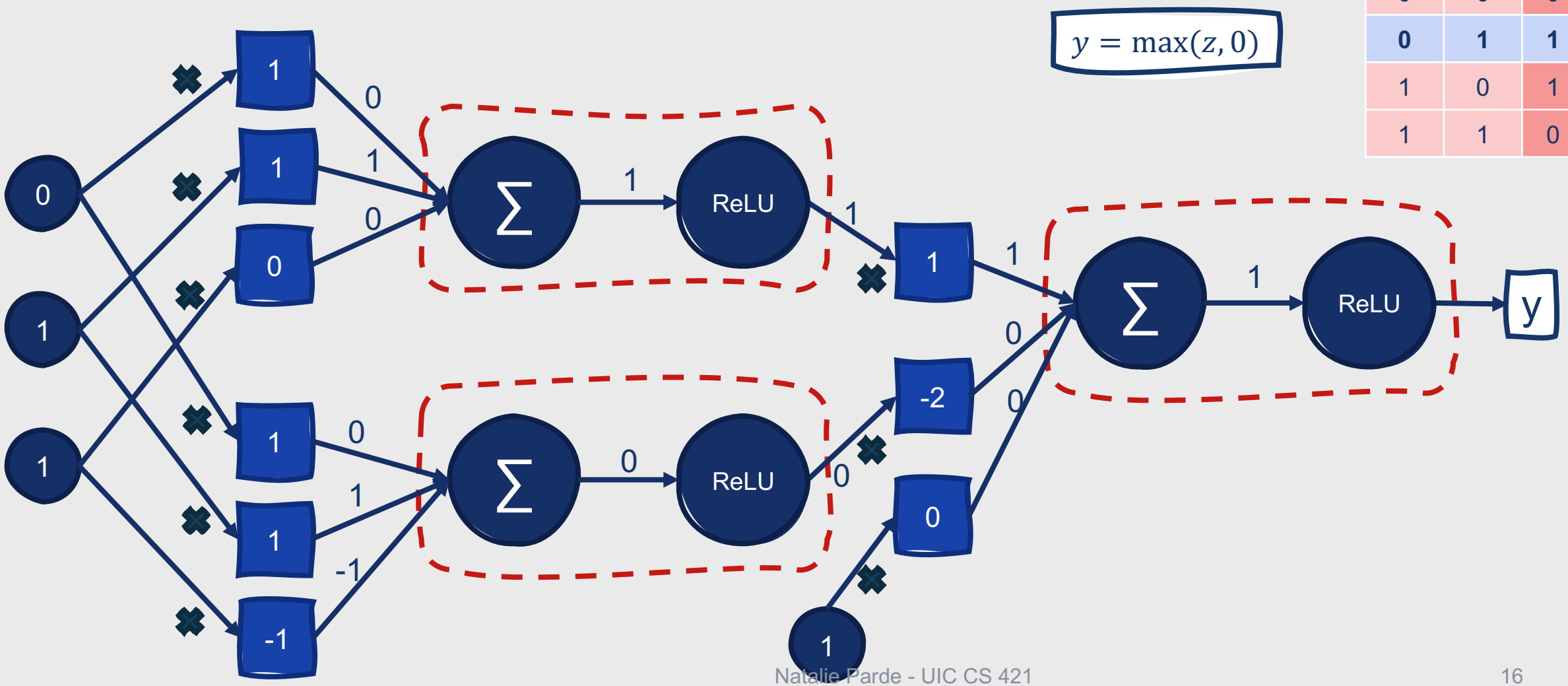
Truth Table Examples: XOR

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



Truth Table Examples: XOR

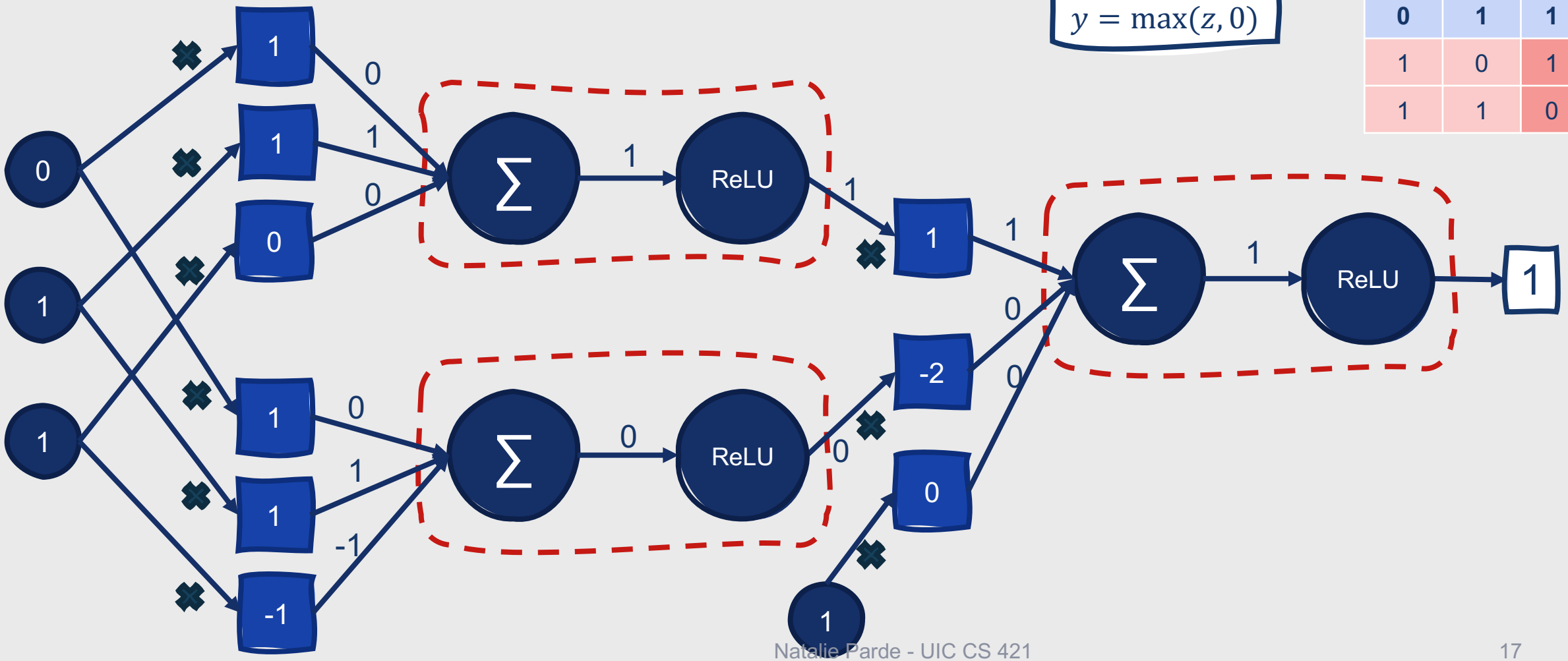
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



Truth Table Examples: XOR

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

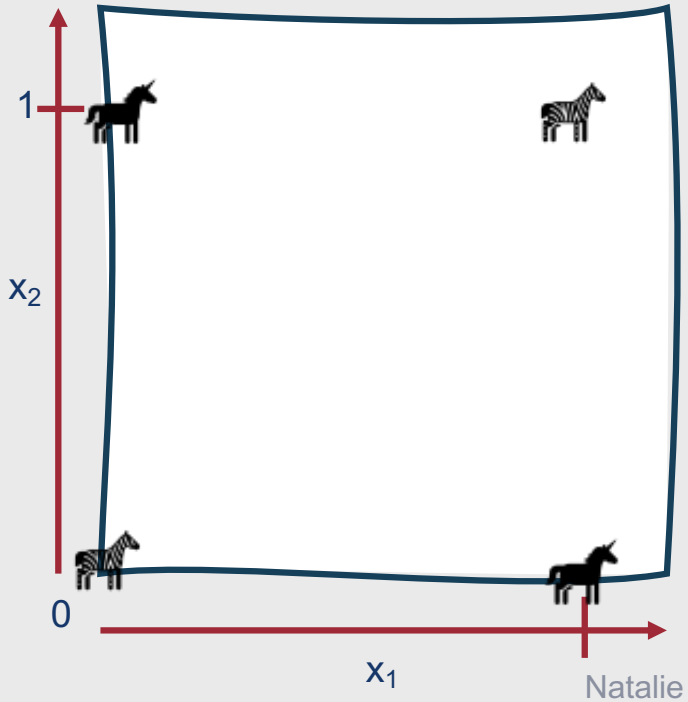
$$y = \max(z, 0)$$



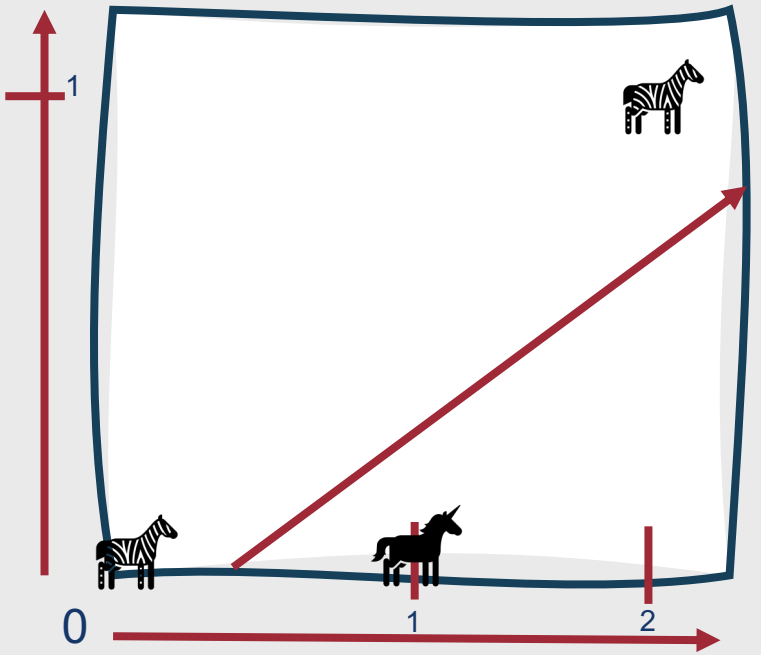
Why does this work?

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

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



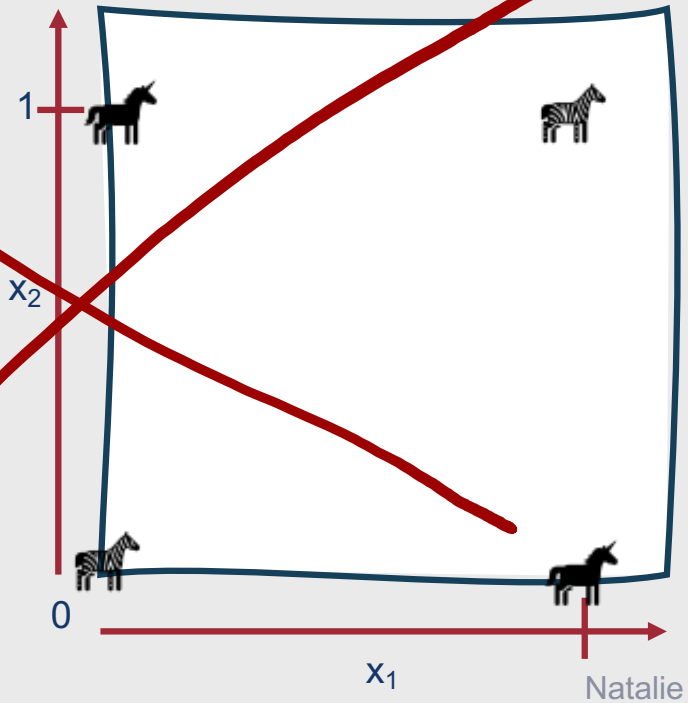
XOR		
h0	h1	y
0	0	0
1	0	1
2	1	0



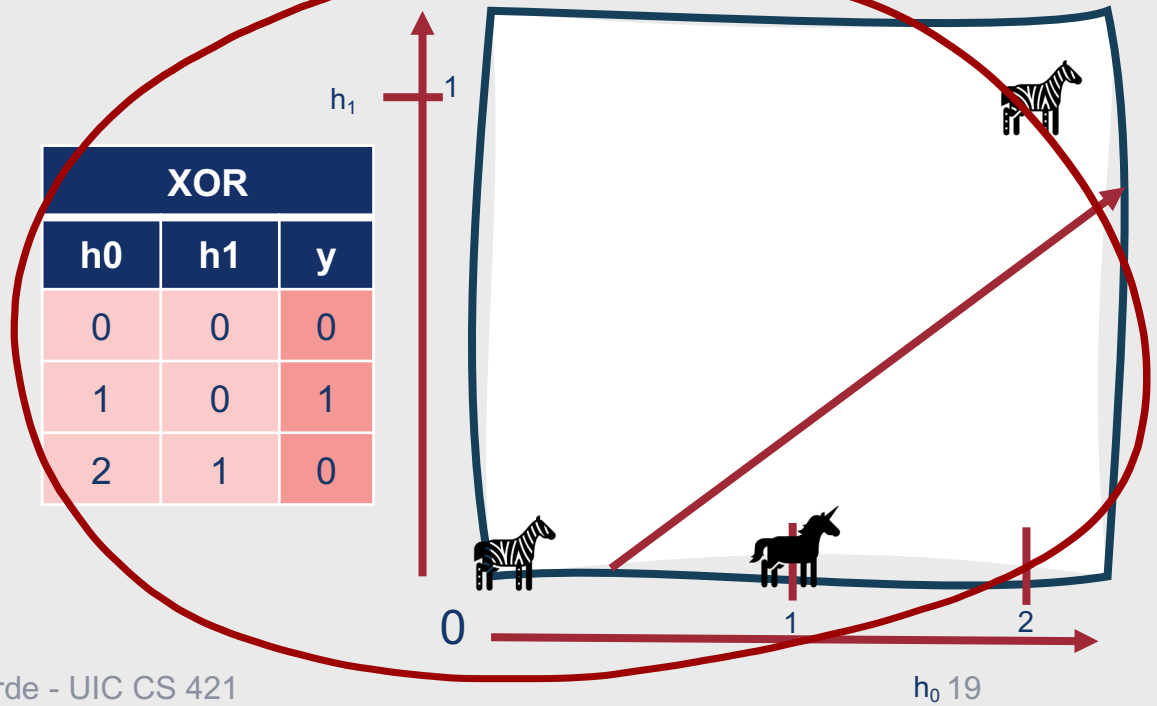
Why does this work?

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

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



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



Combining Computational Units

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

More about specific unit types in feedforward networks….

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

Input Units

0.5	0.2	0.1	0.7	0.4

- Vector of scalar values
  - Word embedding
  - Other feature vector
- No computations performed in input units
- An input (layer 0) vector x has a dimensionality of n_0, where n_0 is the number of inputs
  - So, $x \in \mathbb{R}^{n_0}$

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 a unit in the previous (input or hidden) layer x_i to a unit in the next layer 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:
  - $\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$

Hidden Layer Dimensions

- A hidden layer 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 between layers n_0 and n_1 thus has the dimensionality $W \in \mathbb{R}^{n_1 \times n_0}$

Output Units

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

Output Layer

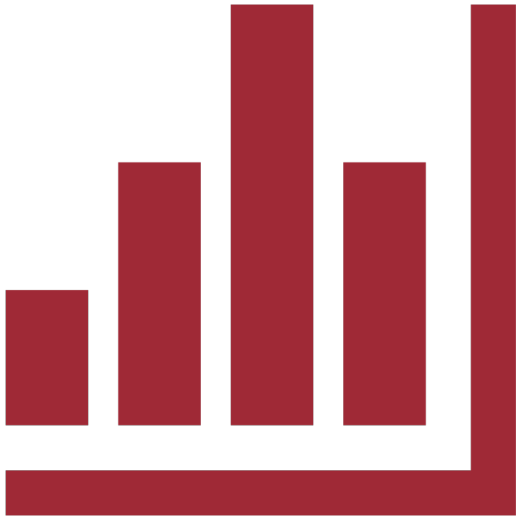
- Provides a probability distribution across the output nodes
- How?
  - Output layer also has a weight matrix, U
  - Following intuition, $z = U\mathbf{h}$, where **h** is the vector of outputs from the previous hidden layer

Output Layer Dimensions

- Letting n_j be the number of nodes in the output layer, $z \in \mathbb{R}^{n_j}$
- The weight matrix U thus has the dimensionality $U \in \mathbb{R}^{n_j \times n_i}$, where n_i is the number of hidden units in the previous layer
  - U_{ab} is the weight from unit *b* in the hidden layer to unit *a* in the output layer

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

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



Feedforward Network

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

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

- Let $W^{[n]}$ be the weight matrix for layer n, $\mathbf{b}^{[n]}$ be the bias vector for layer n, and so forth
- Let $g(\cdot)$ be any activation function
- 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*
    - $z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}$
    - $a^{[i]} = g^{[i]}(z^{[i]})$
  - $y' = a^{[n]}$

Activation functions for the final layer are often different from earlier layers.

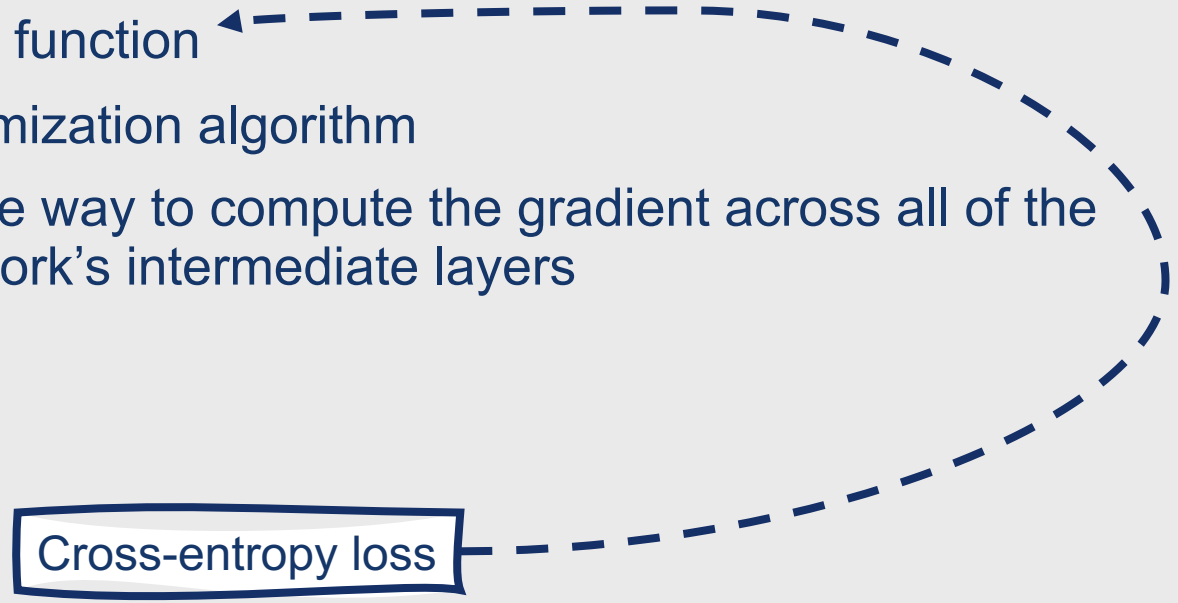
- Earlier layers will more commonly be ReLU or tanh
- Final layers will more commonly be softmax (for multinomial classification) or sigmoid (for binary classification)

How do we train neural networks?

- Loss function
- Optimization algorithm
- Some way to compute the gradient across all of the network's intermediate layers

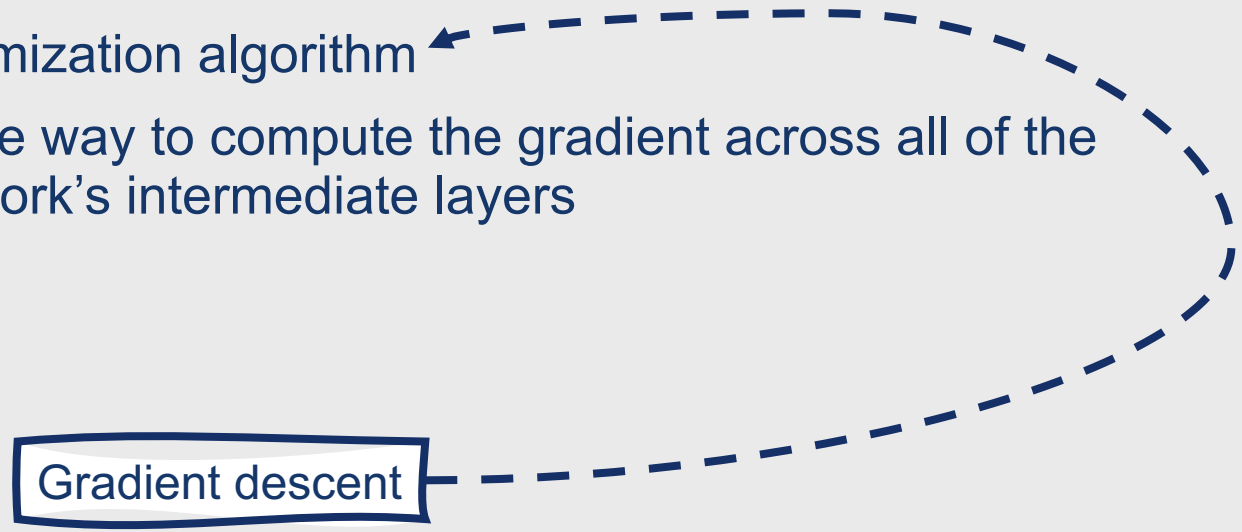
How do we train neural networks?

- ✓ Loss function
- ❑ Optimization algorithm
- ❑ Some way to compute the gradient across all of the network's intermediate layers



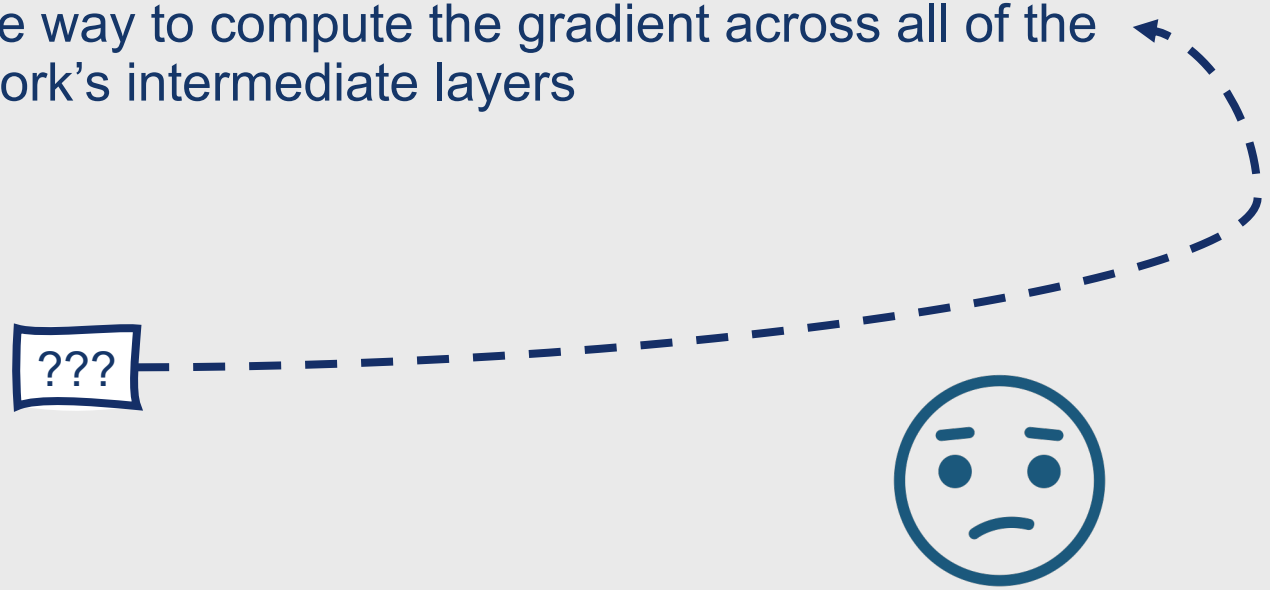
How do we train neural networks?

- ✓ Loss function
- ✓ Optimization algorithm
- ❑ Some way to compute the gradient across all of the network's intermediate layers



How do we train neural networks?

- ✓ Loss function
- ✓ Optimization algorithm
- ❑ Some way to compute the gradient across all of the network's intermediate layers



Backpropagation

- A method for propagating loss values all the way back to the beginning of a deep neural network, even though it's only computed at the end of the network

Why is this necessary?

- Simply taking the derivative like we did for logistic regression only provides the gradient for the most recent (i.e., last) weight layer
- What we need is a way to:
  - Compute the derivative with respect to weight parameters occurring earlier in the network as well
  - Even though we can only compute loss at a single point (the end of the network)

Backpropagation in a nutshell….

- Compute your loss at the final layer
- Propagate your loss backward using the chain rule
  - Given a function f(x) = u(v(x)):
    - Find the derivative of u(x) with respect to v(x)
    - Find the derivative of v(x) with respect to x
    - Multiply the two together
    - $\frac{df}{dx} = \frac{du}{dv} * \frac{dv}{dx}$
- Update weights at each layer based on this information

General Tips for Improving Neural Network Performance

- **Normalize input values** to have a mean of 0
- **Initialize weights** with small random numbers
- 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

Fortunately, you shouldn't need to build your neural networks from scratch!

TensorFlow

- *https://www.tensorflow.org/*

Keras

- *https://keras.io/*

PyTorch

- *https://pytorch.org/*

DL4J

- *https://deeplearning4j.org/*

Neural Language Models

- Popular application of neural networks
- Advantages over *n*-gram language models:
  - Can handle longer histories
  - Can generalize over contexts of similar words
- Disadvantage:
  - Slower to train
- Neural language models have 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.

There are many different architectures for neural language modeling.

- GPT-3: https://beta.openai.com/playground
- XLNet: https://huggingface.co/xlnet-base-cased
- OPT: https://huggingface.co/facebook/opt-125m
- BLOOM: https://huggingface.co/bigscience/bloom-560m
- …and many more!

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.

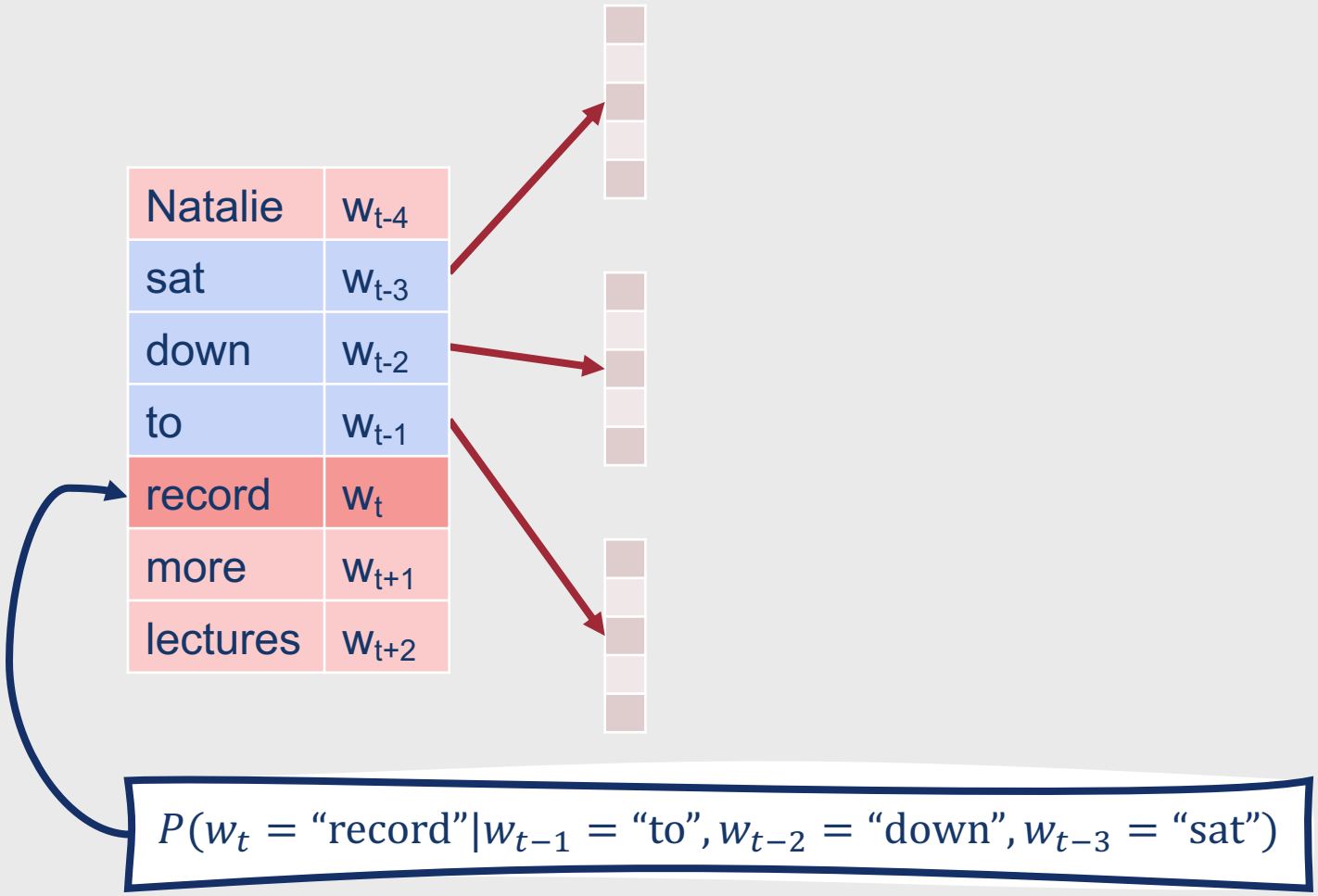
- Allows them to generalize to unseen data better than *n*-gram models
- Embeddings can come from various sources
  - E.g., pretrained Word2Vec embeddings

Neural Language Model

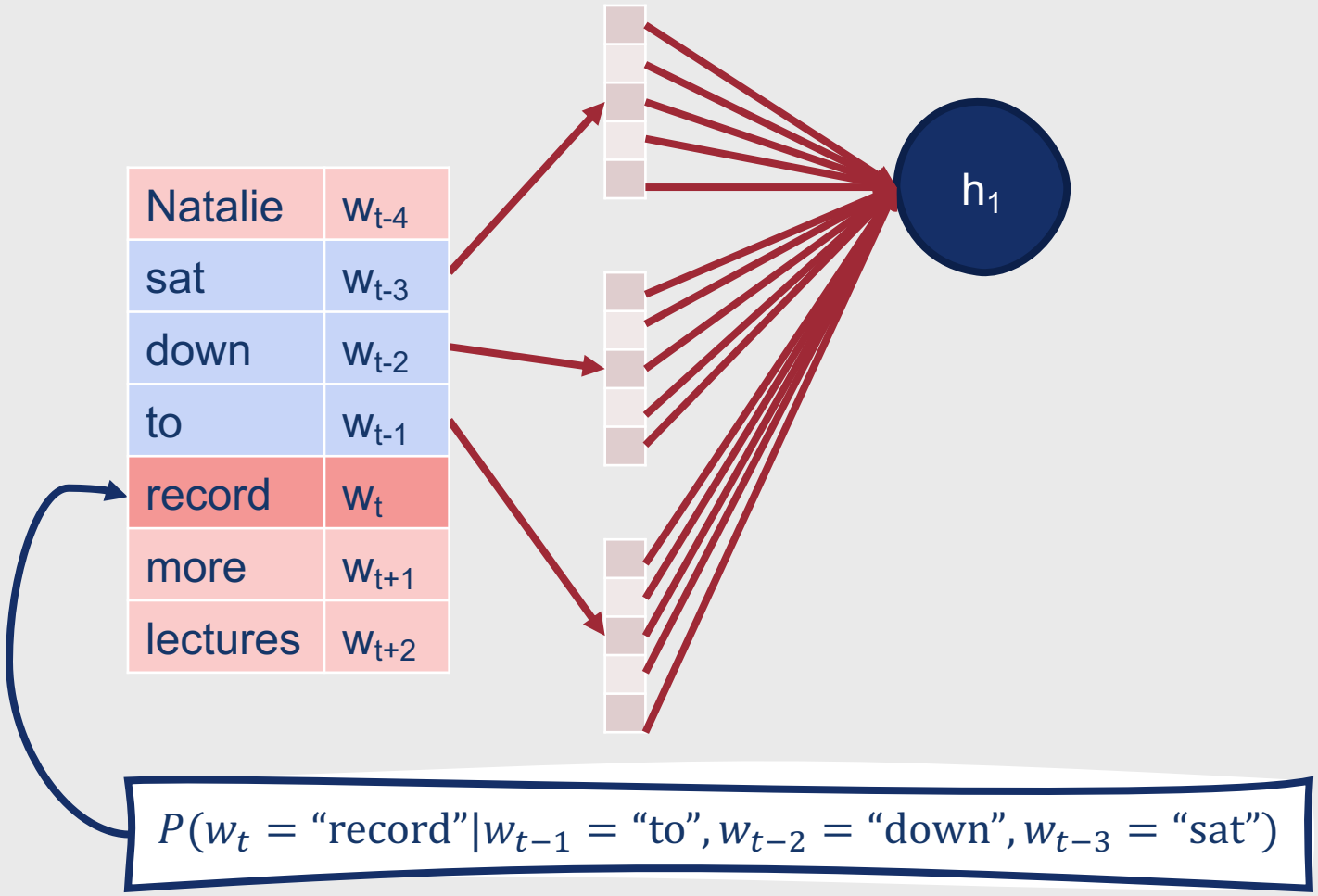
Natalie	w_{t-4}
sat	w_{t-3}
down	w_{t-2}
to	w_{t-1}
record	w_t
more	w_{t+1}
lectures	w_{t+2}

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

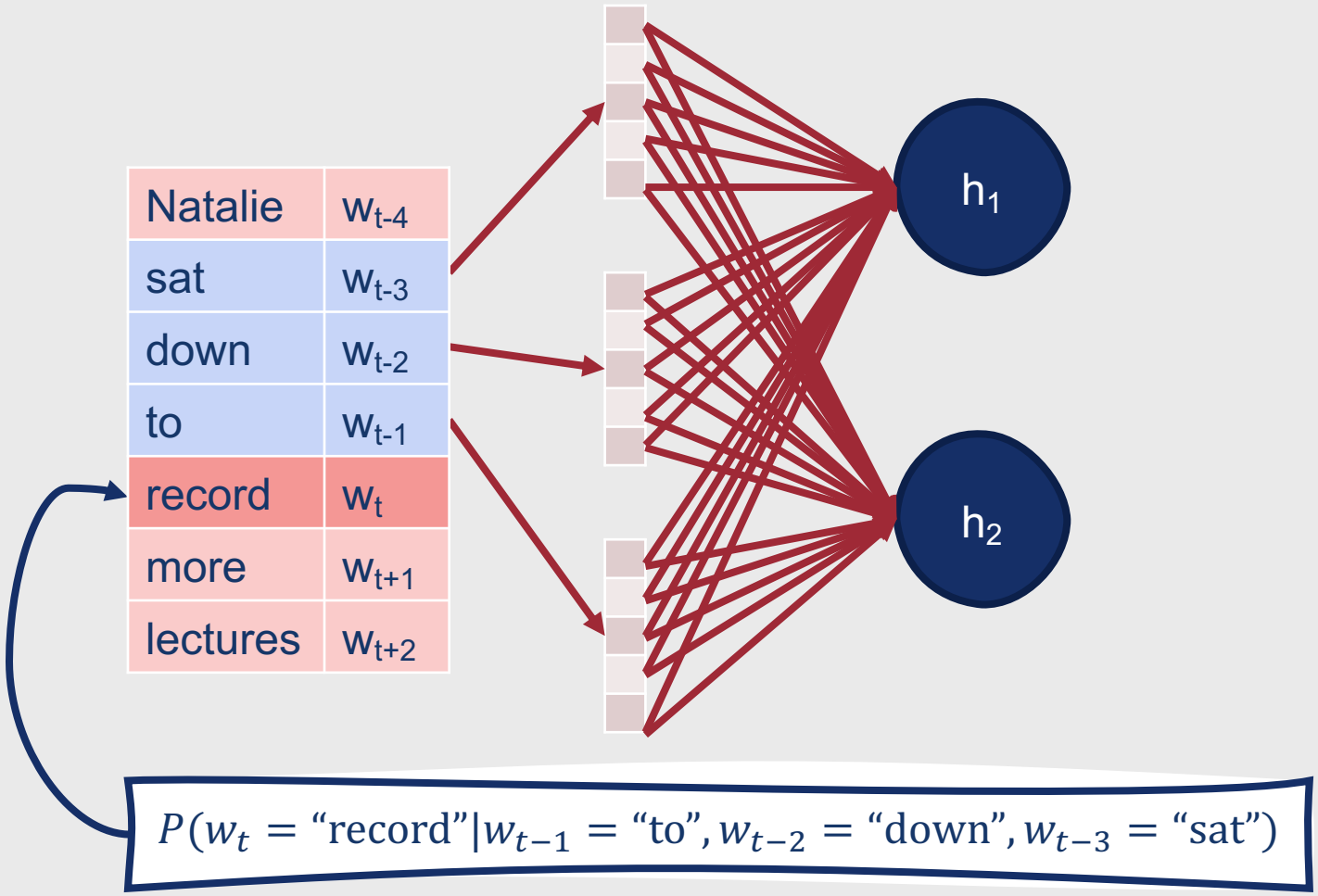
Neural Language Model



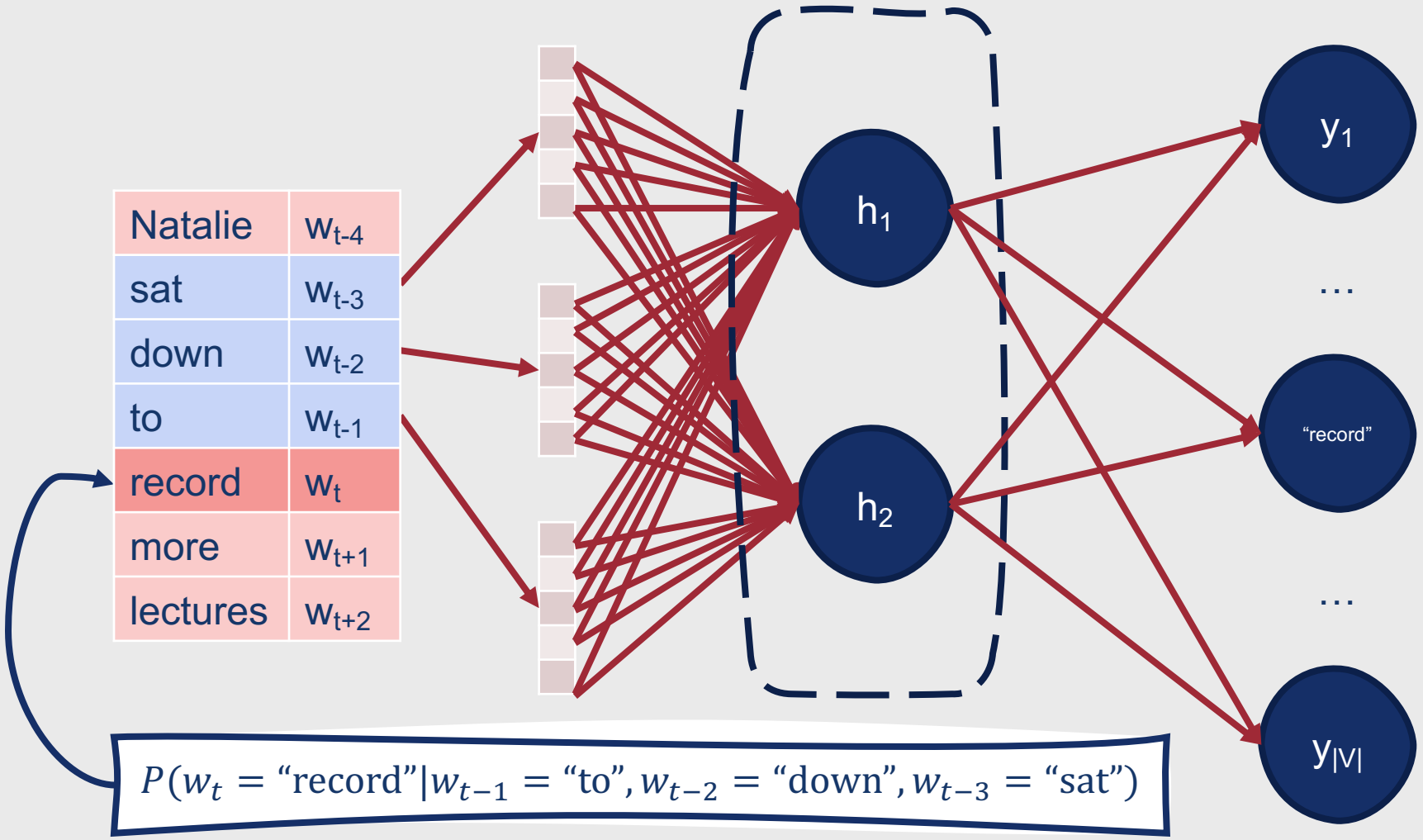
Neural Language Model



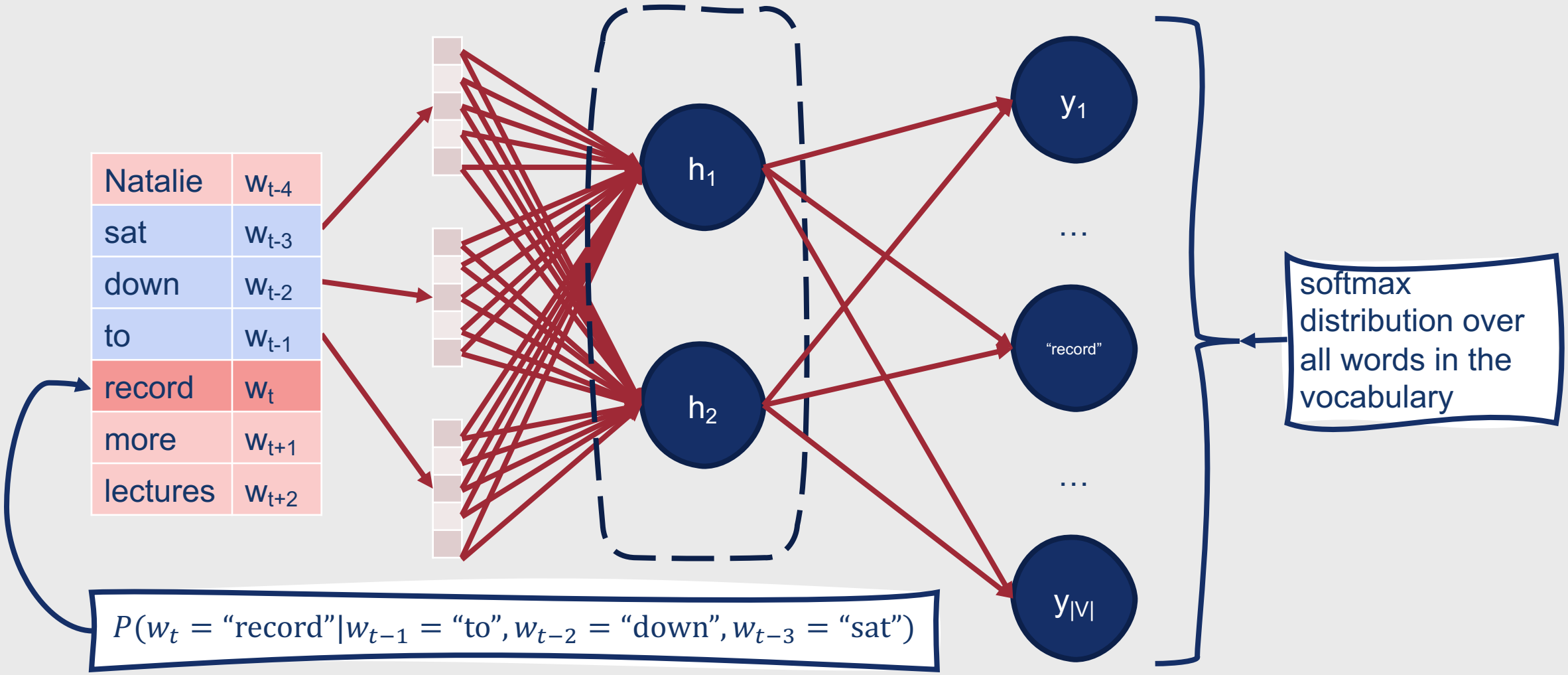
Neural Language Model



Neural Language Model



Neural Language Model



- 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

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

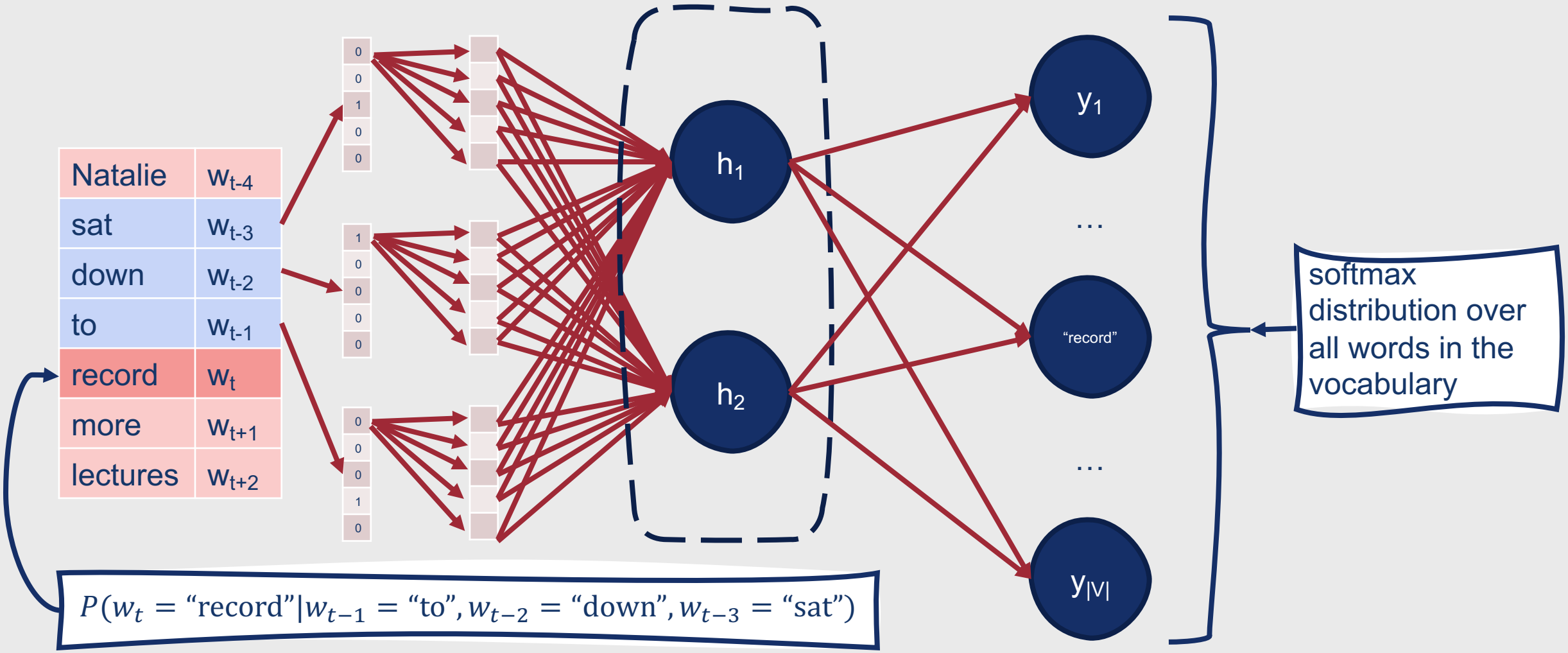
Learning New Embeddings

- Start with a **one-hot vector** for each word in the vocabulary
  - Element for a given word is set to 1
  - All other elements are set to 0
- Randomly initialize the hidden (weight/embedding) layer
- Maintain a separate vector of weights for that layer, for each vocabulary word

Formal Definition: Learning New Embeddings

- Letting *E* be an embedding matrix, 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 model that predicts words
  - A new set of word embeddings that can be used for other tasks

Neural Language Model



Summary: Neural Networks and Neural Language Models

- Computing units can be combined with another to solve complex tasks
- Depending on their location within the network architectures, units may represent **input**, internal compute units (**hidden** units), or **output** classification units
- Loss can be propagated backward through the network from the output layer to earlier layers using **backpropagation**
- Network architectures can be optimized via a **fine-tuning** process
- Neural networks can be used to build **neural language models**
- Neural language models can also be used to learn new language representations