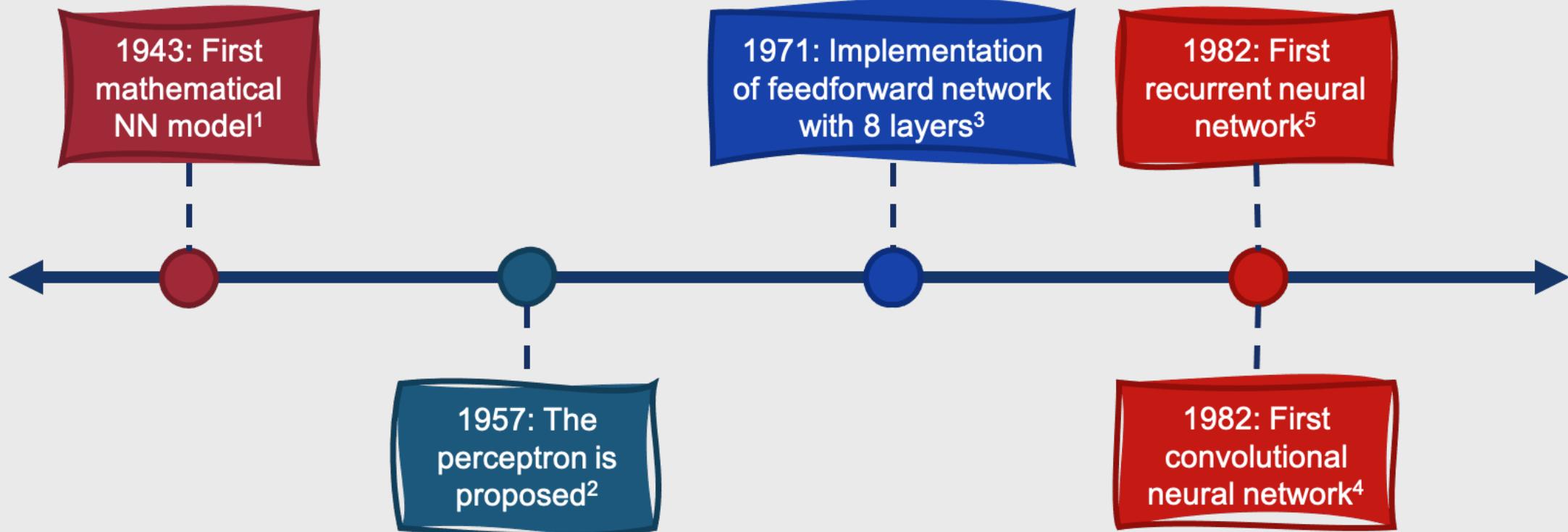


# Deep Learning Architectures for Sequence Processing

Natalie Parde  
UIC CS 521

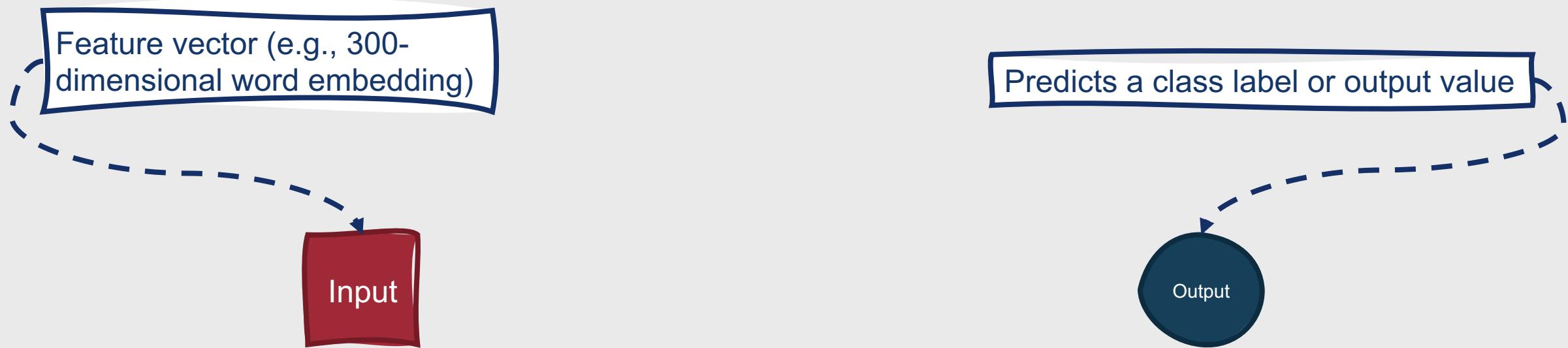
# Review: Neural Networks Basics



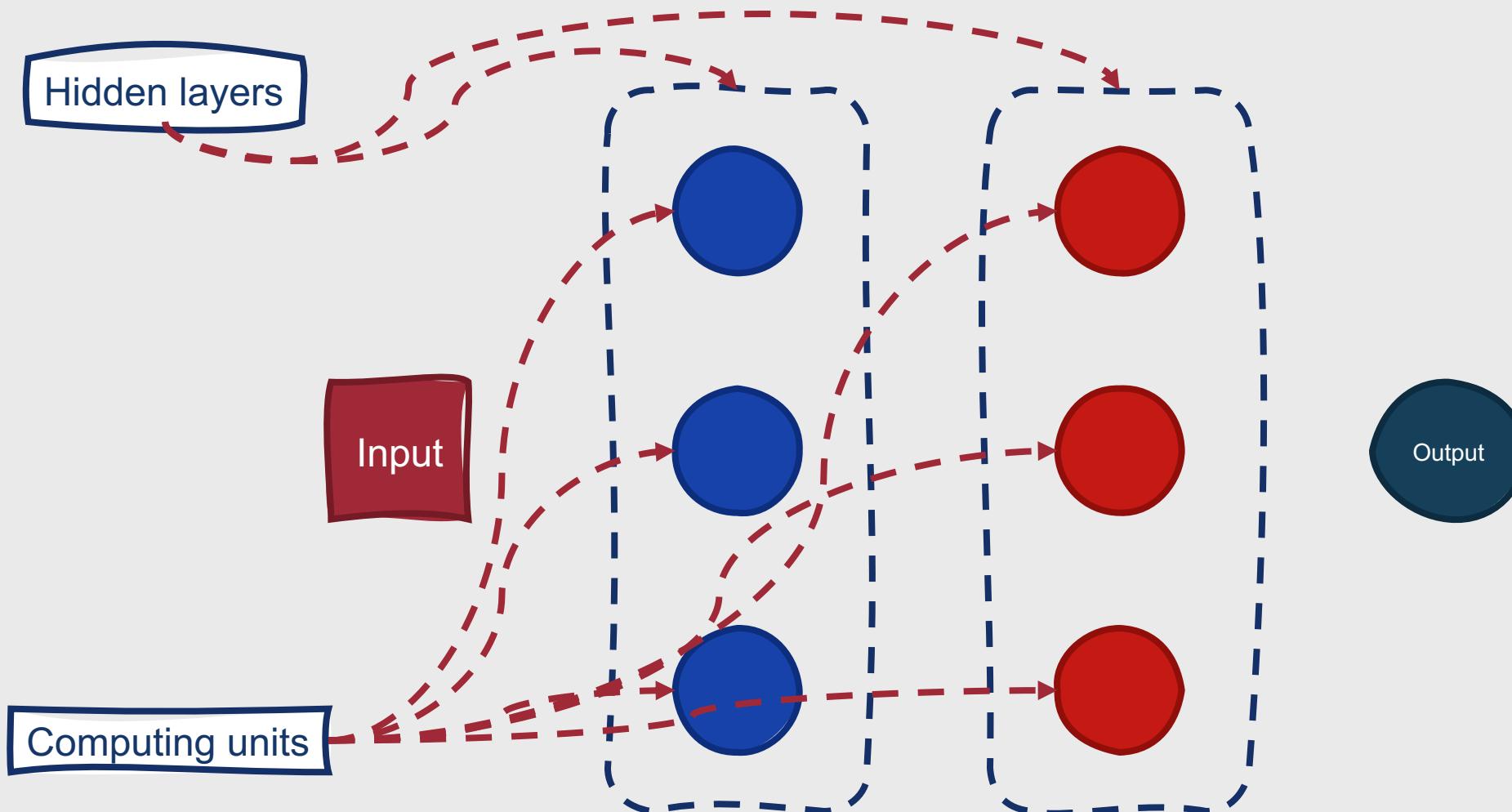
# Feedforward Neural Networks

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

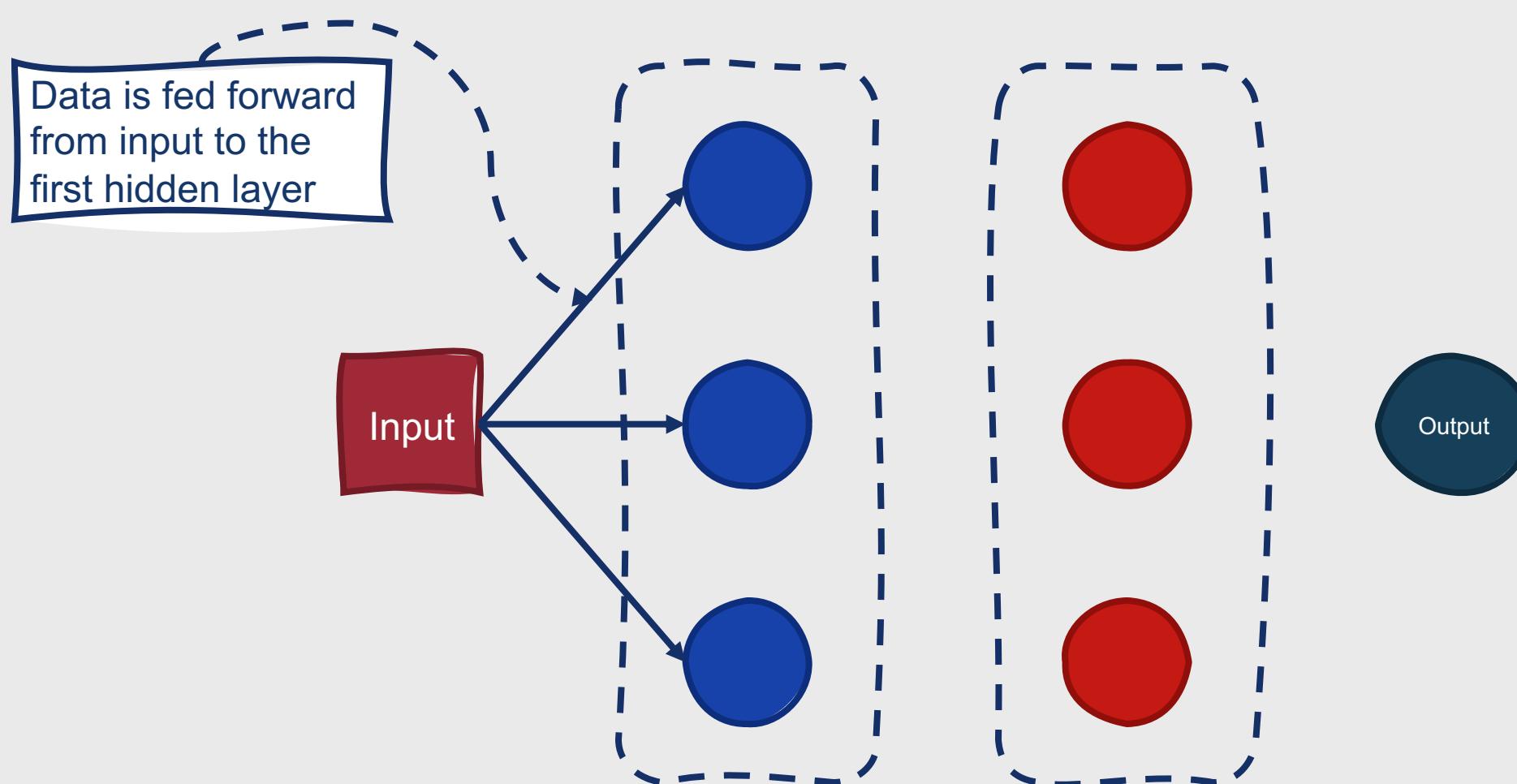
# Feedforward Neural Networks



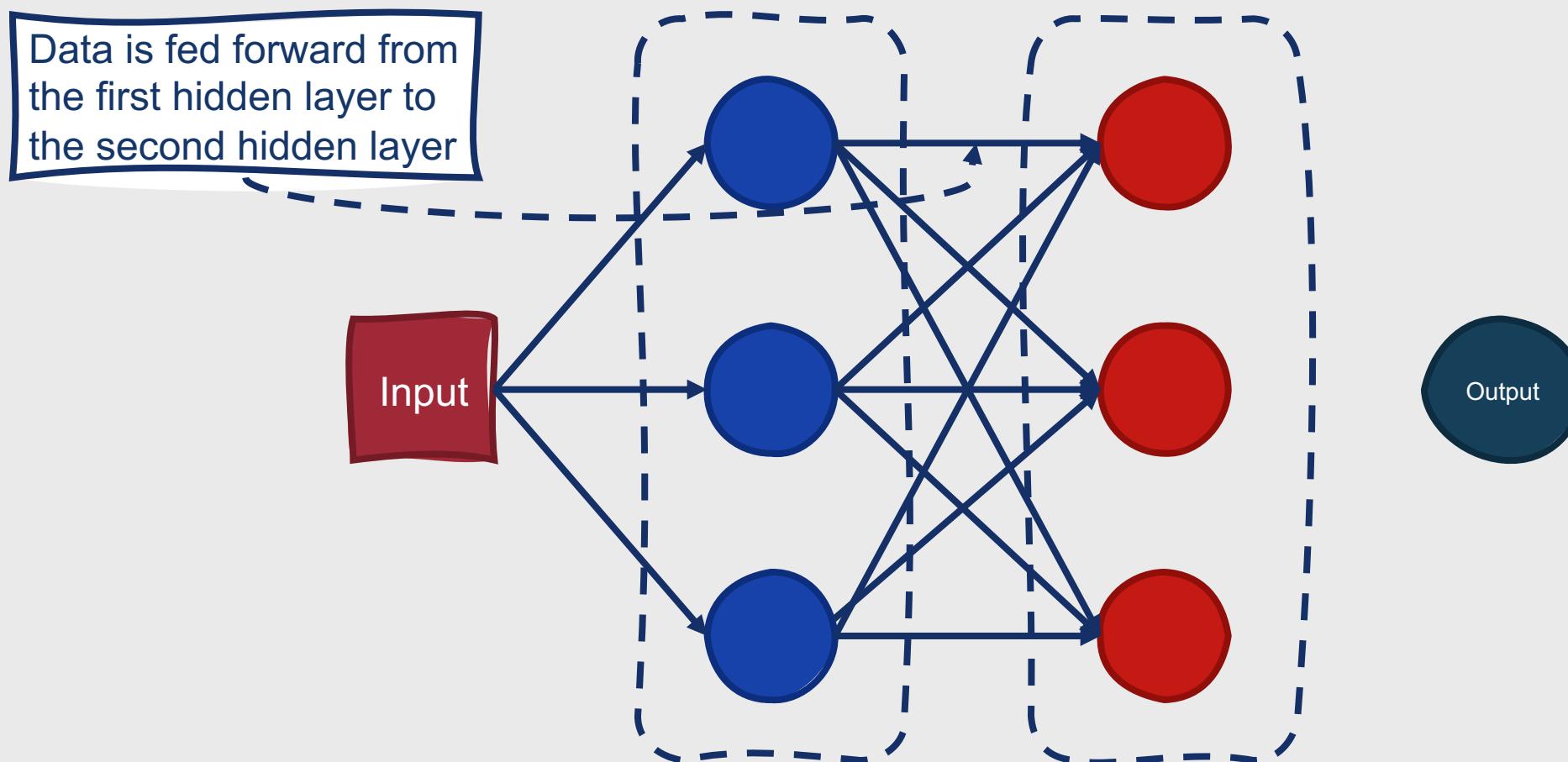
# Feedforward Neural Networks



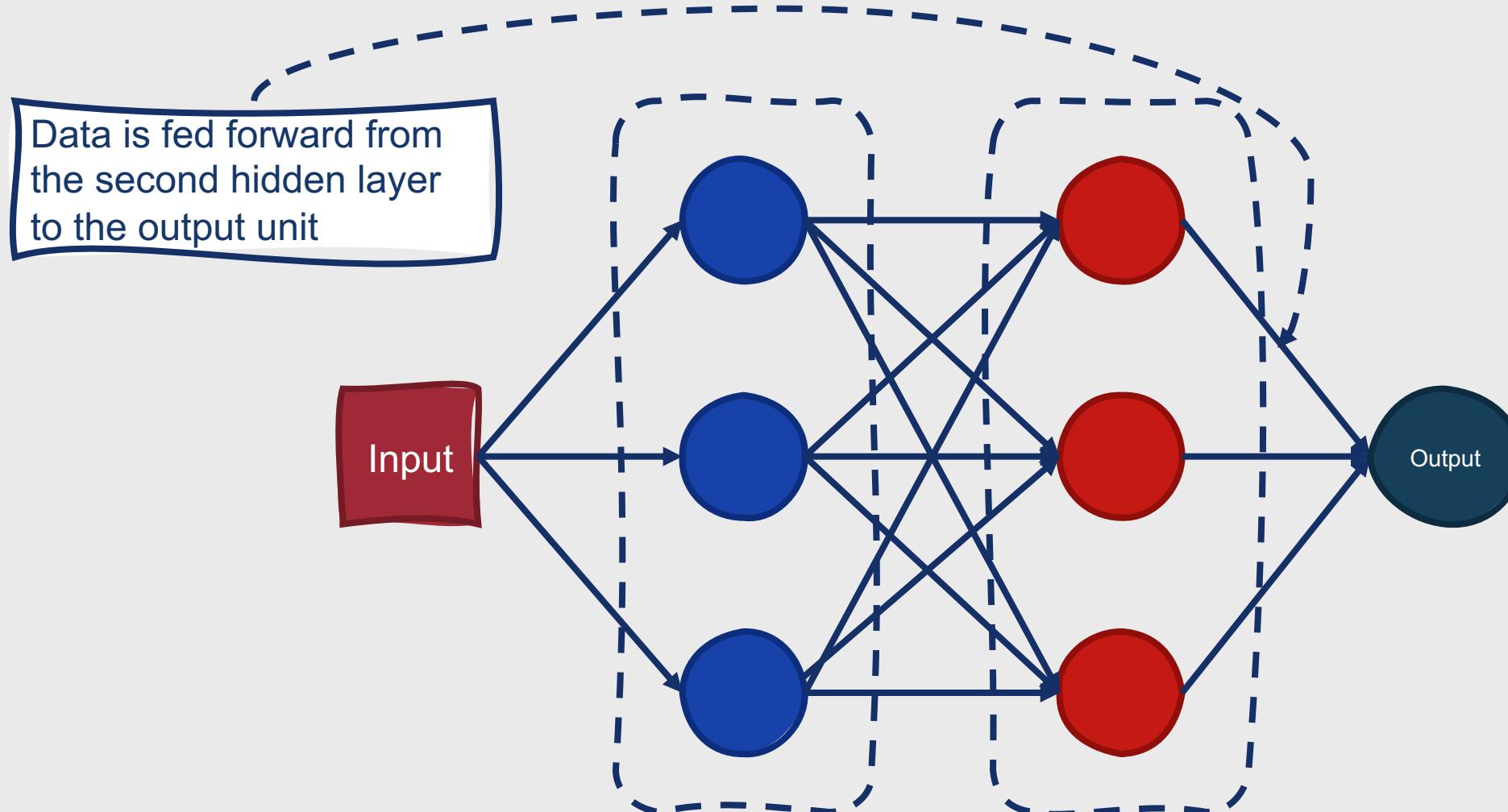
# Feedforward Neural Networks



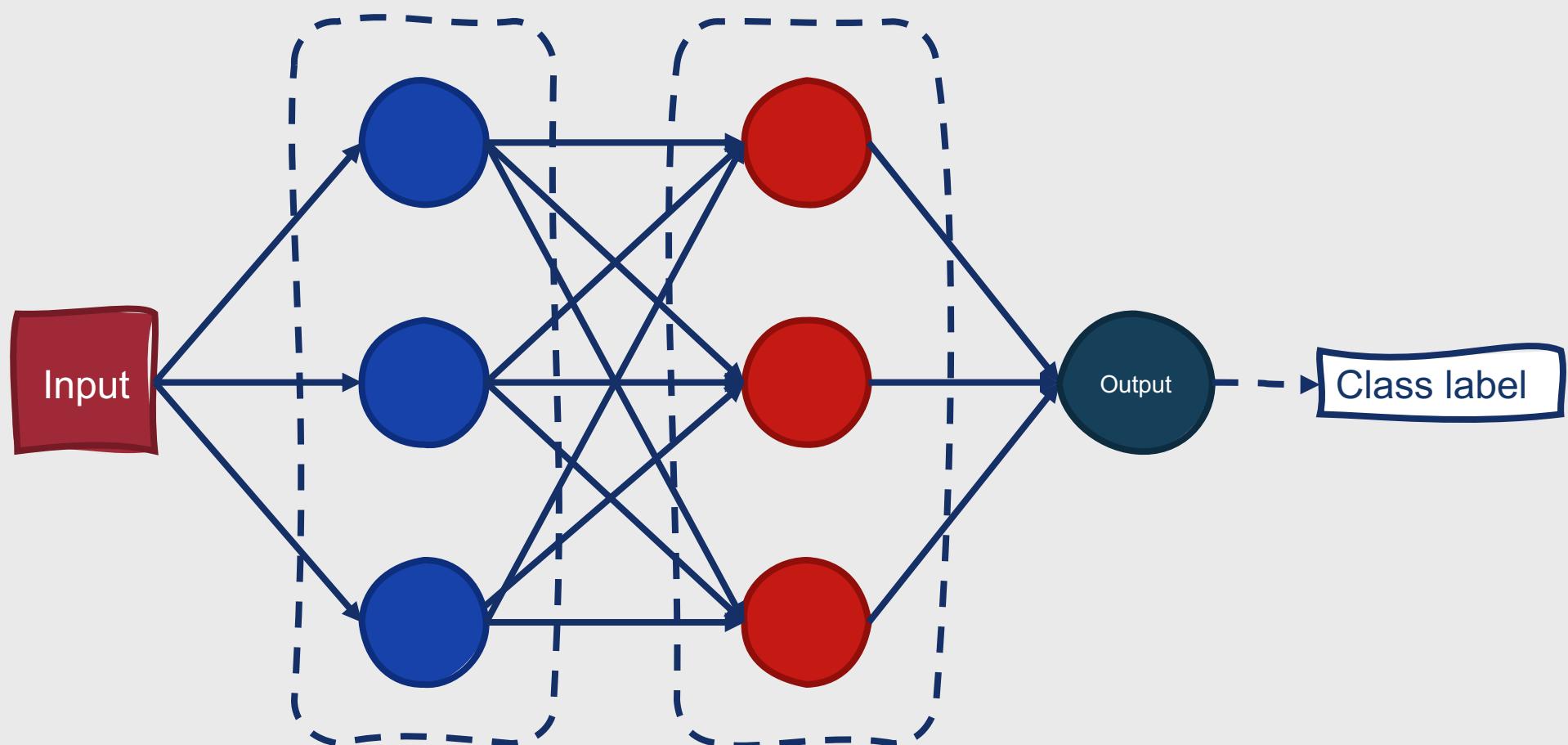
# Feedforward Neural Networks



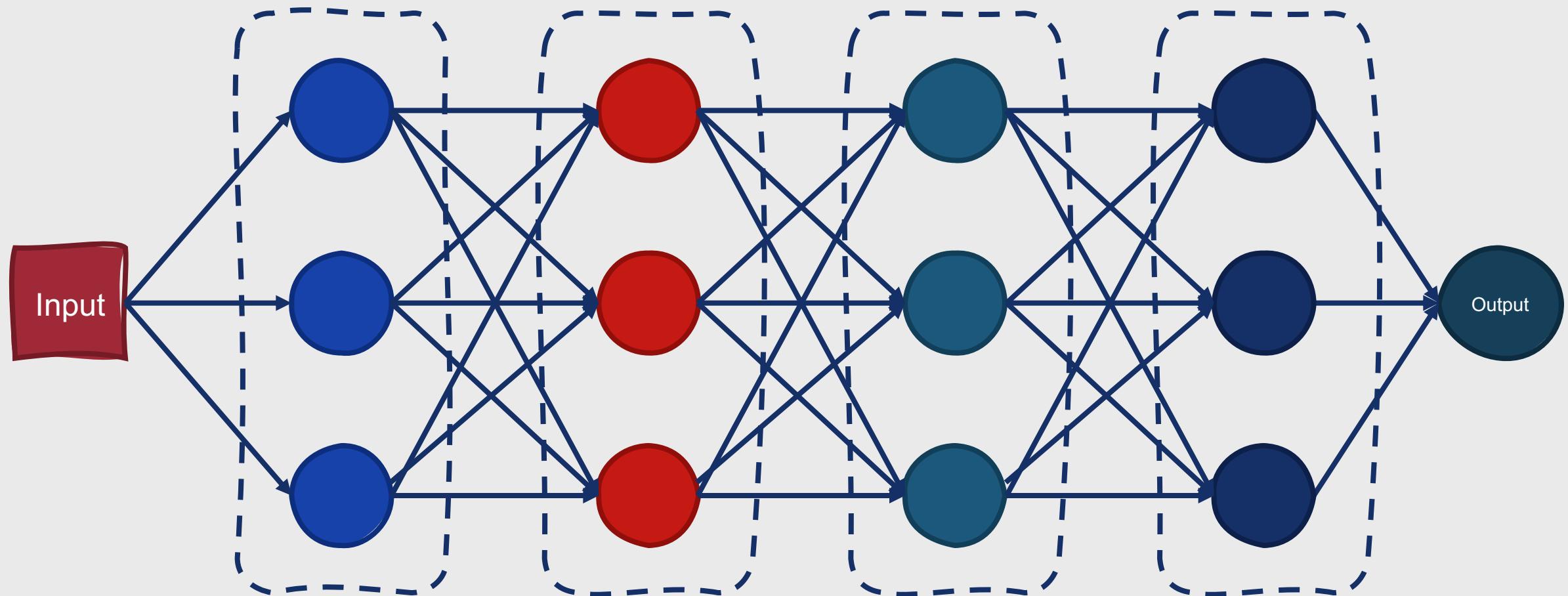
# Feedforward Neural Networks



# Feedforward Neural Networks



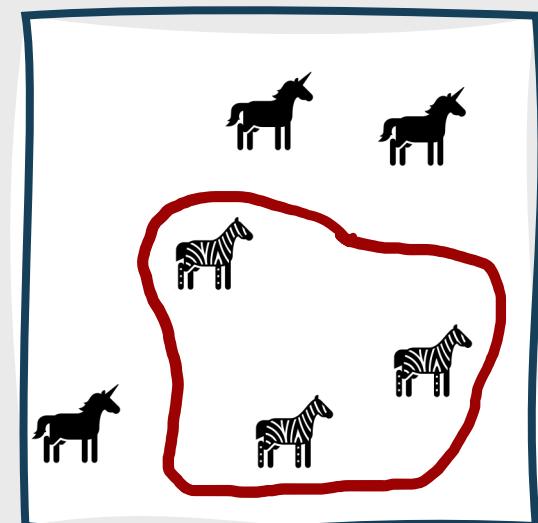
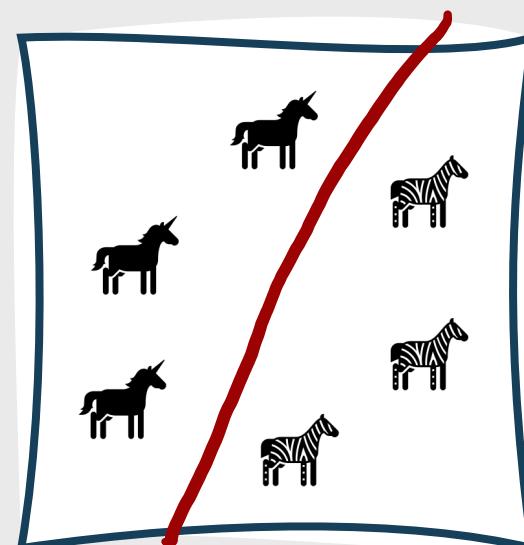
# How many layers is “deep?”





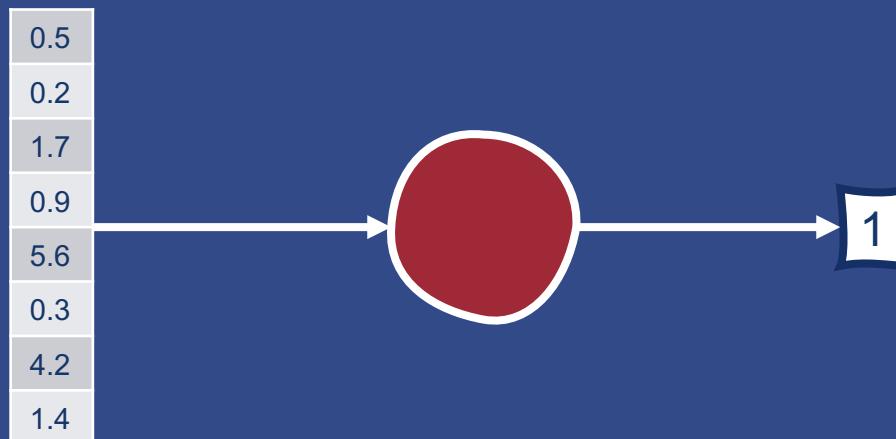
Neural networks tend to be more powerful than traditional classification algorithms.

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



# 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



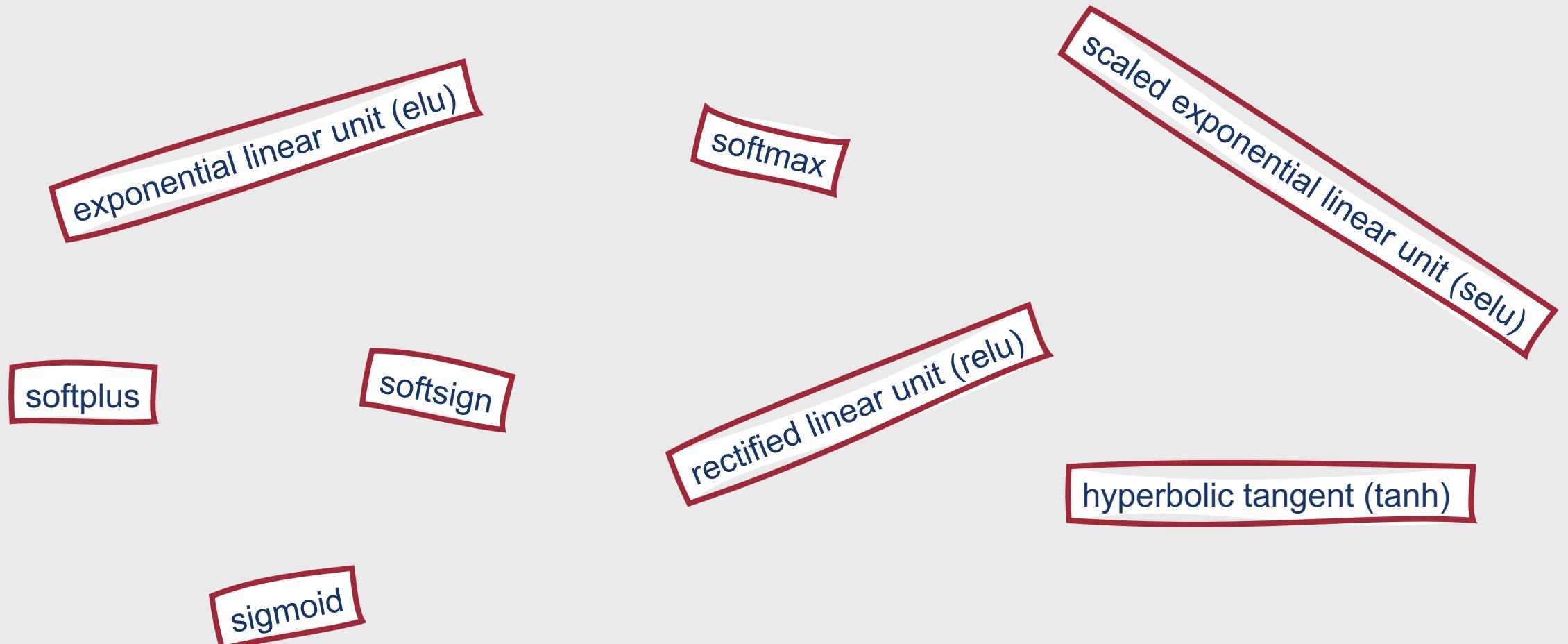
# Computational Units

- 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  $x_1, \dots, x_n$ , a unit has a set of corresponding weights  $w_1, \dots, w_n$  and a bias  $b$ , so the weighted sum  $z$  can be represented as:
  - $$z = b + \sum_i w_i x_i$$

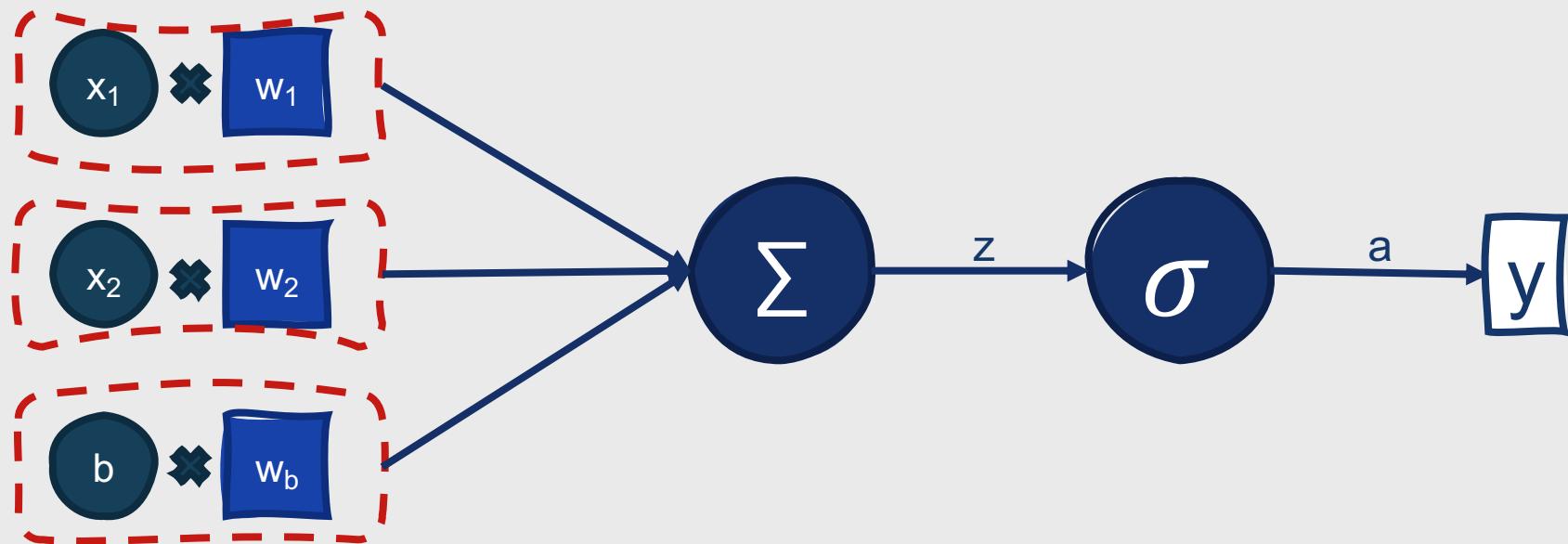
# Computational Units

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

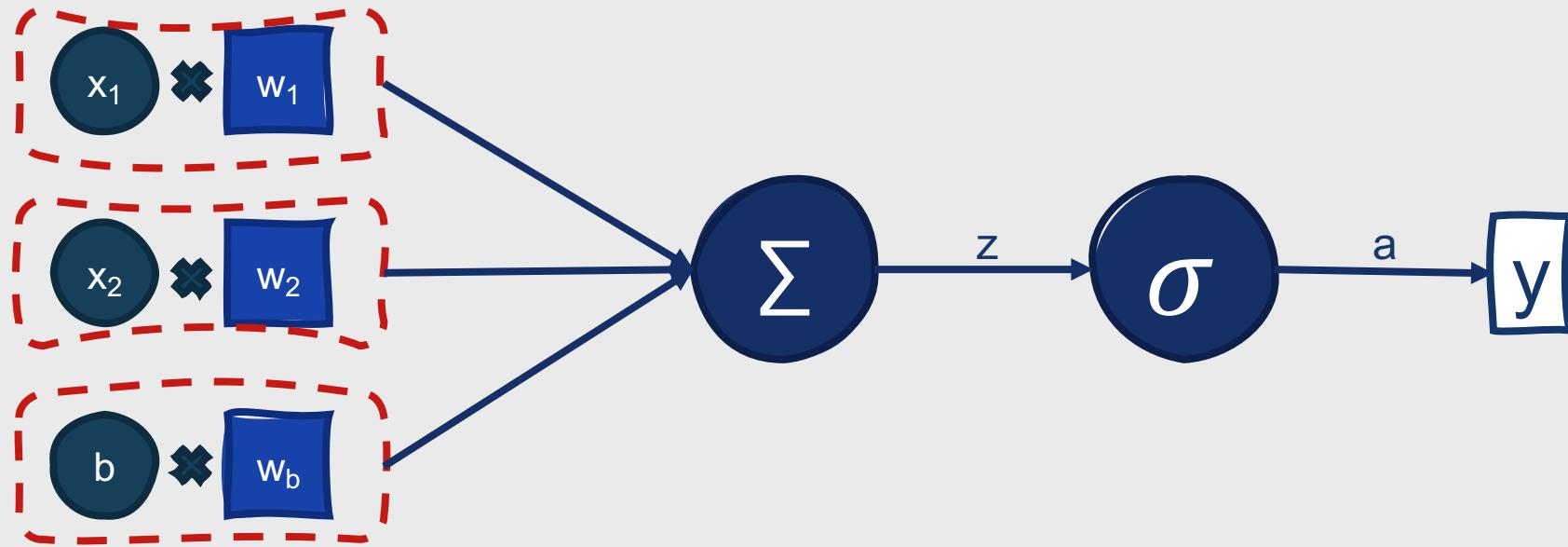
# There are many different activation functions!



# Computational Unit with Sigmoid Activation



# Example: Computational Unit with Sigmoid Activation

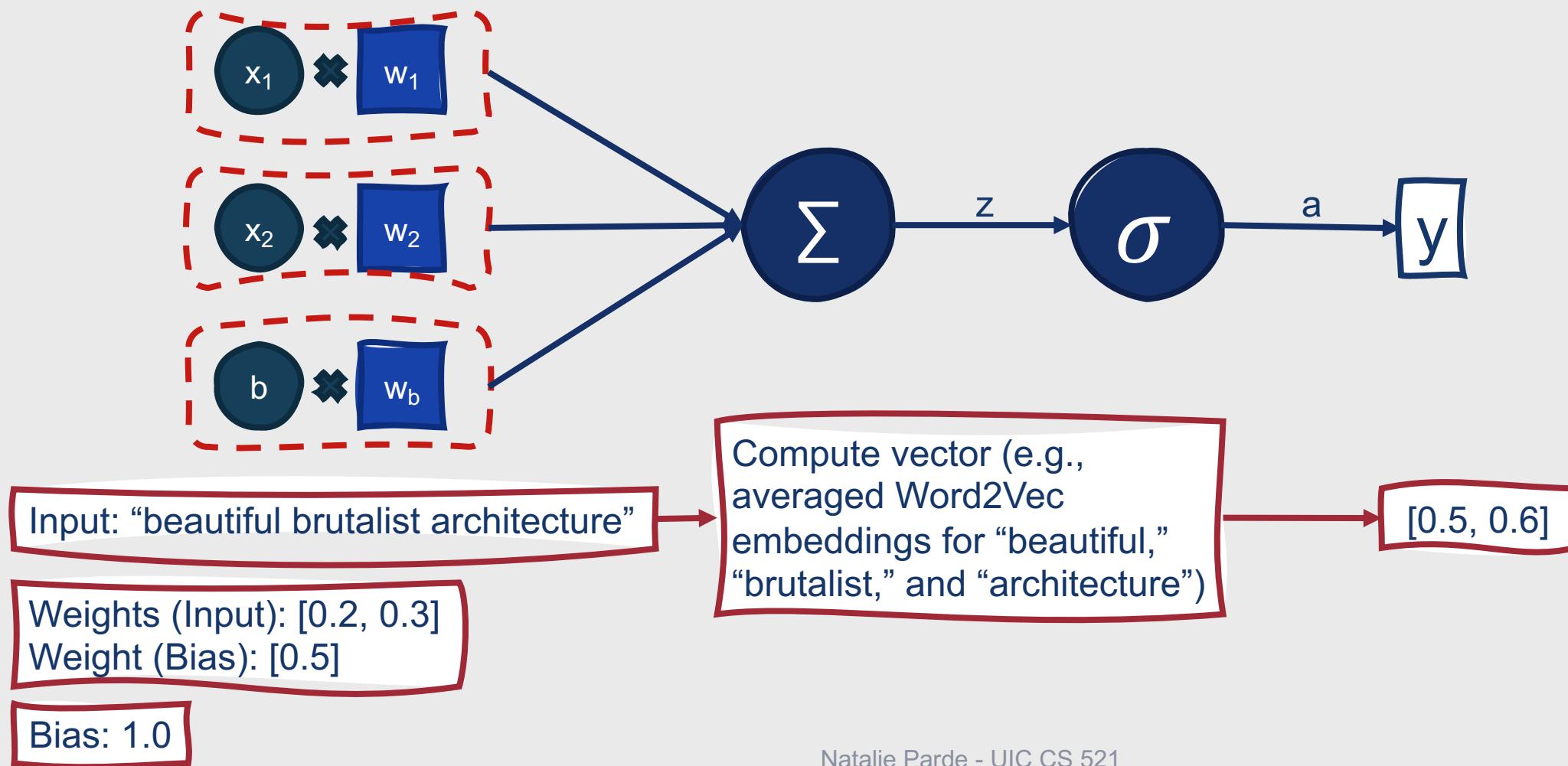


Input: “beautiful brutalist architecture”

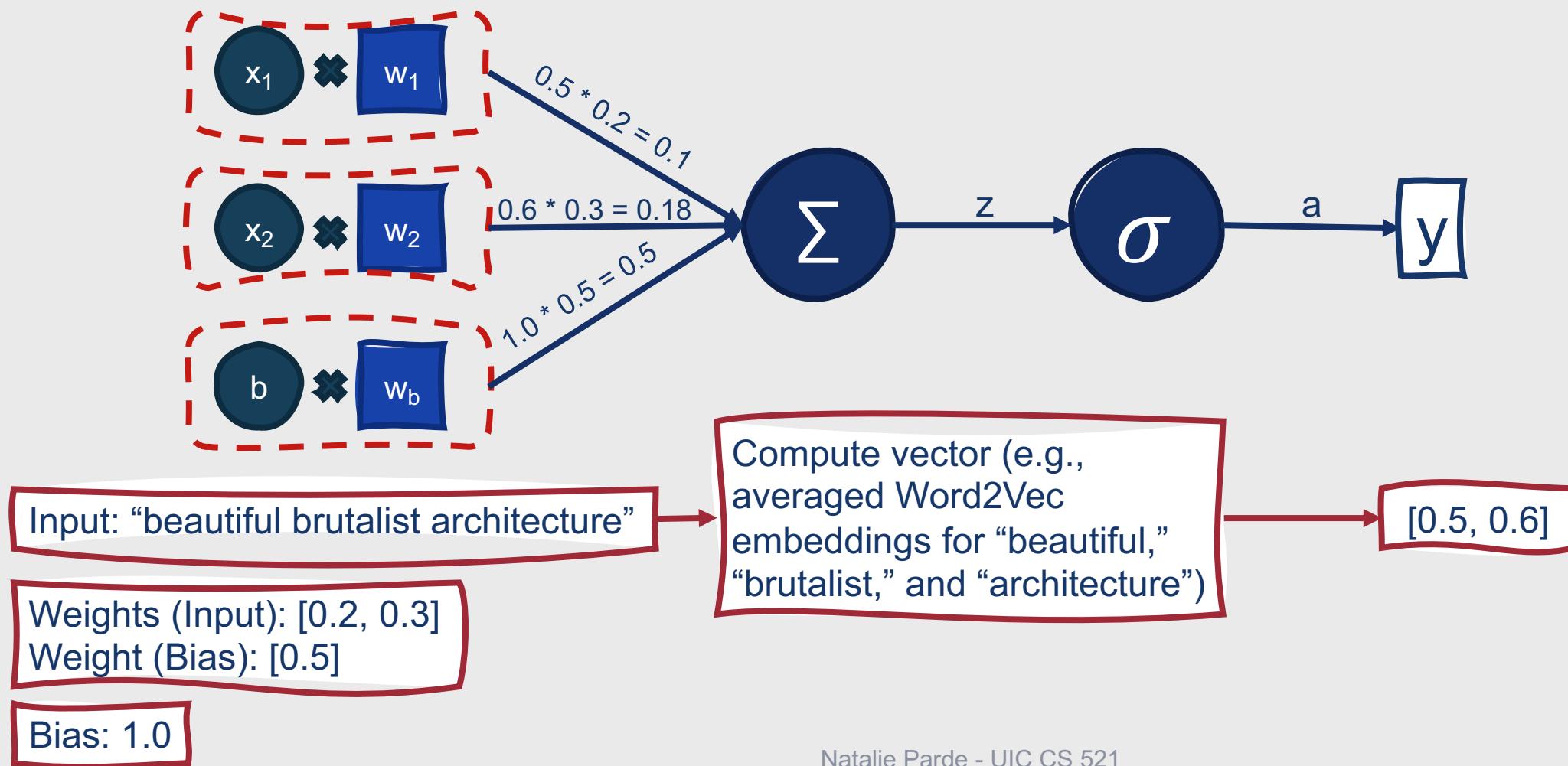
Weights (Input): [0.2, 0.3]  
Weight (Bias): [0.5]

Bias: 1.0

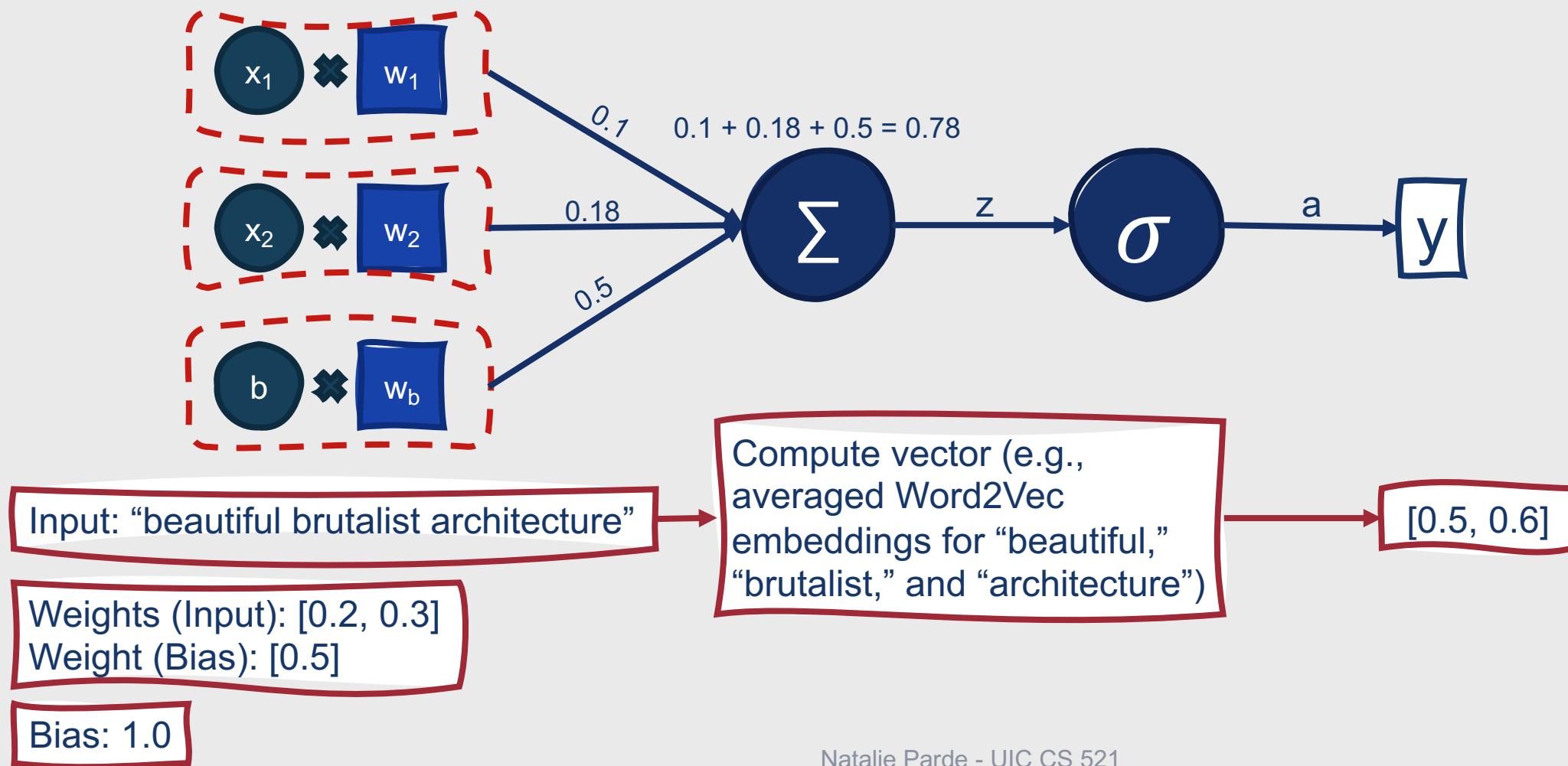
# Example: Computational Unit with Sigmoid Activation



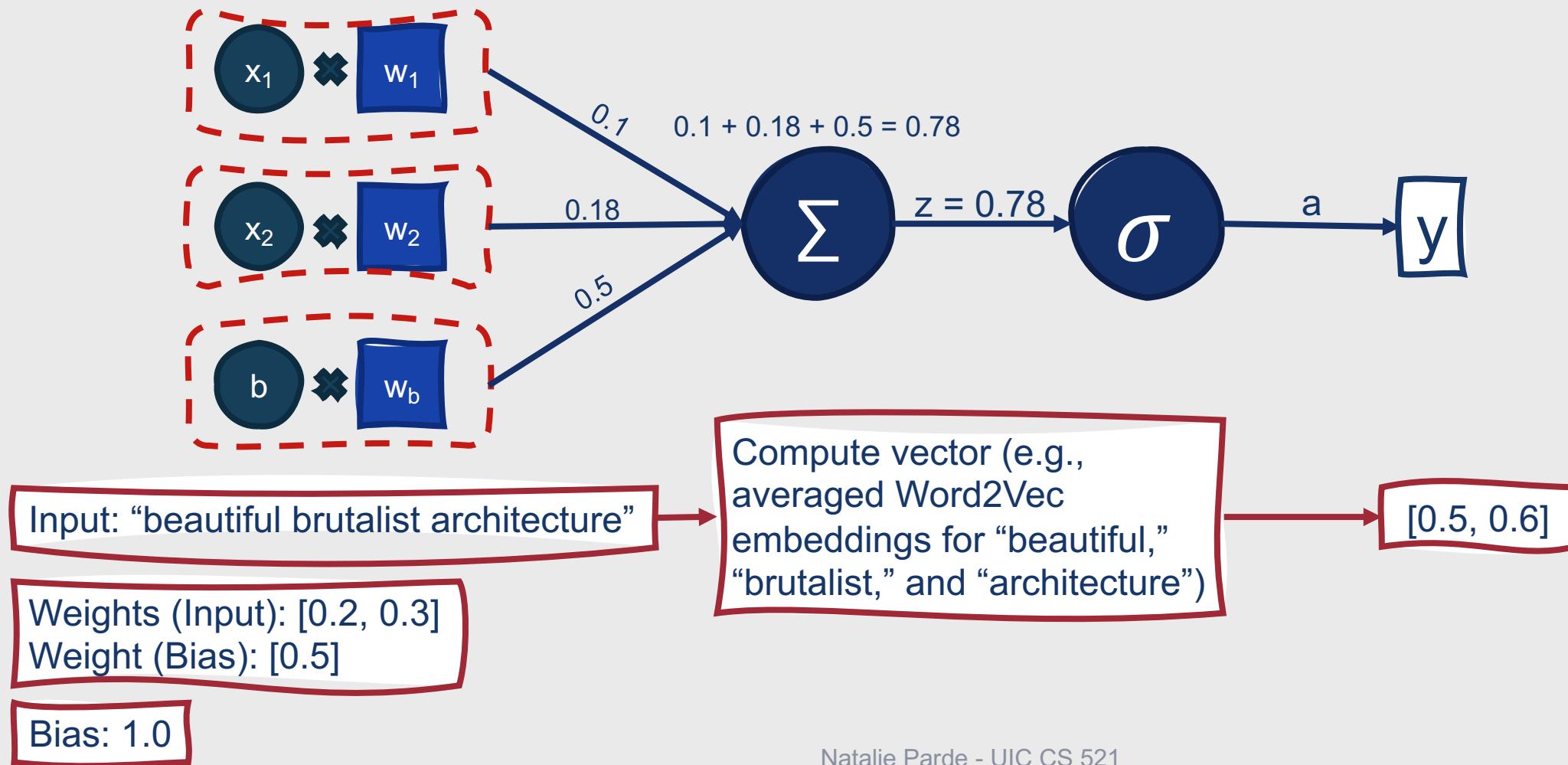
# Example: Computational Unit with Sigmoid Activation



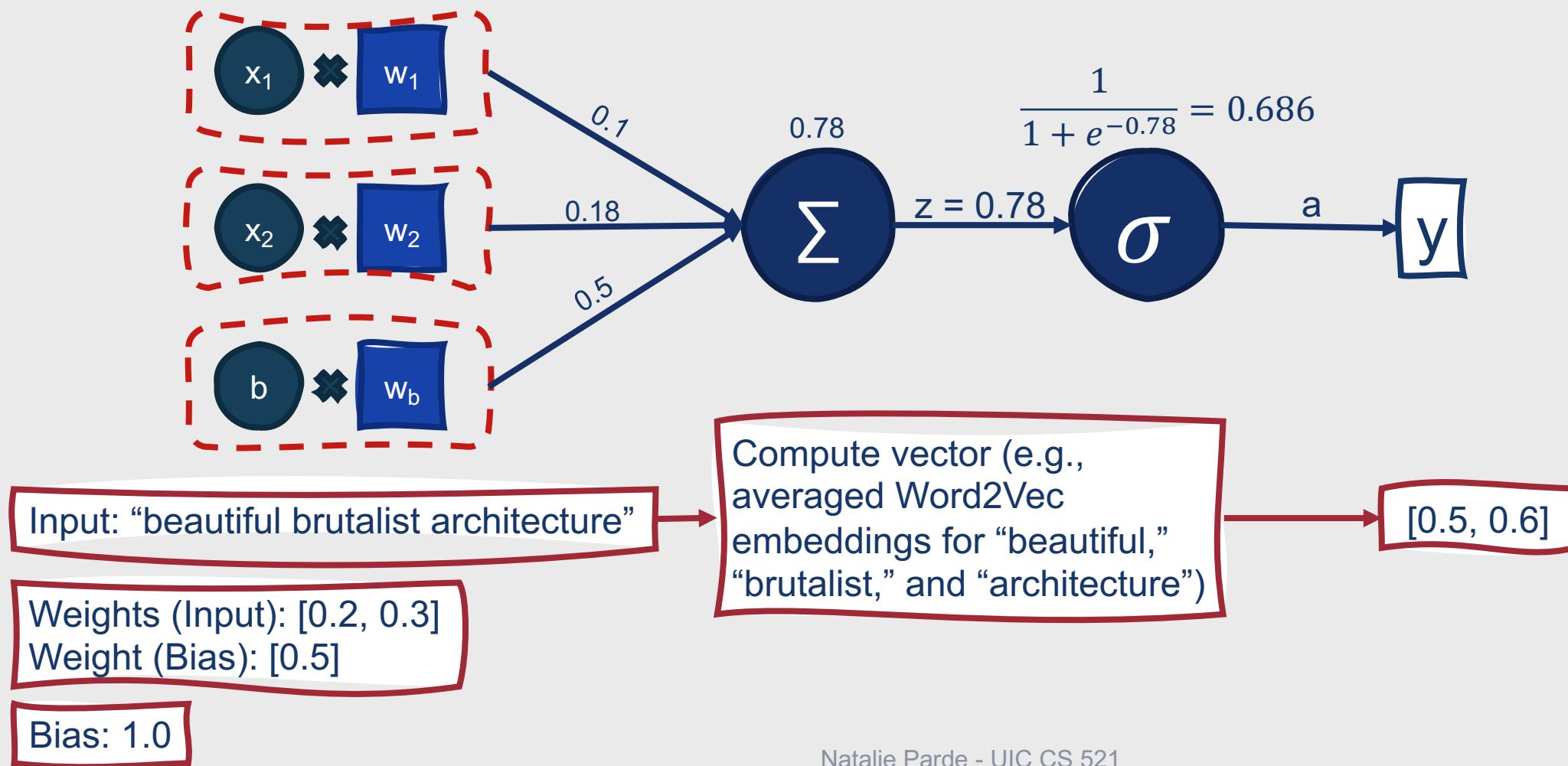
# Example: Computational Unit with Sigmoid Activation



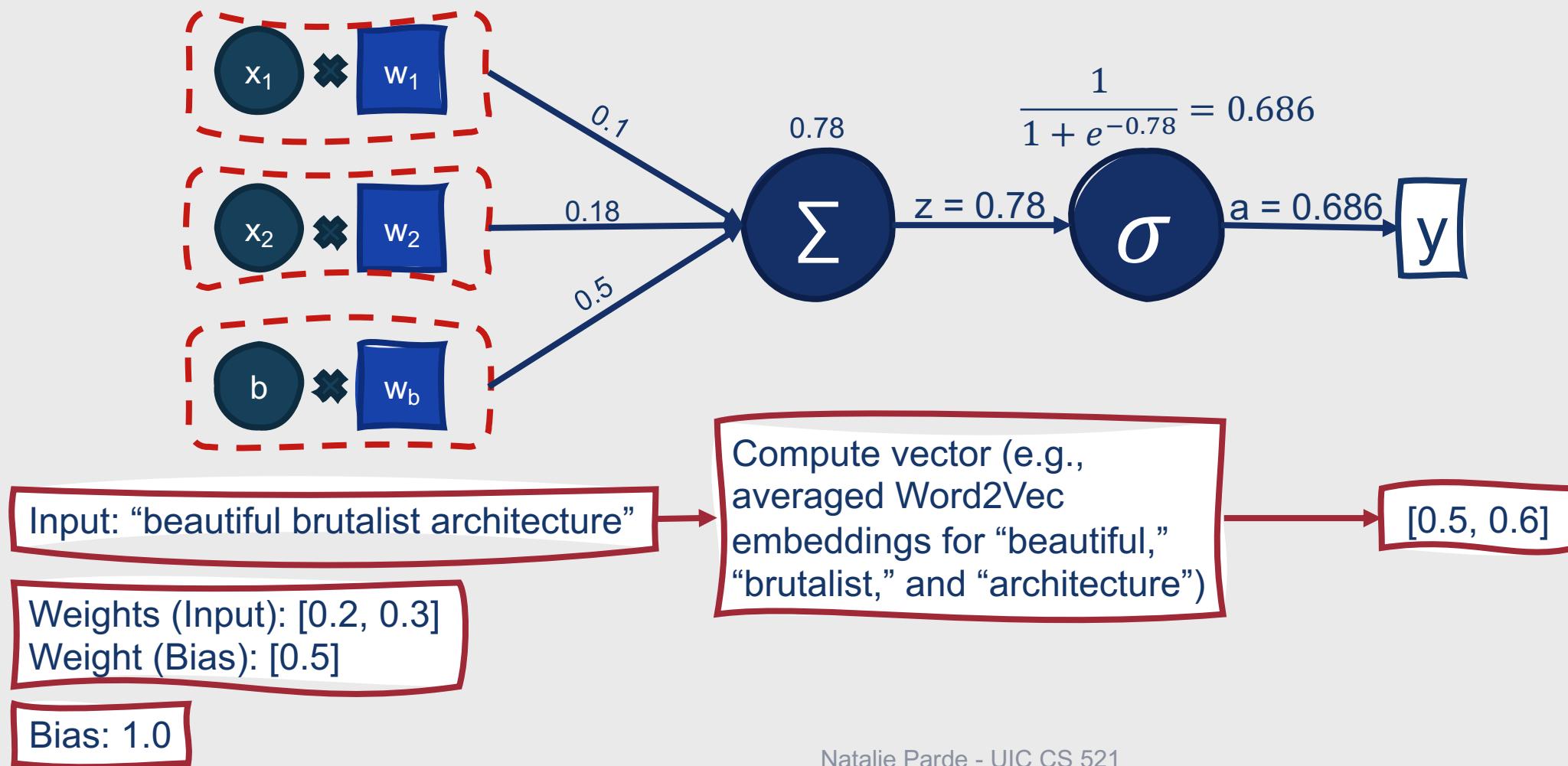
# Example: Computational Unit with Sigmoid Activation



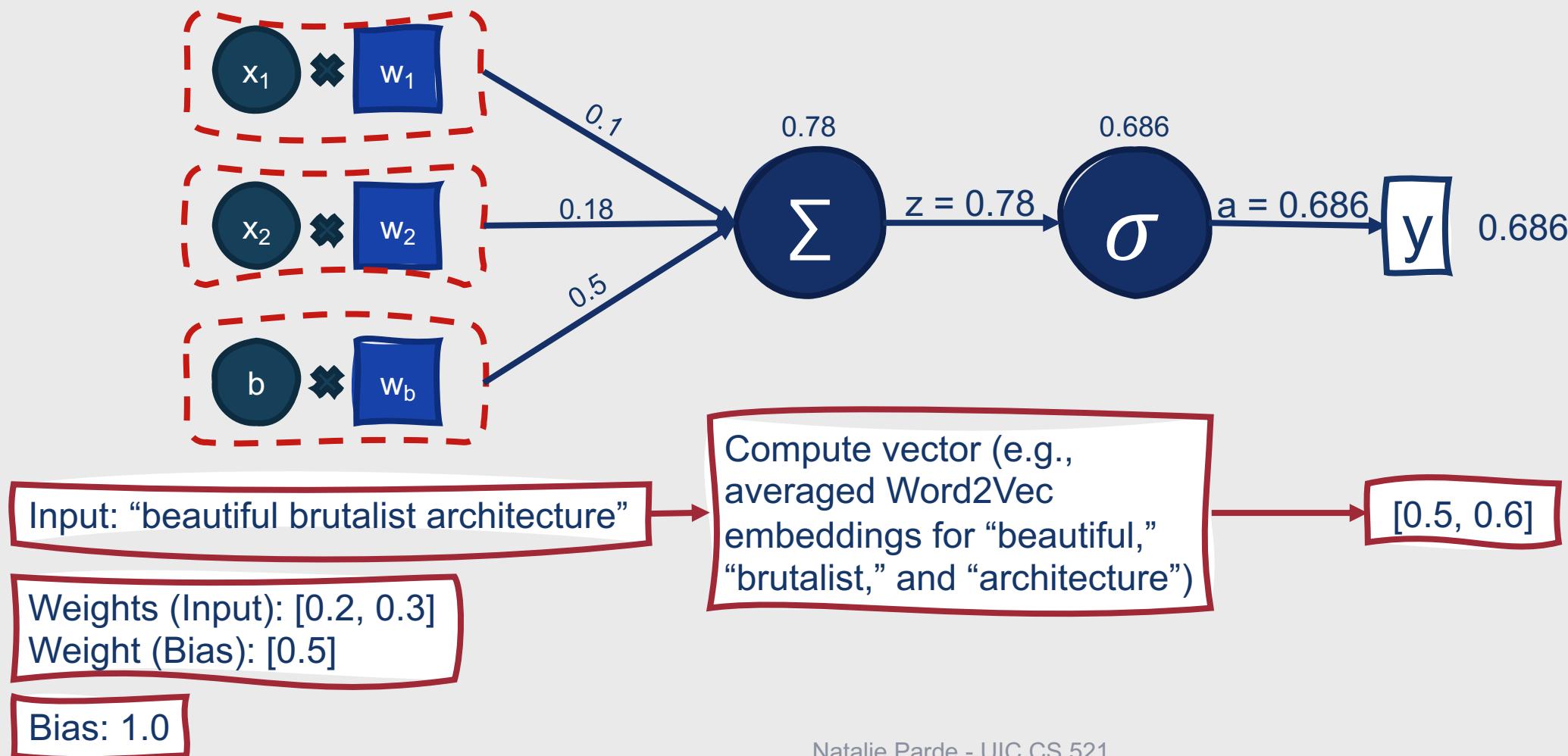
# Example: Computational Unit with Sigmoid Activation



# Example: Computational Unit with Sigmoid Activation



# Example: Computational Unit with Sigmoid Activation



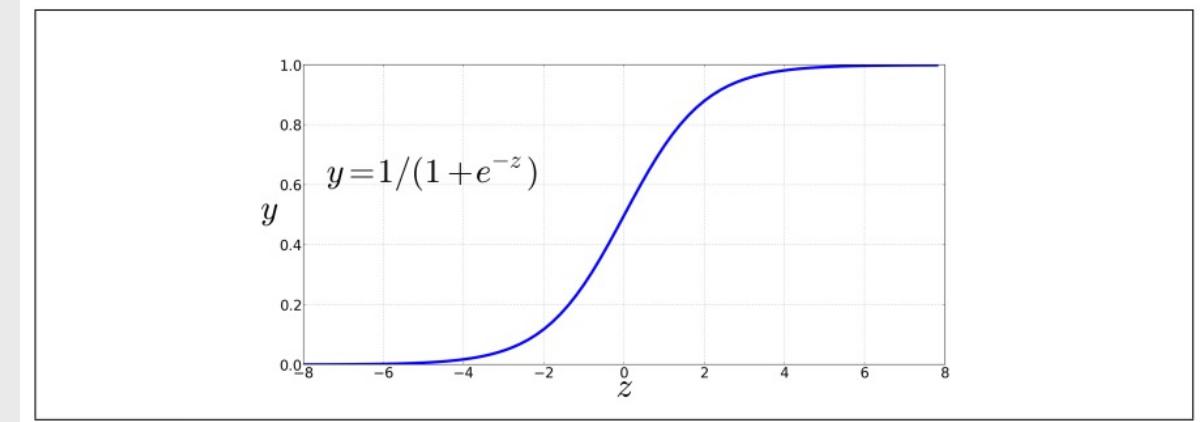
# Activation: tanh

- 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

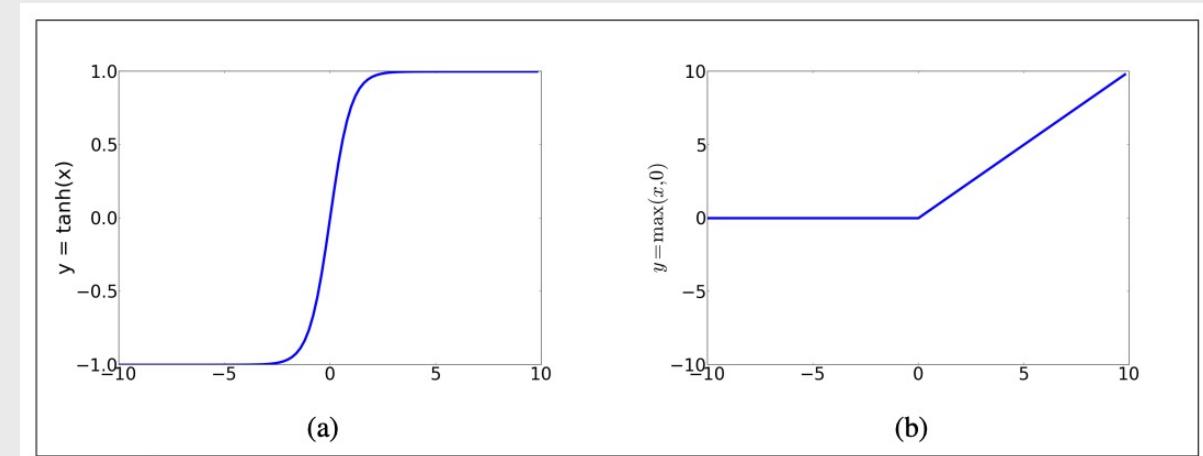
# Activation: ReLU

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

# Comparing sigmoid, tanh, and ReLU



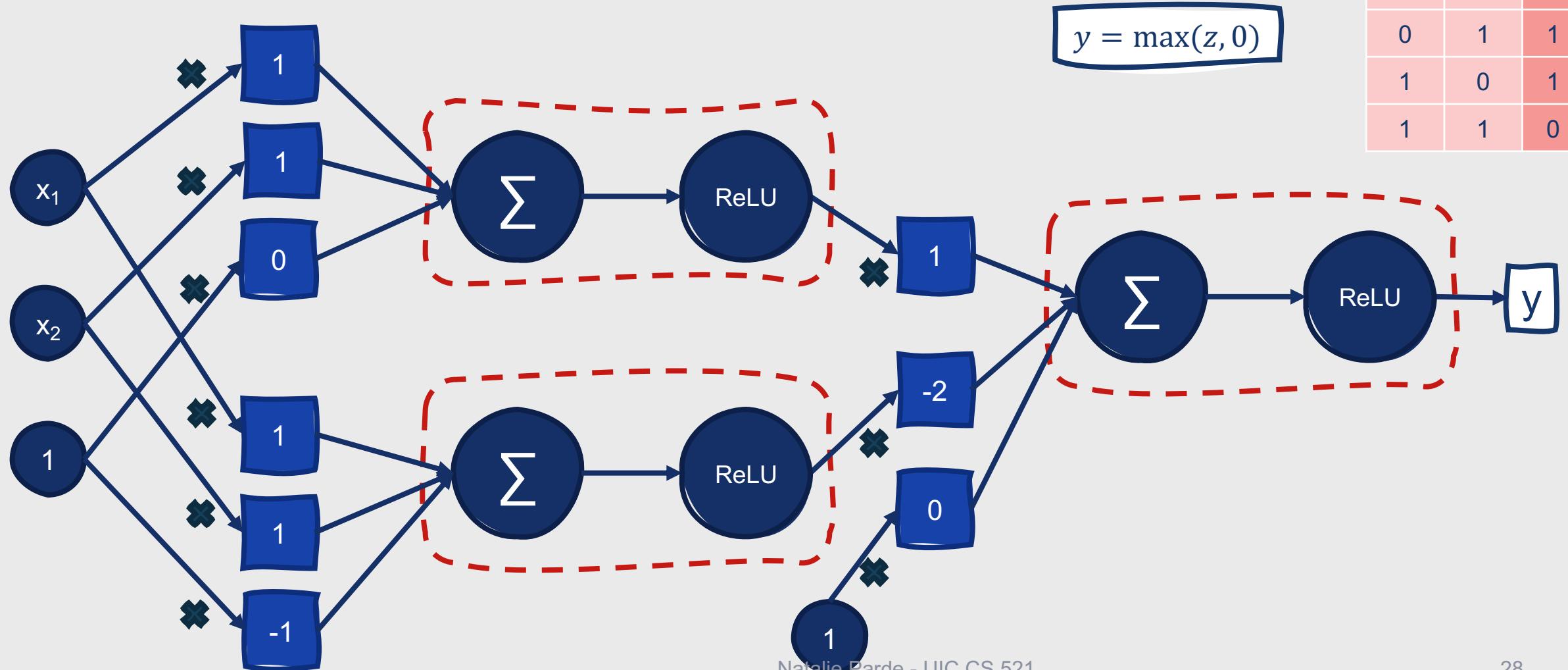
**Figure 7.1** The sigmoid function takes a real value and maps it to the range [0, 1]. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.



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

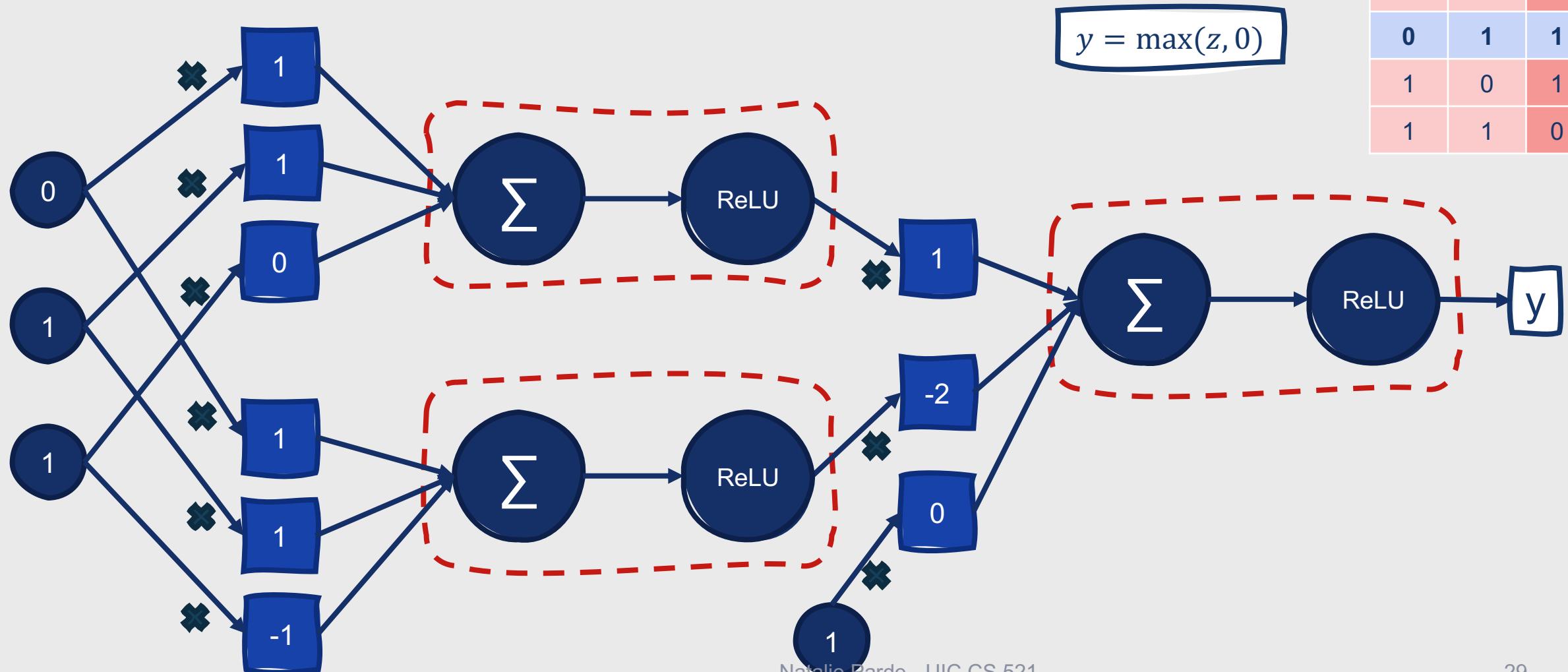
# Truth Table Examples: XOR

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



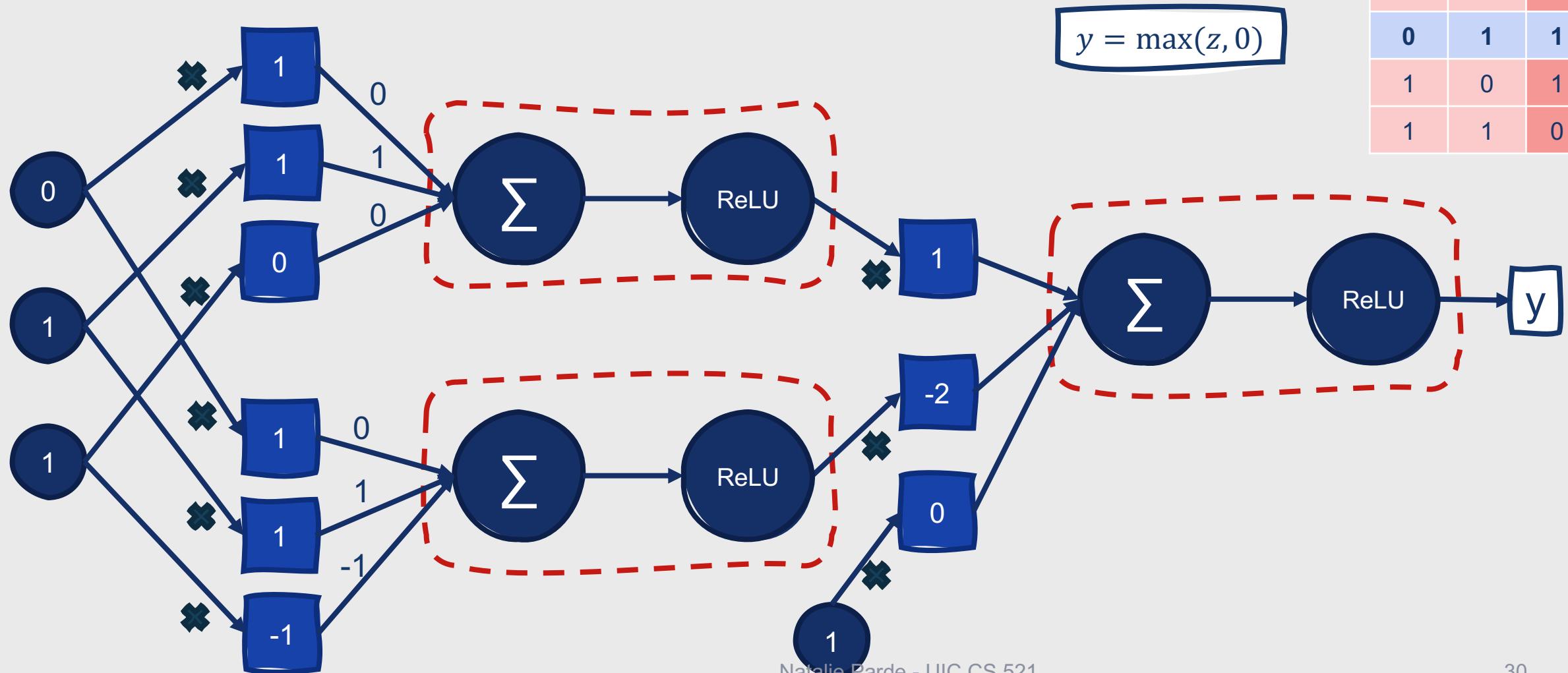
# Truth Table Examples: XOR

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



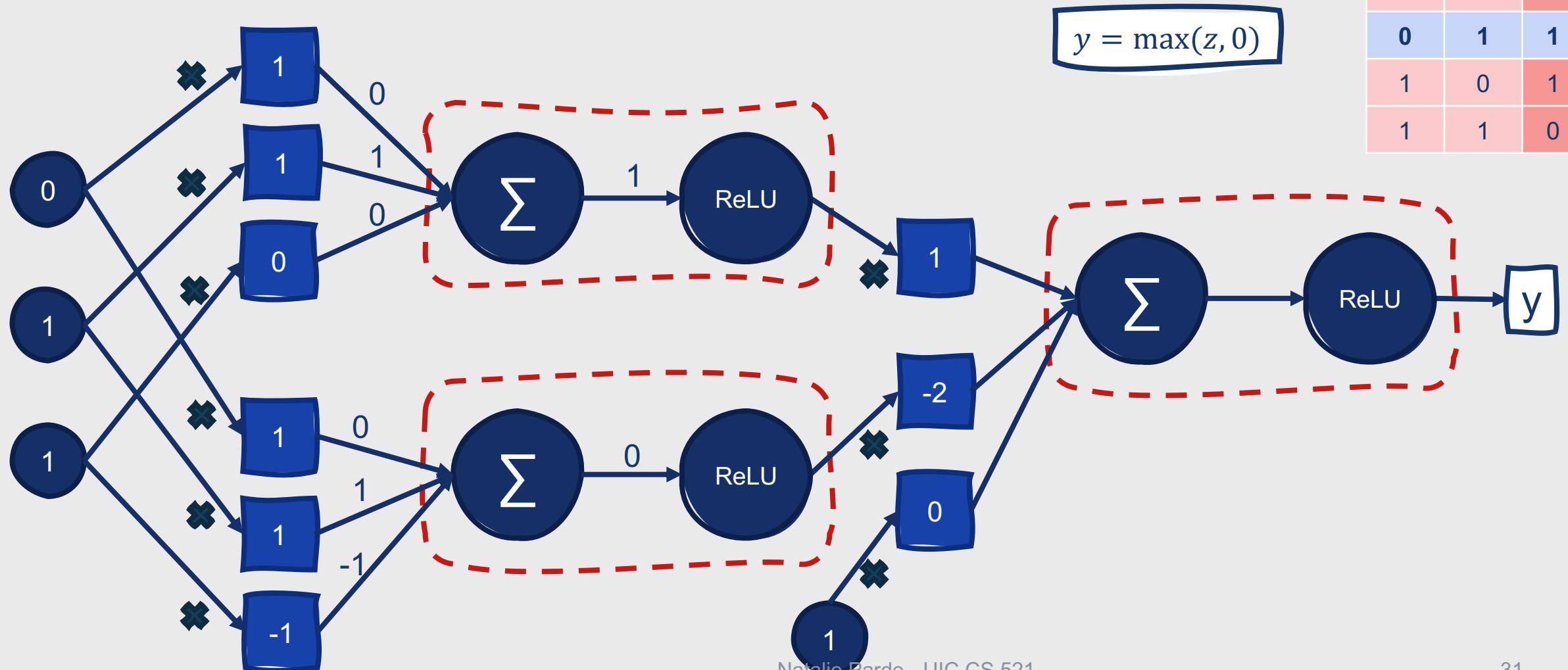
# Truth Table Examples: XOR

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



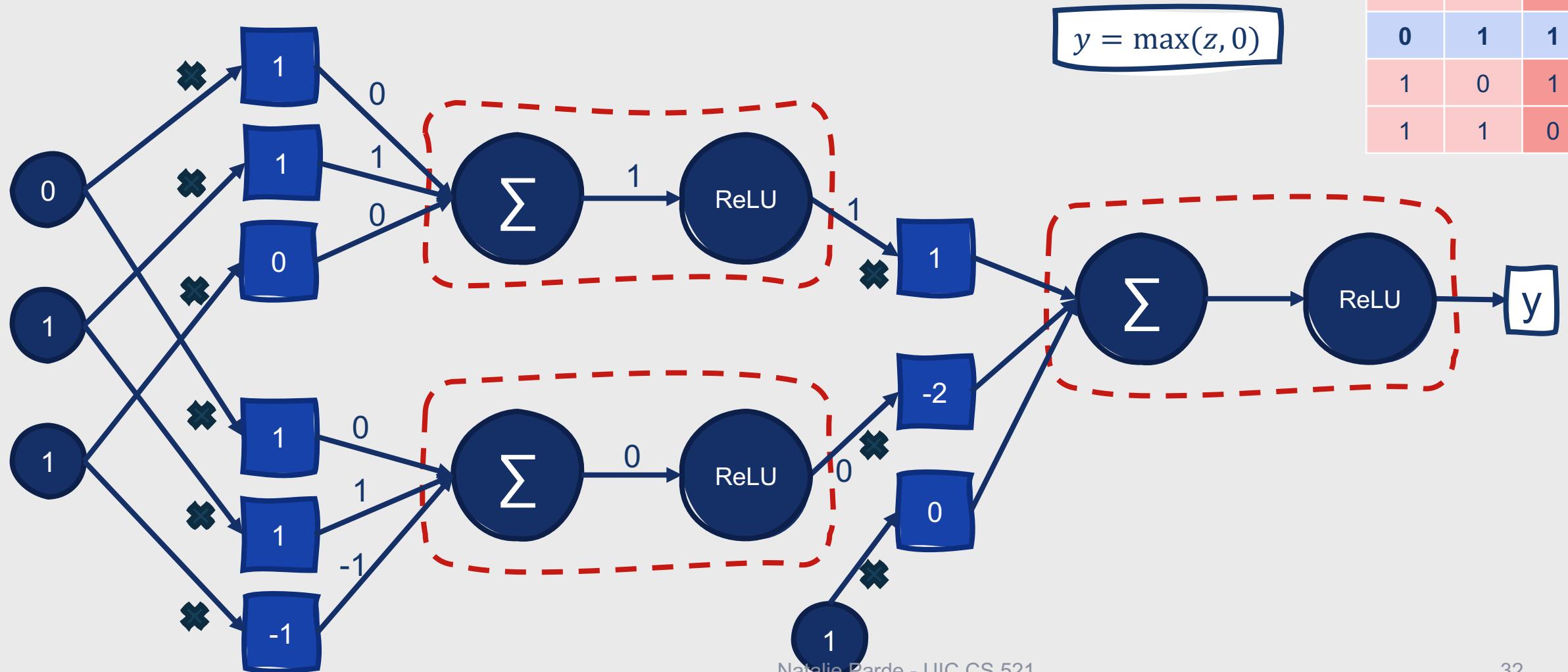
# Truth Table Examples: XOR

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



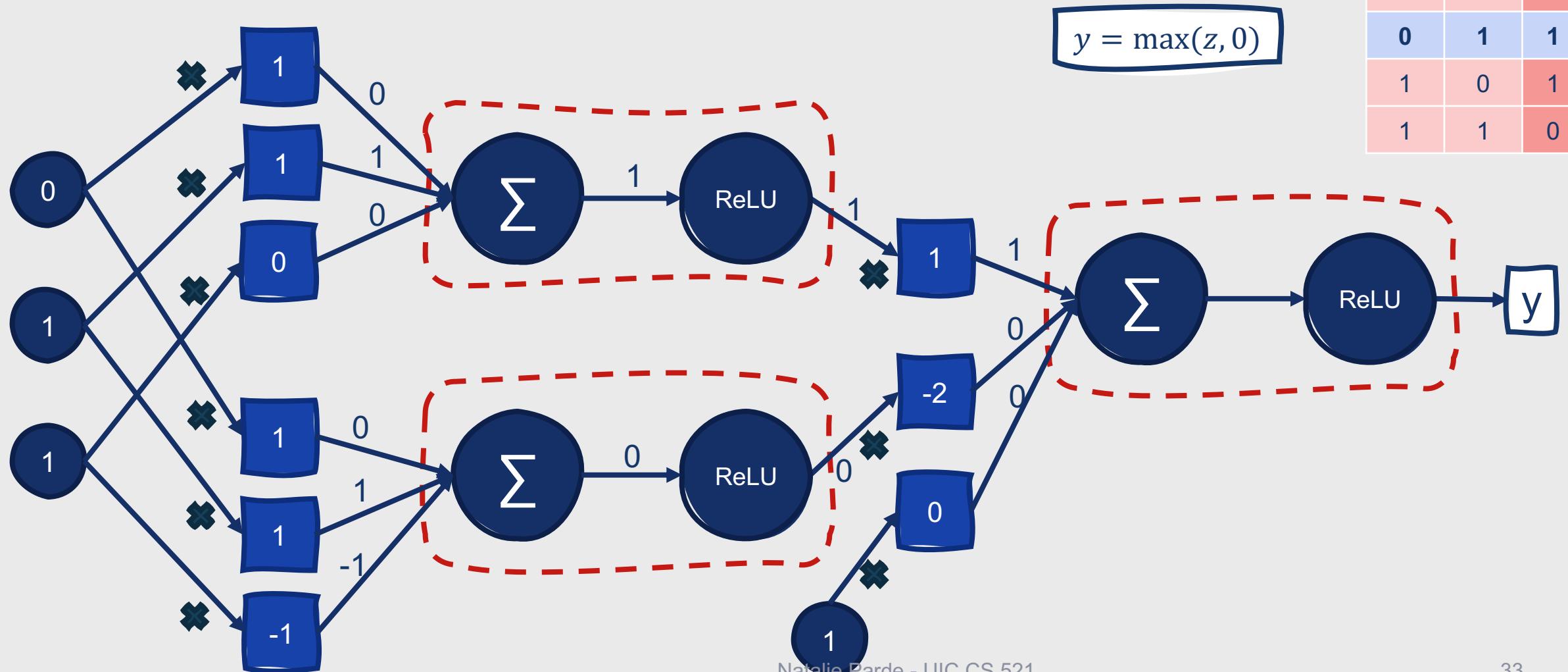
# Truth Table Examples: XOR

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



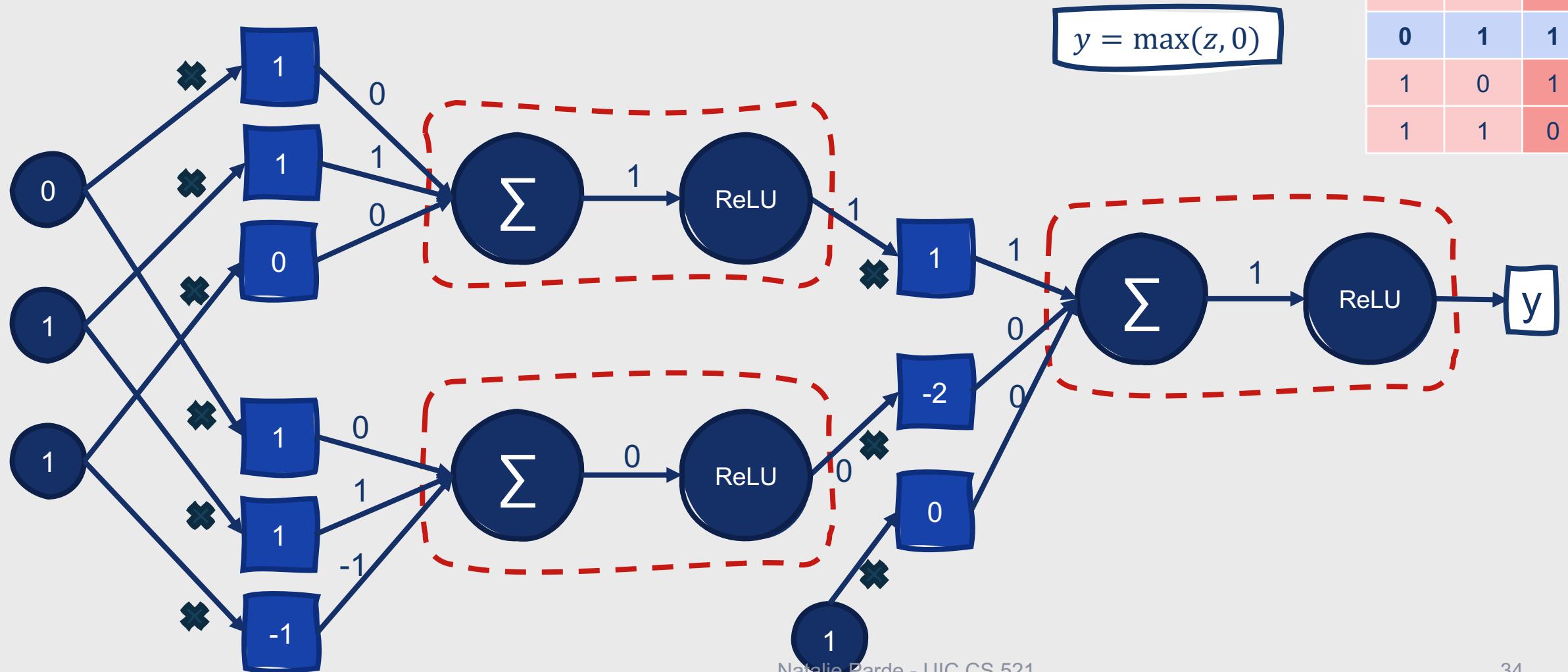
# Truth Table Examples: XOR

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



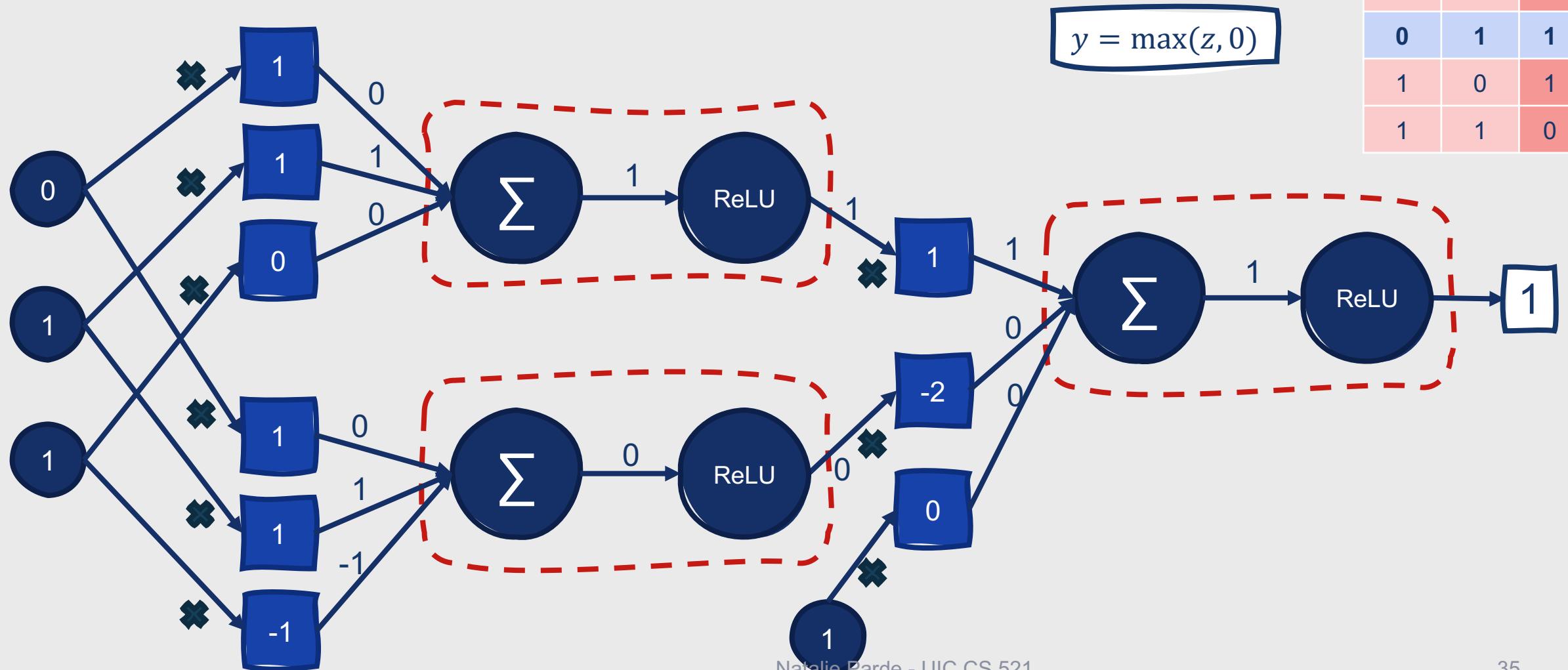
# Truth Table Examples: XOR

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



# Truth Table Examples: XOR

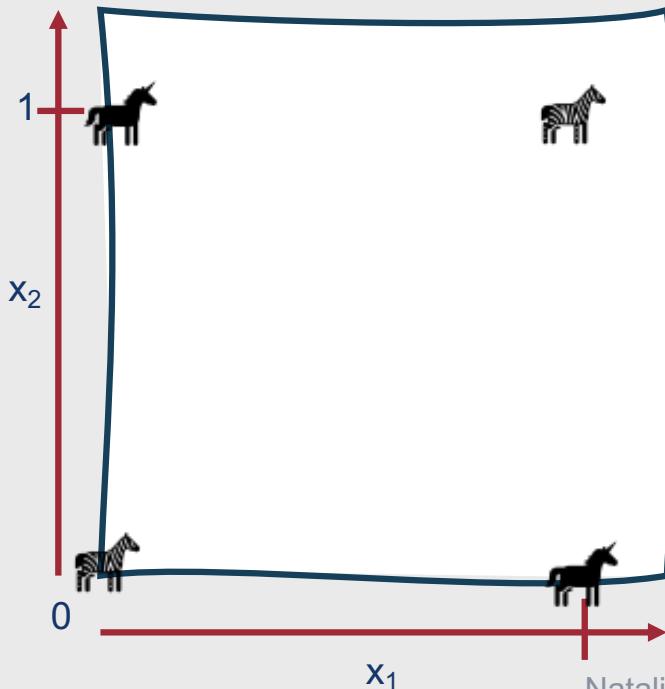
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



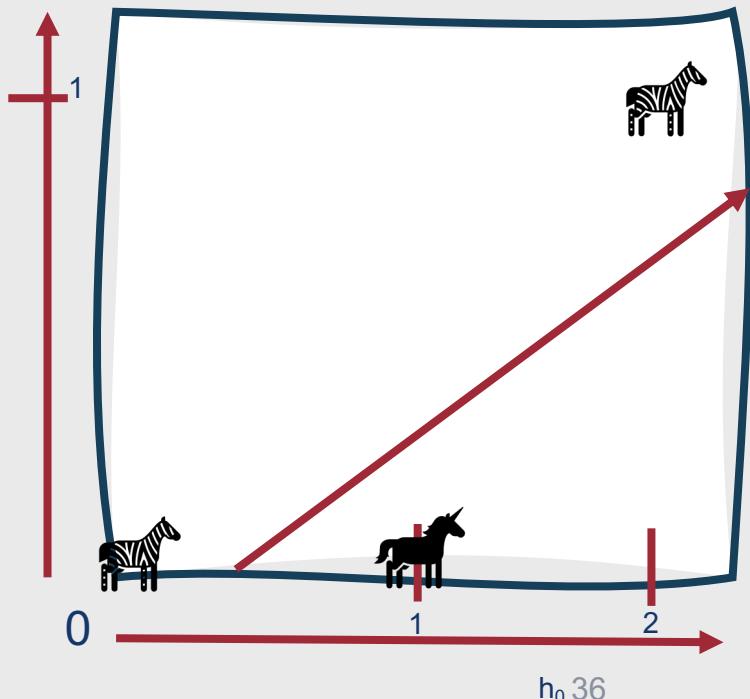
# Why does this work?

- 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



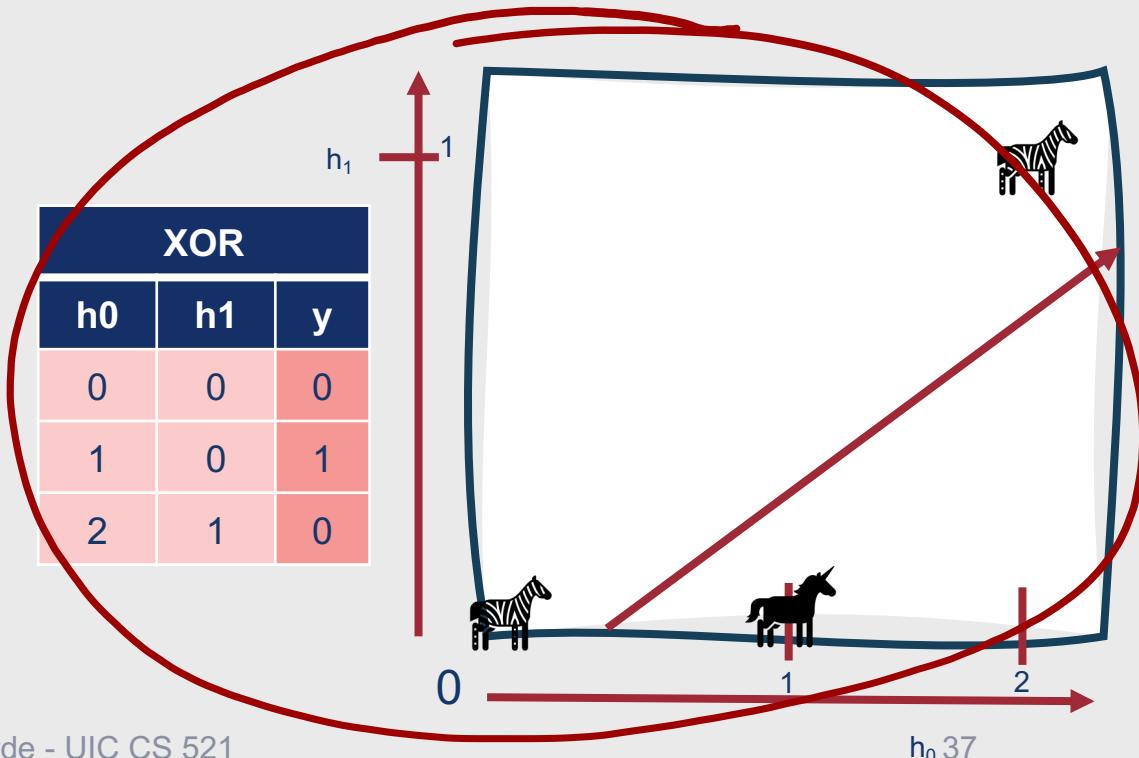
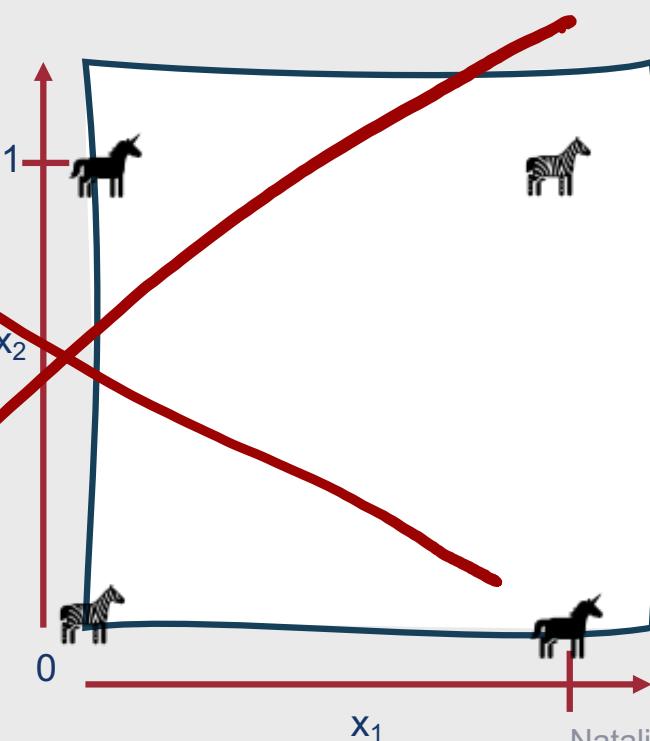
XOR		
h0	h1	y
0	0	0
1	0	1
2	1	0



# 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



# Feedforward Network

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

# 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  $1..n$ 
    - $z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}$
    - $a^{[i]} = g^{[i]}(z^{[i]})$
  - $y' = a^{[n]}$

# Does every layer use the same activation function?

- 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)

A background image featuring a complex network graph with numerous small white dots representing nodes and thin white lines representing connections, set against a gradient background that transitions from dark purple at the top to bright red at the bottom.

# How do we train neural networks?

---

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

A background image showing a complex network graph with numerous small white dots connected by thin white lines on a dark red gradient background.

# How do we train neural networks?

---

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

Cross-entropy loss

A background image showing a complex network graph with numerous small white dots representing nodes and many thin white lines representing connections between them, set against a dark red gradient background.

# How do we train neural networks?

---

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

Gradient descent

A background image showing a complex network graph with numerous white lines connecting small white dots on a red gradient background.

# 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

# Recall....

- For a “neural network” with just one weight layer containing a single computation unit + sigmoid activation (i.e., a logistic regression classifier), we can compute the gradient of our loss function by just taking its derivative:
  - $\frac{\partial L_{CE}(w,b)}{\partial w_j} = (\hat{y} - y)x_j = (\sigma(w \cdot x + b) - y)x_j$

# Recall....

- For a “neural network” with just one weight layer containing a single computation unit + sigmoid activation (i.e., a logistic regression classifier), we can compute the gradient of our loss function by just taking its derivative:

$$\bullet \frac{\partial L_{CE}(w,b)}{\partial w_j} = (\hat{y} - y)x_j = (\sigma(w \cdot x + b) - y)x_j$$

Difference between true and estimated  $y$

Corresponding input observation



**However, we  
can't do that  
with a neural  
network that has  
multiple weight  
layers (“hidden”  
layers).**

- Why?

- 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)

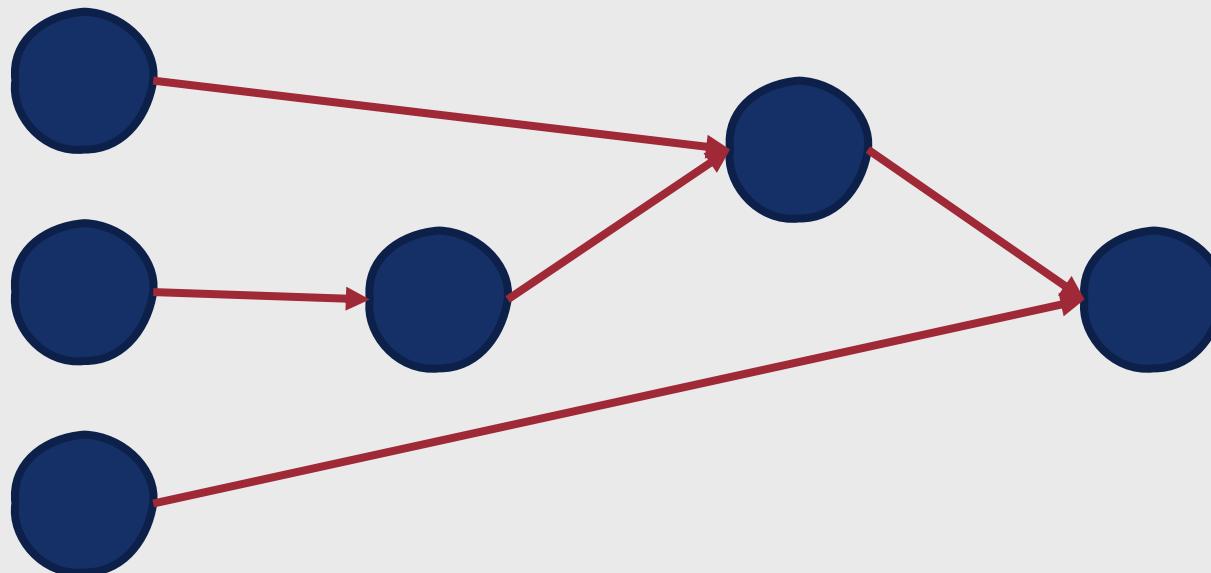
We do this  
using  
backward  
differentiation.

- Usually referred to as **backpropagation** (“**backprop**” for short) in the context of neural networks
- Frames neural networks as **computation graphs**

# What are computation graphs?



- Representations of interconnected mathematical operations
- **Nodes** = Operations
- **Directed edges** = connections between output/input of nodes



There are two different ways that we can pass information through our neural network computation graphs.

- **Forward pass**

- Apply operations in the direction of the arrows
- Pass the output of one computation as the input to the next

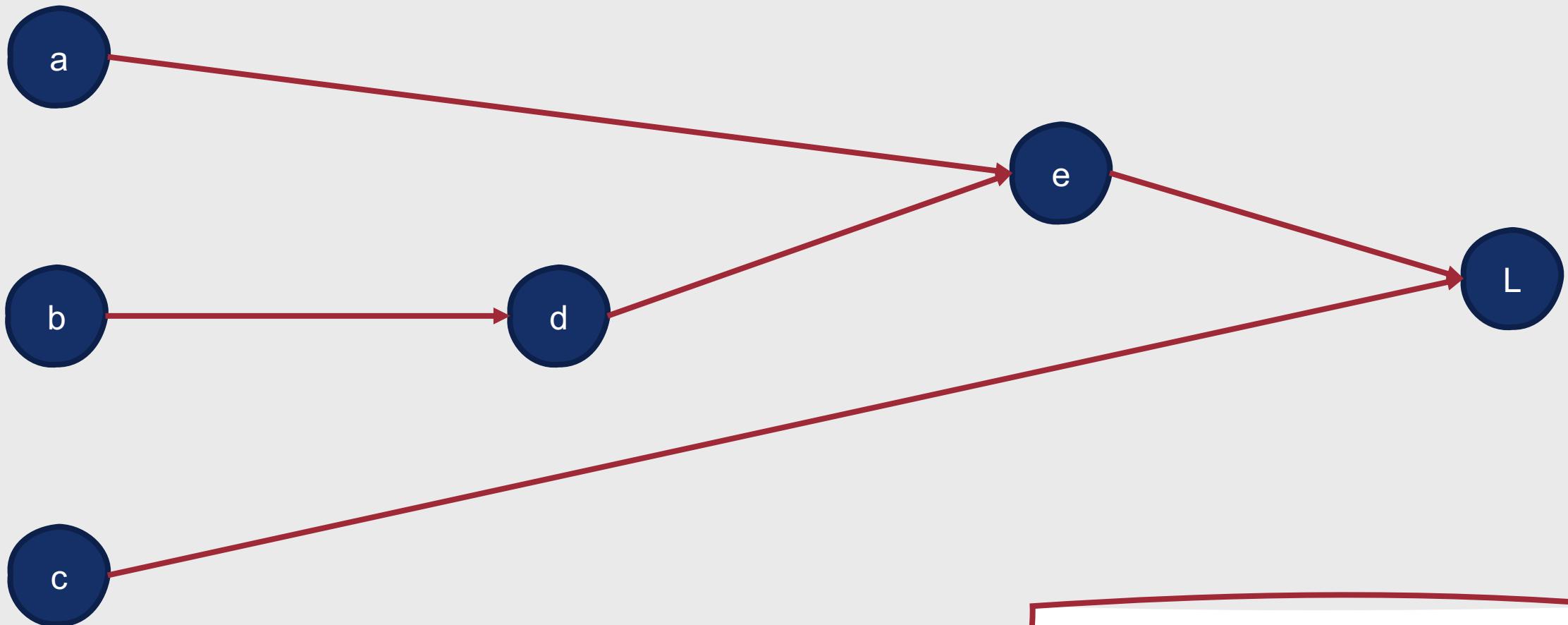
- **Backward pass**

- Compute partial derivatives in the opposite direction of the arrows
- Multiply them by the partial derivatives passed down from the previous step

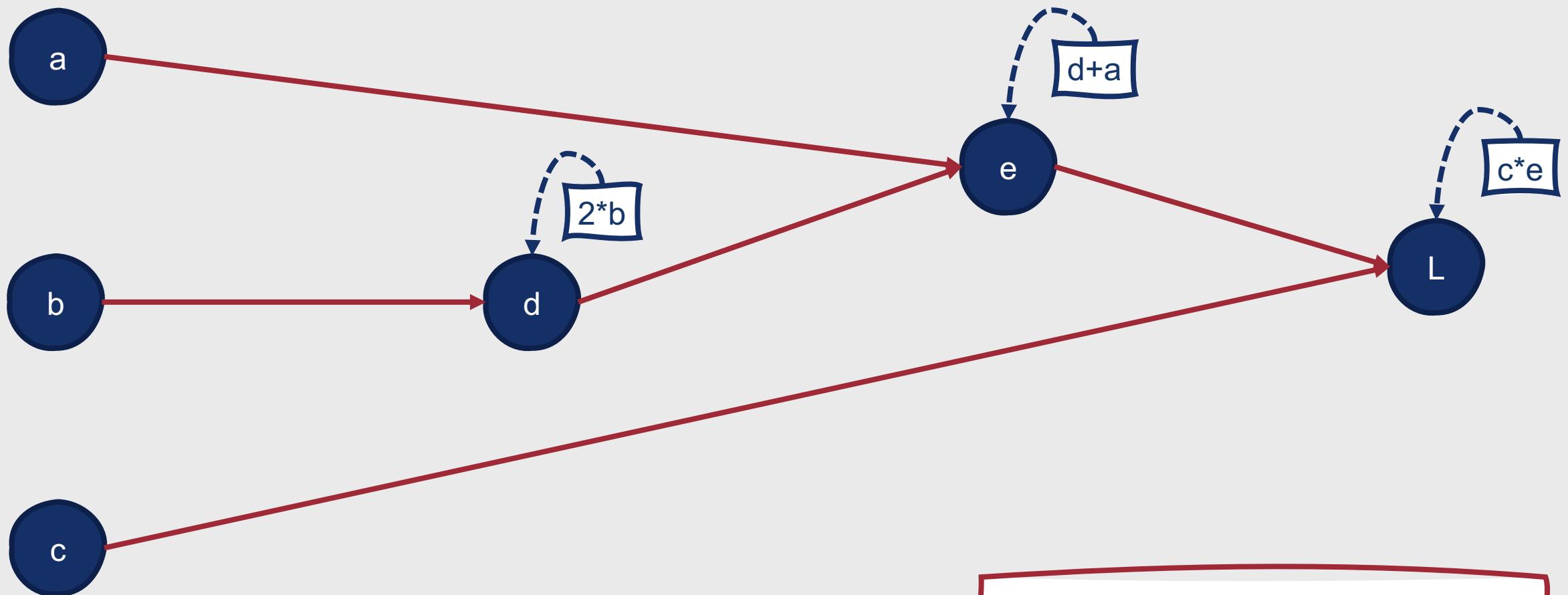
# Example: Forward Pass

Goal: Represent  $L(a, b, c) = c(a + 2b)$

# Example: Forward Pass

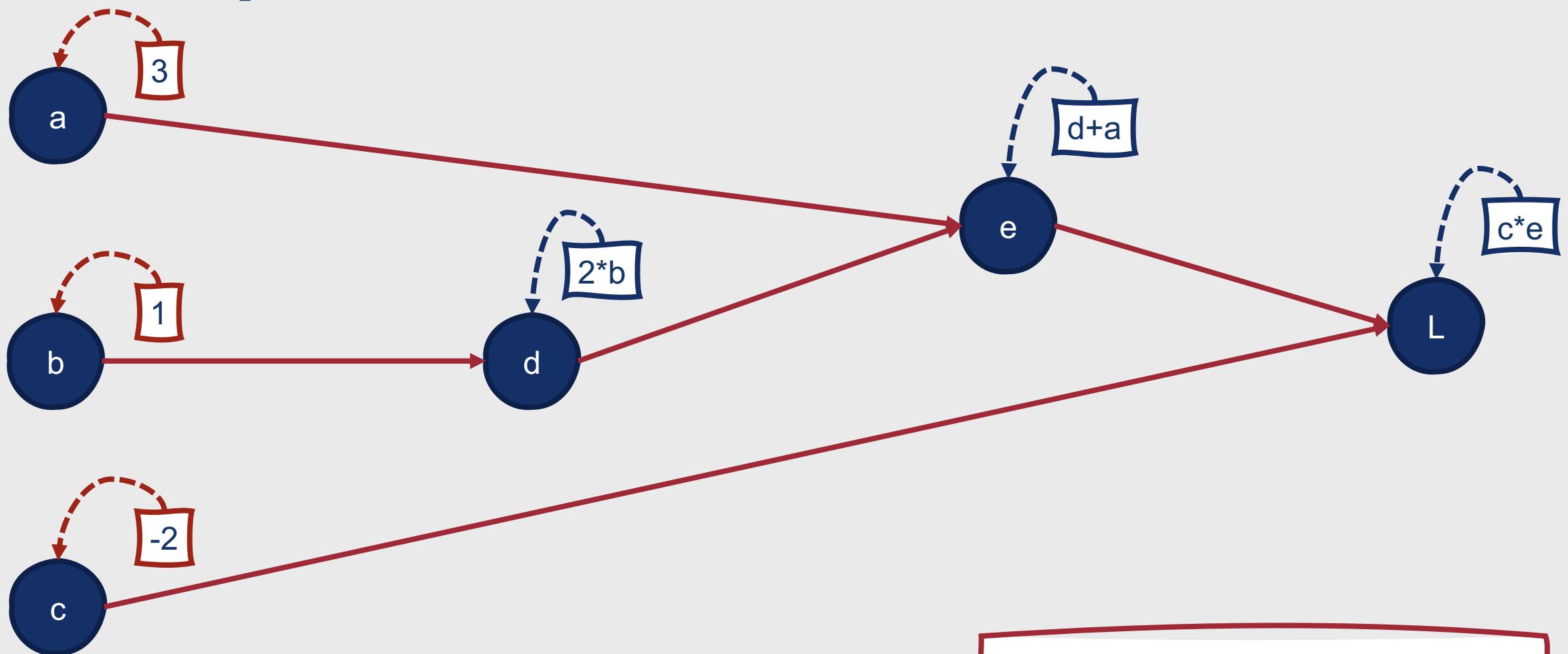


# Example: Forward Pass



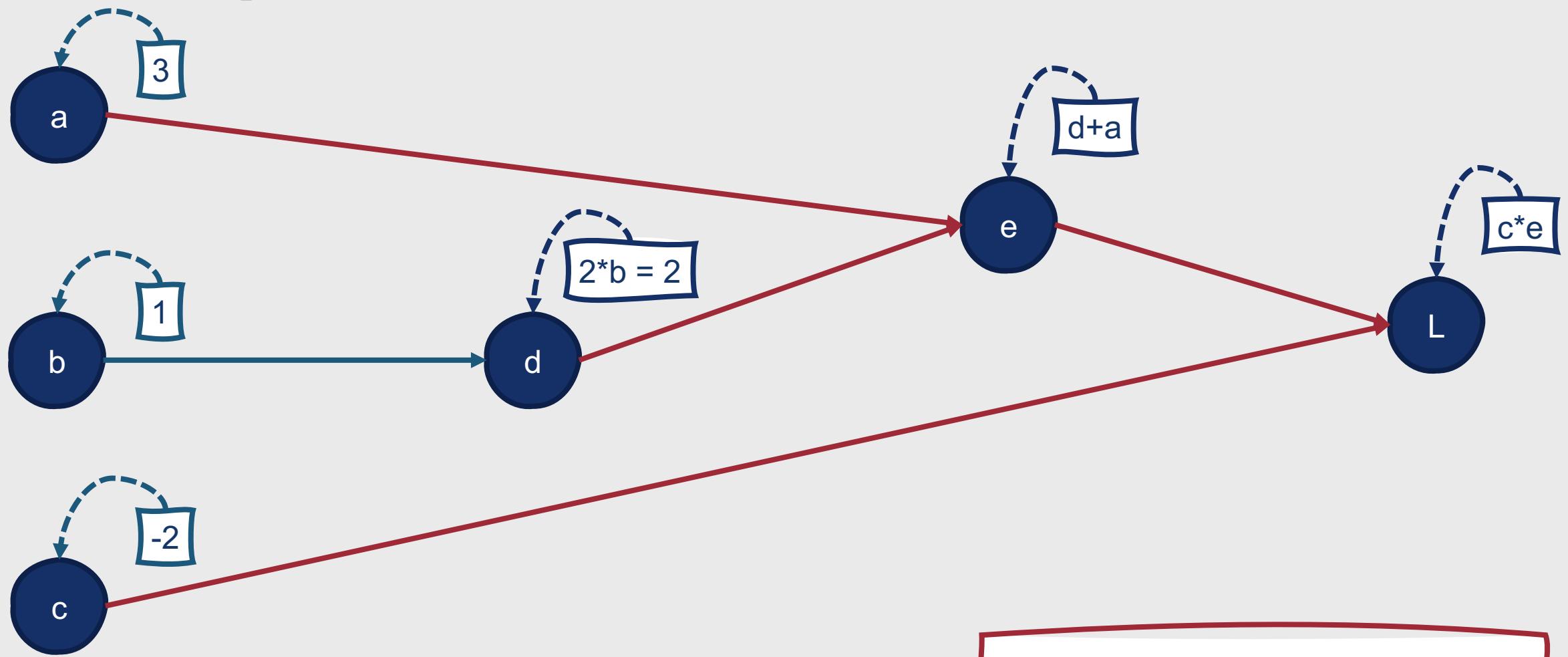
Goal: Represent  $L(a, b, c) = c(a + 2b)$

# Example: Forward Pass



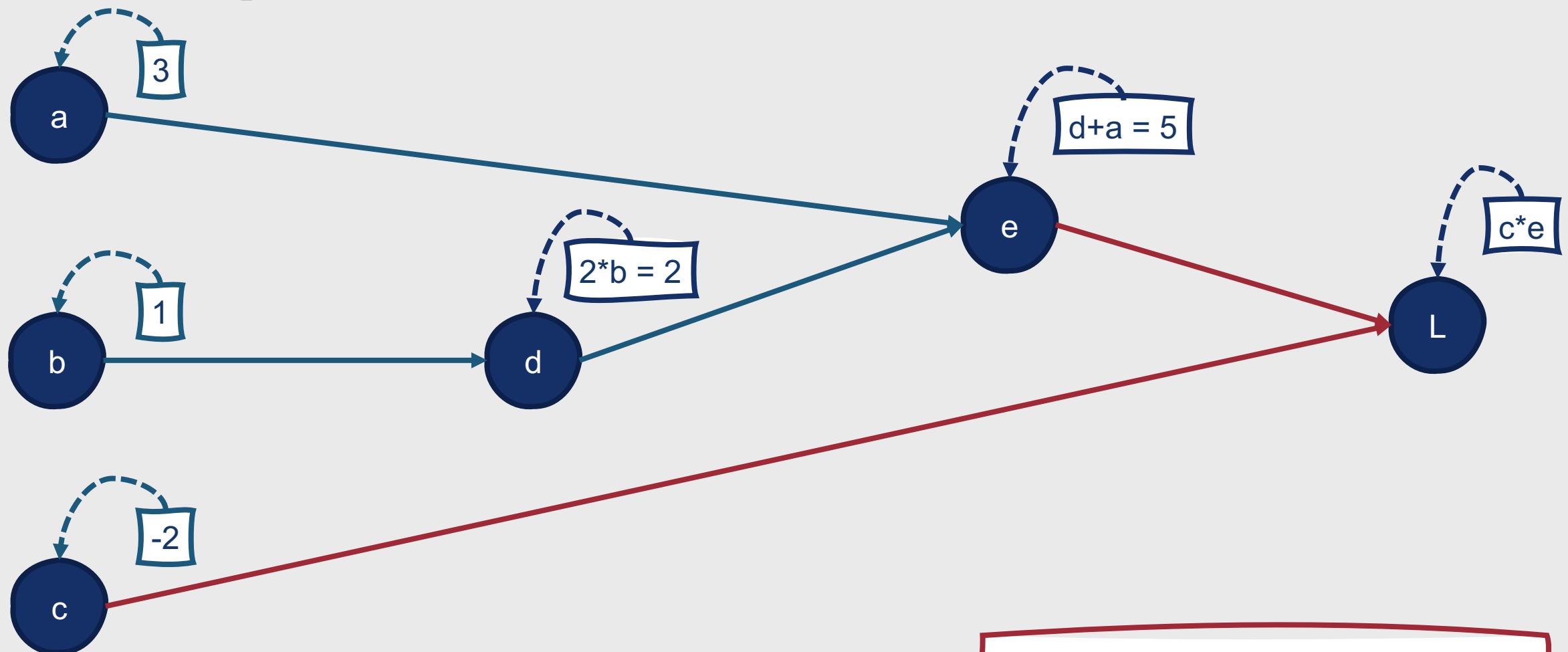
Goal: Represent  $L(a, b, c) = c(a + 2b)$

# Example: Forward Pass



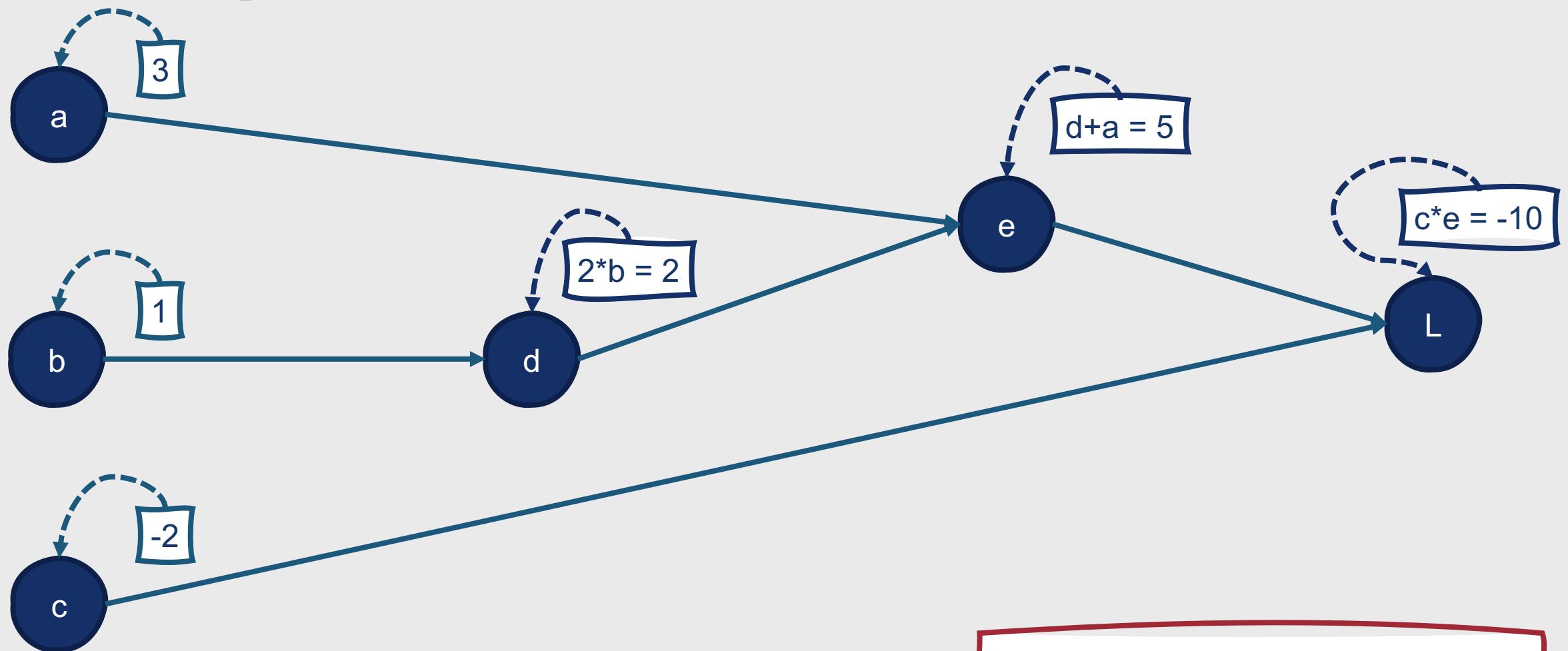
Goal: Represent  $L(a, b, c) = c(a + 2b)$

# Example: Forward Pass

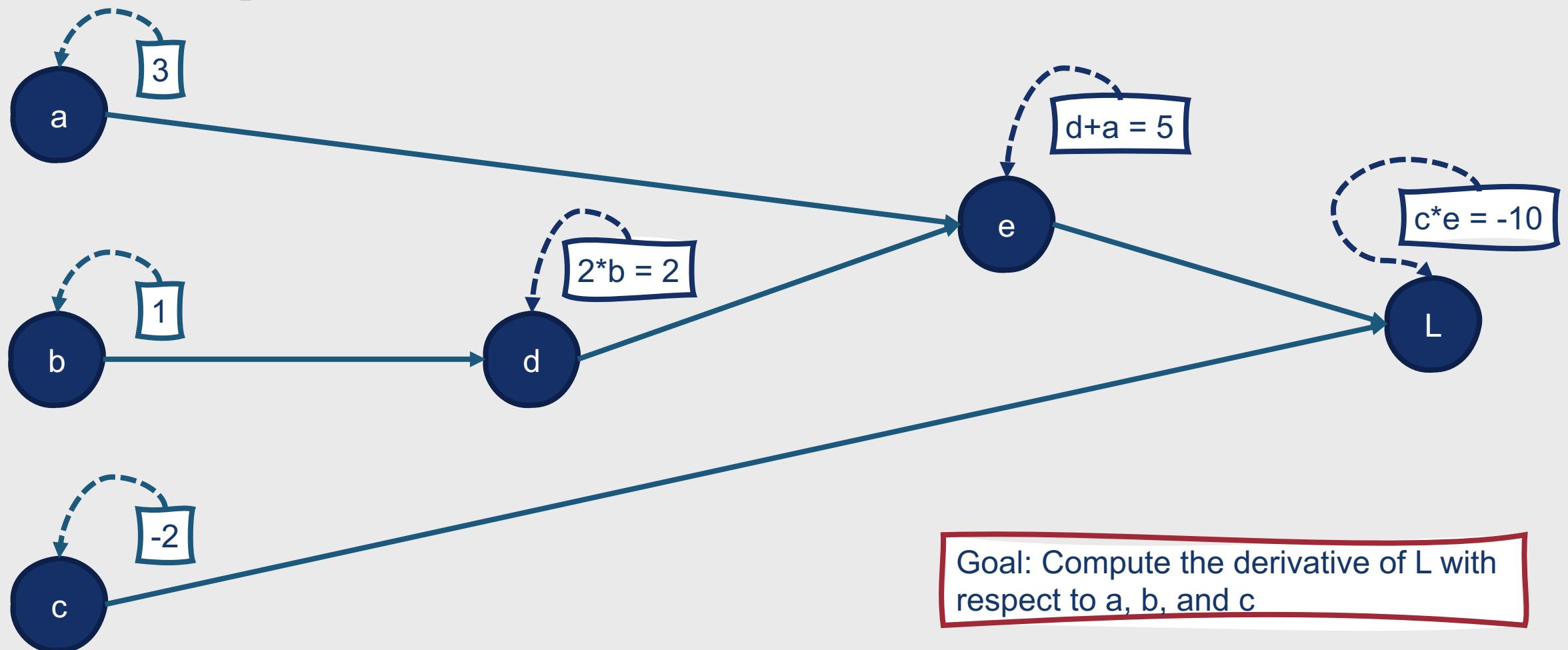


Goal: Represent  $L(a, b, c) = c(a + 2b)$

# Example: Forward Pass



# Example: Backward Pass



# How do we get from L all the way back to a, b, and c?

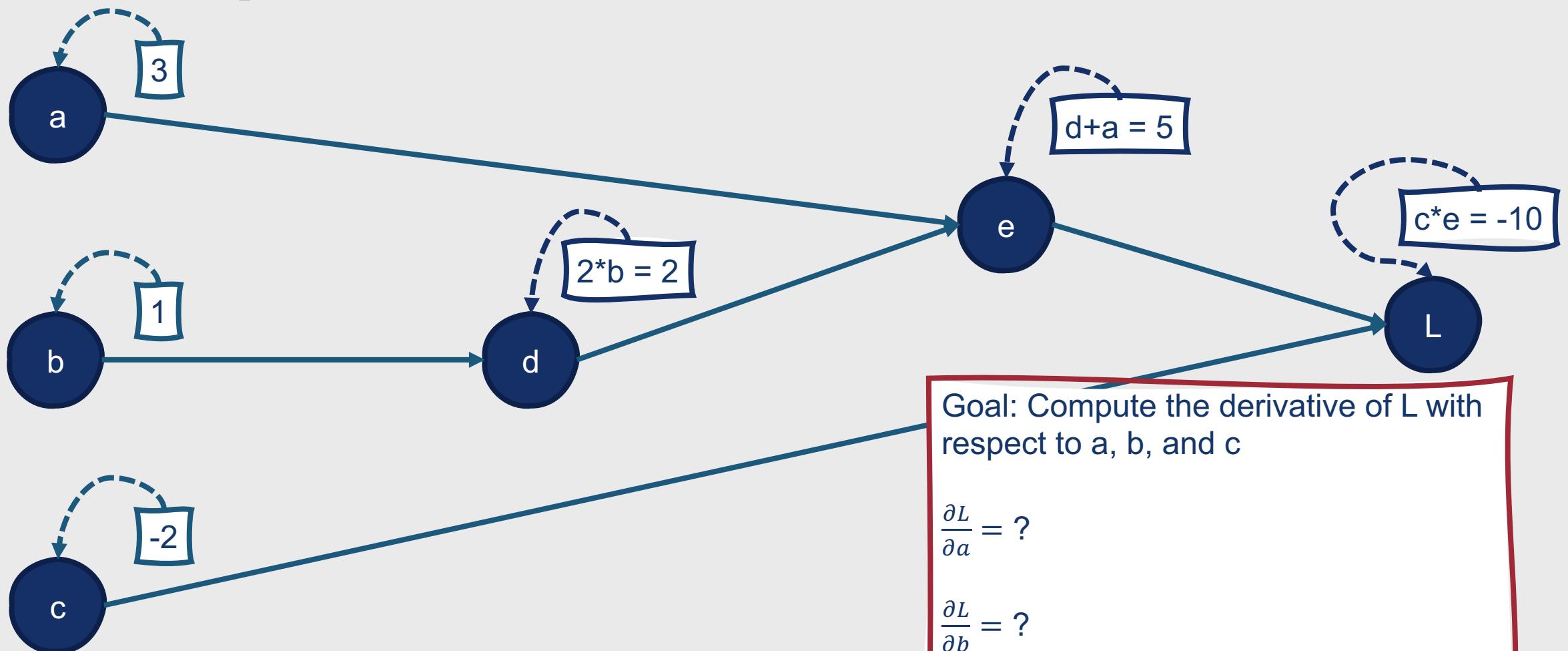
- Chain rule!
  - Given a function  $f(x) = u(v(x))$ :
    - Find the derivative of  $u(x)$  with respect to  $v(x)$
    - Find the derivative of  $v(x)$  with respect to  $x$
    - Multiply the two together
  - $\frac{df}{dx} = \frac{du}{dv} * \frac{dv}{dx}$

Derivatives of popular activation functions:

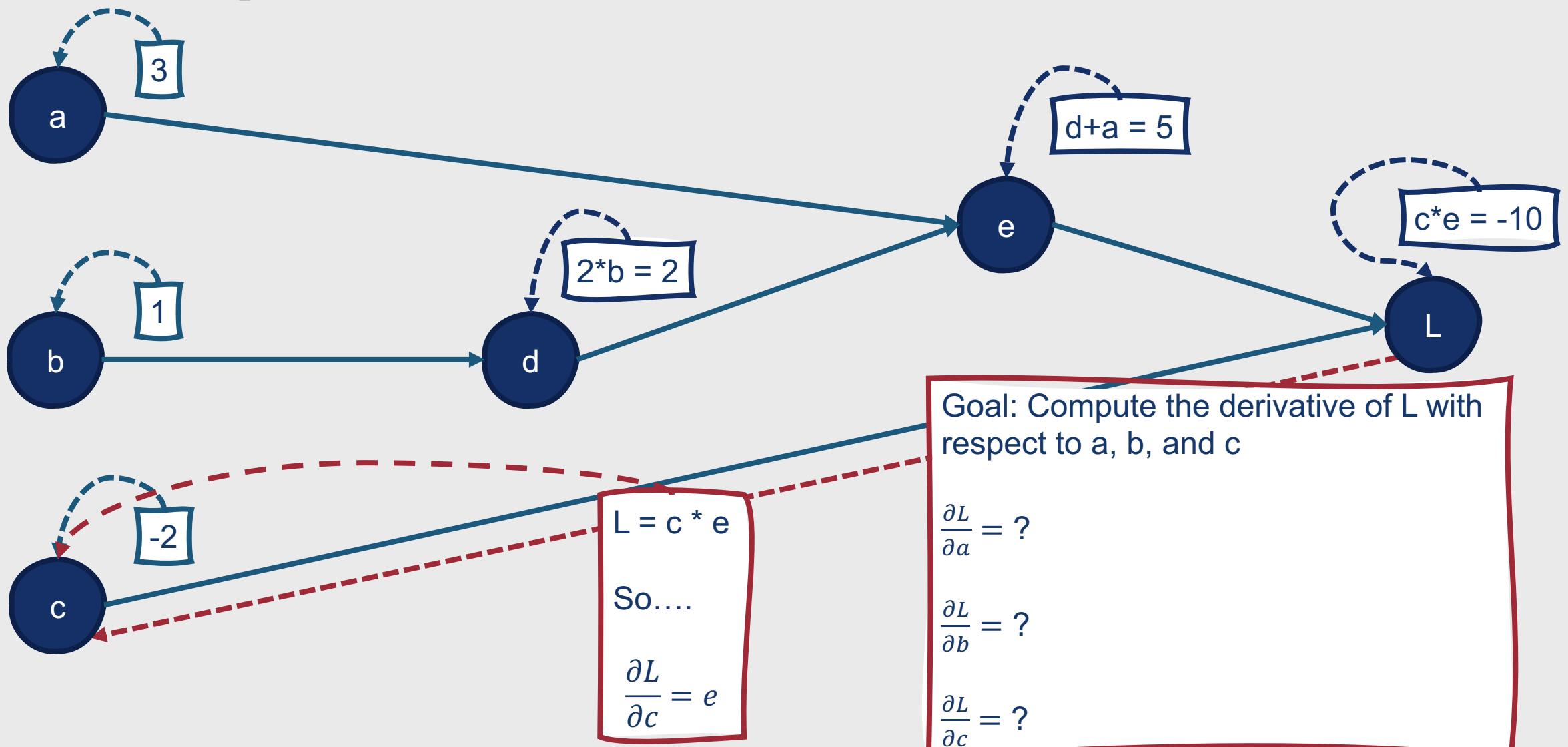
$$\frac{\partial \tanh(z)}{\partial z} = 1 - \tanh^2(z)$$

$$\frac{\partial \text{ReLU}(z)}{\partial z} = \begin{cases} 0 & \text{for } z < 0 \\ 1 & \text{for } z \geq 0 \end{cases}$$

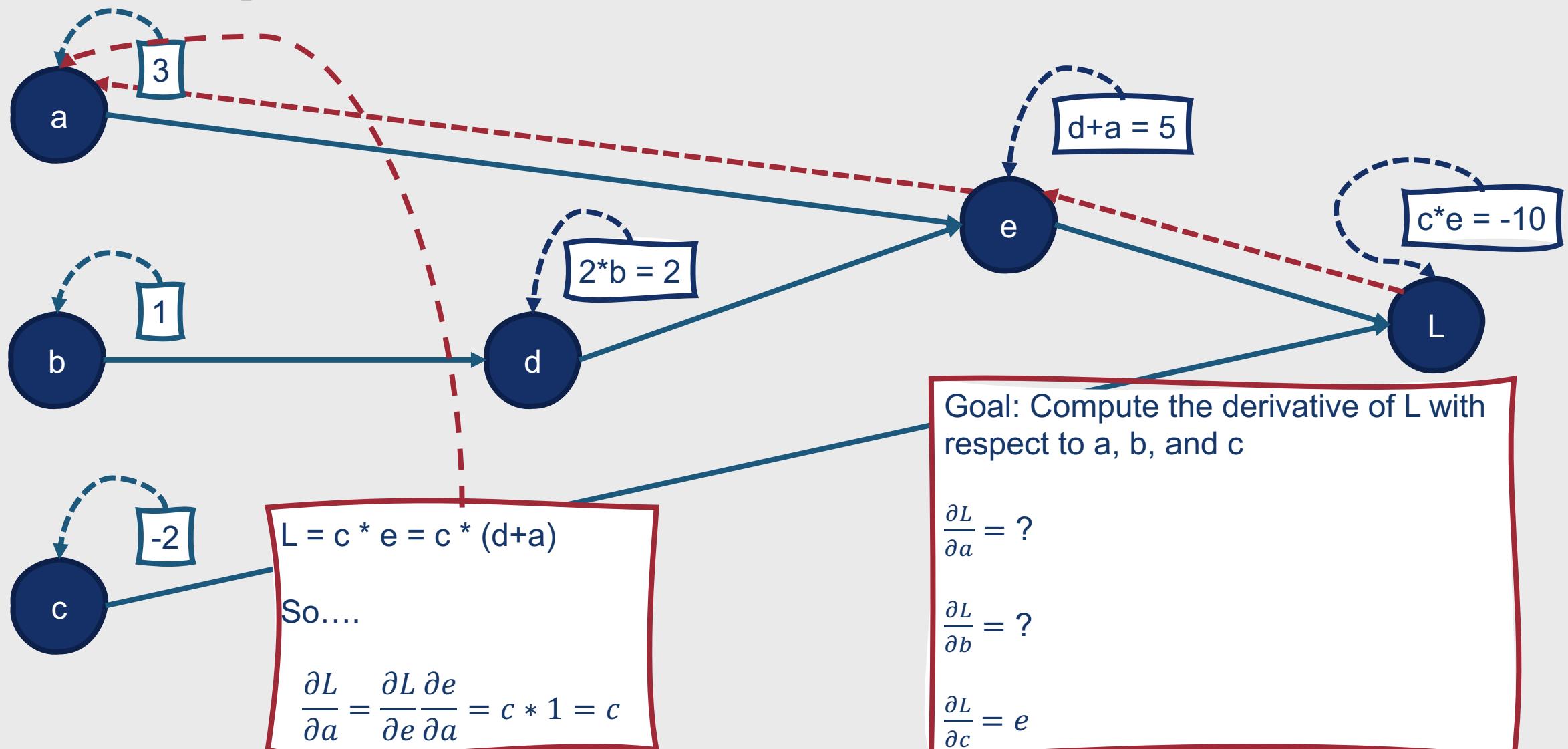
# Example: Backward Pass



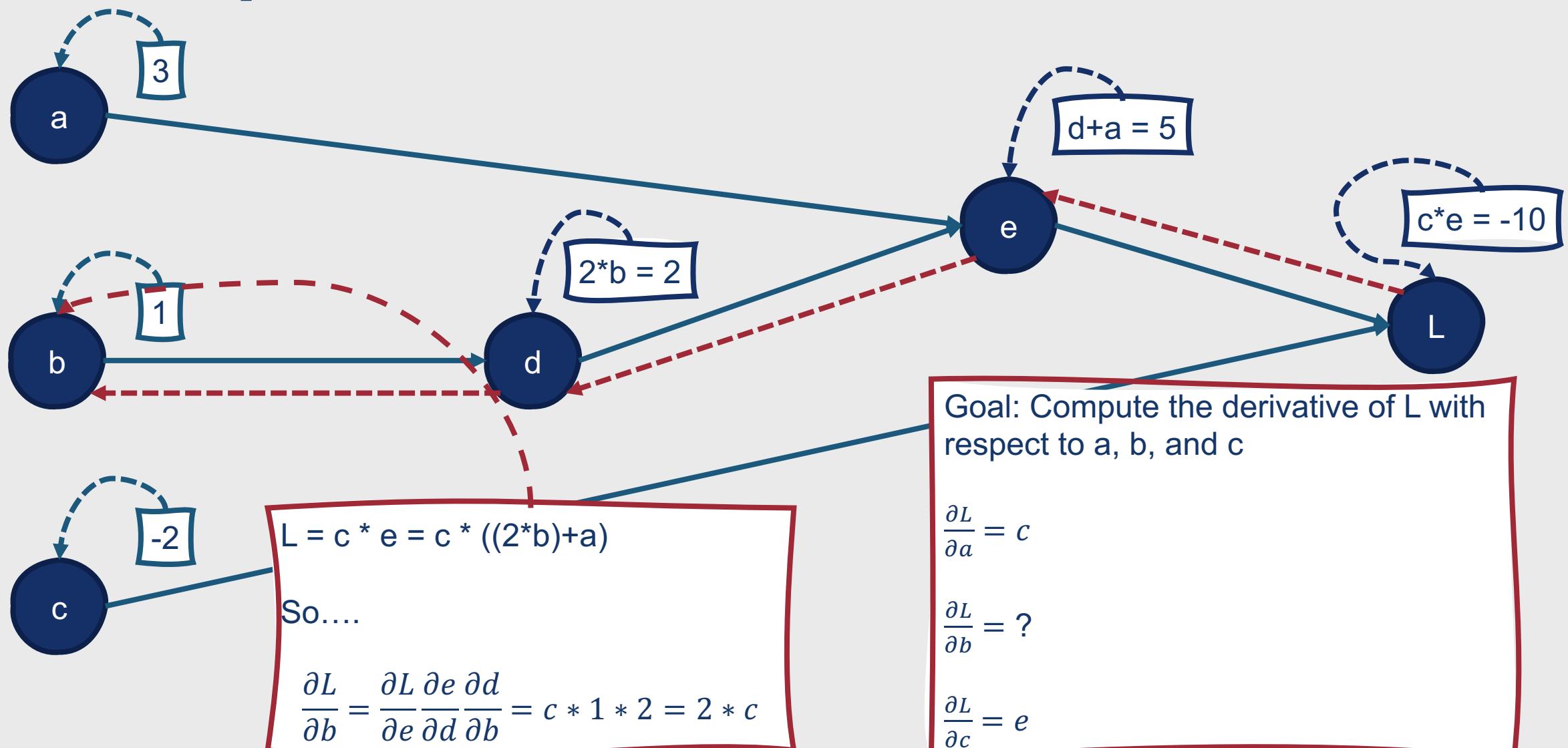
# Example: Backward Pass



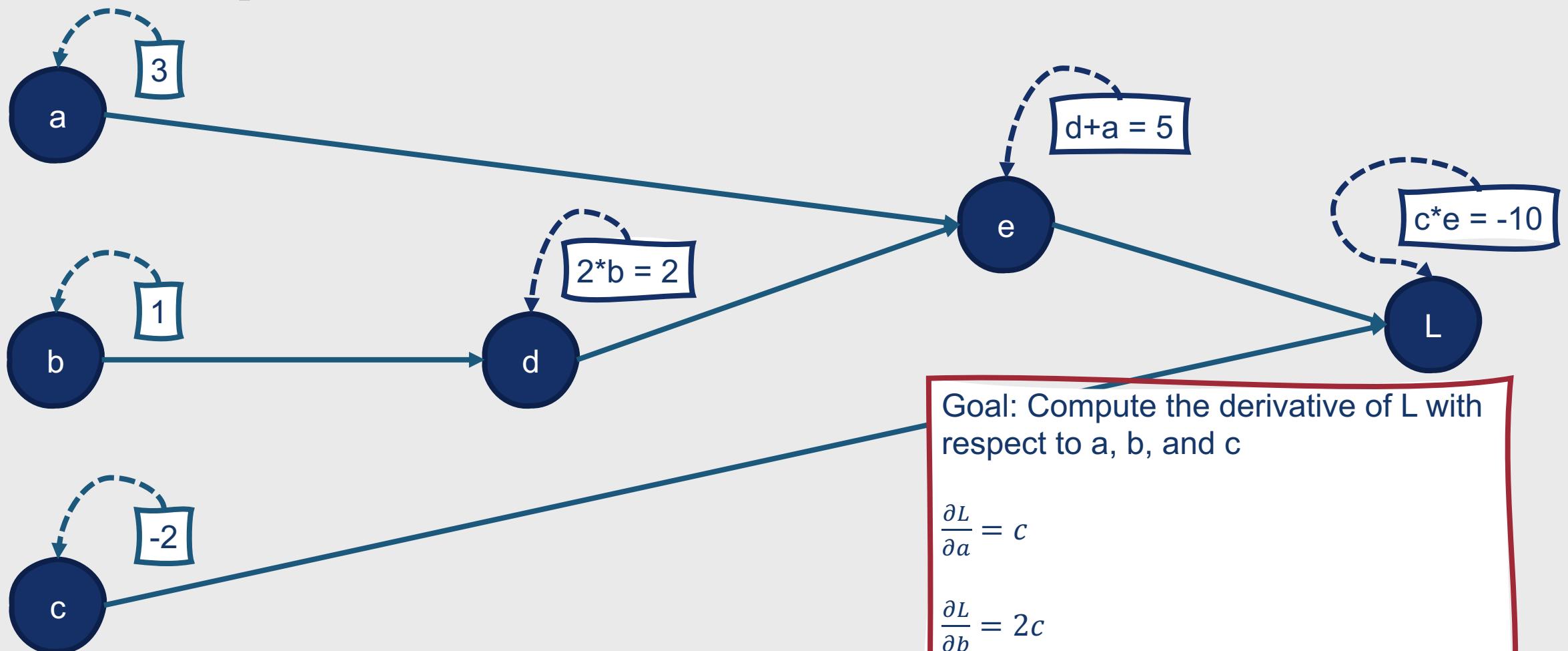
# Example: Backward Pass



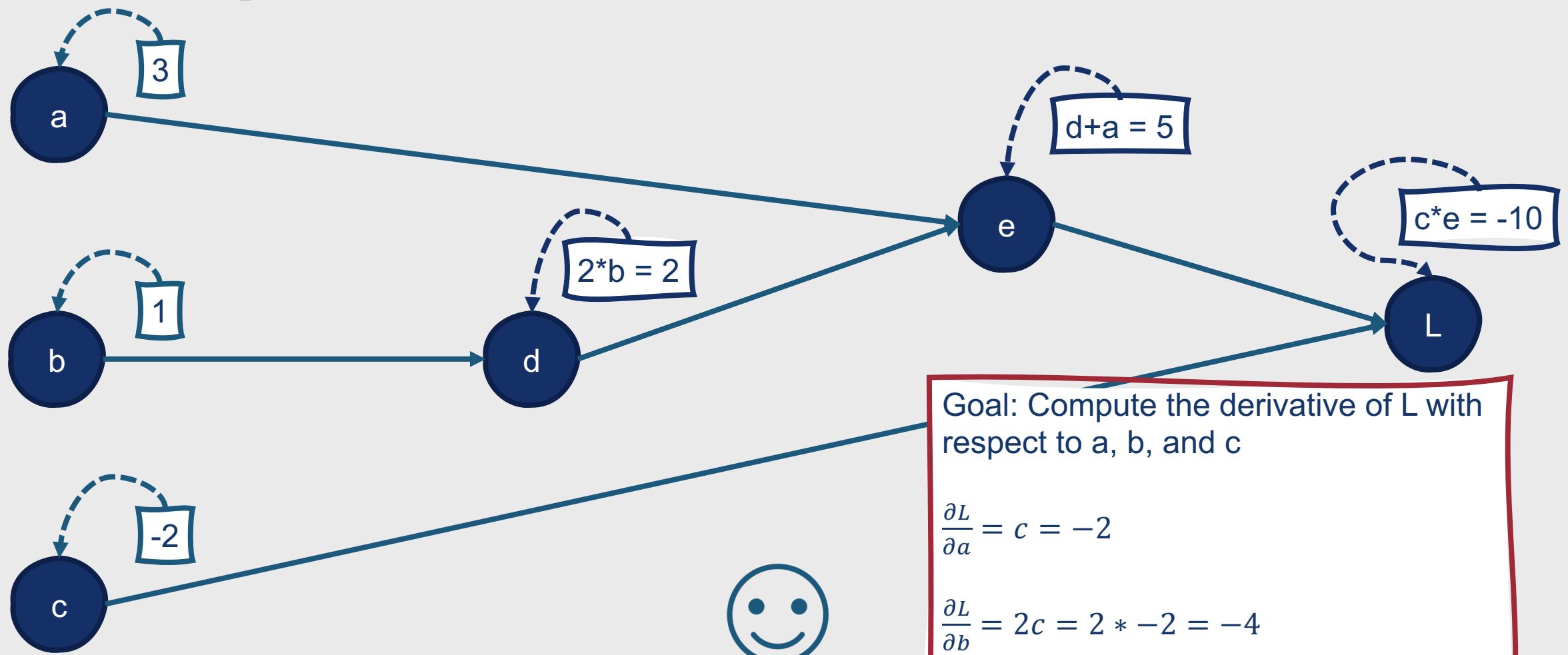
# Example: Backward Pass



# Example: Backward Pass



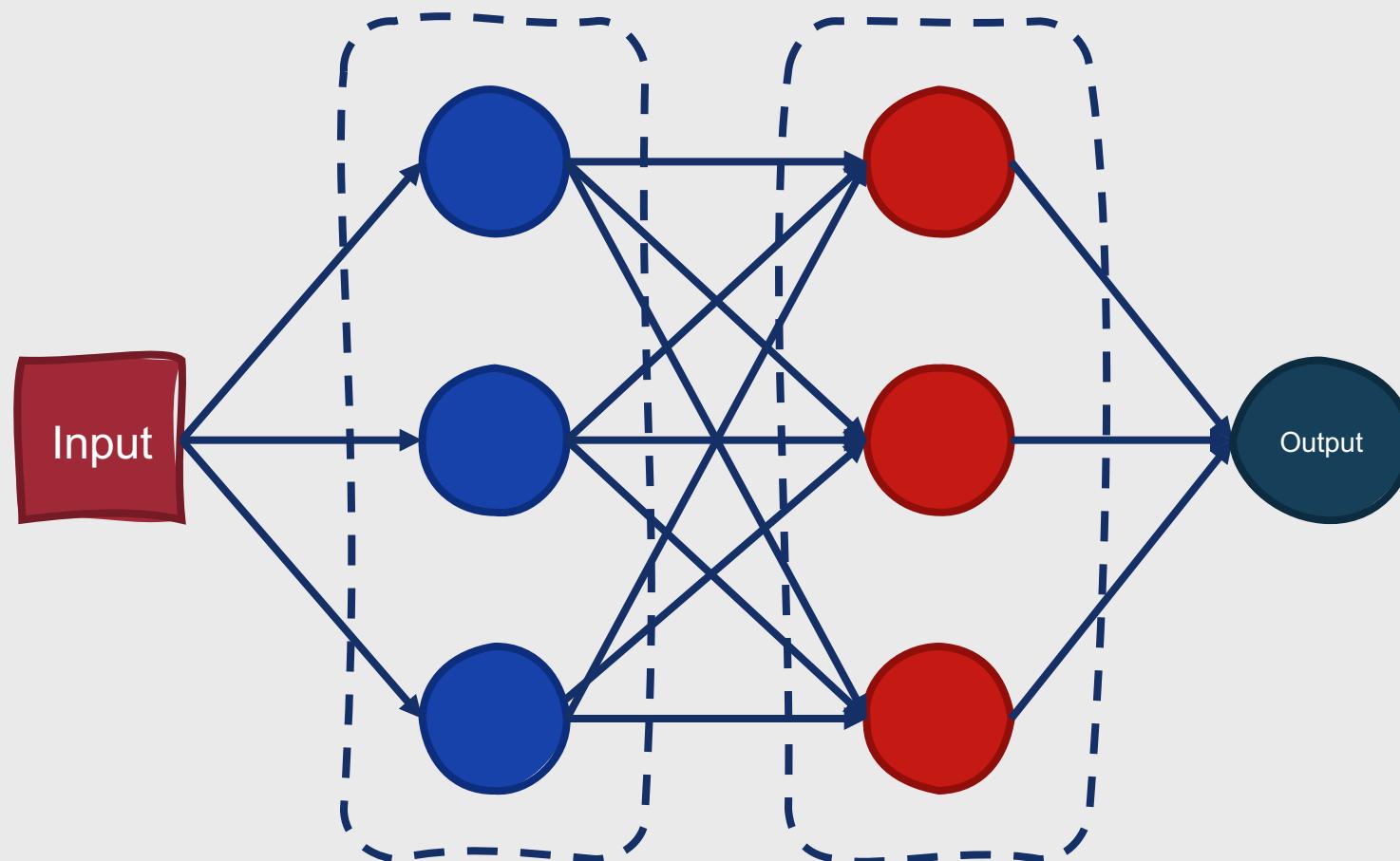
# Example: Backward Pass



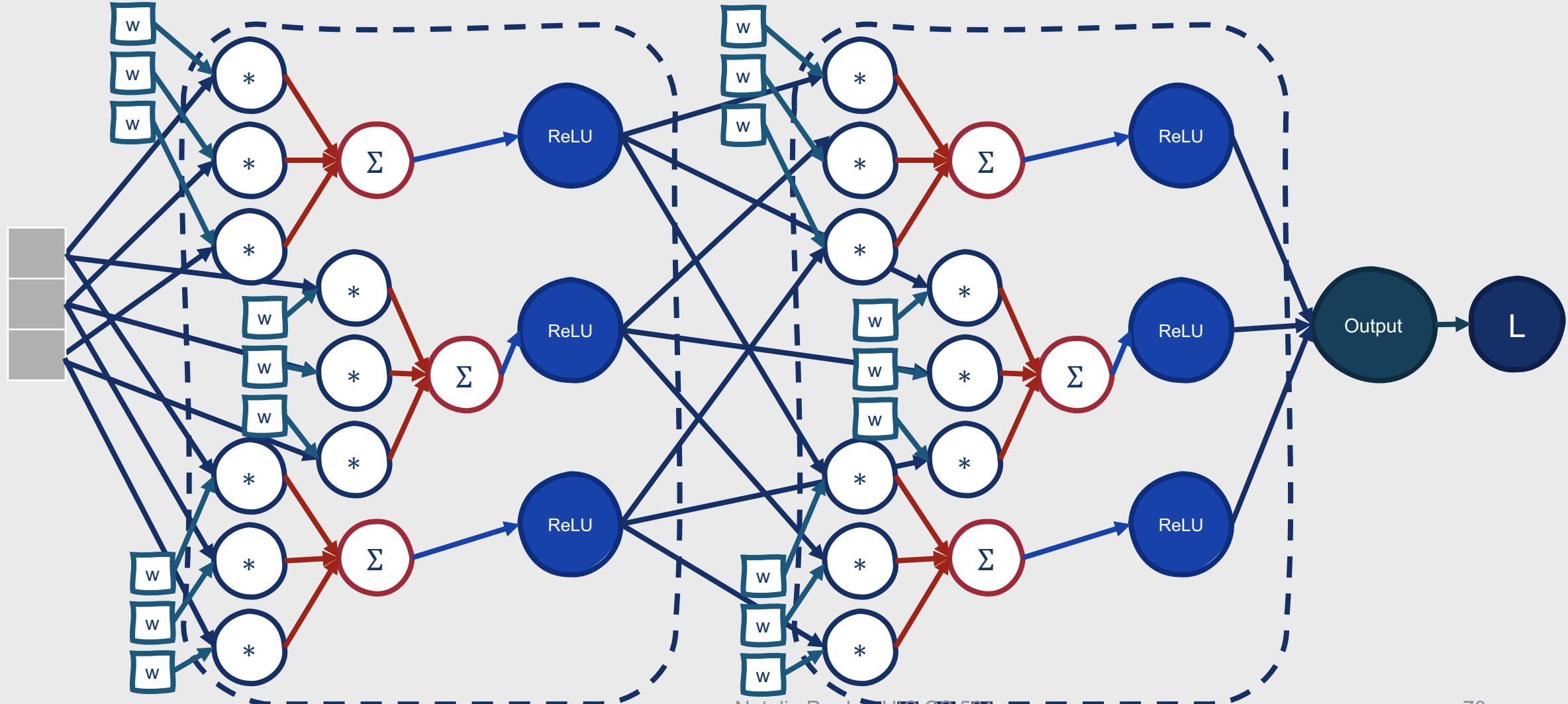
Computation graphs for neural networks are a bit more complex than the previous example.

- More operations:
  - Products (input \* weight)
  - Summations (of weighted inputs)
  - Activation functions

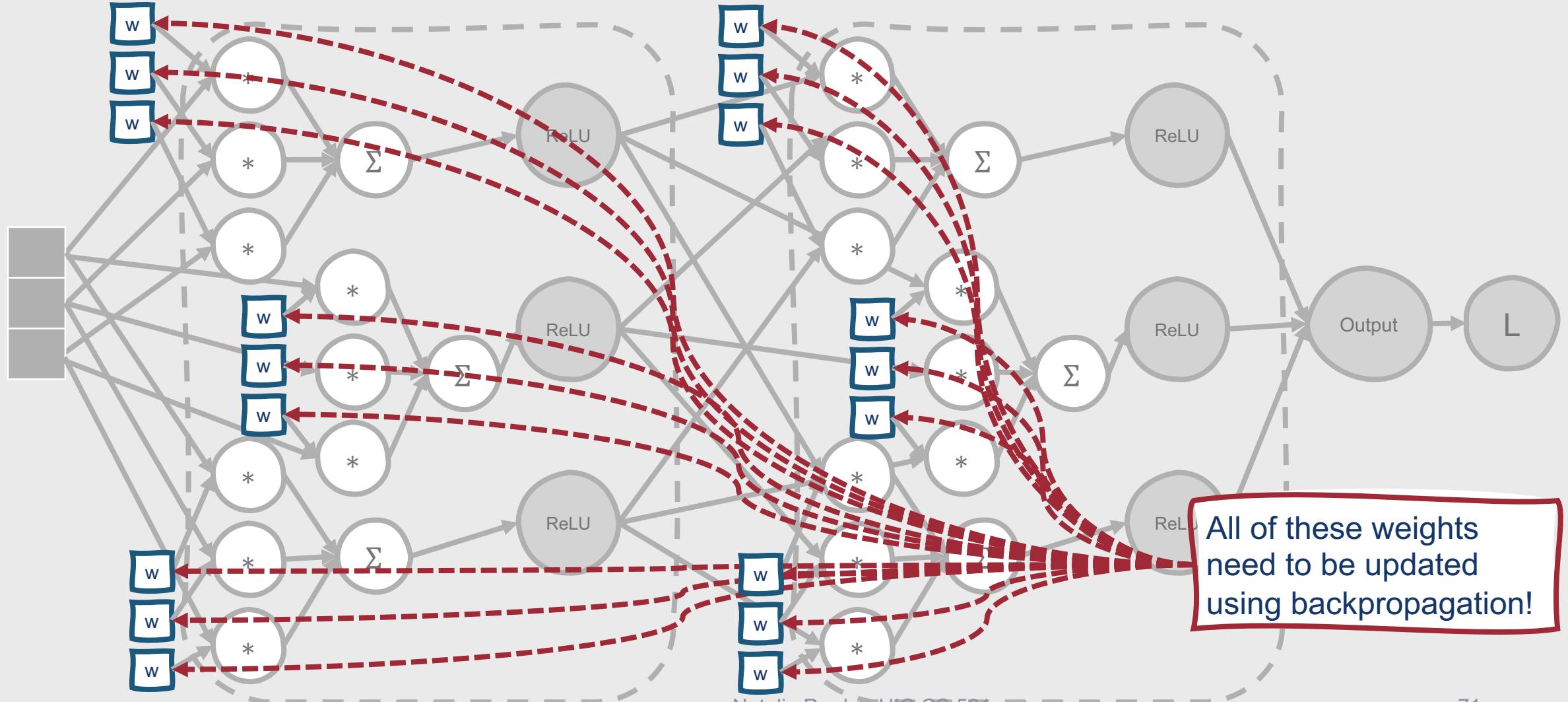
# What would a computation graph look like for a simple neural network?



# What would a computation graph look like for a simple neural network?



# What would a computation graph look like for a simple neural network?

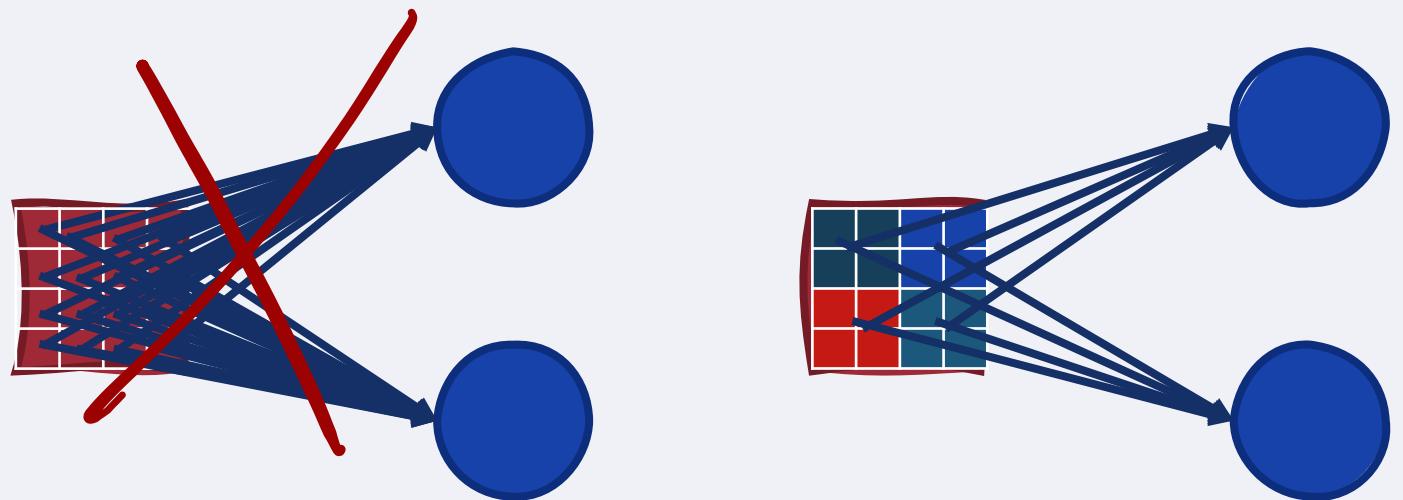


# Convolutional Neural Networks

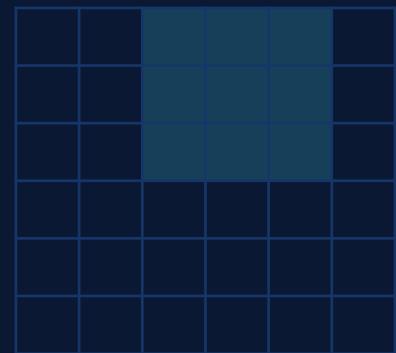
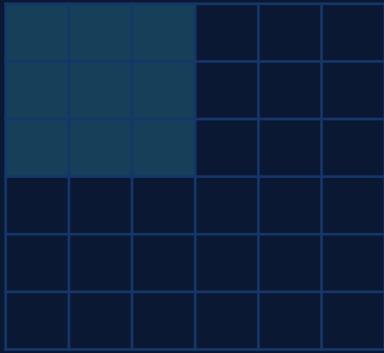
- Neural networks that incorporate one or more **convolutional layers**
- Designed to reflect the inner workings of the visual cortex system
- CNNs require that fewer parameters are learned relative to standard feedforward networks for equivalent input data

# What are convolutional layers?

- **Sliding windows** that perform matrix operations on subsets of the input
- Compute products between those subsets of input and a corresponding weight matrix

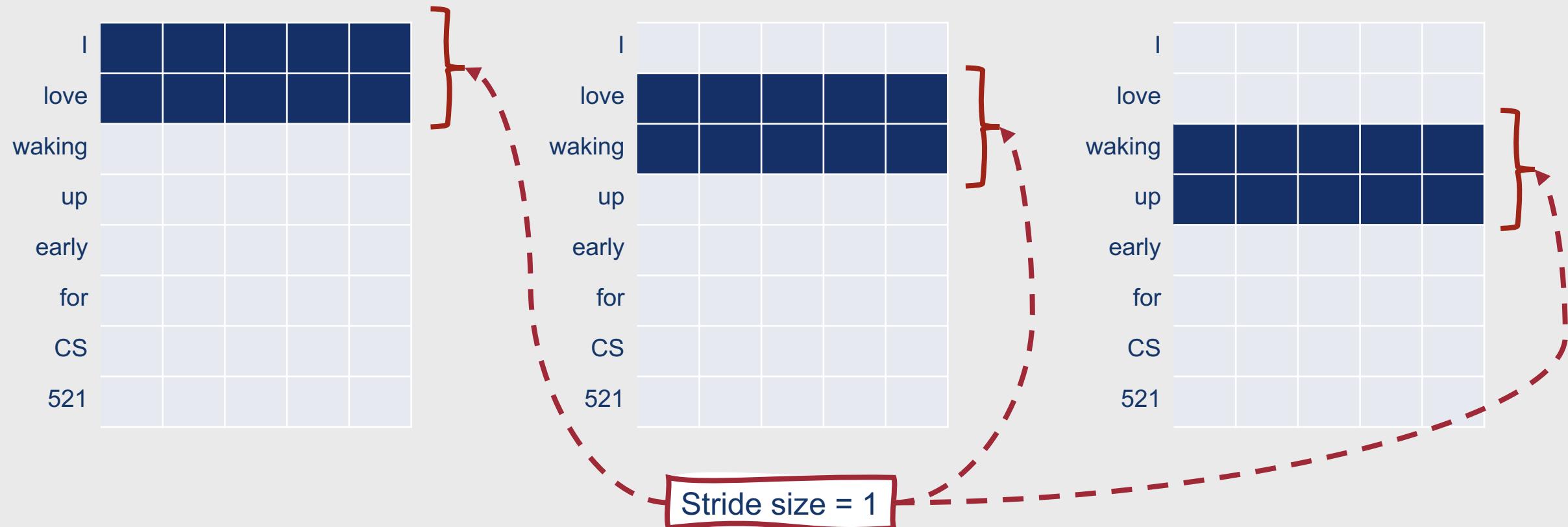


- First layer(s): low-level features
  - Color, gradient orientation
  - N-grams
- Higher layer(s): high-level features
  - Objects
  - Phrases

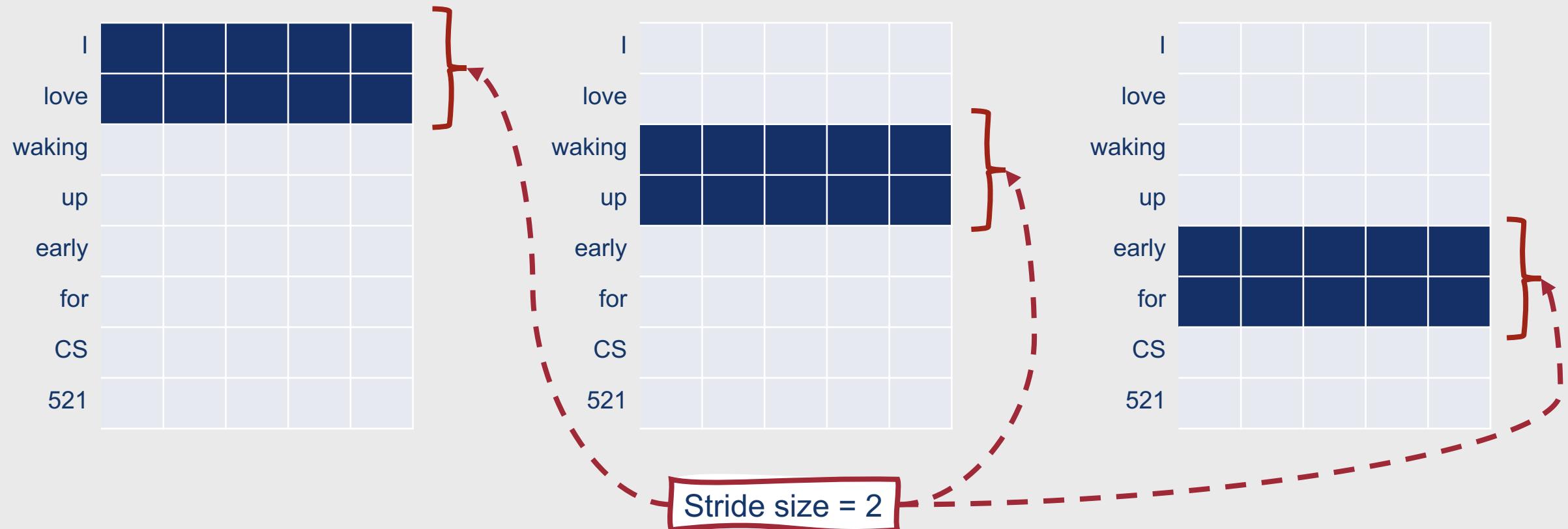


# Convolutional Layers

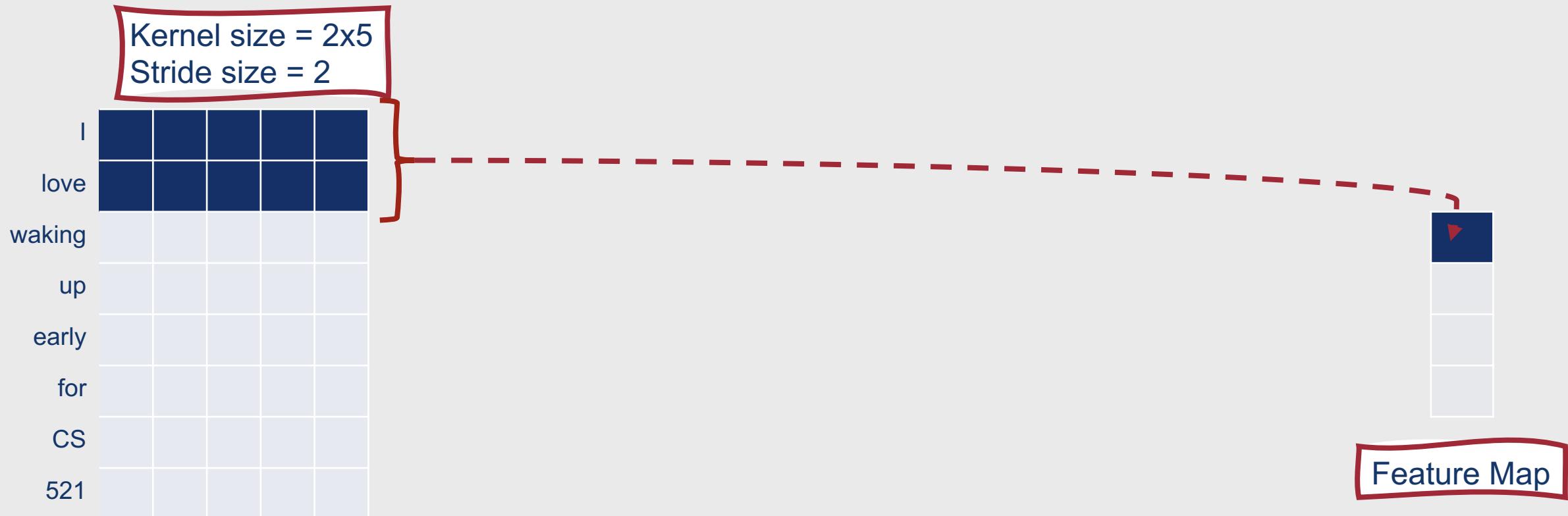
In NLP, convolutions are typically performed on entire rows of an input matrix, where each row corresponds to a word.



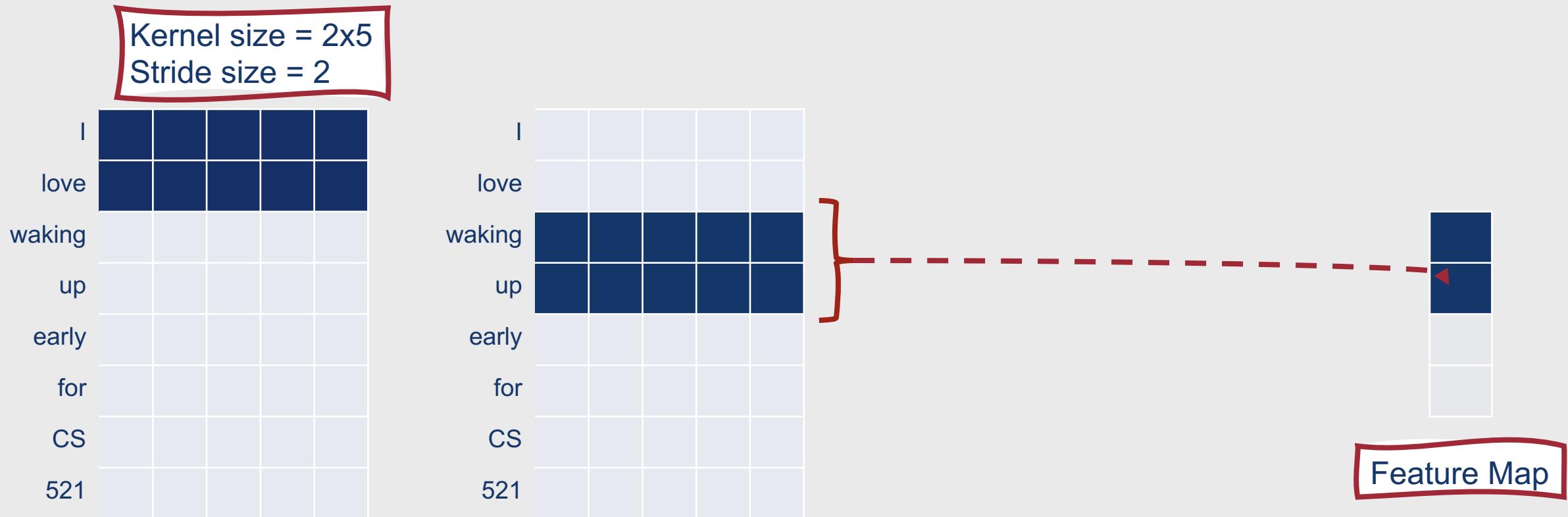
In NLP, convolutions are typically performed on entire rows of an input matrix, where each row corresponds to a word.



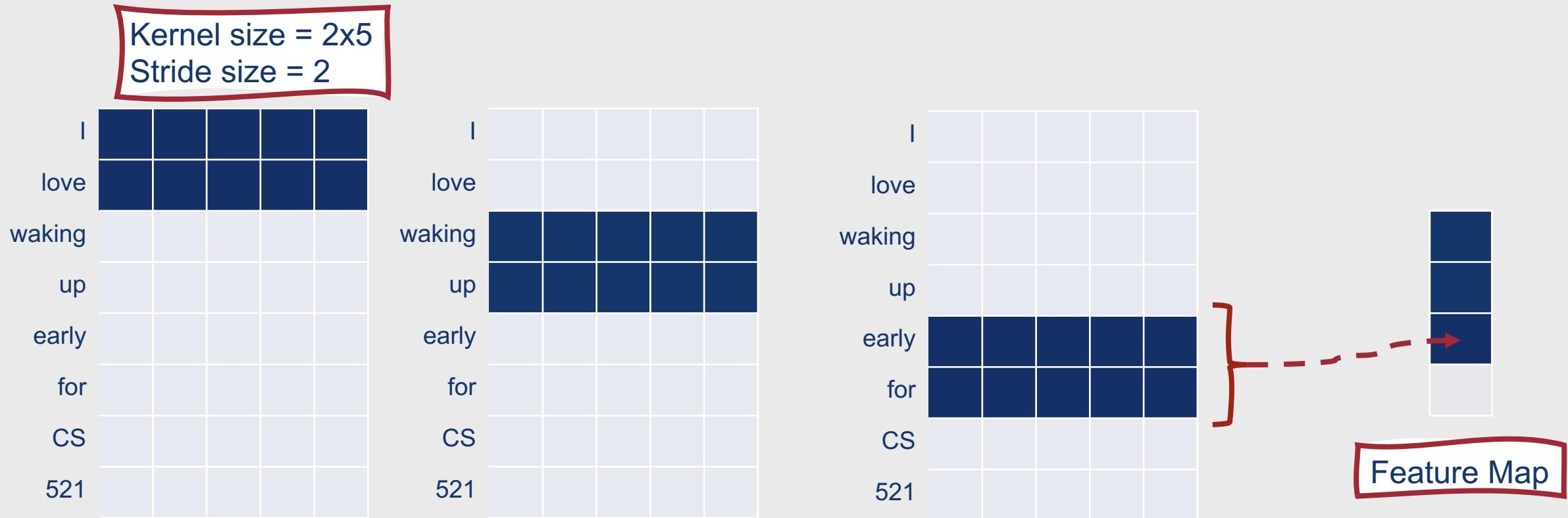
**After applying a convolution with specific region (kernel) and stride sizes to an input matrix, we end up with a feature map.**



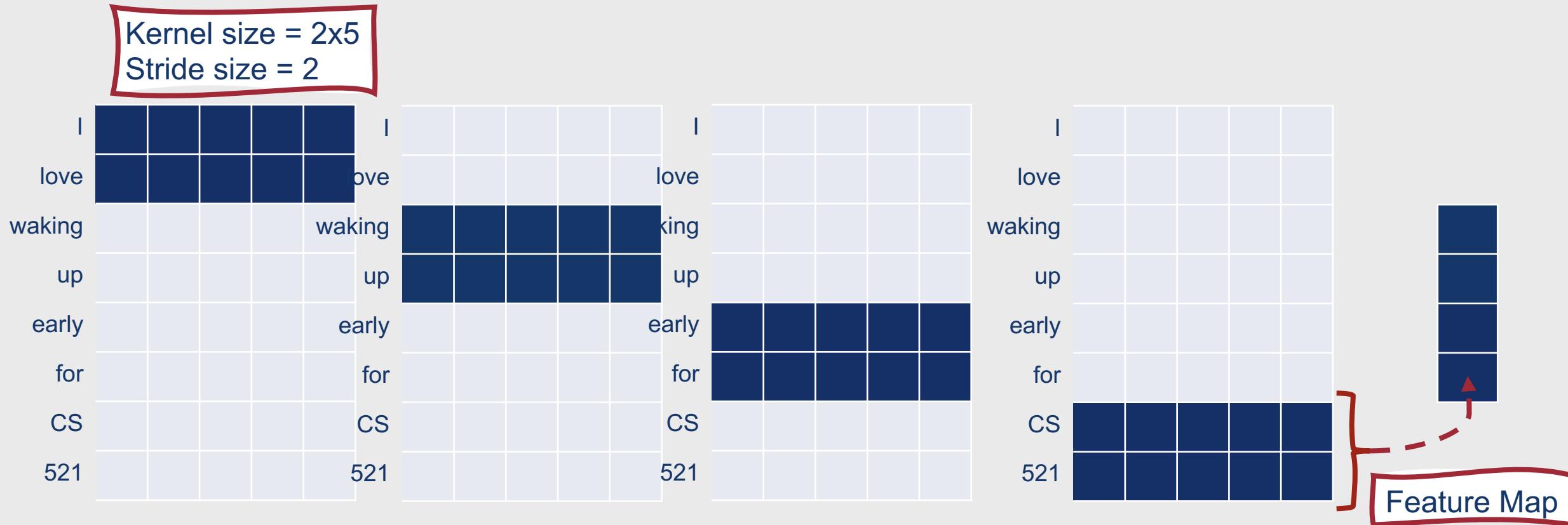
# After applying a convolution with specific region (kernel) and stride sizes to an input matrix, we end up with a feature map.

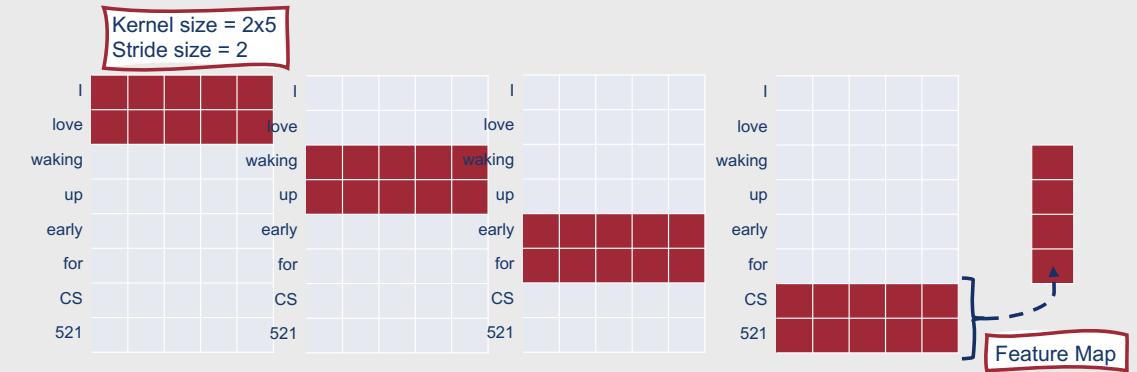
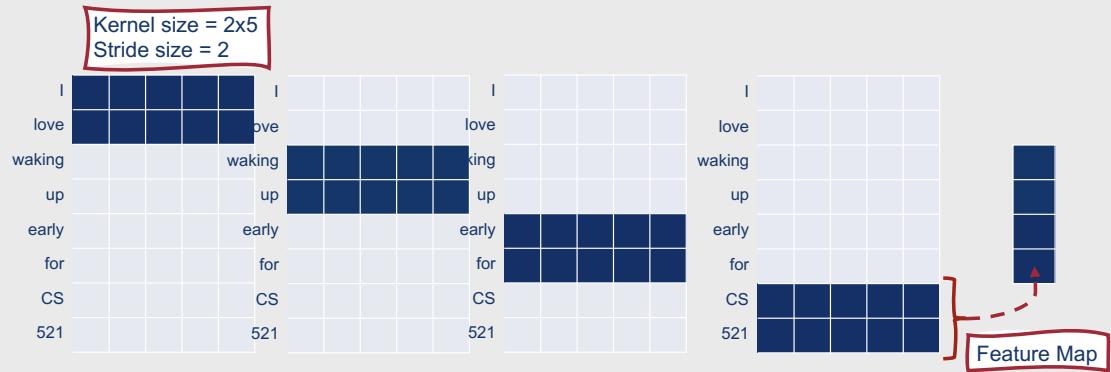


# After applying a convolution with specific region (kernel) and stride sizes to an input matrix, we end up with a feature map.



# After applying a convolution with specific region (kernel) and stride sizes to an input matrix, we end up with a feature map.





**It's common to extract multiple different feature maps from the same input.**

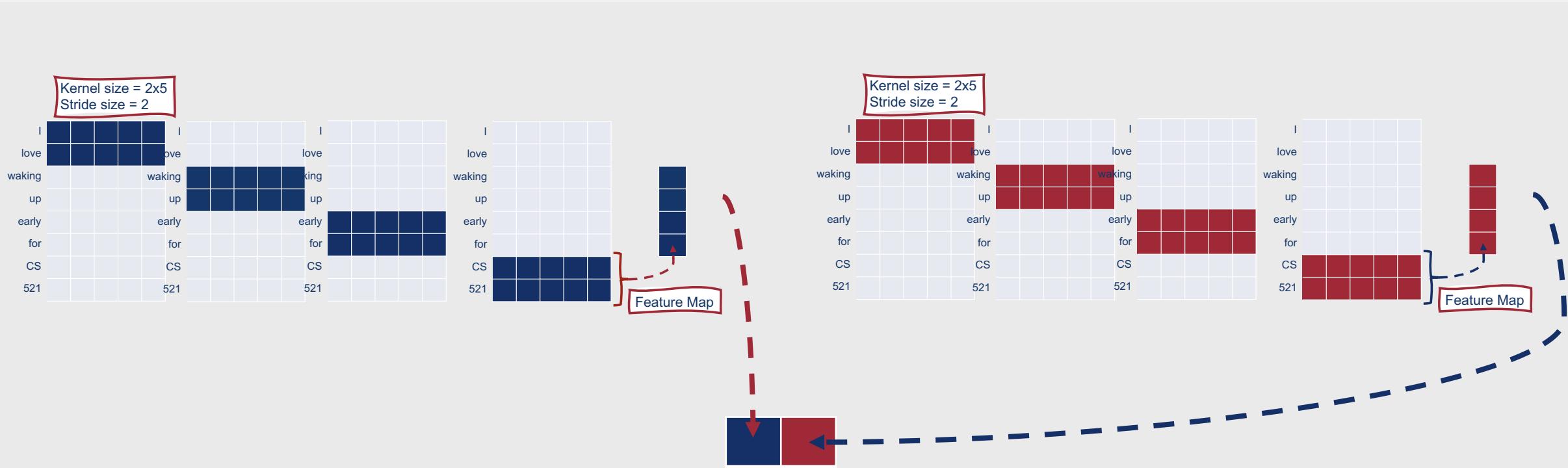


# After extracting feature maps from the input, CNNs utilize pooling layers.

- **Pooling layers:** Layers that reduce the dimensionality of input feature maps by pooling all of the values in a given region
- Why use pooling layers?
  - Further increase **efficiency**
  - Improve the model's ability to be **invariant to small changes**

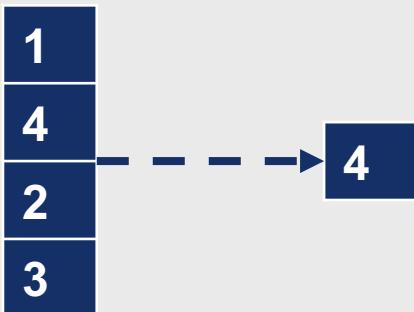


# Pooling Layers



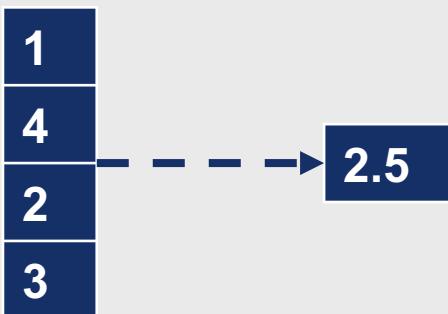
# Common Techniques for Pooling

- **Max pooling**
  - Take the maximum of all values computed in a given window
- **Average pooling**
  - Take the average of all values computed in a given window

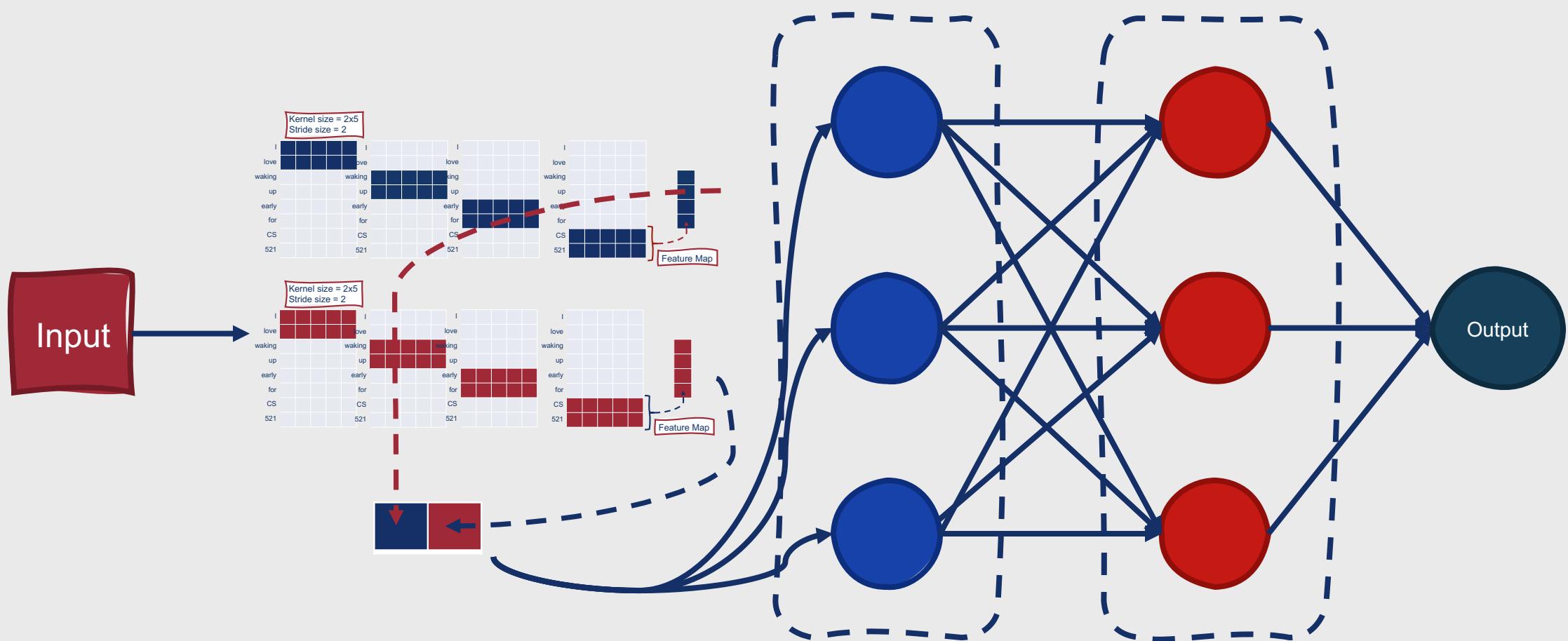


# Common Techniques for Pooling

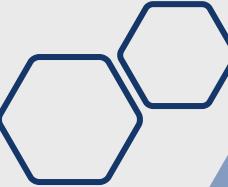
- **Max pooling**
  - Take the maximum of all values computed in a given window
- **Average pooling**
  - Take the average of all values computed in a given window



# The output from pooling layers is typically then passed along as input to one or more feedforward layers.



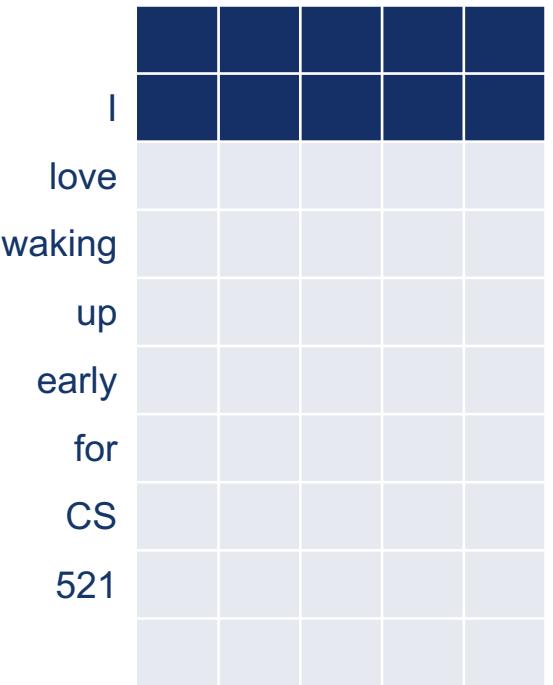
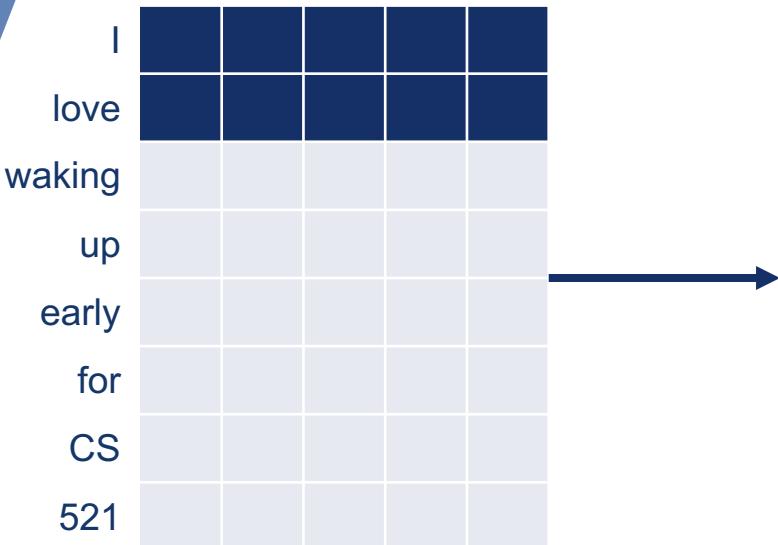
# Convolutional neural network architectures can vary greatly!



- Additional hyperparameters:
  - Kernel size
  - Padding
  - Stride size
  - Number of channels
  - Pooling technique

# Padding?

- Add empty vectors to the beginning and end of your text input
- Why do this?
  - Allows you to apply a filter to every element of the input matrix



# Channels?

For images, generally corresponds to color channels

- Red, green, blue

For text, can mean:

- Different types of word embeddings
  - Word2Vec, GloVe, etc.
- Other feature types
  - POS tags, word length, etc.

# The big question ...why use CNNs at all?

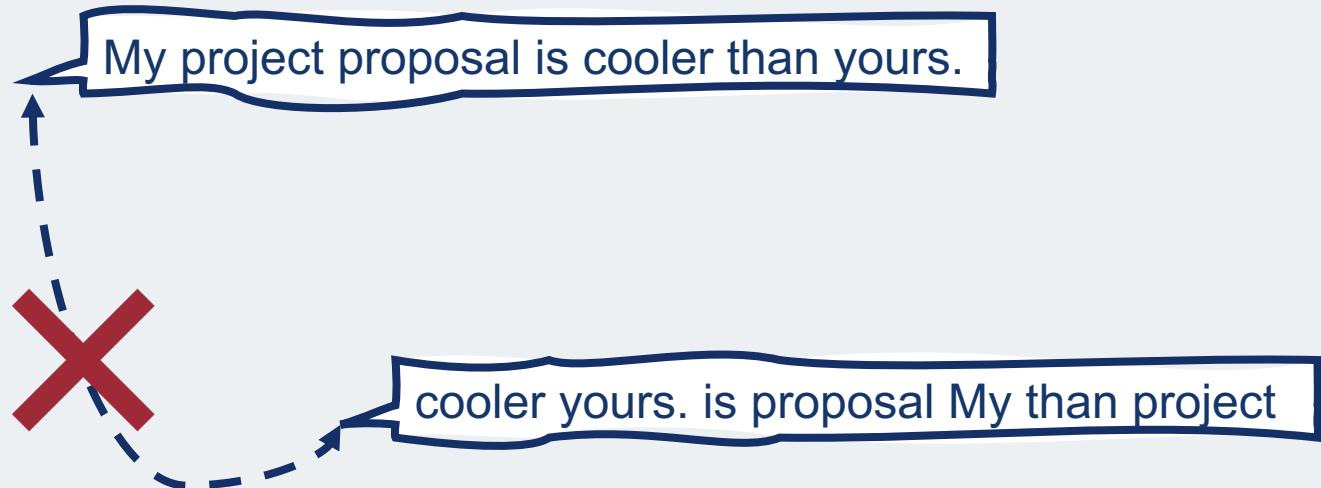
- Traditionally for image classification!
- However, offer unique advantages for NLP tasks:
  - CNNs inherently extract meaningful local structures from input
  - In NLP → implicitly-learned, useful n-grams!

# Language is inherently temporal.

- **Continuous input streams** of **indefinite length** that unfold over **time**
- Even clear from the metaphors we use to describe language:
  - Conversation flow
  - News feed
  - Twitter stream

# What are recurrent neural networks?

- Neural networks that exploit the **temporal** nature of language!
- Also allow variable-length inputs



This makes RNNs particularly useful for performing sequence processing.

- **Sequence Processing:** Automated processing of **sequential** items (e.g., words in a sentence) while taking into account **temporal** information (e.g.,  $w_1$  occurs before  $w_2$ )

# Sequence processing is particularly useful for some tasks!

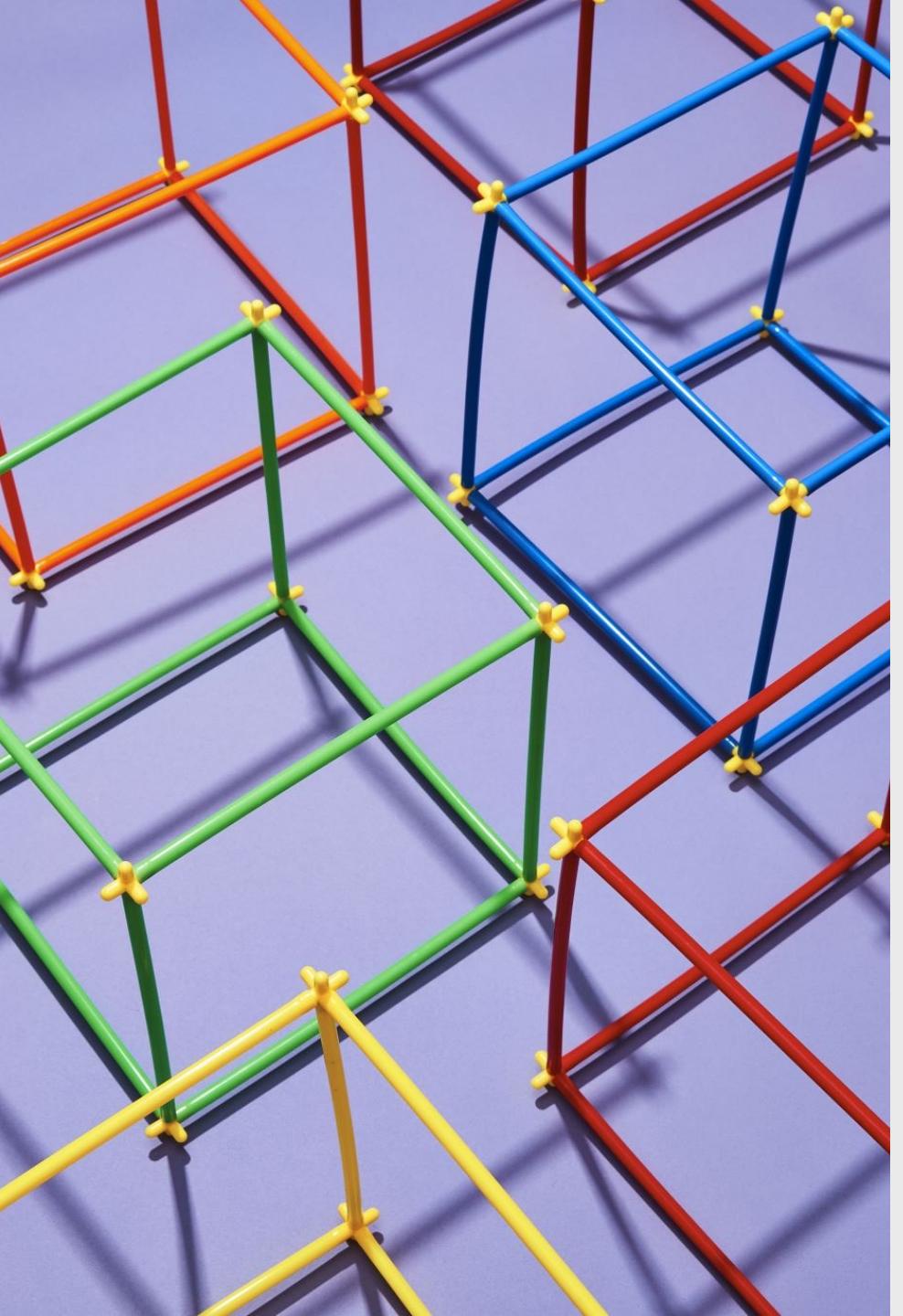
- Syntactic parsing
- Part of speech tagging
- Language modeling

Natalie did not like social events so she politely declined the party invitation.

verb? noun? adjective?

Natalie's tweet had a like within thirty seconds of posting it.

verb? noun? adjective?

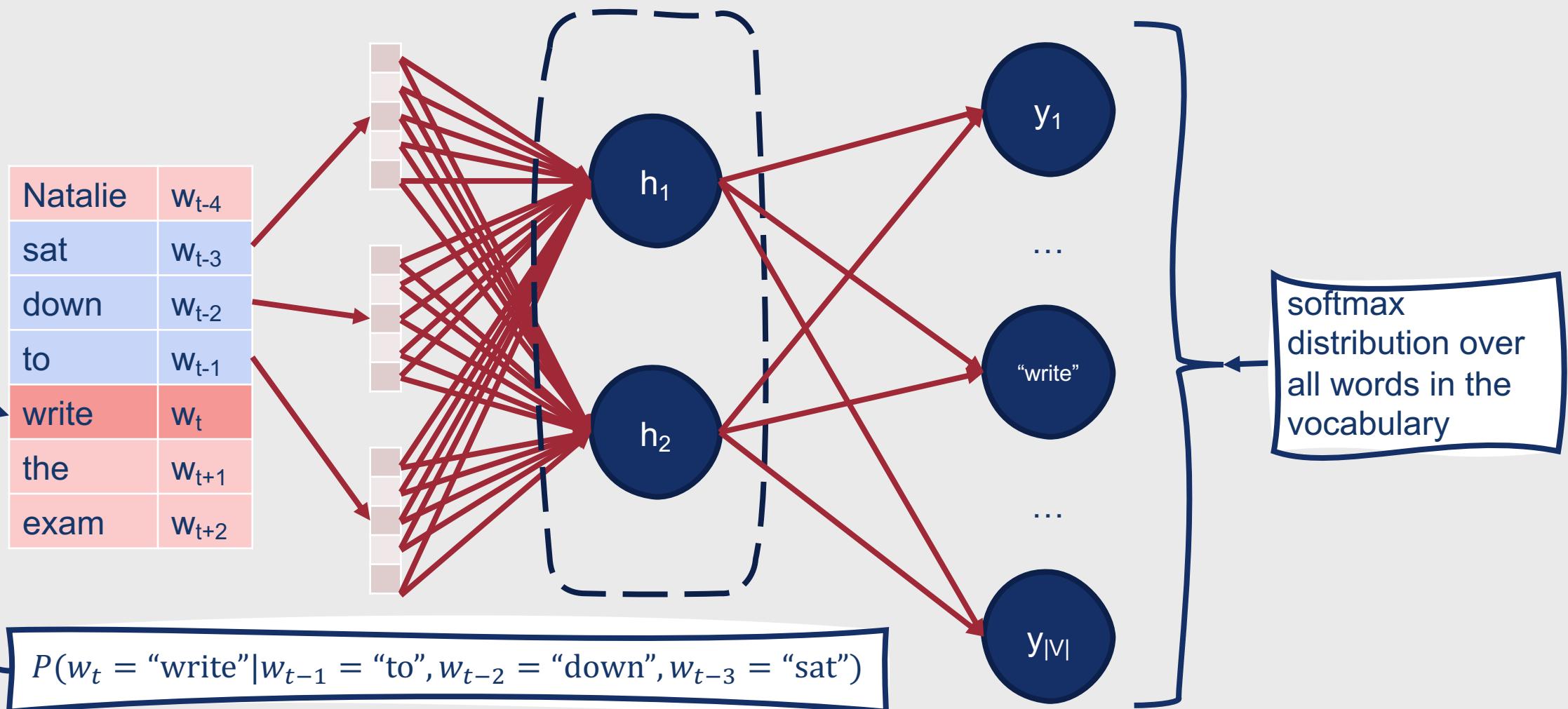


Aren't other neural network models (e.g., feedforward networks) already able to capture temporal information?

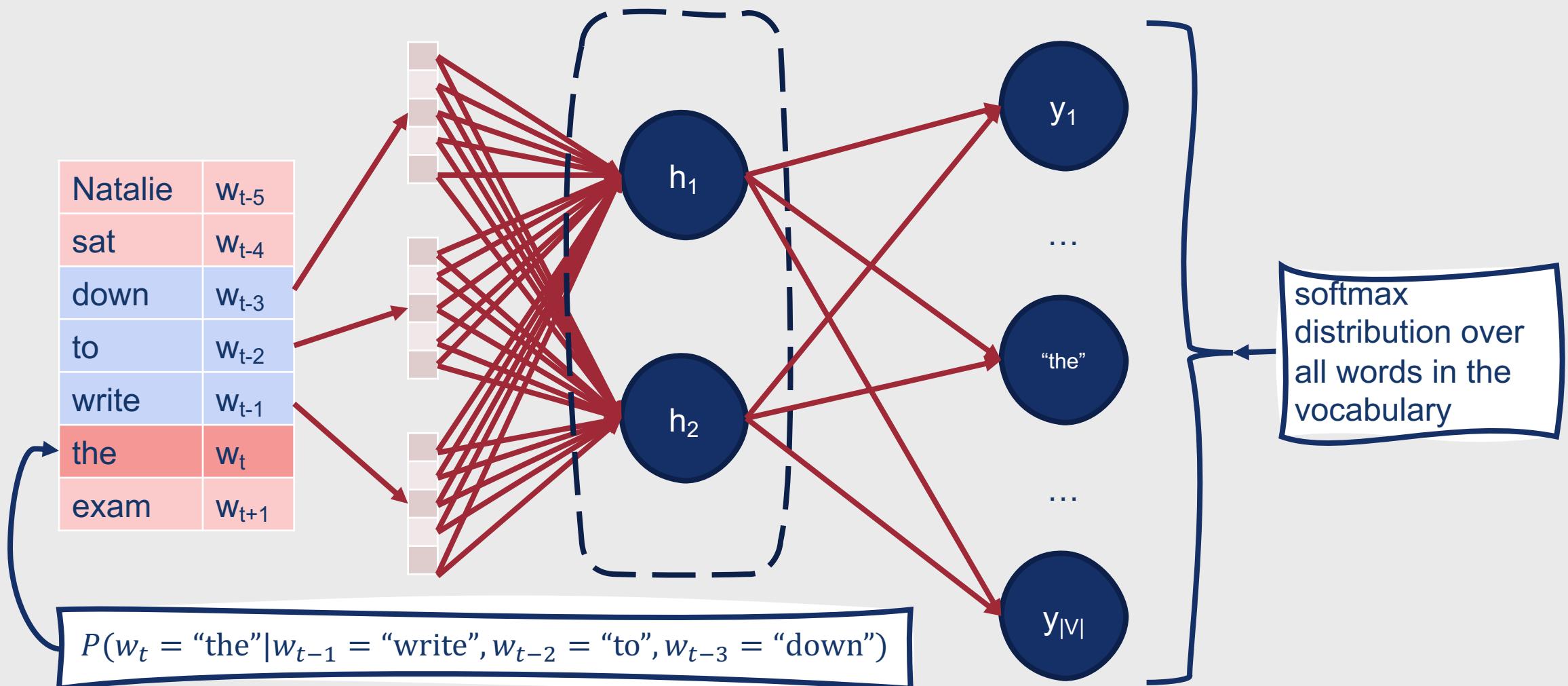
---

- In a sense, yes
- How?
  - **Sliding window approach**

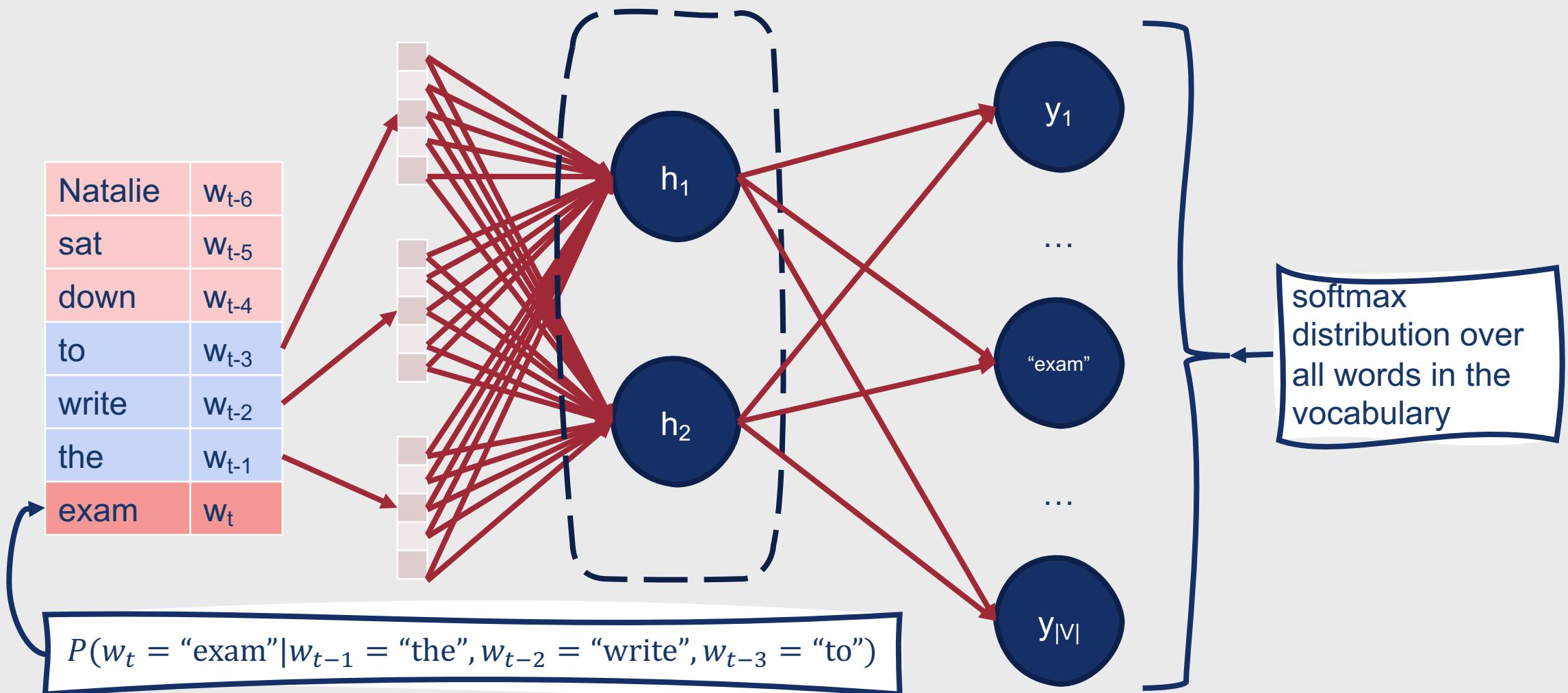
# Sliding Window Approach



# Sliding Window Approach

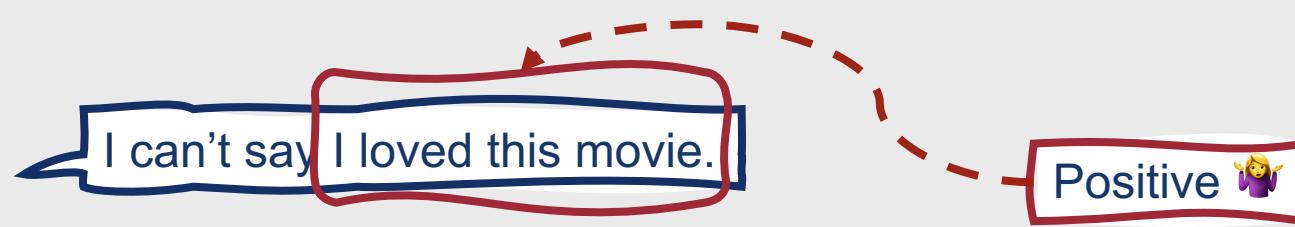


# Sliding Window Approach



# However, this method has some limitations.

- **Constrains the context** from which information can be extracted
  - Only items within the predetermined context window can impact the model's decision
- Makes it difficult to learn **systematic patterns**
  - Particularly problematic when learning grammatical information (e.g., constituent parses)



# However, this method has some limitations.

- **Constrains the context** from which information can be extracted
  - Only items within the predetermined context window can impact the model's decision
- Makes it difficult to learn **systematic patterns**
  - Particularly problematic when learning grammatical information (e.g., constituent parses)

I can't say I loved this movie.

I can't say I loved this movie.

I can't say I loved this movie.

**Recurrent neural  
networks (RNNs) are  
designed to overcome  
these limitations.**

- Built-in capacity to handle temporal information
- Can accept variable length inputs without the use of fixed-size windows

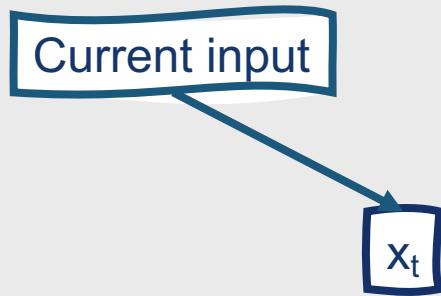


# Recurrent Neural Networks

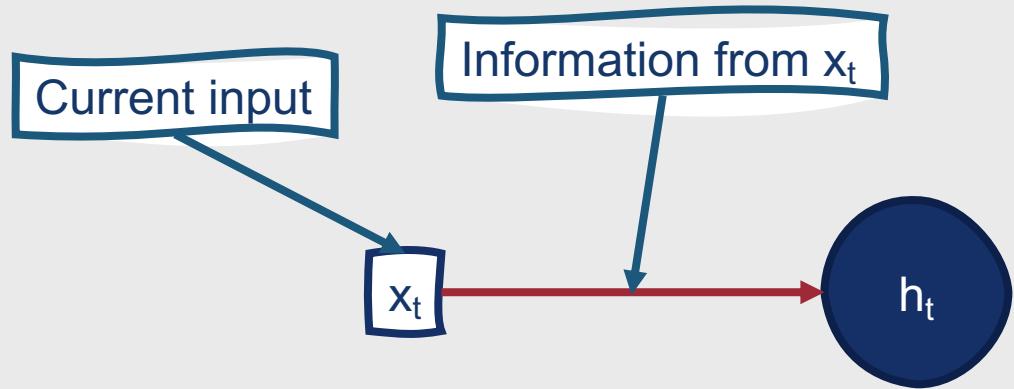


- Contain cycles within their connections
  - The value of a unit is dependent upon outputs from previous timesteps
- Many varieties exist
  - “Vanilla” RNNs
  - Long short-term memory networks (LSTMs)
  - Gated recurrent units (GRUs)

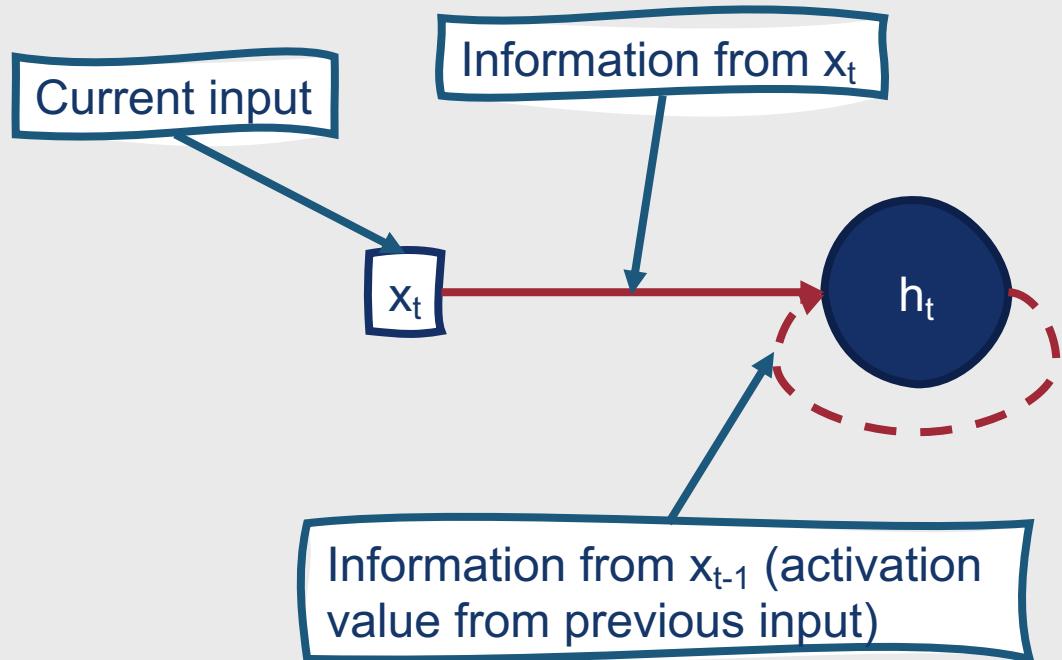
# Vanilla RNN Layer



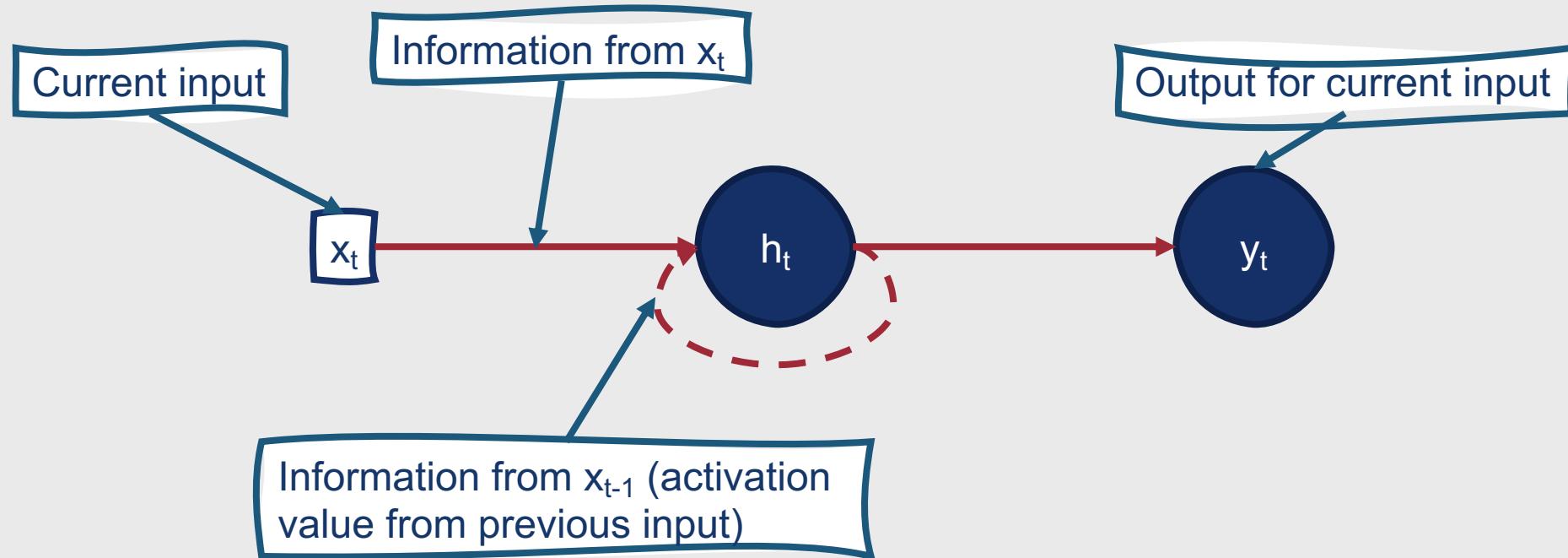
# Vanilla RNN Layer



# Vanilla RNN Layer



# Vanilla RNN Layer



# Thus, hidden layers in RNNs are more complex than in feedforward networks.

---

Outputs from earlier timesteps serve as additional context

---

Makes decisions based on both current input and outputs from prior timesteps

---

Can include information extending all the way back to the beginning of the sequence

# However, computation units still perform the same core actions.

Given:

- Input vector
- (New!) activation values for the hidden layer from the previous timestep

Compute:

- Weighted sum of inputs

# Most Significant Change

- New set of weights that connect the hidden layer from the previous timestep to the current hidden layer
- These weights determine how the network should make use of prior context

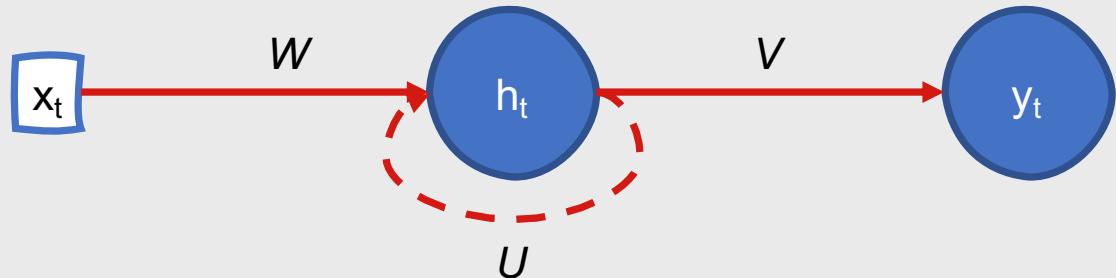


# Formal Equations

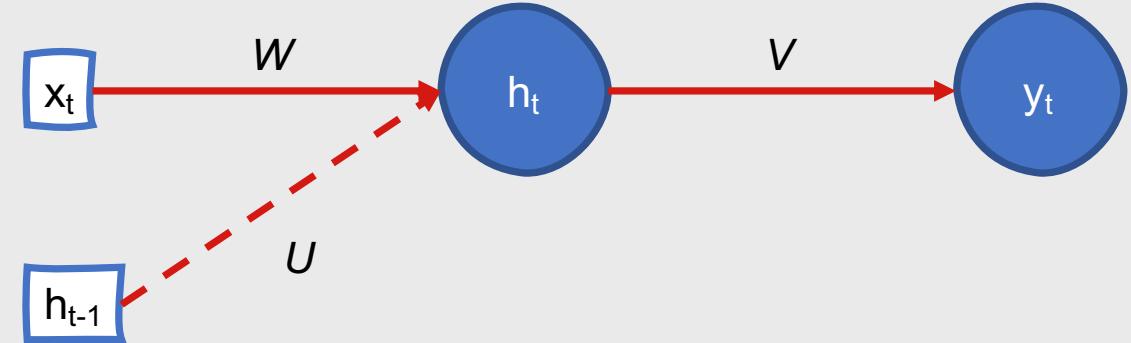
- Similar to what we've seen with feedforward networks
- Recall the basic set of equations for a feedforward neural network:
  - $\mathbf{h} = \sigma(\mathbf{Wx} + \mathbf{b})$
  - $\mathbf{z} = \mathbf{Uh}$
  - $y = \text{softmax}(\mathbf{z})$

# Formal Equations

- Just add (weights X activation values from previous timestep) product to the current (weights X inputs) product
  - $\mathbf{h} = \sigma(W\mathbf{x}_t + U\mathbf{h}_{t-1} + \mathbf{b})$
  - $\mathbf{z} = V\mathbf{h}_t$
  - $y = \text{softmax}(\mathbf{z})$
- $W$ ,  $U$ , and  $V$  are shared across all timesteps



Recurrent View



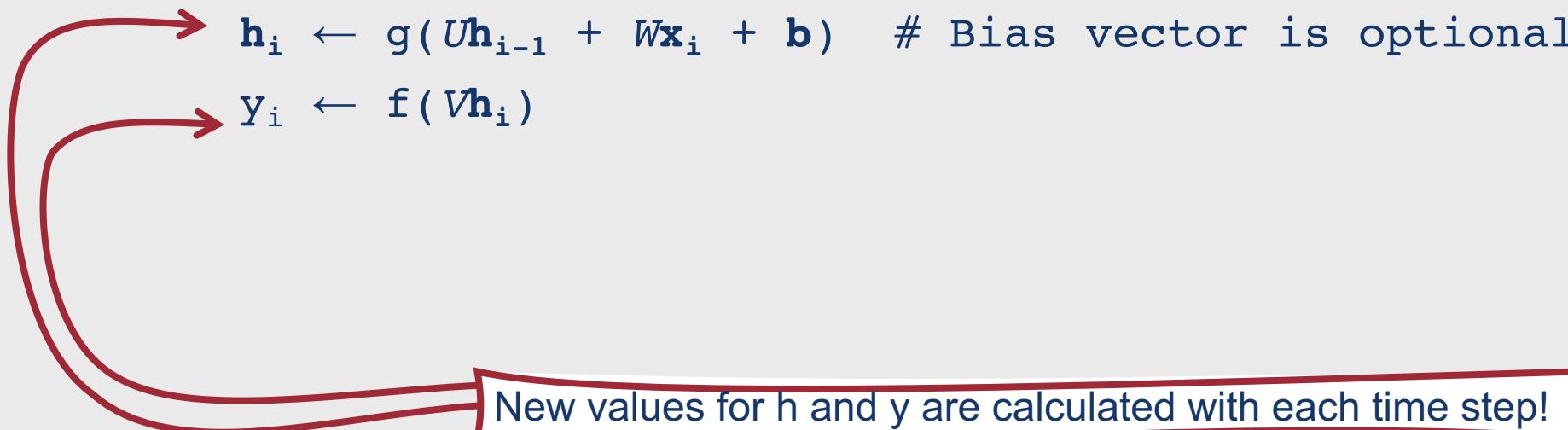
Unrolled View

What does this look like  
when unrolled?

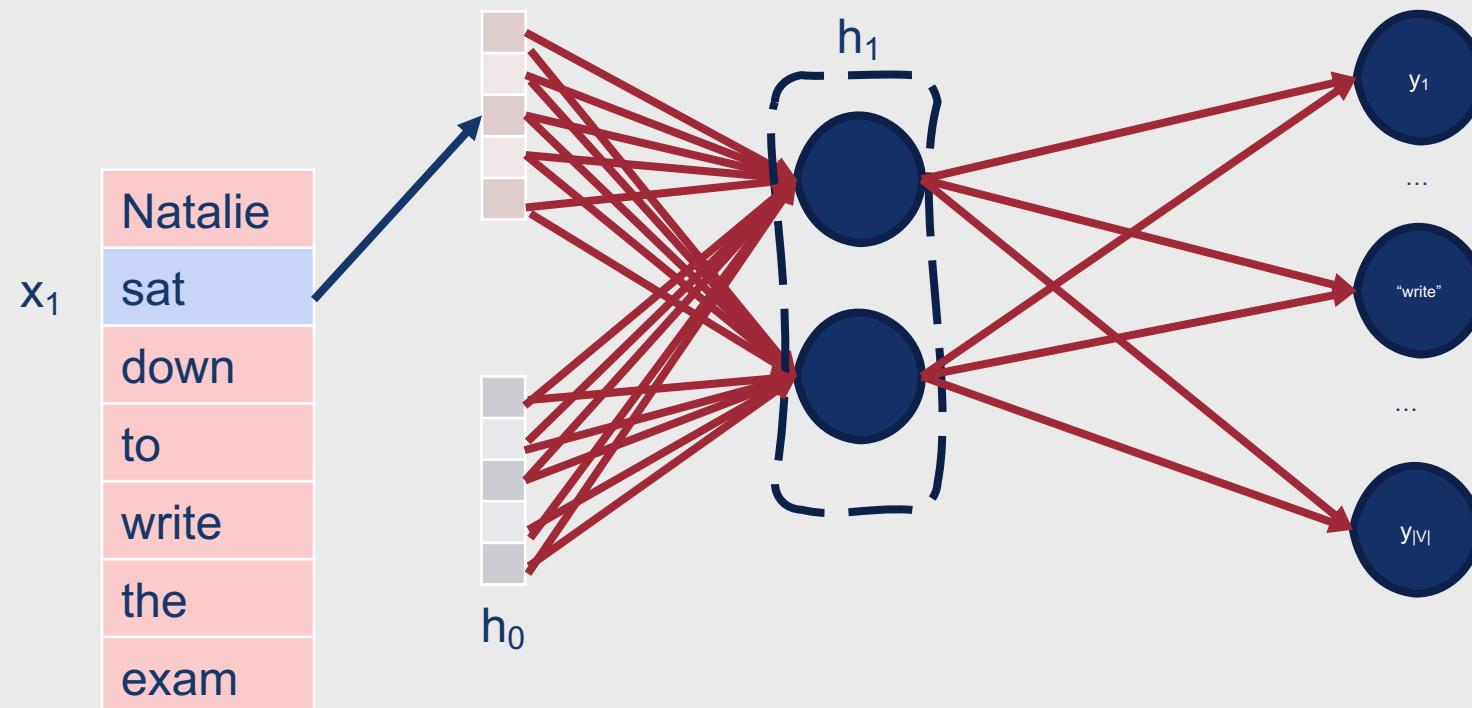


# Formal Algorithm

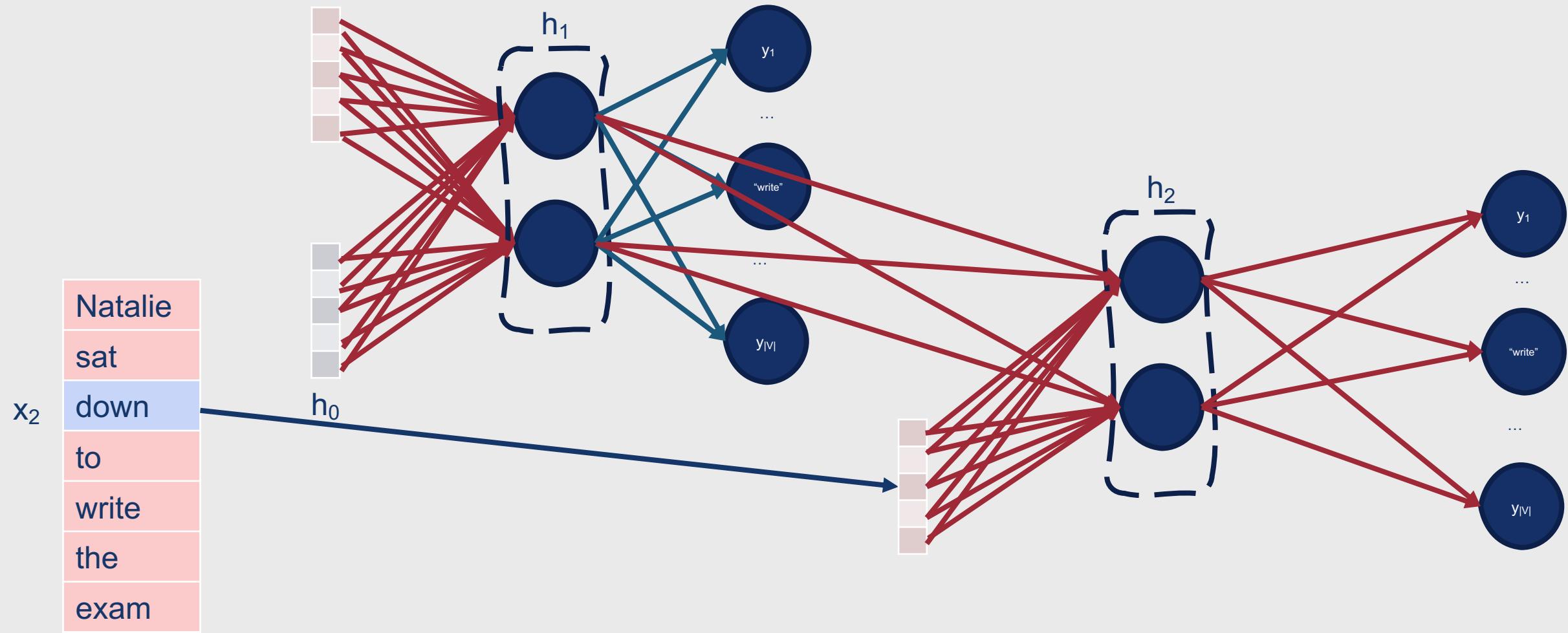
```
h0 ← 0 # Initialize activations from the hidden layer to 0  
  
# Iterate through each input element in temporal order  
for i ← 1 to length(x) do:  
    hi ← g(Uhi-1 + Wxi + b) # Bias vector is optional  
    yi ← f(Vhi)
```



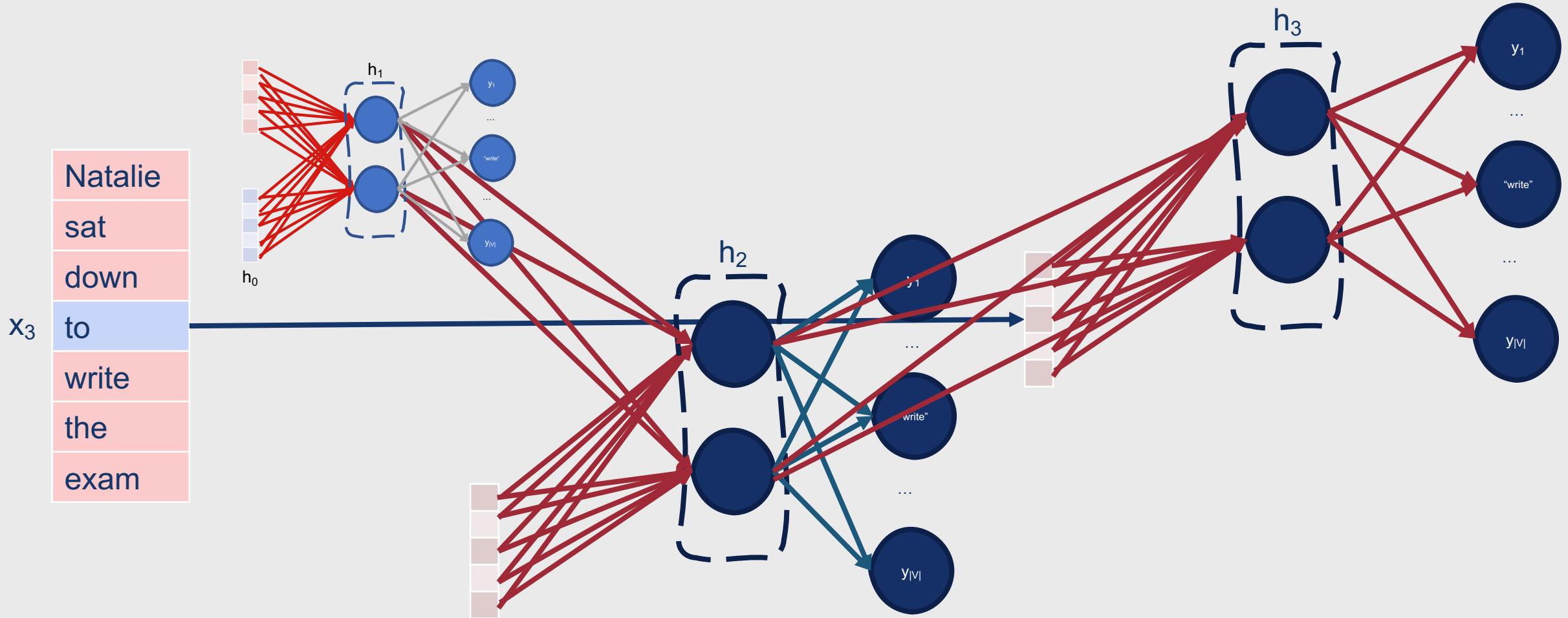
# Earlier Example: RNN Edition



# Earlier Example: RNN Edition



# Earlier Example: RNN Edition

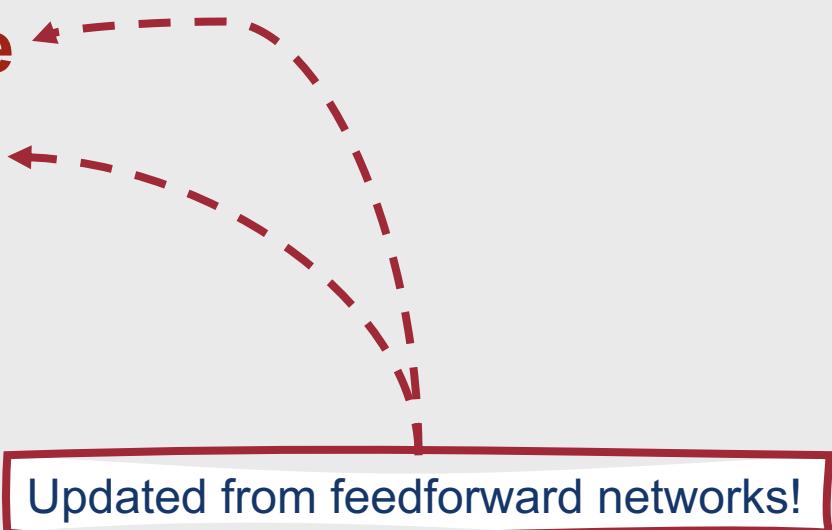


# Training RNNs

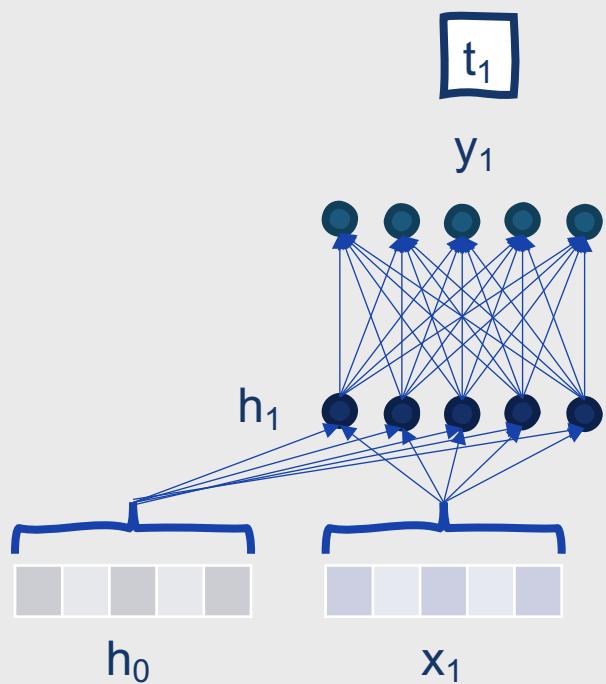
- Same core elements:
  - Loss function
  - Optimization function
  - Backpropagation
- One extra set of weights to update
  - Hidden layer from  $t-1$  to current hidden layer at  $t$

# Forward Inference

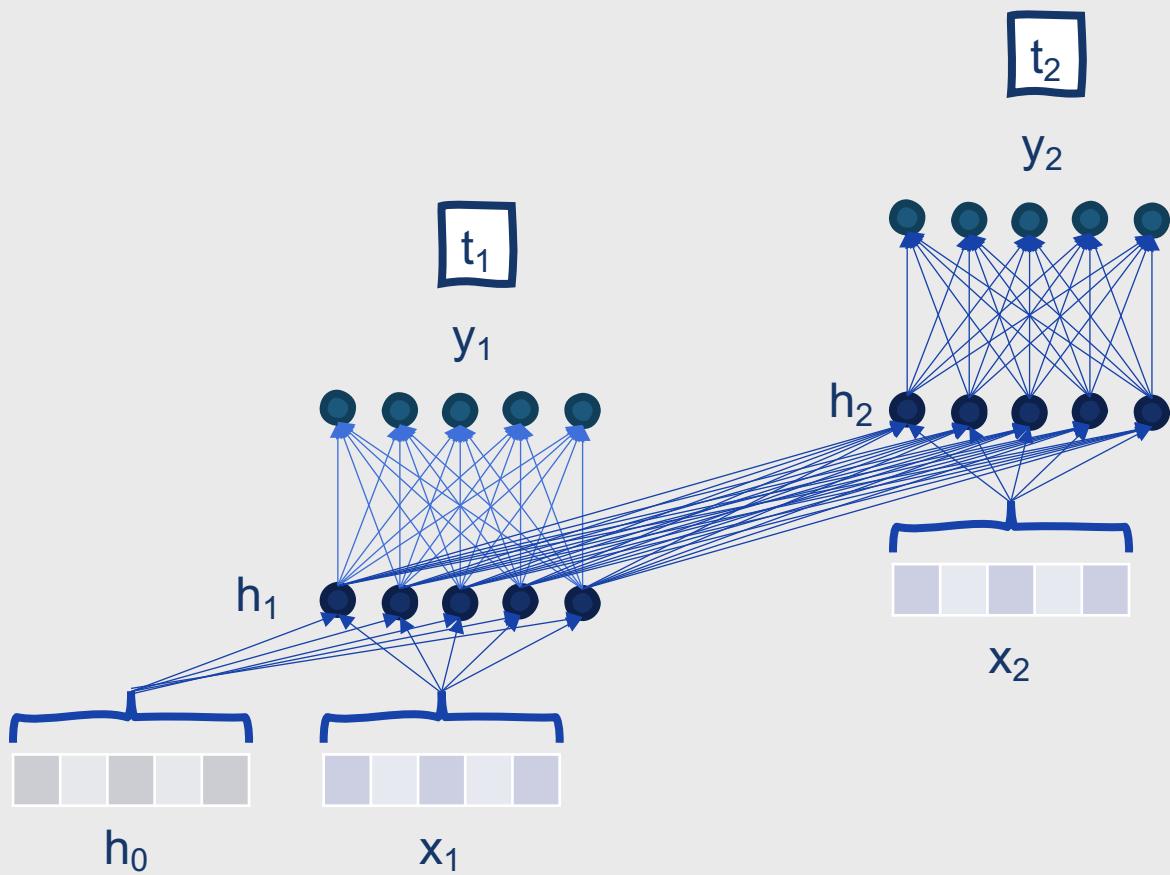
- Compute  $h_t$  and  $y_t$  at **each step in time**
- Compute the loss at **each step in time**



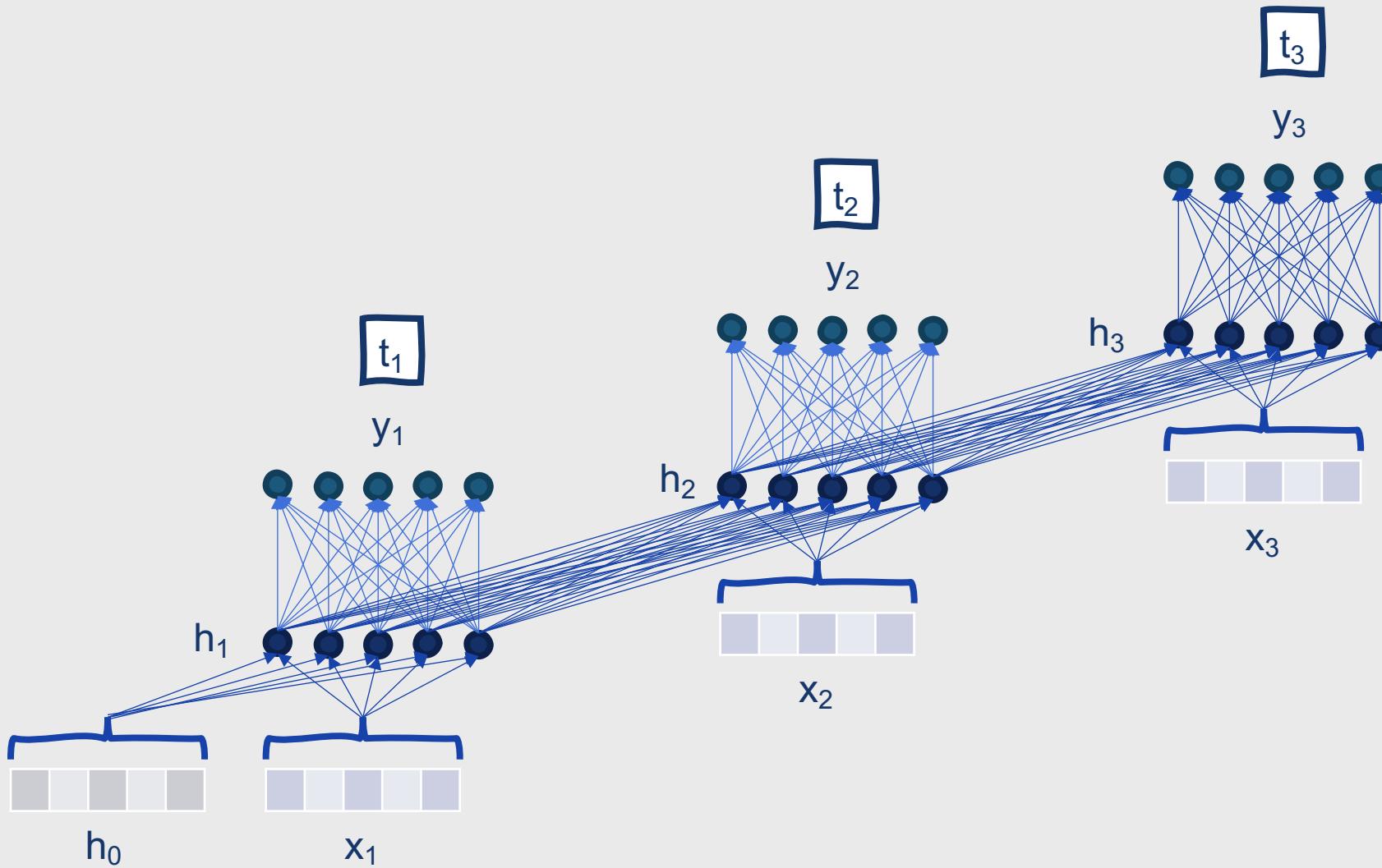
# Forward Pass



# Forward Pass



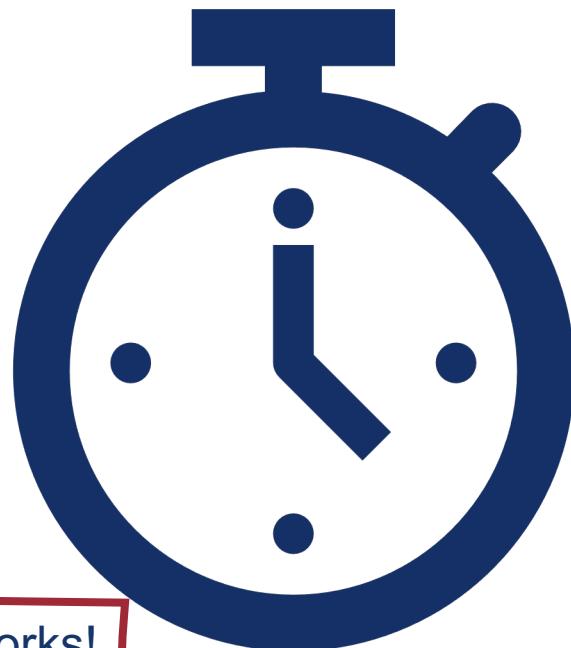
# Forward Pass



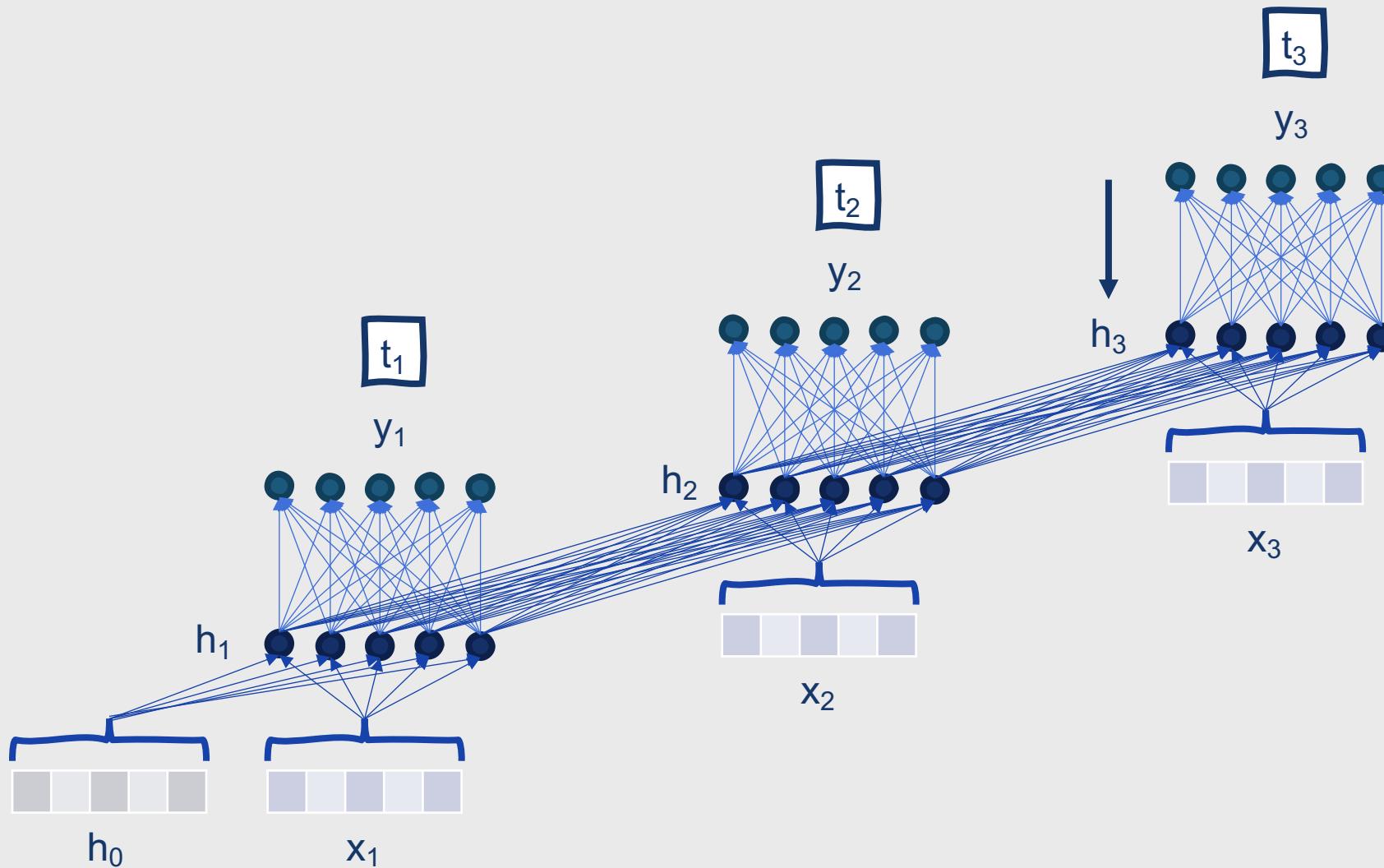
# Backpropagation Through Time

- Process the sequence in reverse
- Compute the required error gradients at **each step backward in time**

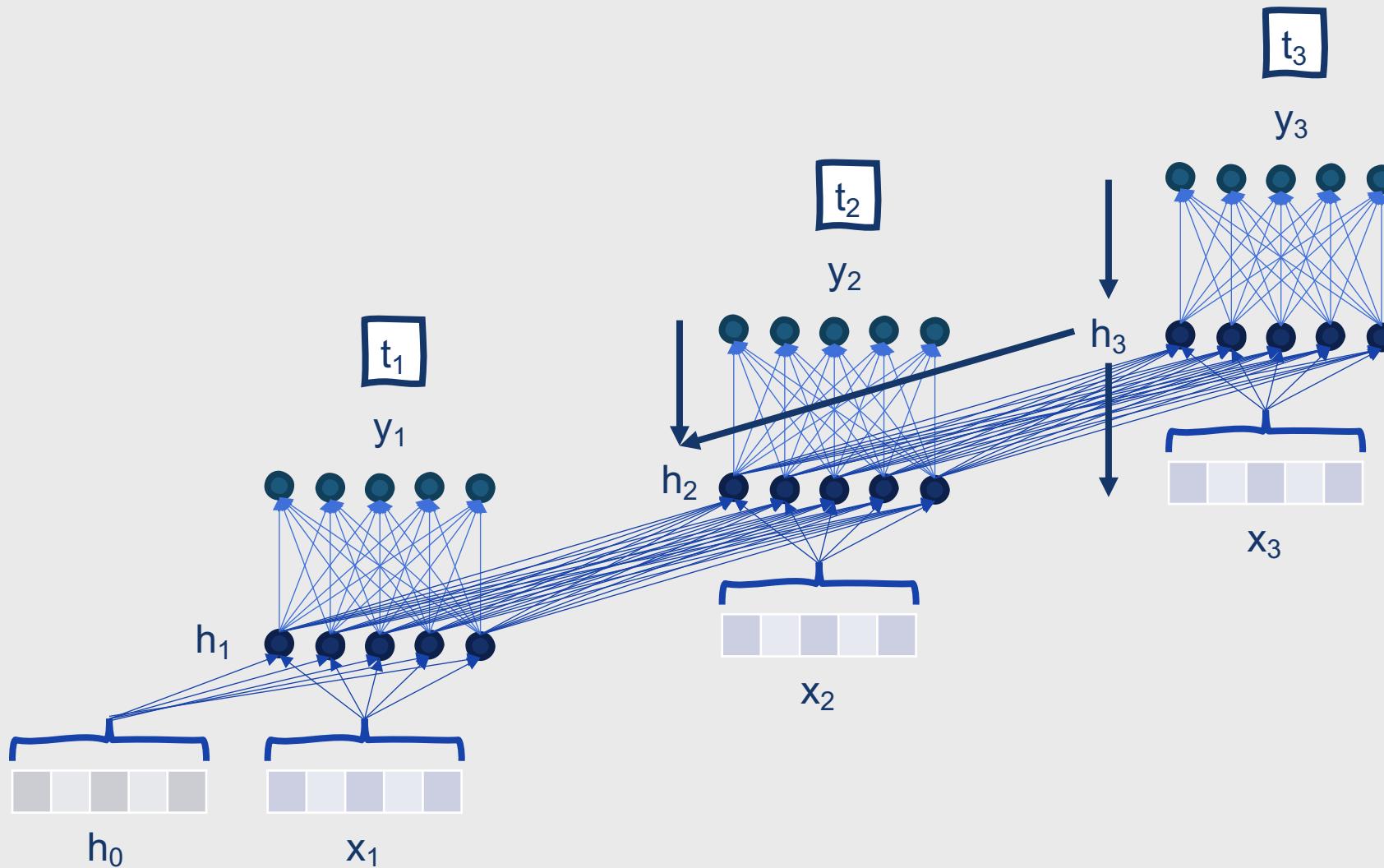
Updated from feedforward networks!



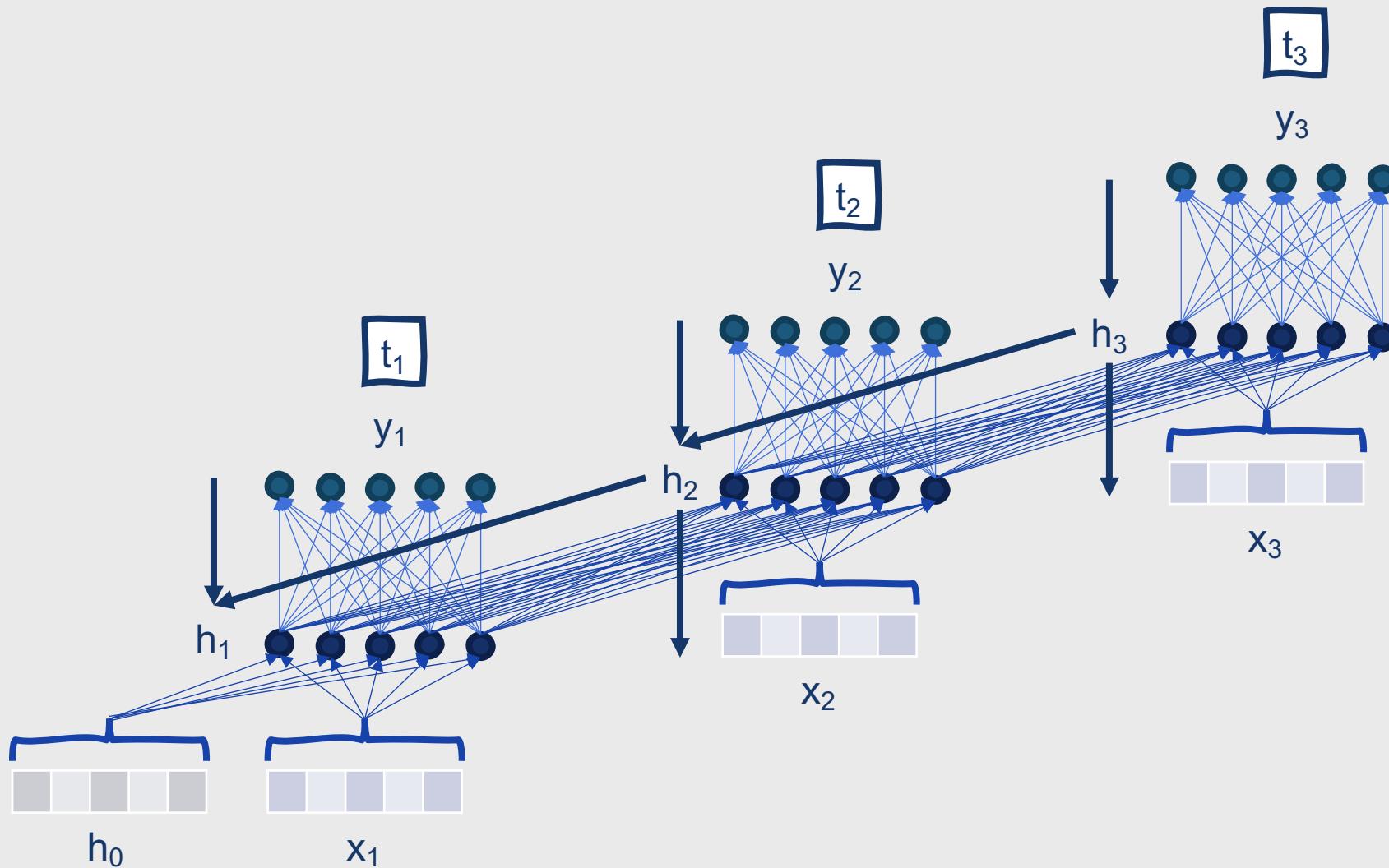
# Backward Pass



# Backward Pass



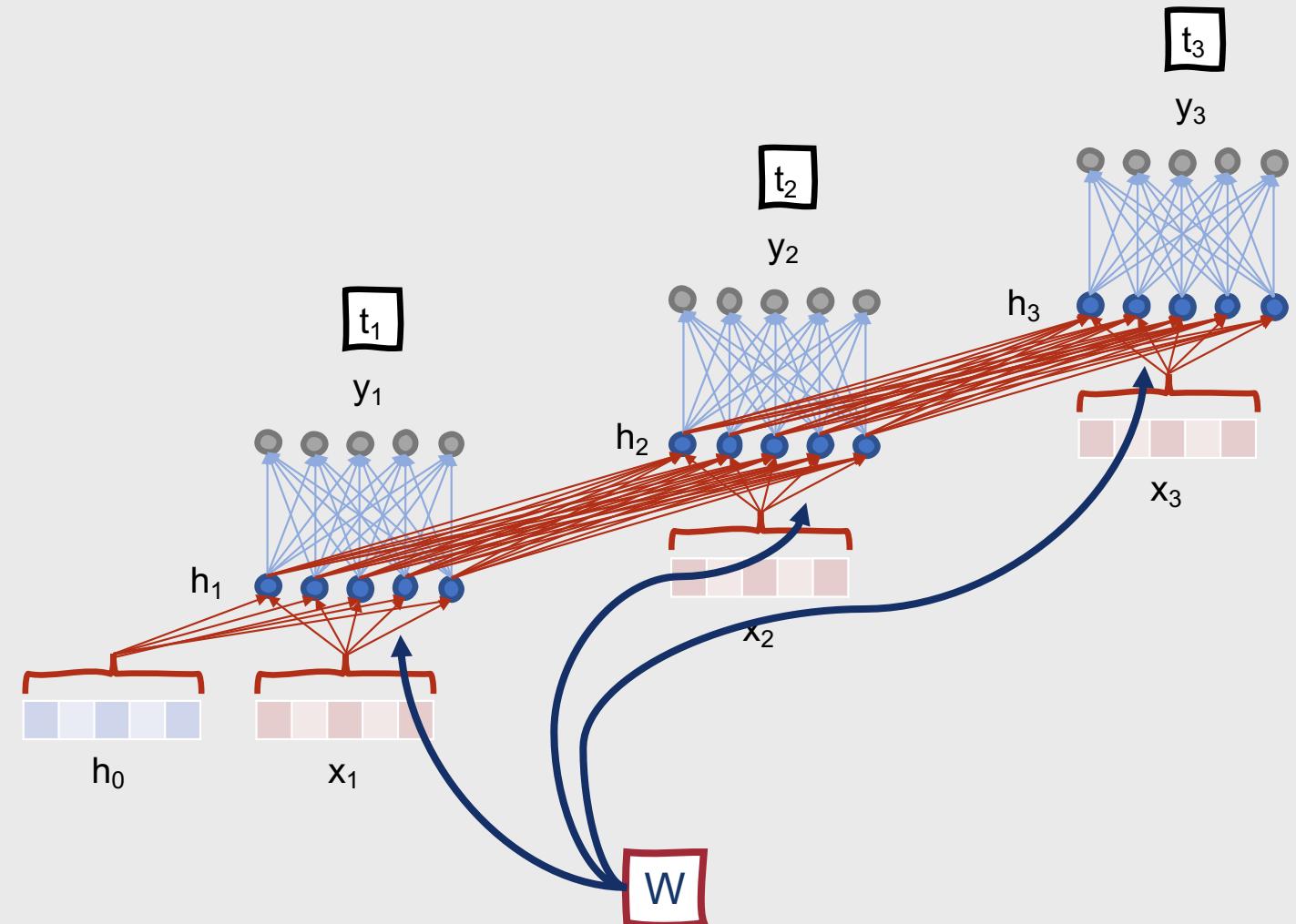
# Backward Pass





# Updated Backpropagation Equations

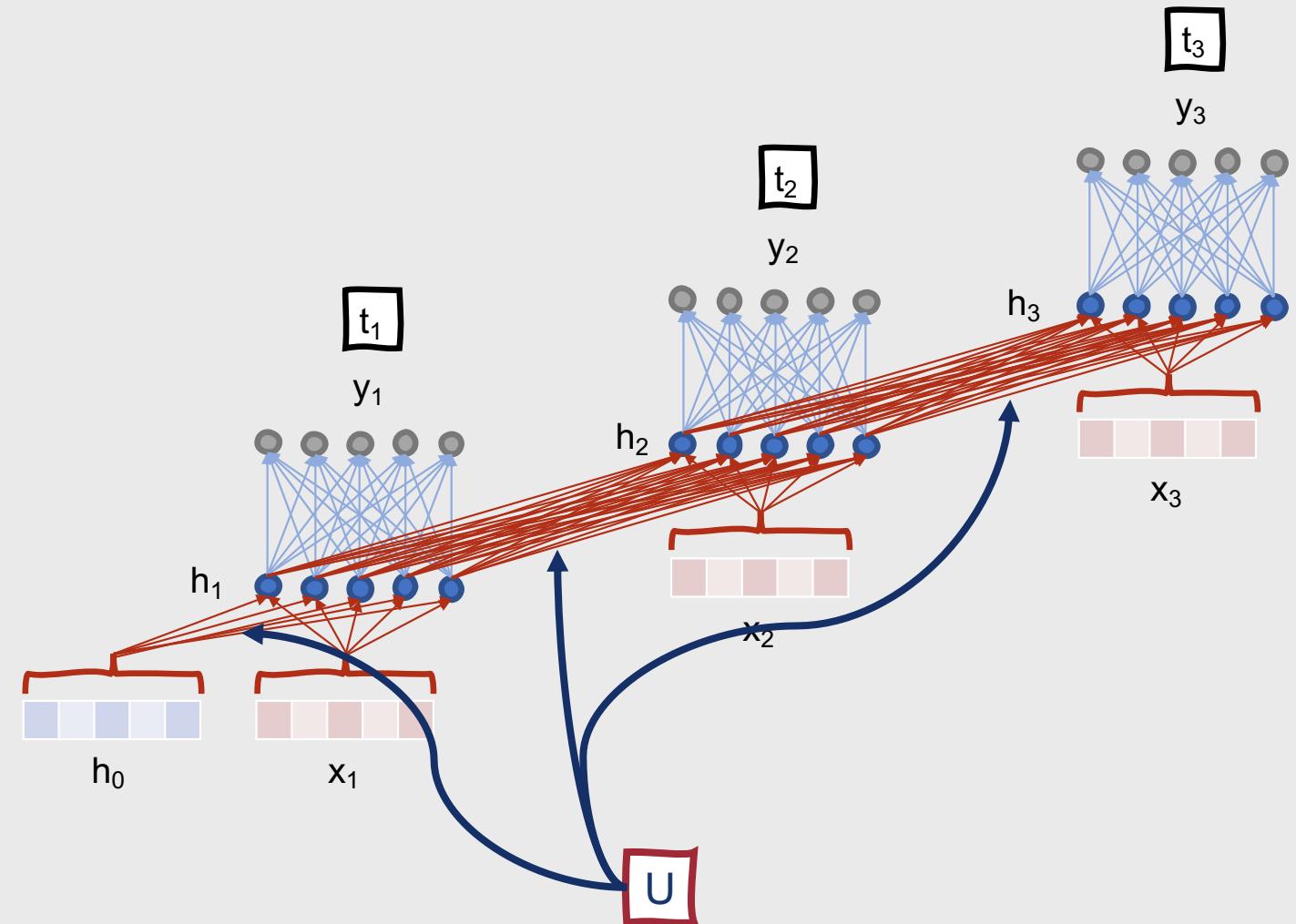
- Now we have three sets of weights we need to update
  - $W$ , the weights from the input layer to the hidden layer
  - $U$ , the weights from the previous hidden layer to the current hidden layer
  - $V$ , the weights from the hidden layer to the output layer





# Updated Backpropagation Equations

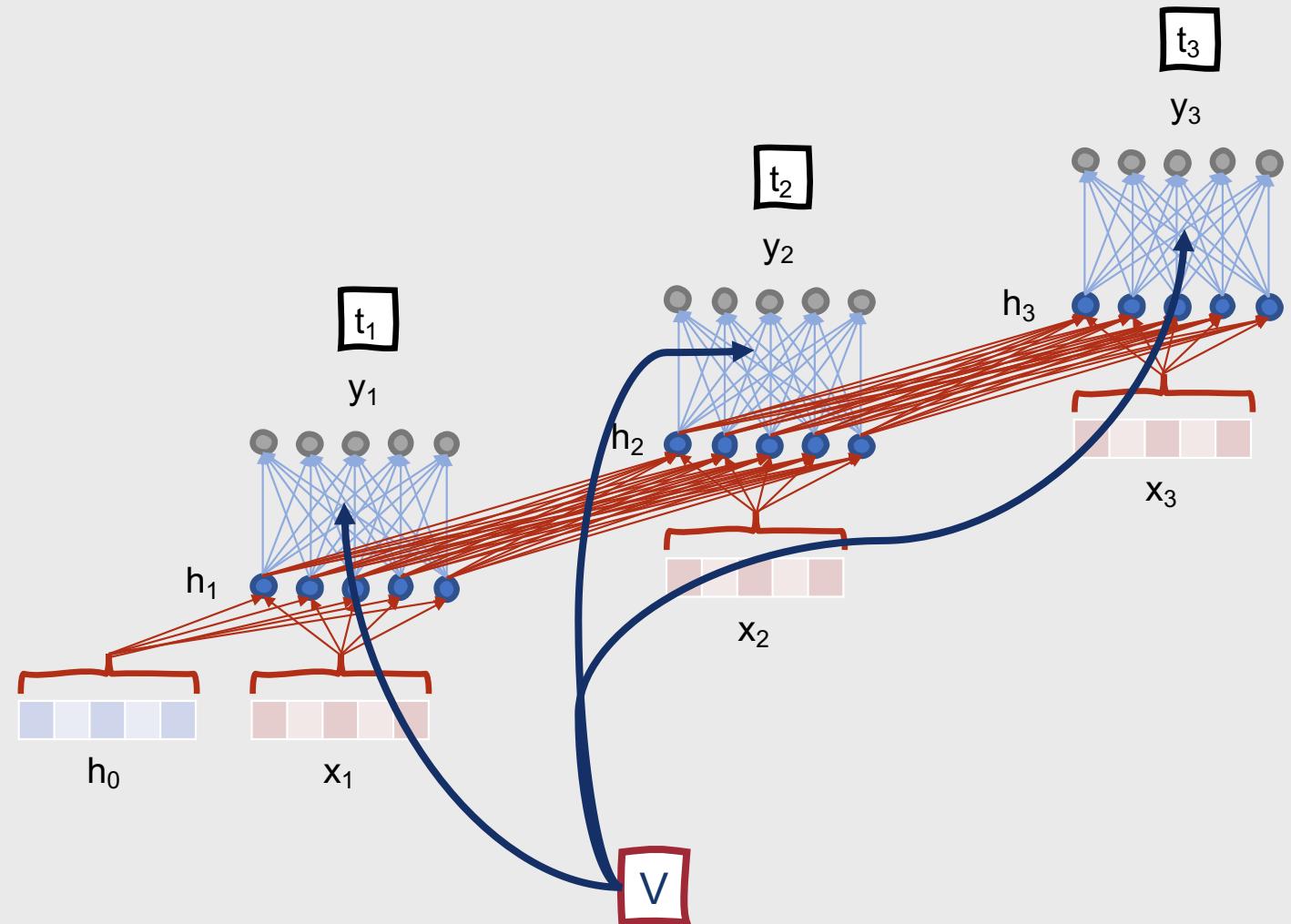
- Now we have three sets of weights we need to update:
  - $W$ , the weights from the input layer to the hidden layer
  - $U$ , the weights from the previous hidden layer to the current hidden layer
  - $V$ , the weights from the hidden layer to the output layer





# Updated Backpropagation Equations

- Now we have three sets of weights we need to update:
  - $W$ , the weights from the input layer to the hidden layer
  - $U$ , the weights from the previous hidden layer to the current hidden layer
  - $V$ , the weights from the hidden layer to the output layer



**Updating the  
weights for  $V$   
works no  
differently from  
feedforward  
networks.**

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a} \frac{\partial a}{\partial z} \frac{\partial z}{\partial V}$$

Chain rule

# Updating the weights for $W$ and $U$ works a little bit differently.

- Error term for a hidden layer,  $\delta_h$ , must be the sum of the error term from the current output and the error term from the next timestep
  - $\delta_h = g'(z)V\delta_t + \delta_{t+1}$

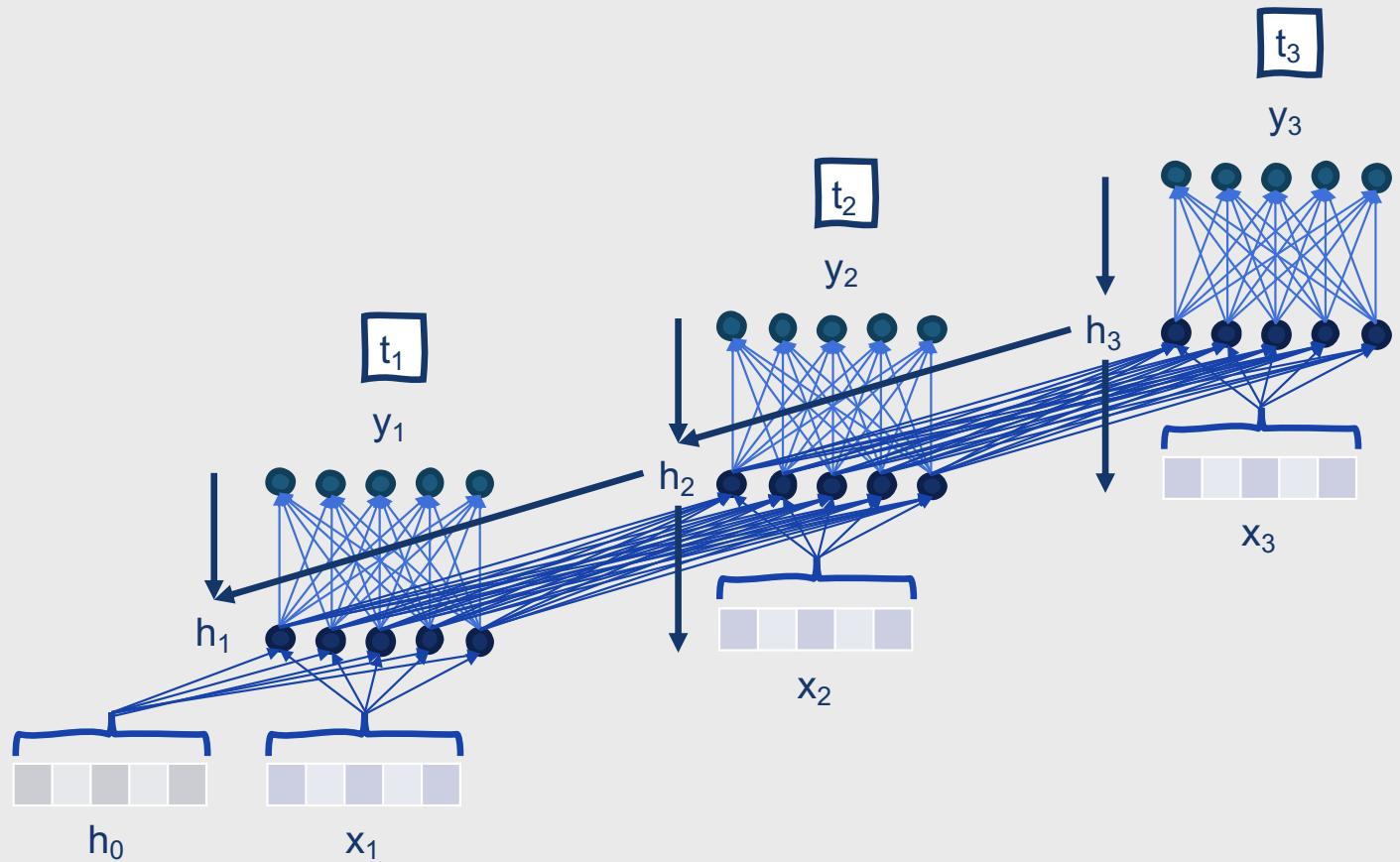
Once we have this updated error term for the hidden layer, we can proceed as usual to compute the gradients for  $U$  and  $W$ .

- $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial a} \frac{\partial a}{\partial W} = \delta_h x_t$
- $\frac{\partial L}{\partial U} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial a} \frac{\partial a}{\partial U} = \delta_h h_{t-1}$



# One remaining step....

- Backpropagate the error from  $\delta_h$  to  $h_{t-1}$  based on the weights in  $U$ 
  - $\delta_{t+1} = g'(z)U\delta_h$
- At this point, we have all of the necessary gradients to update  $U$ ,  $V$ , and  $W$ !



**At this point,  
we've seen a  
few types of  
language  
models.**

- N-gram language models
- Feedforward neural network language models

**These models attempt to predict the next word in a sequence given a prior context of fixed length.**

- What's challenging about this approach?
  - Model quality is dependent on context size
  - Anything outside the fixed context window has no impact on the model's decision

# Recurrent Neural Language Models

- Recurrent neural language models process sequences one word at a time
- This means that they **avoid constraining the context size**
- The **hidden state embodies information about all of the preceding words**, all the way back to the beginning of the sequence

# Recurrent Neural Language Models

- At each timestep:
  1. **Retrieve an embedding** for the current input word
  2. **Combine the weighted sums** of (a) the input embedding values and (b) the activations of the hidden layer from the previous step, to compute a new set of activation values from the hidden layer
  3. **Generate a set of outputs** based on the activations from the hidden layer
  4. **Pass the outputs through a softmax function** to generate a probability distribution over the entire vocabulary

# Recurrent Neural Language Models

- At each timestep:
  1. **Retrieve an embedding** for the current input word
  2. **Combine the weighted sums** of (a) the input embedding values and (b) the activations of the hidden layer from the previous step, to compute a new set of activation values from the hidden layer
  3. **Generate a set of outputs** based on the activations from the hidden layer
  4. **Pass the outputs through a softmax function** to generate a probability distribution over the entire vocabulary

# Recurrent Neural Language Models

- At each timestep:
  1. **Retrieve an embedding** for the current input word
  2. **Combine the weighted sums** of (a) the input embedding values and (b) the activations of the hidden layer from the previous step, to compute a new set of activation values from the hidden layer
  3. **Generate a set of outputs** based on the activations from the hidden layer
  4. **Pass the outputs through a softmax function** to generate a probability distribution over the entire vocabulary

# Recurrent Neural Language Models

- At each timestep:
  1. **Retrieve an embedding** for the current input word
  2. **Combine the weighted sums** of (a) the input embedding values and (b) the activations of the hidden layer from the previous step, to compute a new set of activation values from the hidden layer
  3. **Generate a set of outputs** based on the activations from the hidden layer
  4. **Pass the outputs through a softmax function** to generate a probability distribution over the entire vocabulary

# How can we generate text with neural language models?

**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.

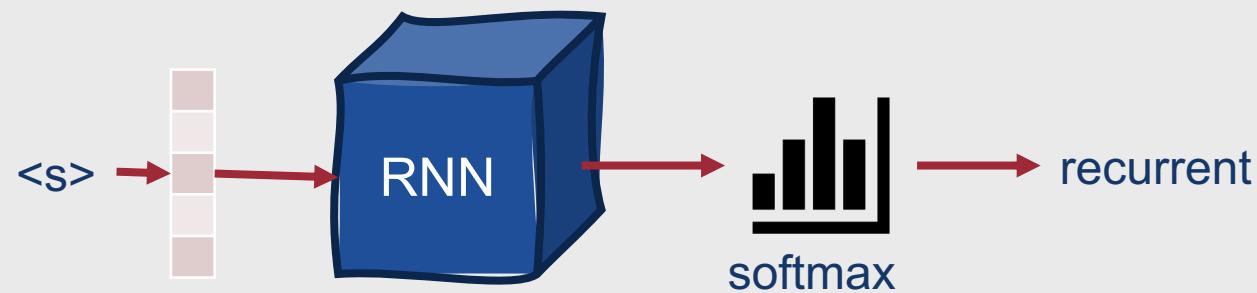
# Generation with Neural Language Models

1. Sample the first word in the output from the softmax distribution that results from using the **beginning of sentence marker** (<s>) as input
2. Get the embedding for that word
3. Use it as input to the network at the next time step, and sample the following word as in (1)
4. Repeat until the **end of sentence marker** (</s>) is sampled, or a fixed length limit is reached

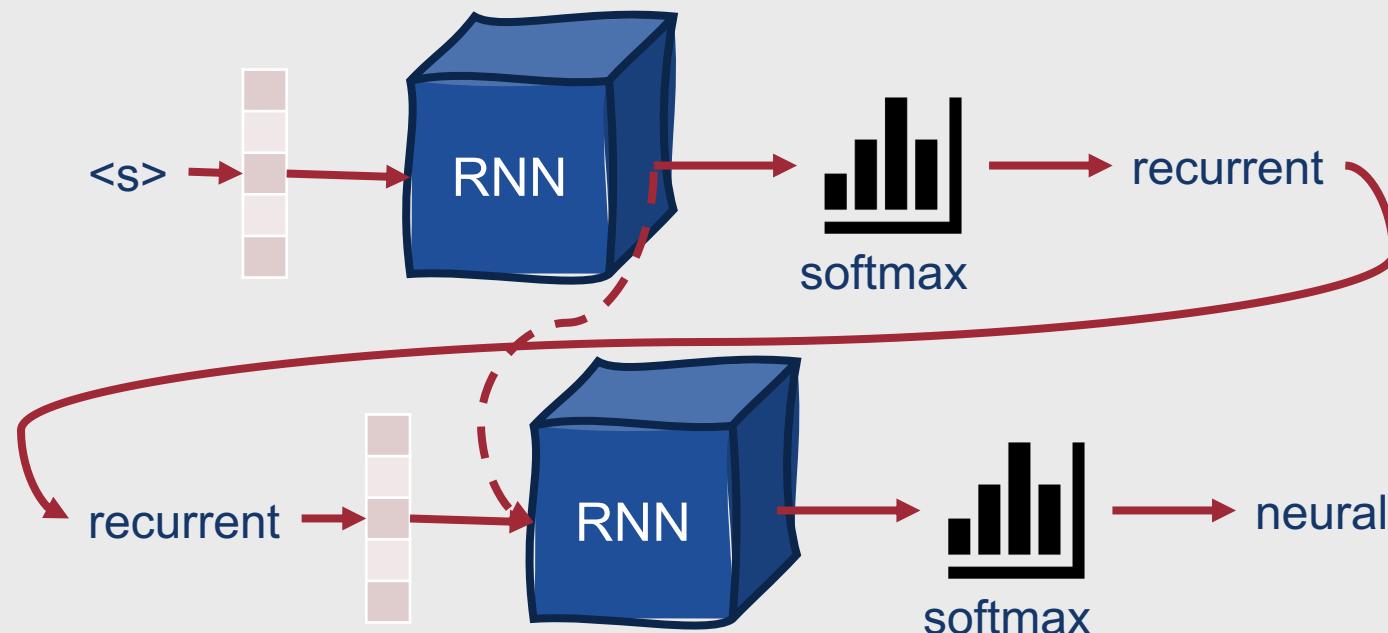
# Autoregressive Generation

- This technique is referred to as **autoregressive generation**
  - Word generated at each timestep is conditioned on the word generated previously by the model

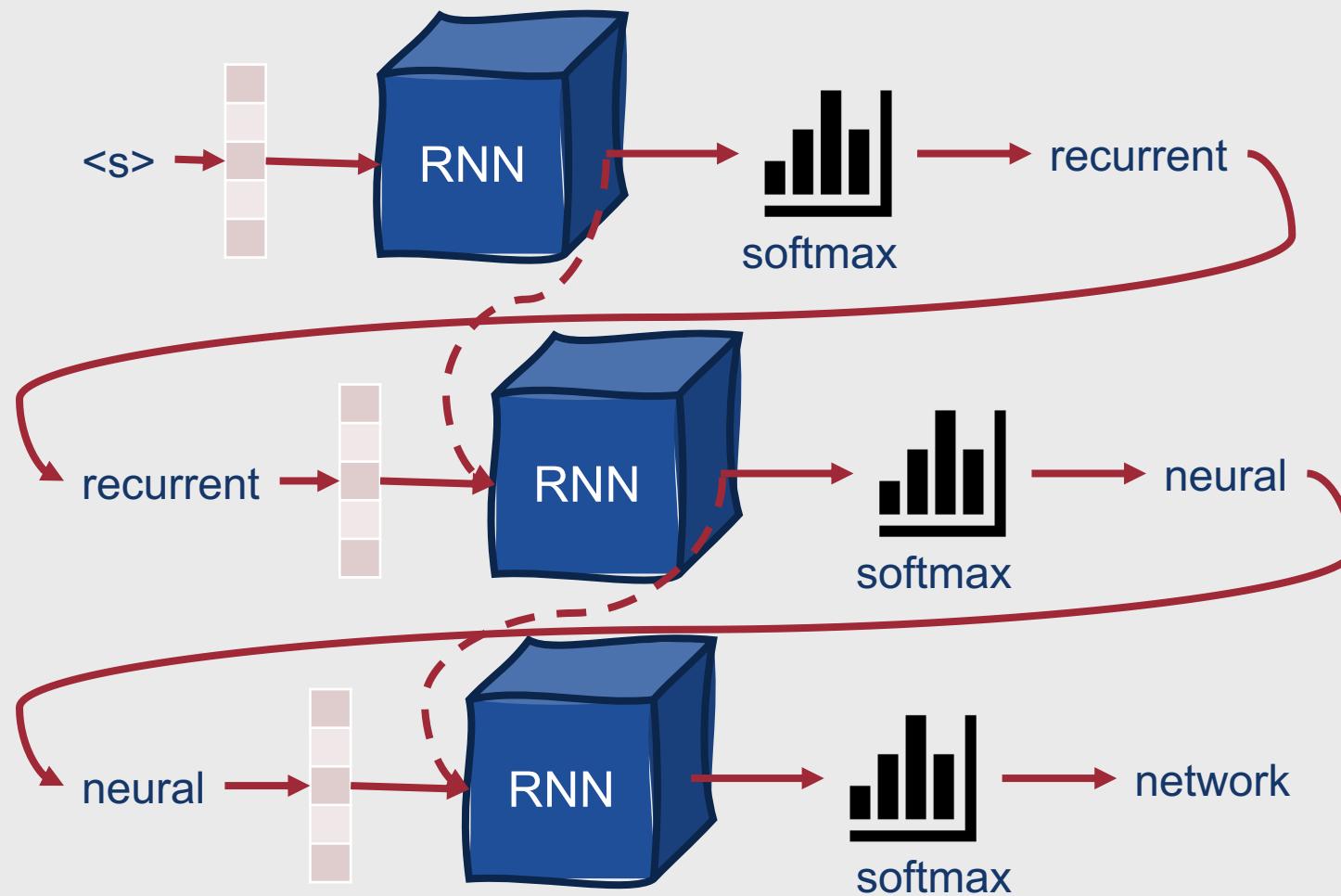
# Autoregressive Generation



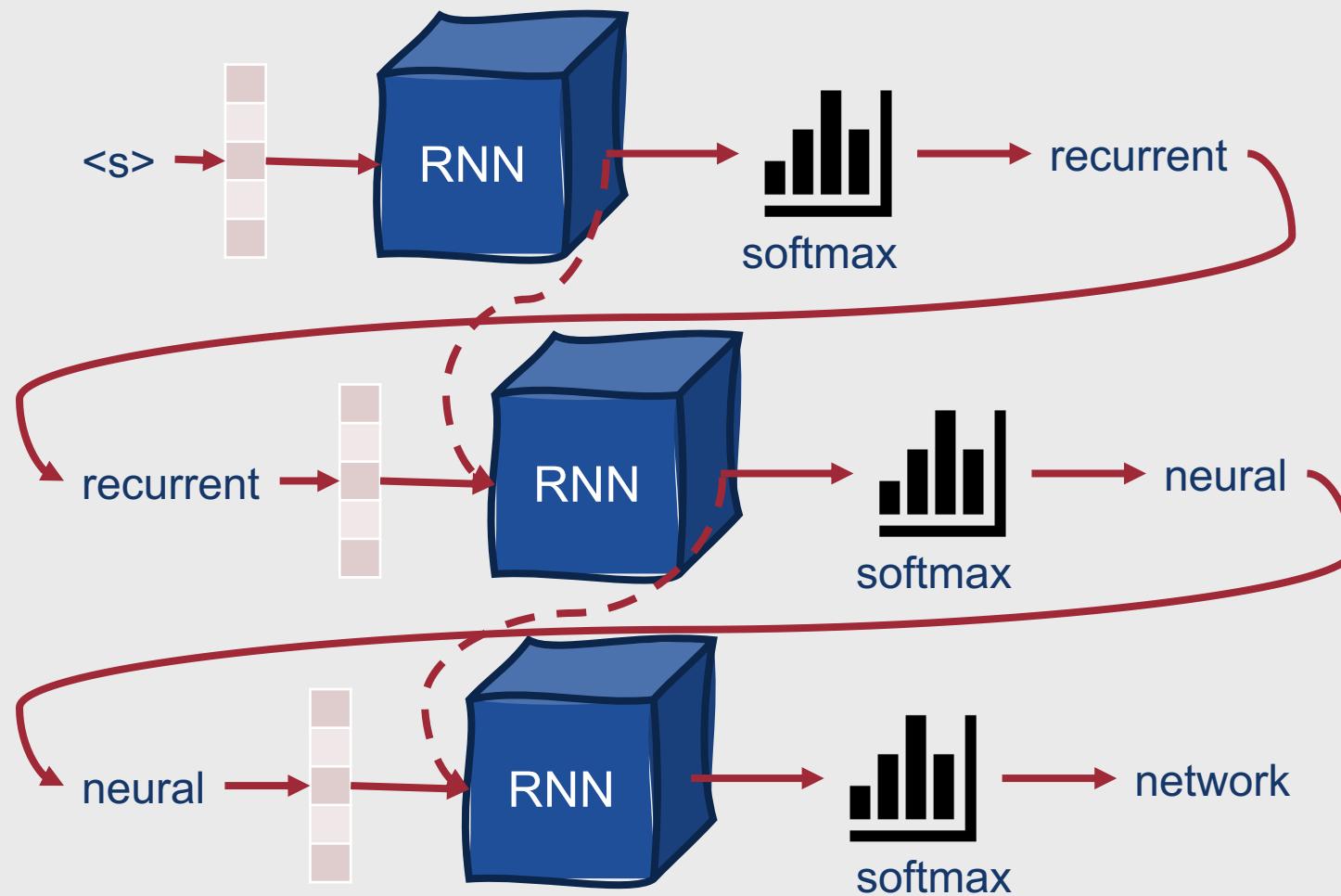
# Autoregressive Generation



# Autoregressive Generation

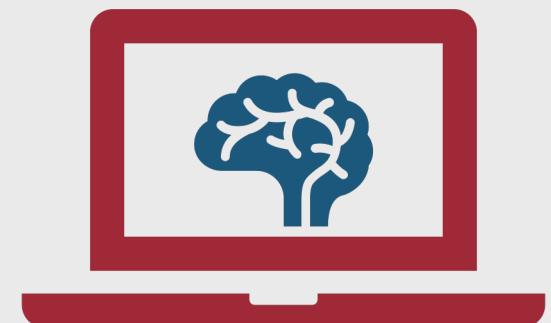


# Autoregressive Generation



**Key to successful autoregressive generation?**

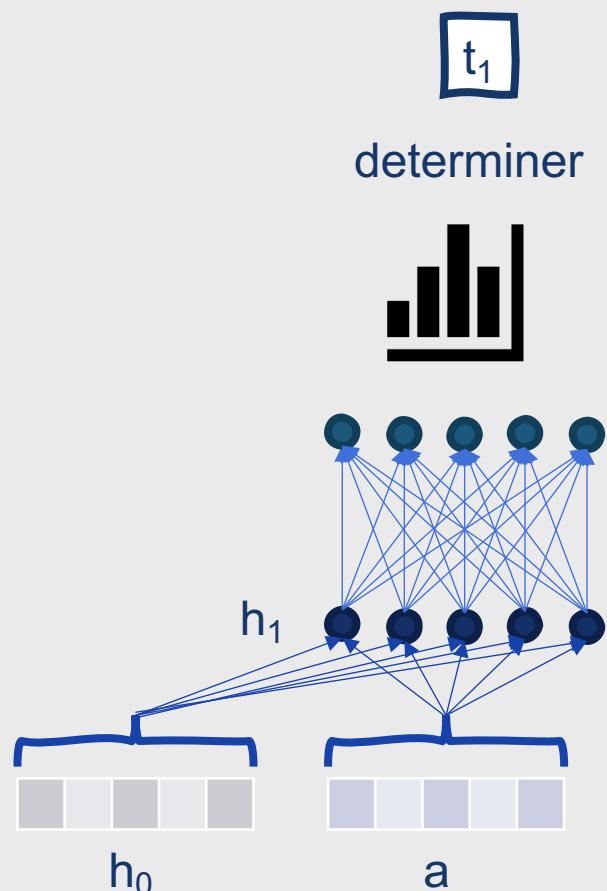
Prime the generation component with **appropriate context** (e.g., something more useful than <s>)



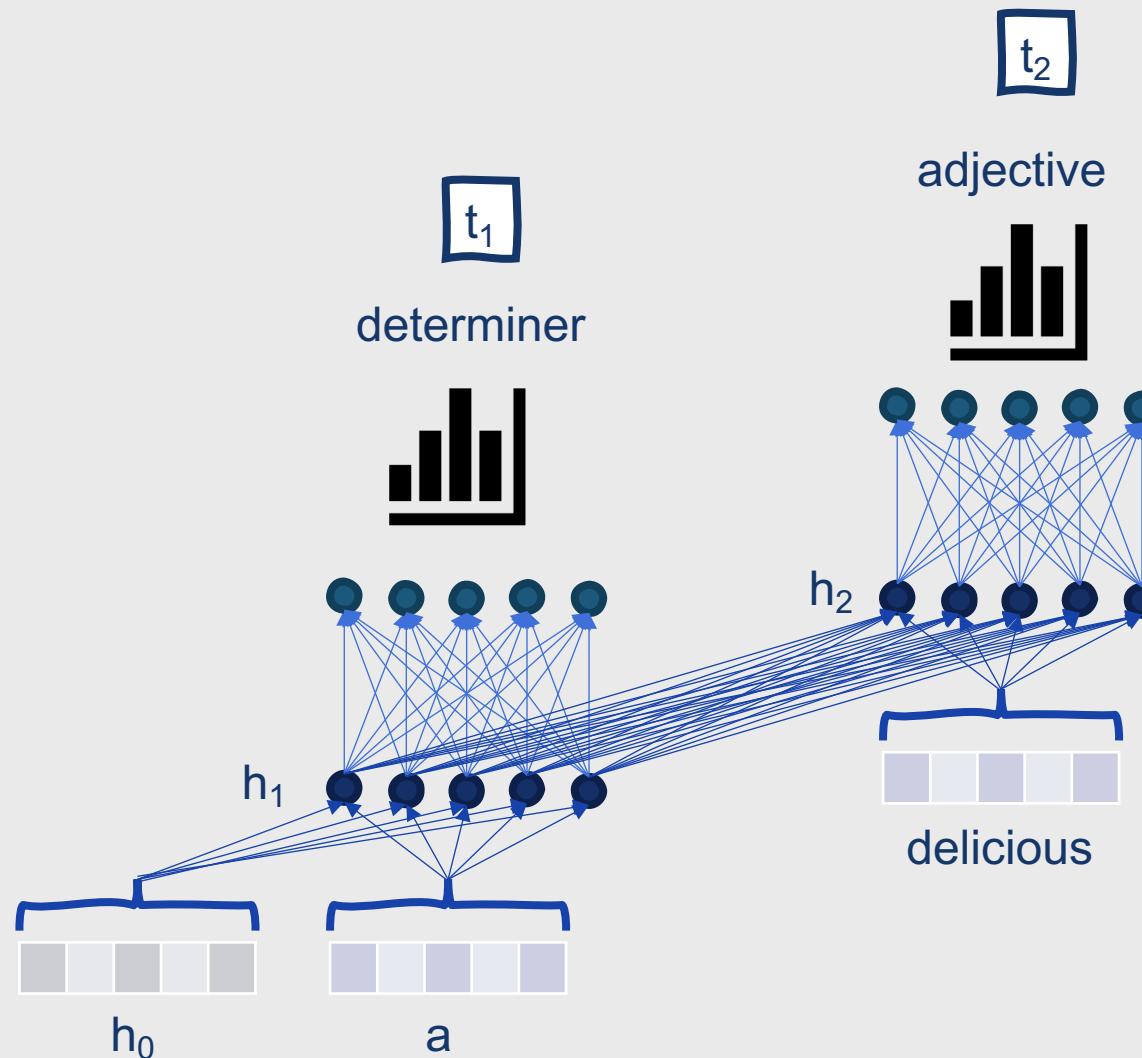
# RNNs are also highly useful for sequence labeling.

- Task: Given a fixed set of labels, assign a label to each element of a sequence
  - Example: Part-of-speech tagging
- Inputs → word embeddings
- Outputs → label probabilities generated by the softmax (or other activation) function over the set of all labels

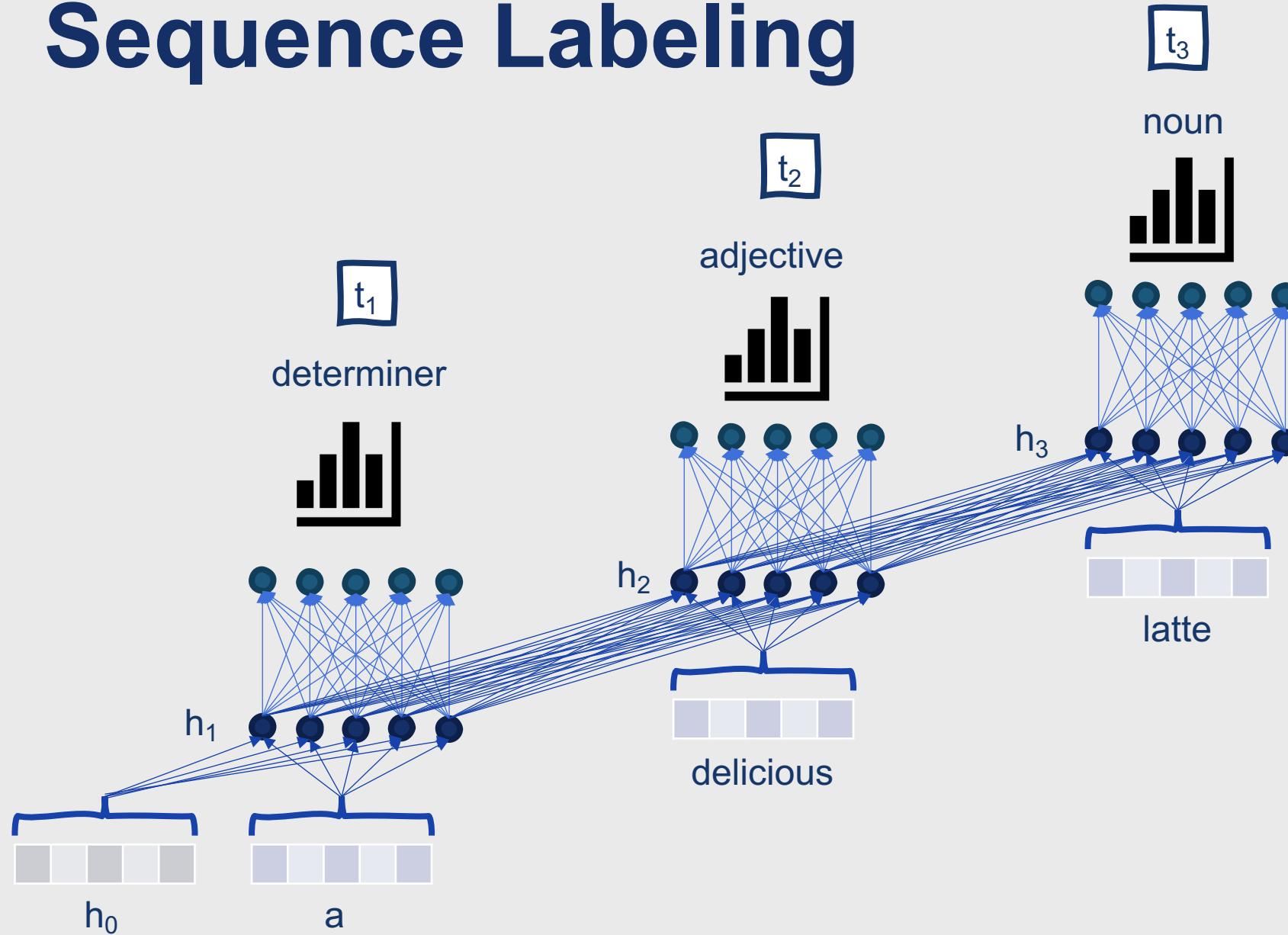
# Sequence Labeling



# Sequence Labeling



# Sequence Labeling



They're also  
useful for  
sequence  
classification!

- Task: Given an input sequence, **assign the entire sequence to a class** (rather than the individual tokens within it)



# How to use RNNs for sequence classification?



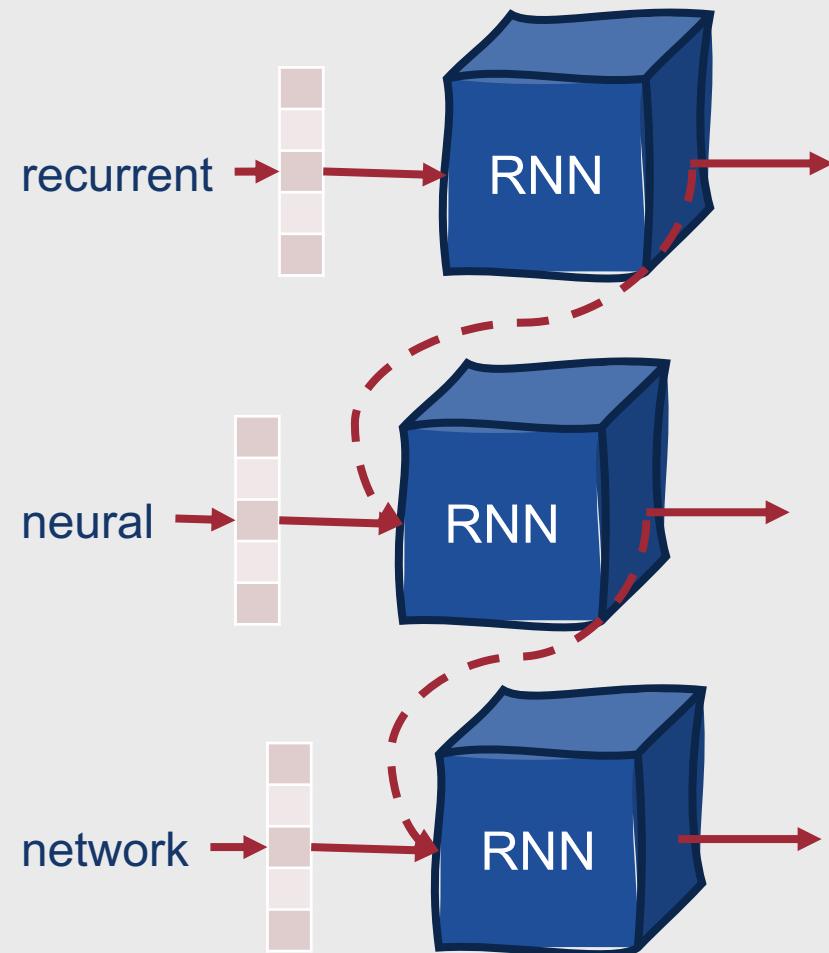
Pass the sequence through an RNN one word at a time, as usual

Assume that the hidden layer for the final word,  $h_n$ , acts as a compressed representation of the entire sequence

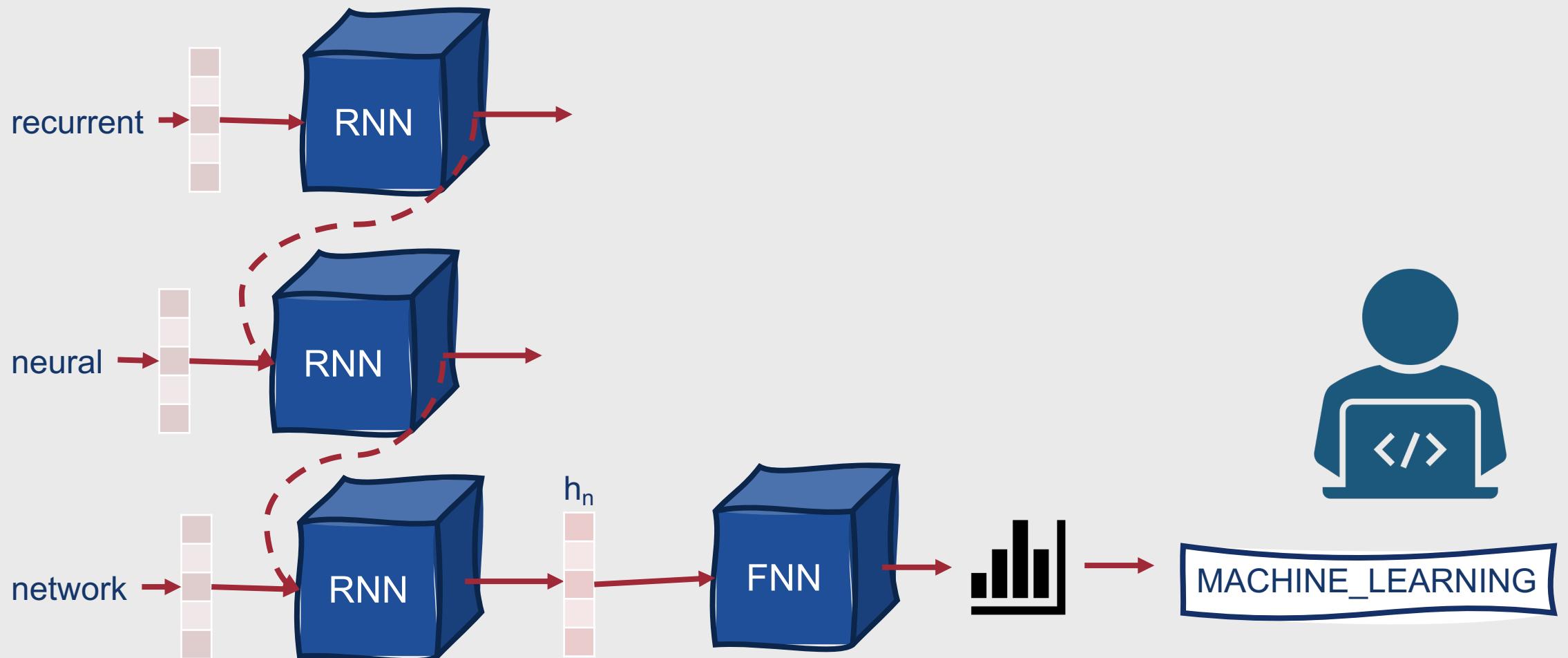
Use  $h_n$  as input to a subsequent feedforward neural network

Choose a class via softmax over all the possible classes

# Sequence Classification



# Sequence Classification

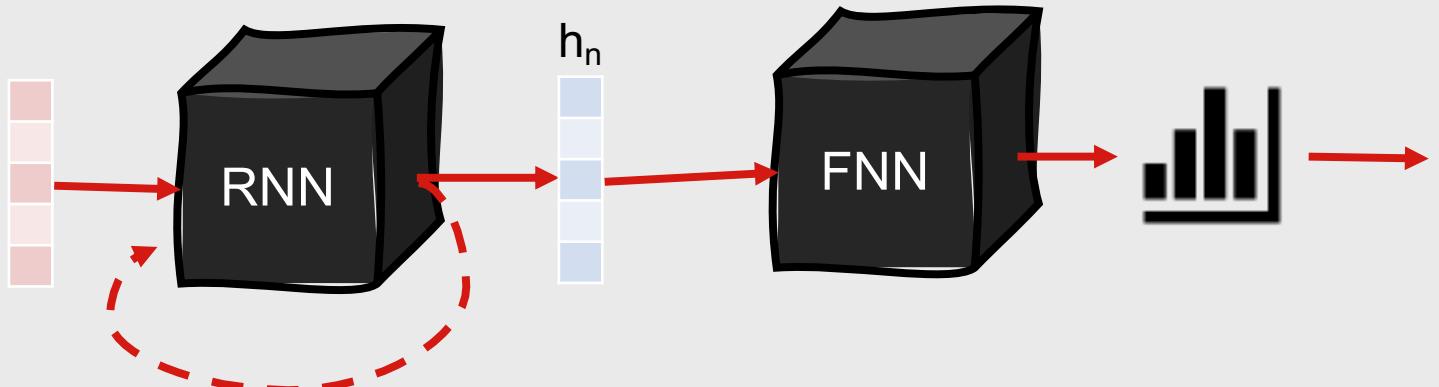


# Notes about Sequence Classification

- No loss associated with intermediate outputs
- Loss function is based entirely on the final classification task!
- Errors are still backpropagated all the way through the RNN
- The process of adjusting weights the entire way through the network based on the loss from a downstream application is often referred to as **end-to-end training**

# Where do we go from here?

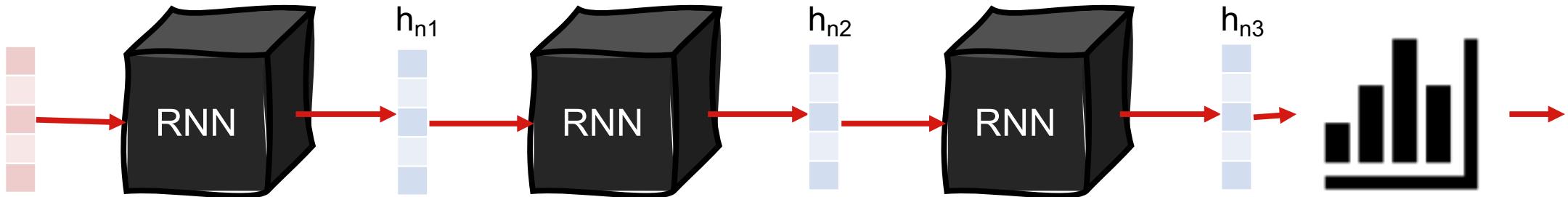
- So far, we've discussed "vanilla" RNNs
- Many additional varieties exist!
- Extensions to the vanilla RNN model:
  - RNN + Feedforward layers
  - Stacked RNNs
  - Bidirectional RNNs



# Stacked RNNs

- Use the entire sequence of outputs from one RNN as the input sequence to another
- Capable of outperforming single-layer networks
- Why?
  - Having more layers allows the network to learn representations at differing levels of **abstraction** across layers
    - Early layers → more fundamental properties
    - Later layers → more meaningful groups of fundamental properties

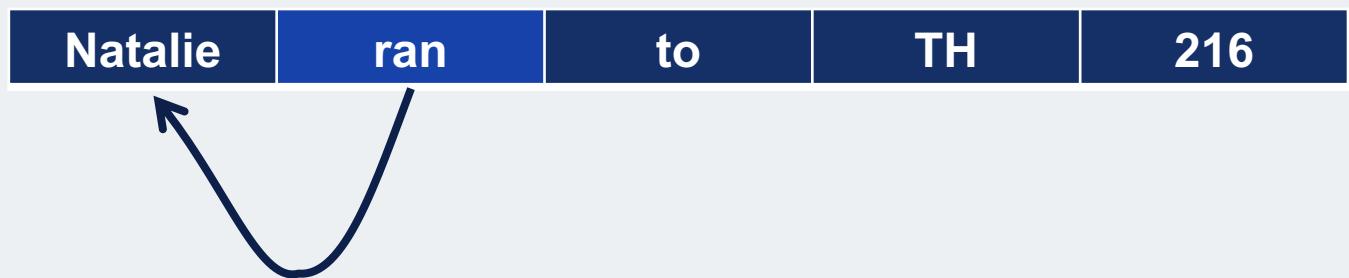
# Stacked RNNs



- Optimal number of RNNs to stack together?
  - Depends on application and training set
  - More RNNs in the stack → increased training costs

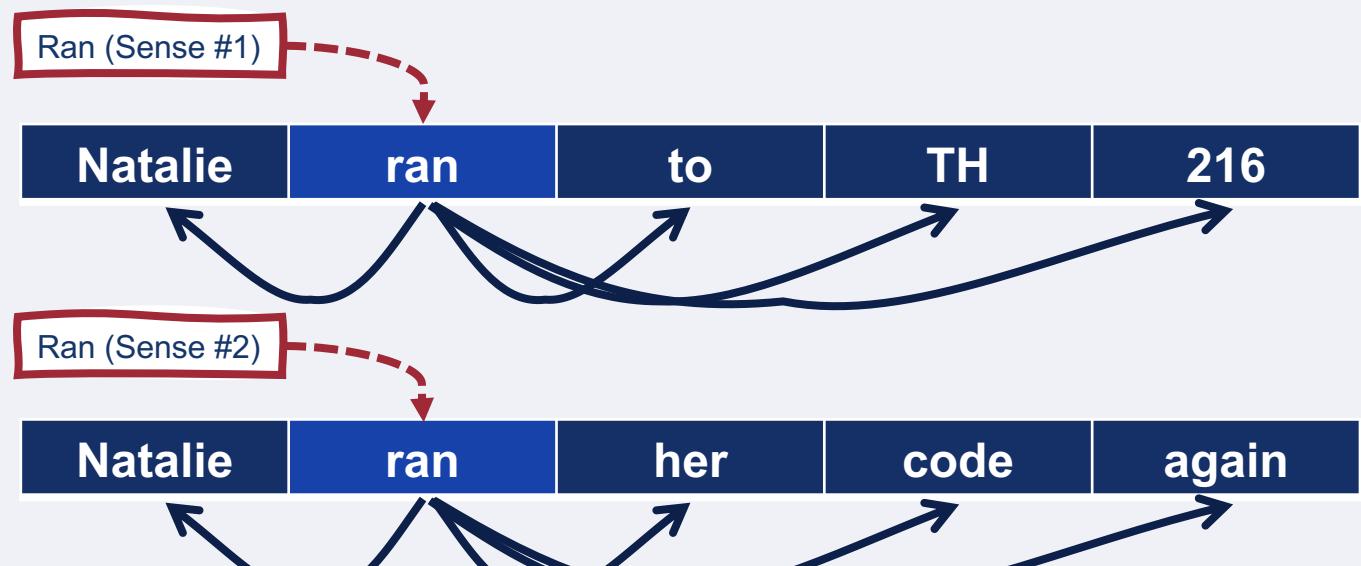
# Bidirectional RNNs

- Simple RNNs only consider the information in a sequence leading up to the current timestep
  - $h_t^f = RNN_{forward}(x_1^t)$
  - $h_t^f$  corresponds to the normal hidden state at time  $t$
- This could be visualized as the context to the left of the current time



# Bidirectional RNNs

- However, in many cases the context after the current timestep (to the right of the current time) could be useful as well!
- In many applications we have access to the entire input sequence at once anyway



# Bidirectional RNNs

- How can we make use of information from both sides of the current timestep?
- Simple solution:
  - Train an RNN on an input sequence in **reverse**
    - $h_t^b = RNN_{backward}(x_t^n)$
    - $h_t^b$  corresponds to information from the current timestep to the end of the sequence
  - **Combine** the forward and backward networks

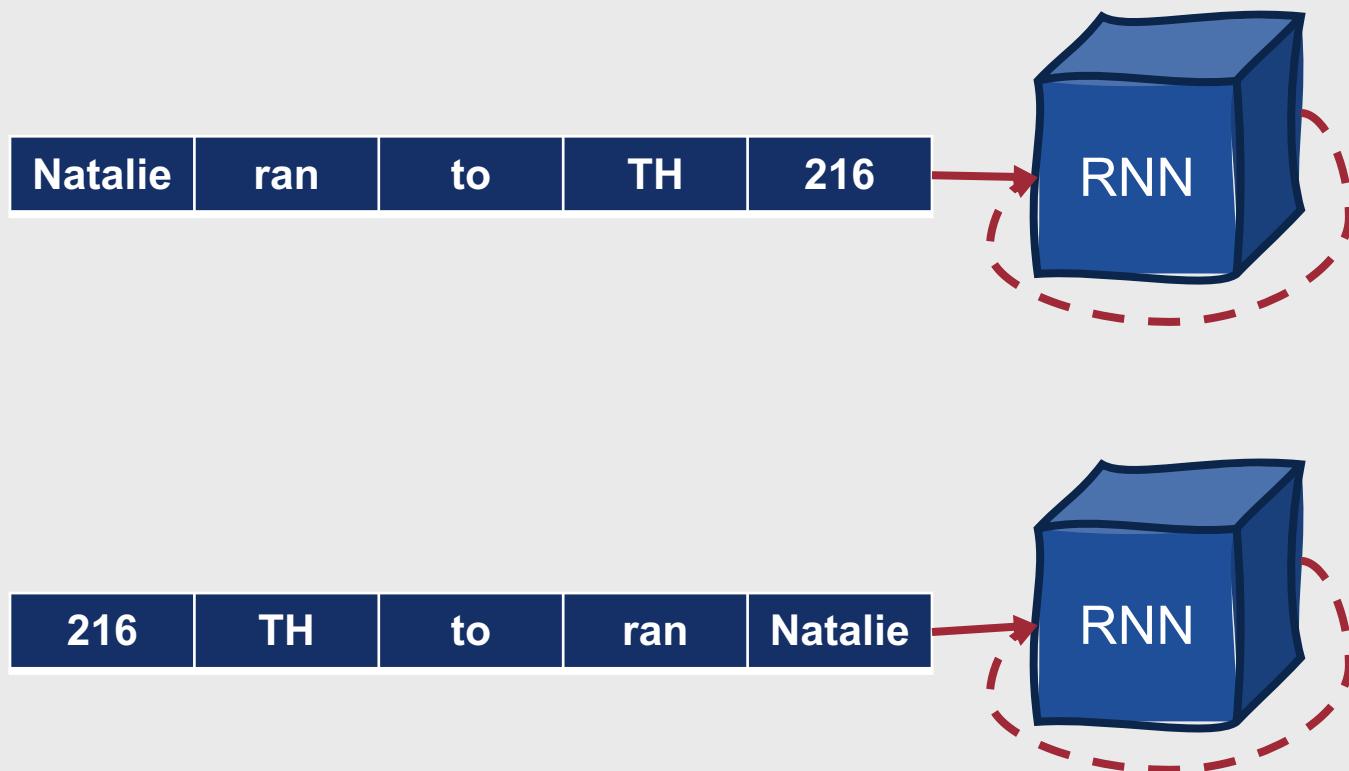
# Bidirectional RNNs

- Two independent RNNs
  - One where the input is processed from start to end
  - One where the input is processed from end to start
- Outputs combined into a single representation that captures both the left and right contexts of an input at each timestep
  - $h_t = h_t^f \oplus h_t^b$
- How to combine the contexts?
  - Concatenation
  - Element-wise addition, multiplication, etc.

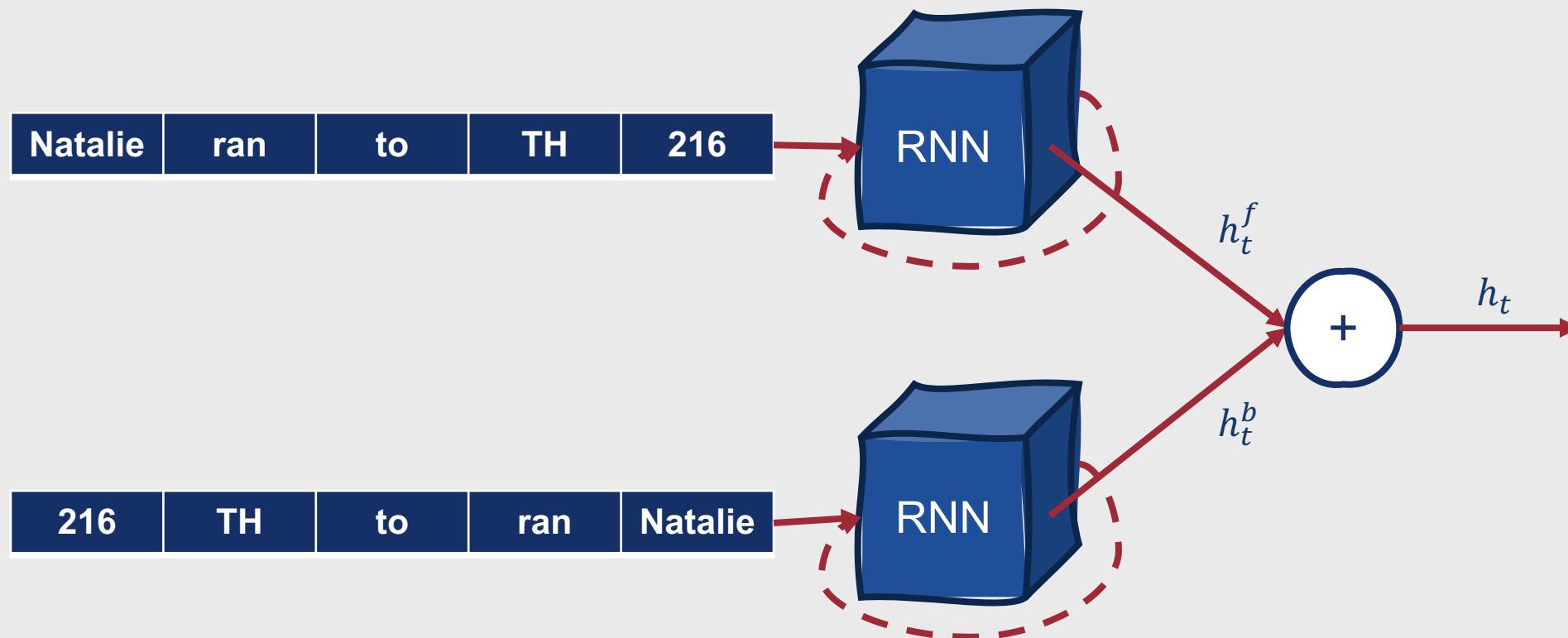
# Bidirectional RNNs



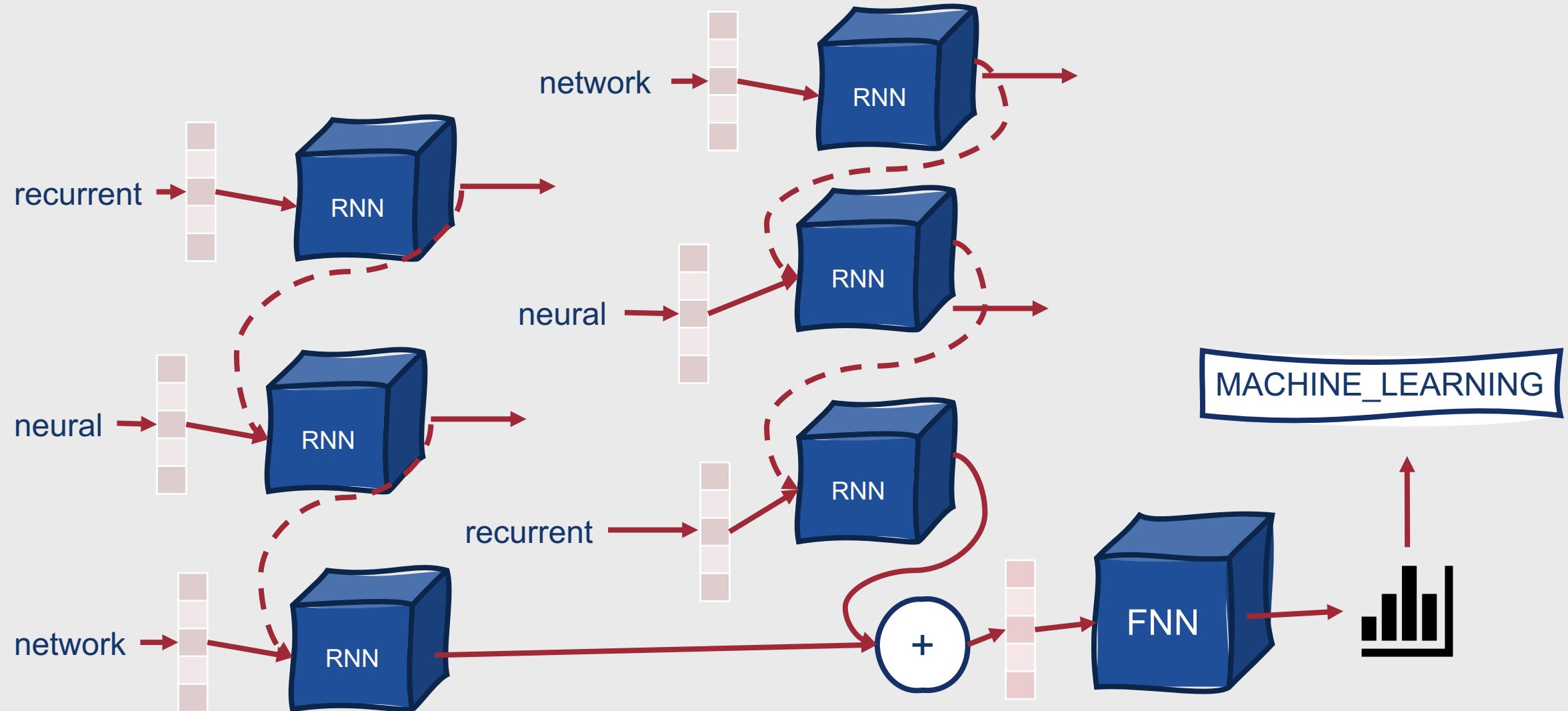
# Bidirectional RNNs



# Bidirectional RNNs



# Sequence Classification with a Bidirectional RNN



# More advanced variations to come....

- Additional ways to combine RNNs
- Architectural modifications to allow better context management

**“Vanilla” RNNs hold many advantages over feedforward networks for NLP tasks.**

- Temporal context
- Variable-length input
- However ...they're not perfect (no networks are!)

---

# In particular, RNNs may struggle with managing context.

- In a simple RNN, the final state tends to reflect more information about **recent items** than those at the beginning of the sequence
- **Distant timesteps → less information**



This long-distance information can be critical to many tasks!

# Why is it so hard to maintain long-distance context?



- Hidden layers must perform two tasks simultaneously:
  - Provide information useful for the current decision (input at  $t$ )
  - Update and carry forward information required for future decisions (input at time  $t+1$  and beyond)
- These tasks may not always be perfectly aligned with one another

## There's also the issue of “vanishing gradients”....

- When small derivatives are repeatedly multiplied together, the products can become extremely small
- This means that when backpropagating through time for a long sequence, gradients can become so close to zero that they are no longer effective for model training!



---

# How can we address this?

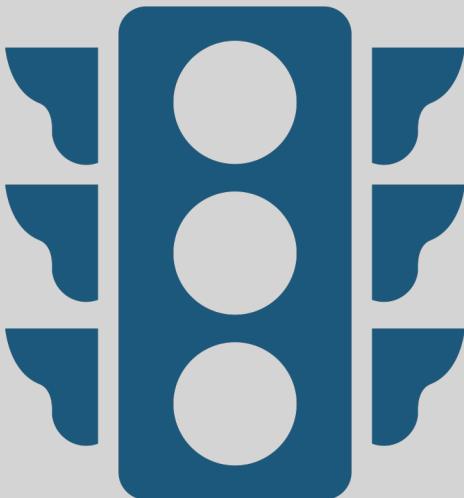
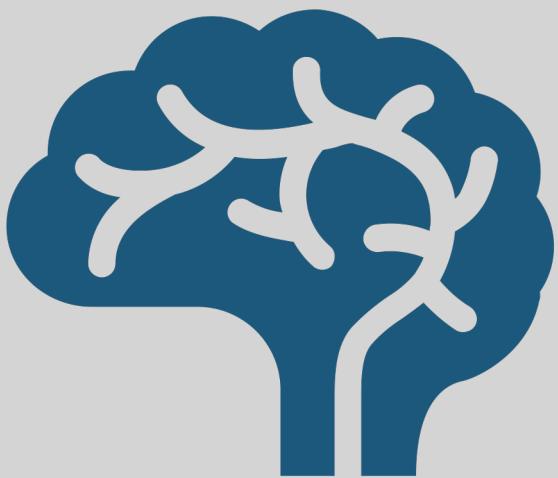
- Design more complex RNNs that learn to:
  - **Forget** information that is no longer needed
  - **Remember** information still required for future decisions

# Long Short-Term Memory Networks (LSTMs)

- Remove information no longer needed from the context, and add information likely to be needed later
- Do this by:
  - Adding an **explicit context layer** to the architecture
  - This layer controls the flow of information into and out of network layers using specialized neural units called **gates**



# LSTM Gates



- Feedforward layer + sigmoid activation + pointwise multiplication with the layer being gated
- Combination of sigmoid activation and pointwise multiplication essentially creates a **binary mask**
  - Values near 1 in the mask are passed through **nearly unchanged**
  - Values near 0 are **nearly erased**



# LSTM Gates

- Three main gates:
  - **Forget gate:** Should we erase this existing information from the context?
  - **Add gate:** Should we write this new information to the context?
  - **Output gate:** What information should be leveraged for the current hidden state?

# Forget Gate

- Goal: Delete information from the context that is no longer needed

$$\begin{aligned} \bullet & f_t = \sigma(U_f h_{t-1} + W_f x_t) \\ \bullet & k_t = c_{t-1} \odot f_t \end{aligned}$$

Weighted sum of:

- Hidden layer at the previous timestep
- Current input

# Forget Gate

- Goal: Delete information from the context that is no longer needed

$$\begin{aligned} \bullet & f_t = \sigma(U_f h_{t-1} + W_f x_t) \\ \bullet & k_t = c_{t-1} \odot f_t \end{aligned}$$

Context vector from the previous timestep

# Add Gate

- Goal: Select the information to add to the current context

- $g_t = \tanh(U_g h_{t-1} + W_g x_t)$
- $i_t = \sigma(U_i h_{t-1} + W_i x_t)$
- $j_t = g_t \odot i_t$
- $c_t = j_t + k_t$

Regular RNN computation

# Add Gate

- Goal: Select the information to add to the current context

- $g_t = \tanh(U_g h_{t-1} + W_g x_t)$
- $i_t = \sigma(U_i h_{t-1} + W_i x_t)$
- $j_t = g_t \odot i_t$
- $c_t = j_t + k_t$

Weighted sum of:  
• Hidden layer at the previous timestep  
• Current input

# Add Gate

- Goal: Select the information to add to the current context

- $g_t = \tanh(U_g h_{t-1} + W_g x_t)$
- $i_t = \sigma(U_i h_{t-1} + W_i x_t)$
- $j_t = g_t \odot i_t$
- $c_t = j_t + k_t$

New information to be added

# Add Gate

- Goal: Select the information to add to the current context

- $g_t = \tanh(U_g h_{t-1} + W_g x_t)$
- $i_t = \sigma(U_i h_{t-1} + W_i x_t)$
- $j_t = g_t \odot i_t$
- $c_t = j_t + k_t$



Updated context vector contains:

- New information to be added
- Existing information from context vector that was not removed by the forget gate

# Output Gate

- Goal: Decide what information is required for the *current* hidden state

$$\begin{aligned} \bullet \quad o_t &= \sigma(U_o h_{t-1} + W_o x_t) \\ \bullet \quad h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

Weighted sum of:

- Hidden layer at the previous timestep
- Current input

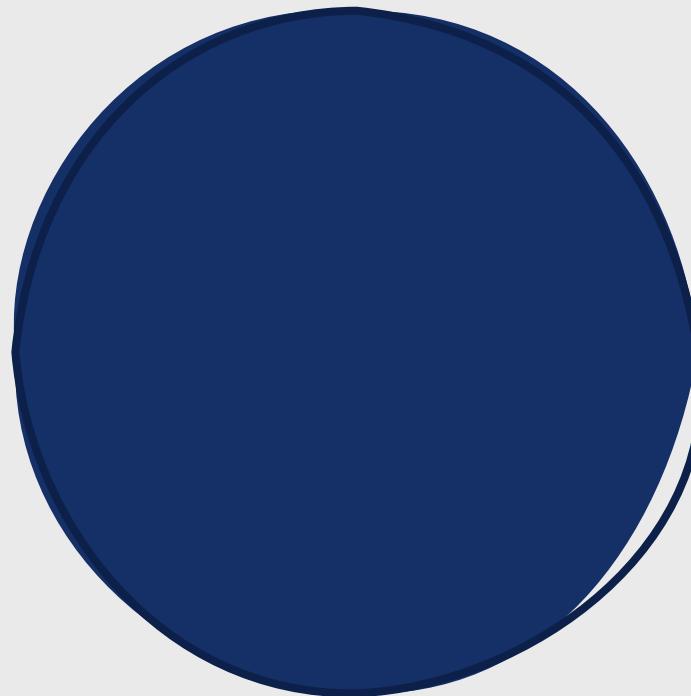
# Output Gate

- Goal: Decide what information is required for the *current* hidden state

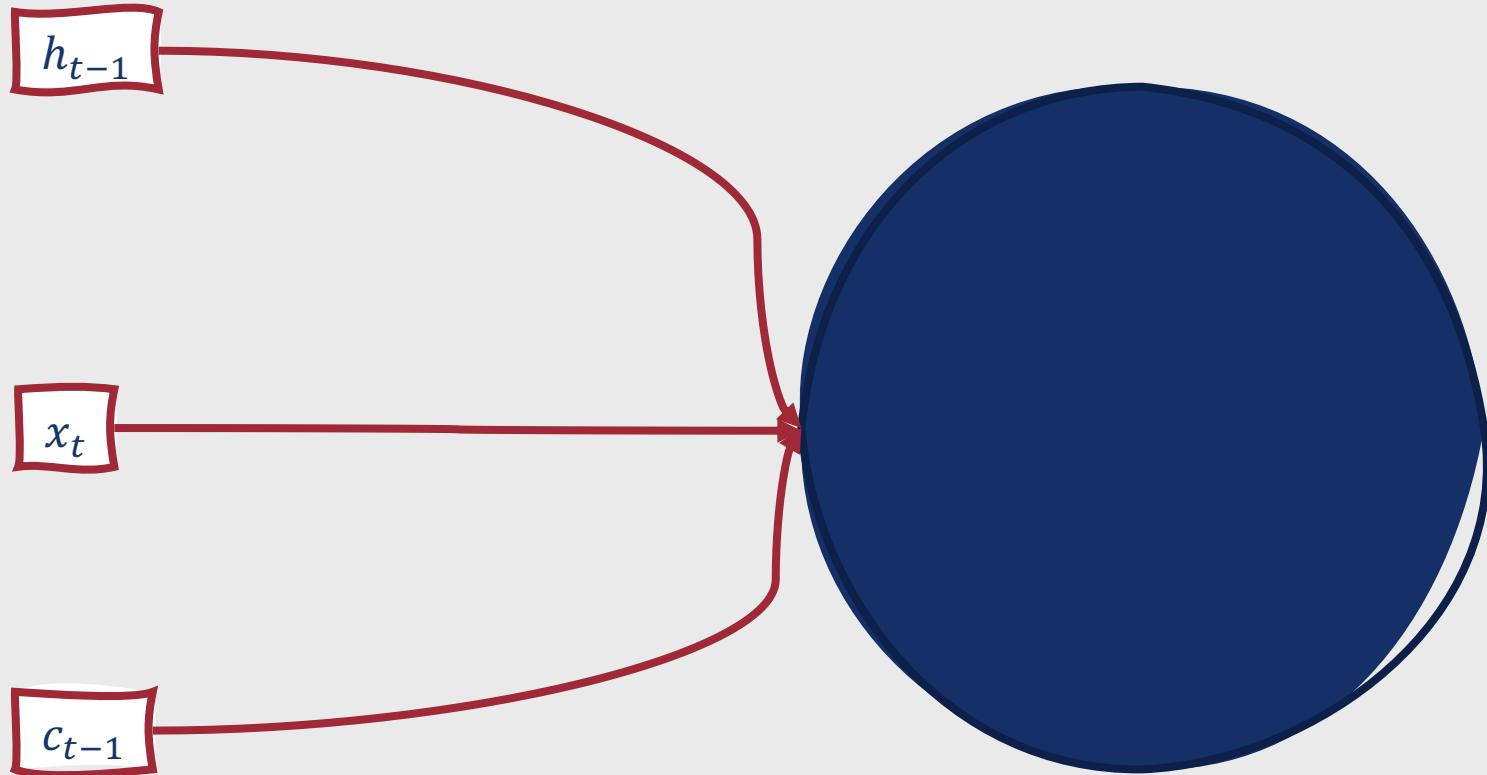
$$\begin{aligned} \bullet & o_t = \sigma(U_o h_{t-1} + W_o x_t) \\ \bullet & h_t = o_t \odot \tanh(c_t) \end{aligned}$$

Updated hidden layer output

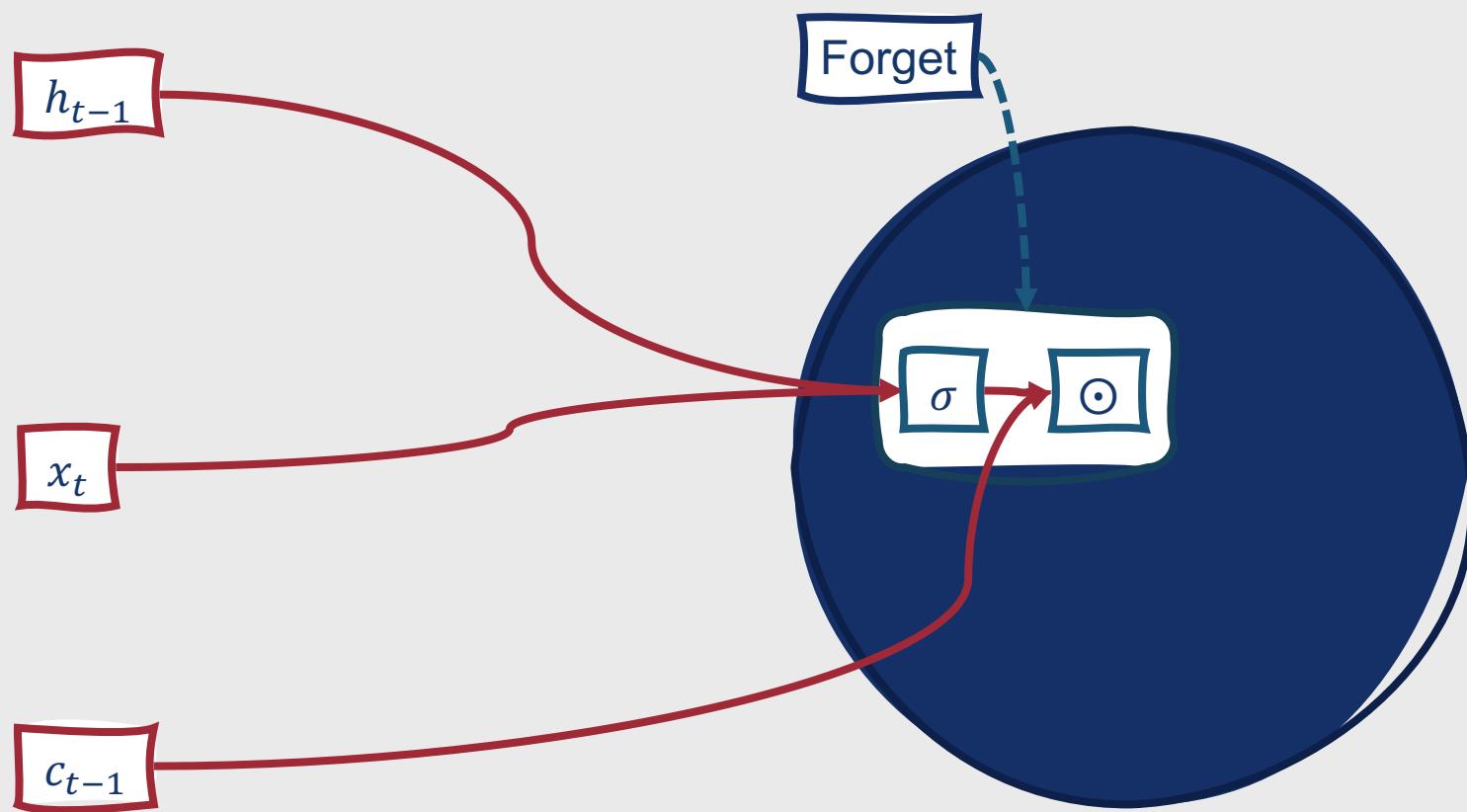
# What does this process look like in a single LSTM unit?



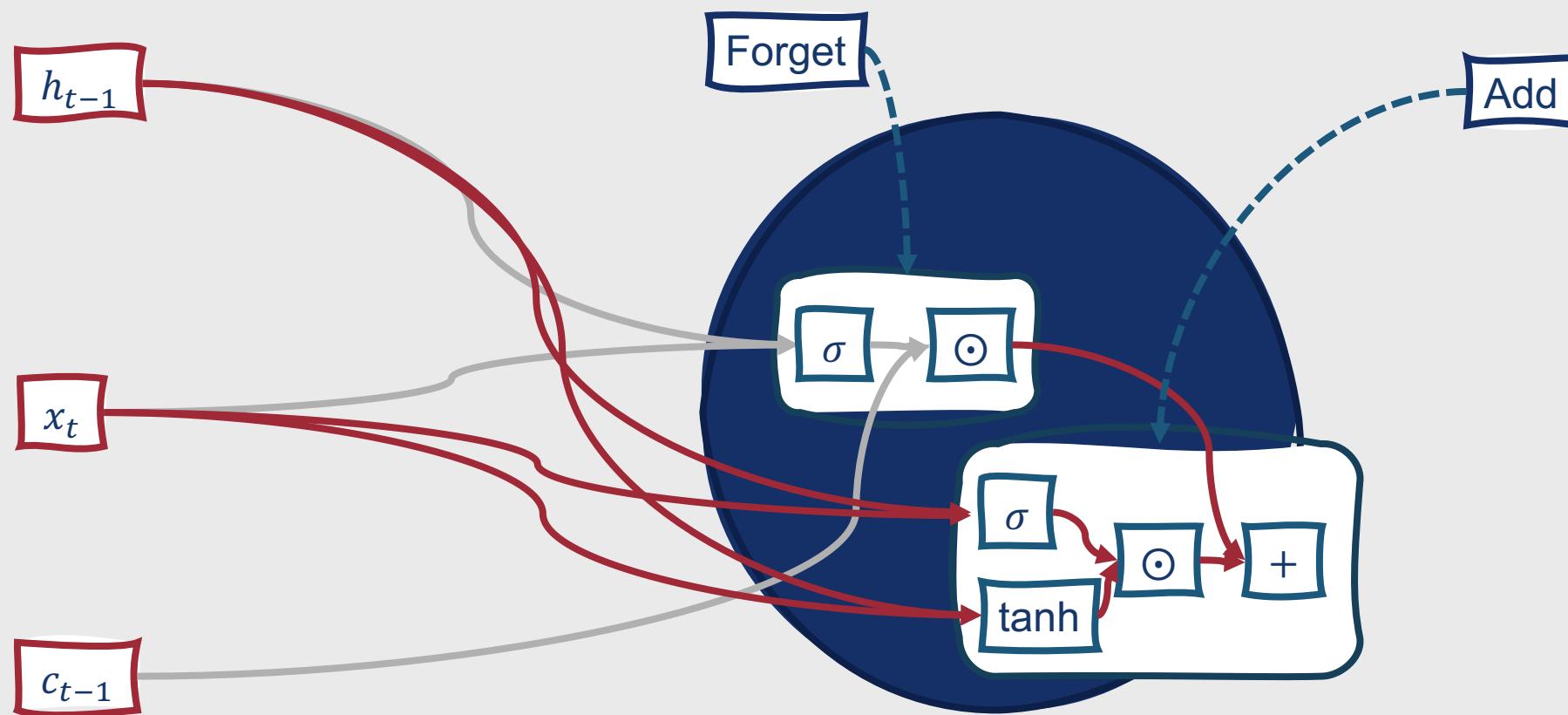
# What does this process look like in a single LSTM unit?



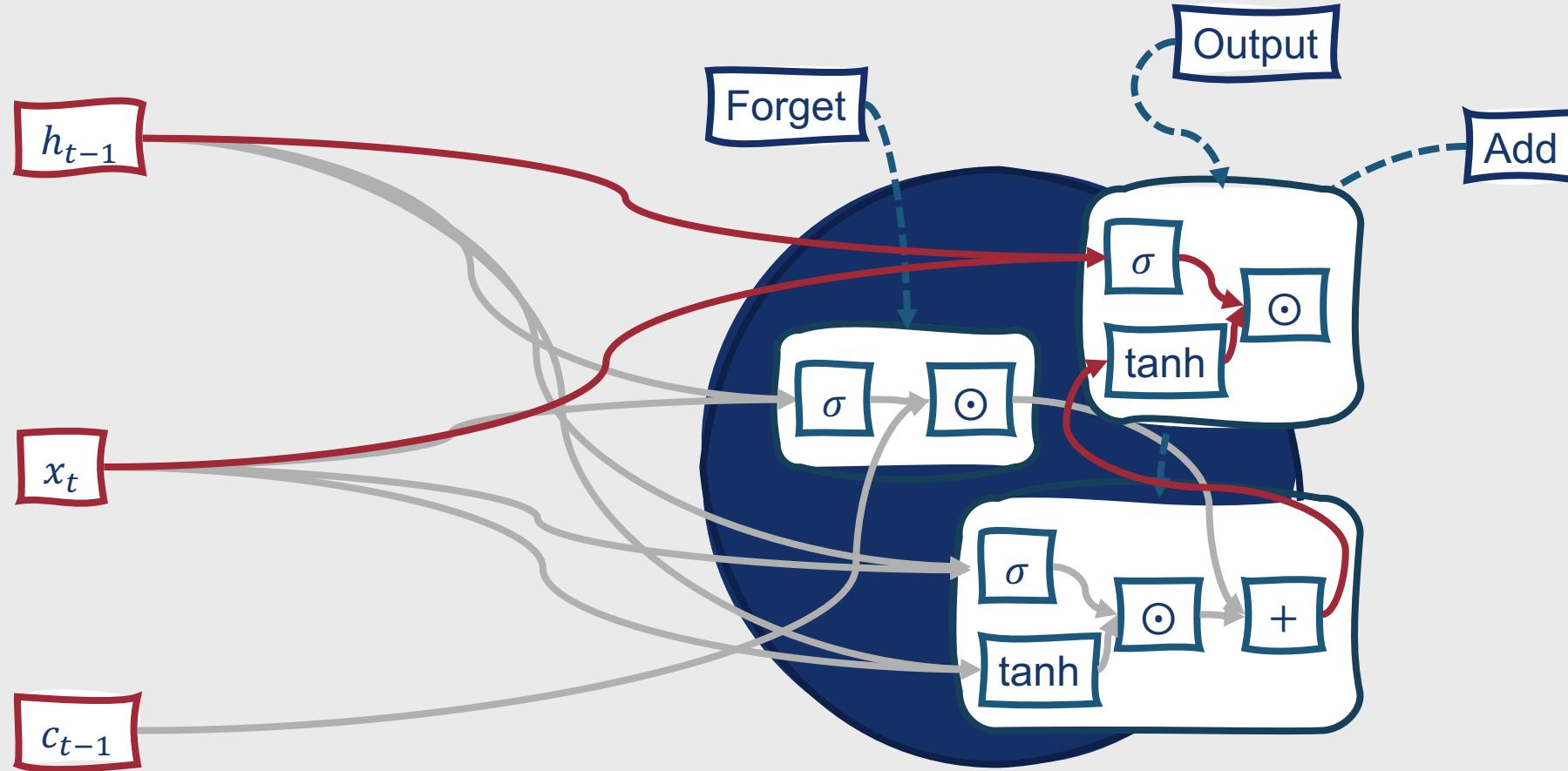
# What does this process look like in a single LSTM unit?



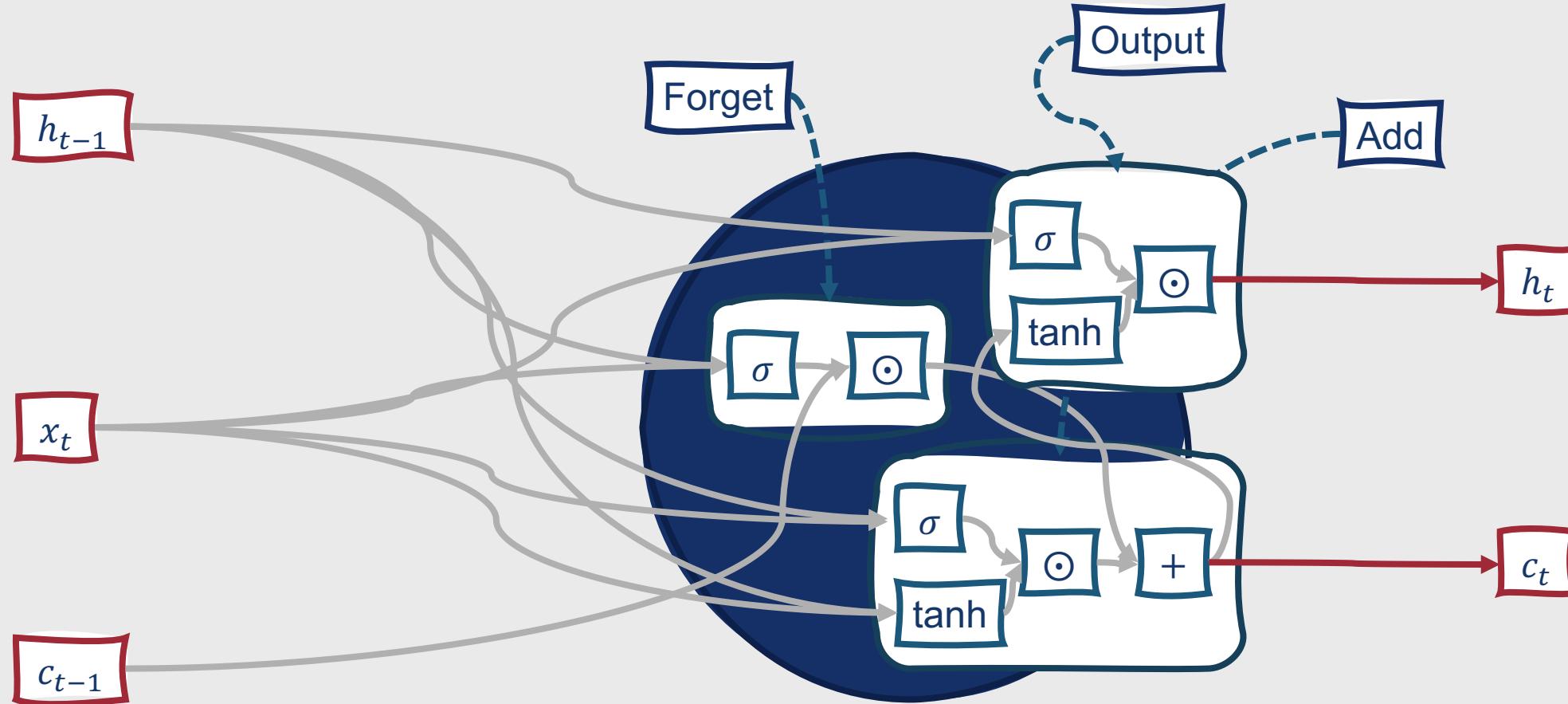
# What does this process look like in a single LSTM unit?



# What does this process look like in a single LSTM unit?



# What does this process look like in a single LSTM unit?



# Long Short-Term Memory Networks (LSTMs)

- LSTMs thus accept as input:
  - **Context layer**
  - **Hidden outputs** from previous timestep
  - **Current input vector**
- They return as output:
  - **Context layer**
  - **Hidden outputs** from the current timestep
- The output of the hidden layer can be used as input to subsequent layers in a stacked RNN, or to the network's output layer

# Gated Recurrent Units (GRUs)

- Also manage the context that is passed through to the next timestep, but do so by utilizing a simpler architecture than LSTMs
  - No separate context vector
  - Only two gates
    - **Reset** gate
    - **Update** gate
- Gates still use a similar design to that seen in LSTMs
  - **Feedforward layer + sigmoid activation + pointwise multiplication** with the layer being gated, resulting in a **binary-like mask**

# Reset Gate

- Goal: Decide which aspects of the previous hidden state are relevant to the current context

$$\begin{aligned} \bullet \quad r_t &= \sigma(U_r h_{t-1} + W_r x_t) \\ \bullet \quad \tilde{h}_t &= \tanh(U(r_t \odot h_{t-1}) + W x_t) \end{aligned}$$

Weighted sum of:

- Hidden layer at the previous timestep
- Current input

# Reset Gate

- Goal: Decide which aspects of the previous hidden state are relevant to the current context

$$\bullet r_t = \sigma(U_r h_{t-1} + W_r x_t)$$

$$\bullet \tilde{h}_t = \tanh(U(r_t \odot h_{t-1}) + W x_t)$$

Intermediate representation for  $h_t$

# Update Gate

- Goal: Decide which aspects of the intermediate hidden state and which aspects of the previous hidden state need to be preserved for future use

$$\begin{aligned} \bullet \quad z_t &= \sigma(U_z h_{t-1} + W_z x_t) \\ \bullet \quad h_t &= (1 - z_t)h_{t-1} + z_t \tilde{h}_t \end{aligned}$$

Weighted sum of:

- Hidden layer at the previous timestep
- Current input

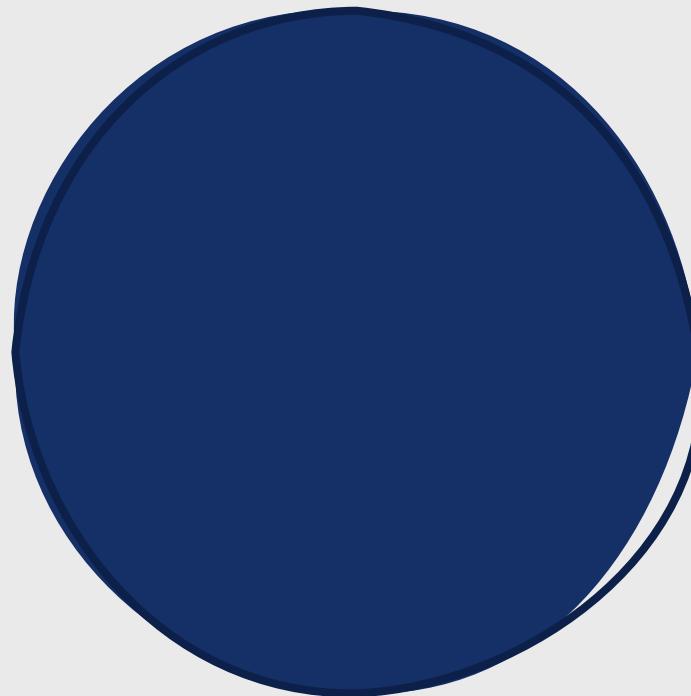
# Update Gate

- Goal: Decide which aspects of the intermediate hidden state and which aspects of the previous hidden state need to be preserved for future use

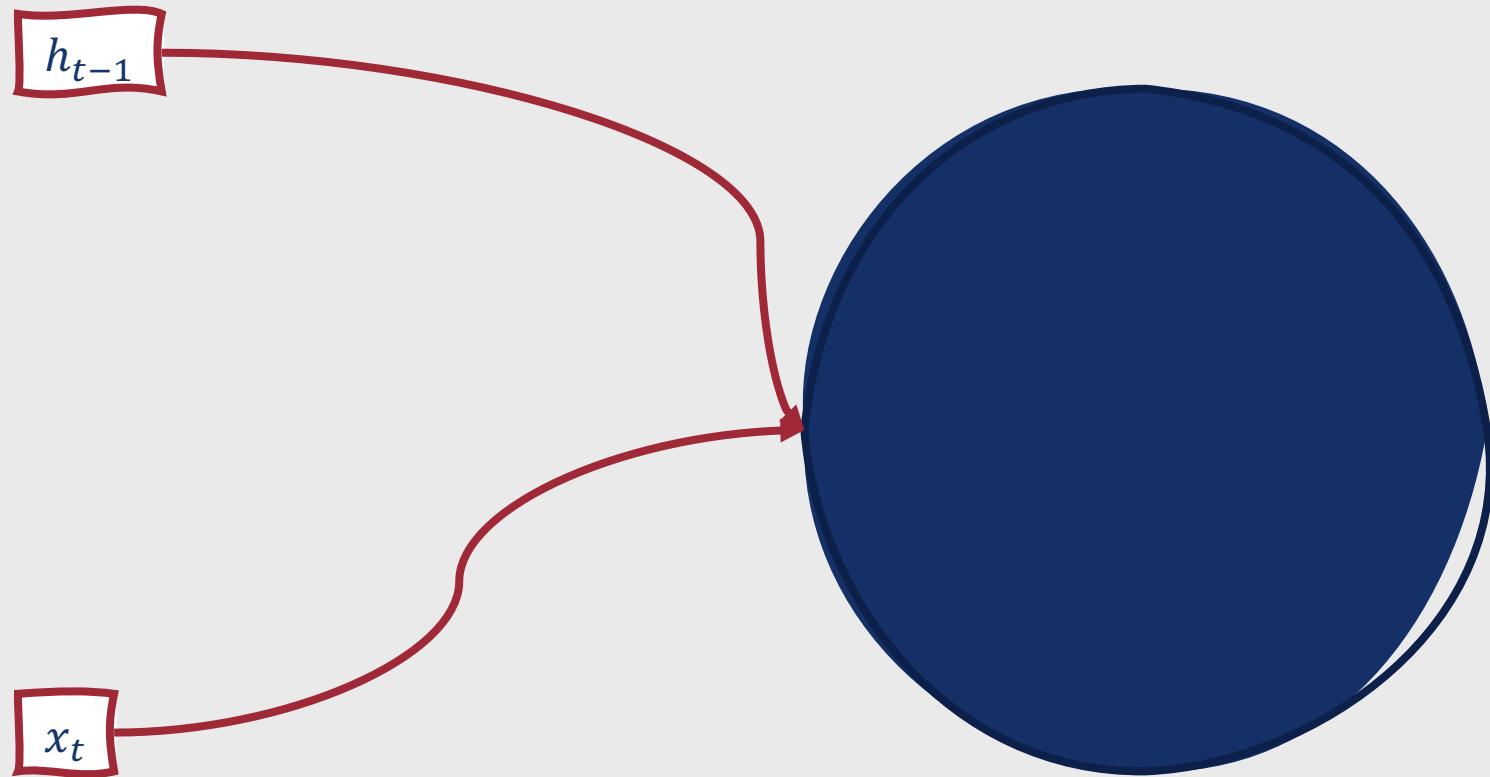
$$\begin{aligned} \bullet & z_t = \sigma(U_z h_{t-1} + W_z x_t) \\ \bullet & h_t = (1 - z_t)h_{t-1} + z_t \tilde{h}_t \end{aligned}$$

Updated hidden layer output

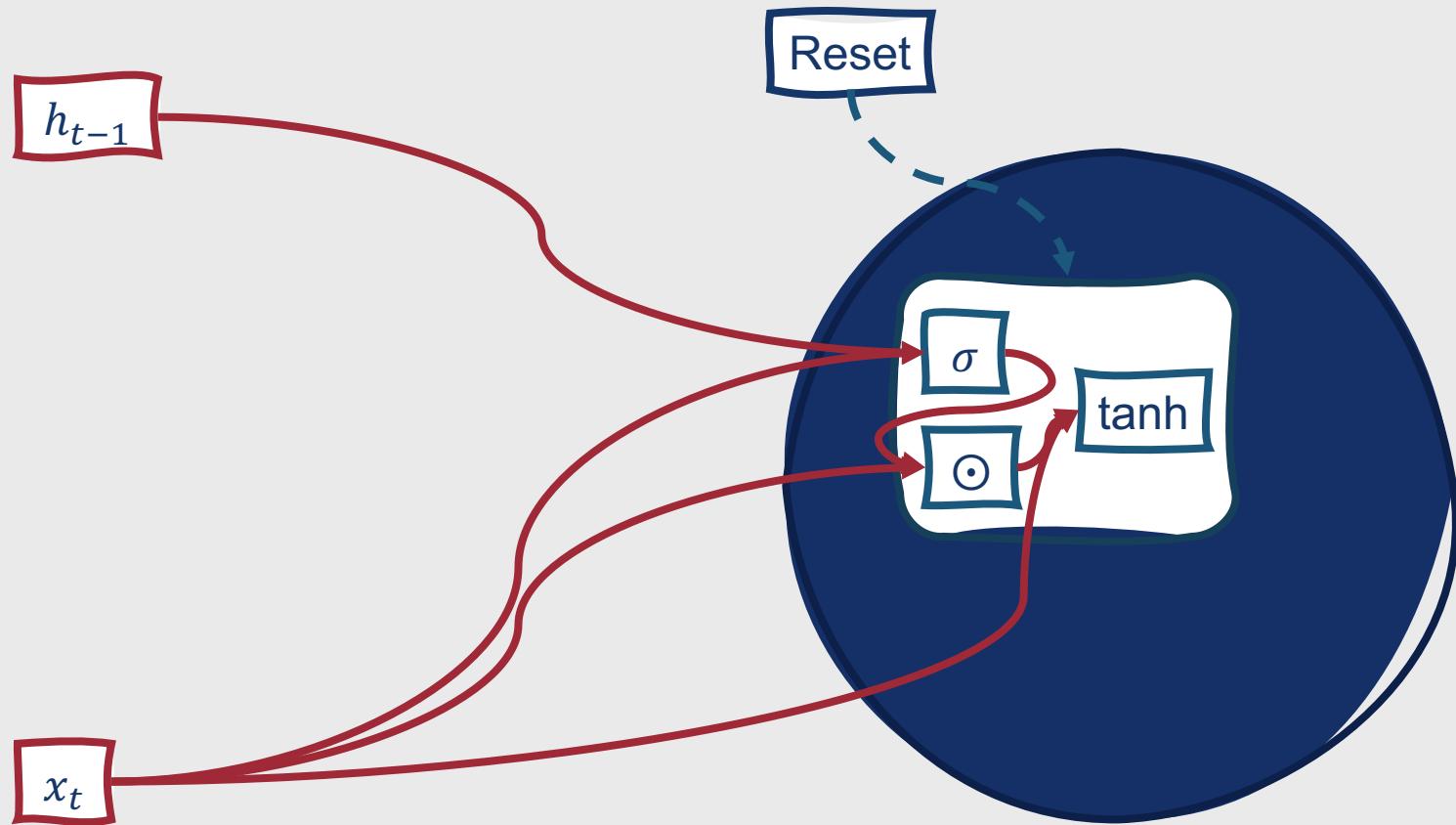
# What does this process look like in a single GRU unit?



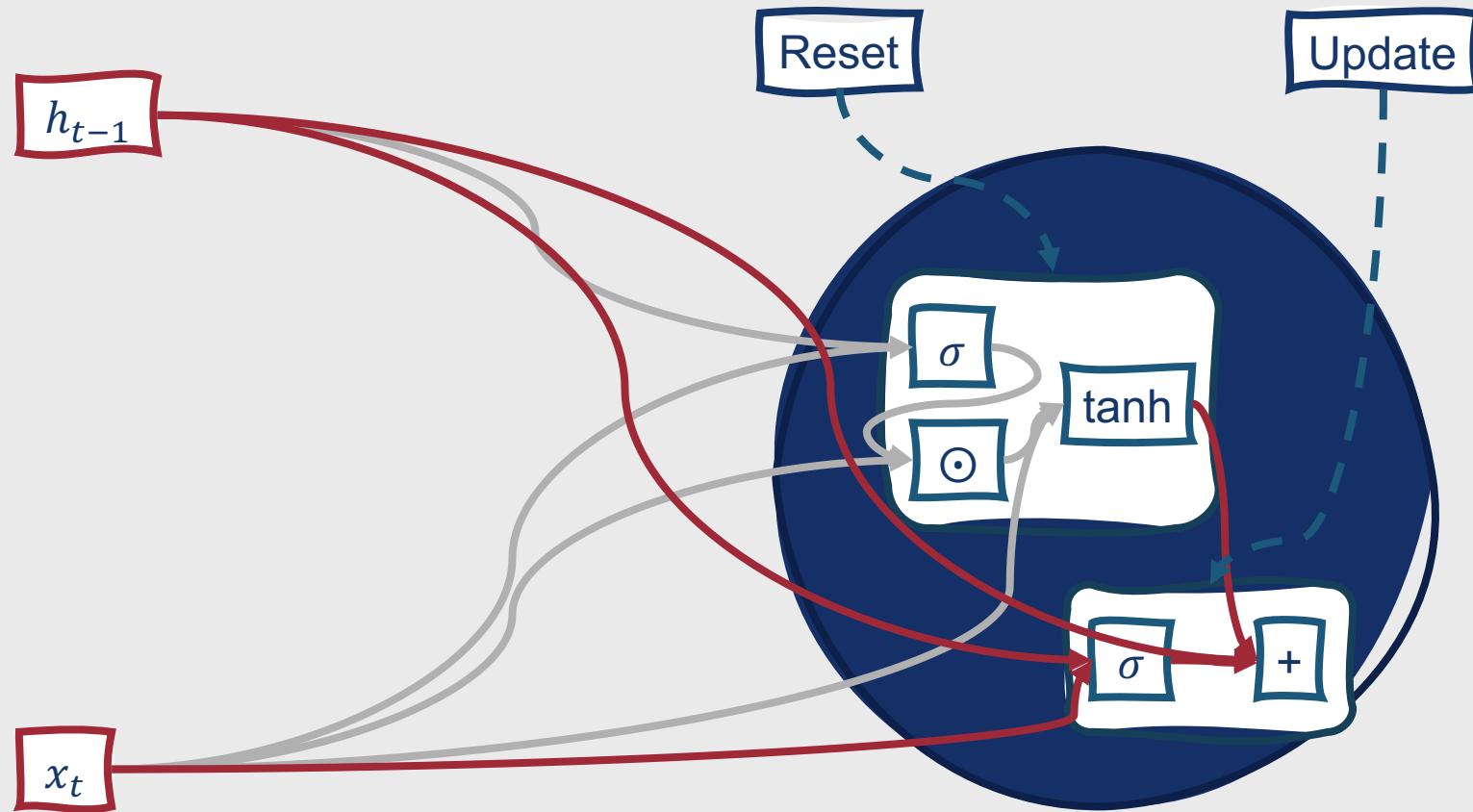
# What does this process look like in a single GRU unit?



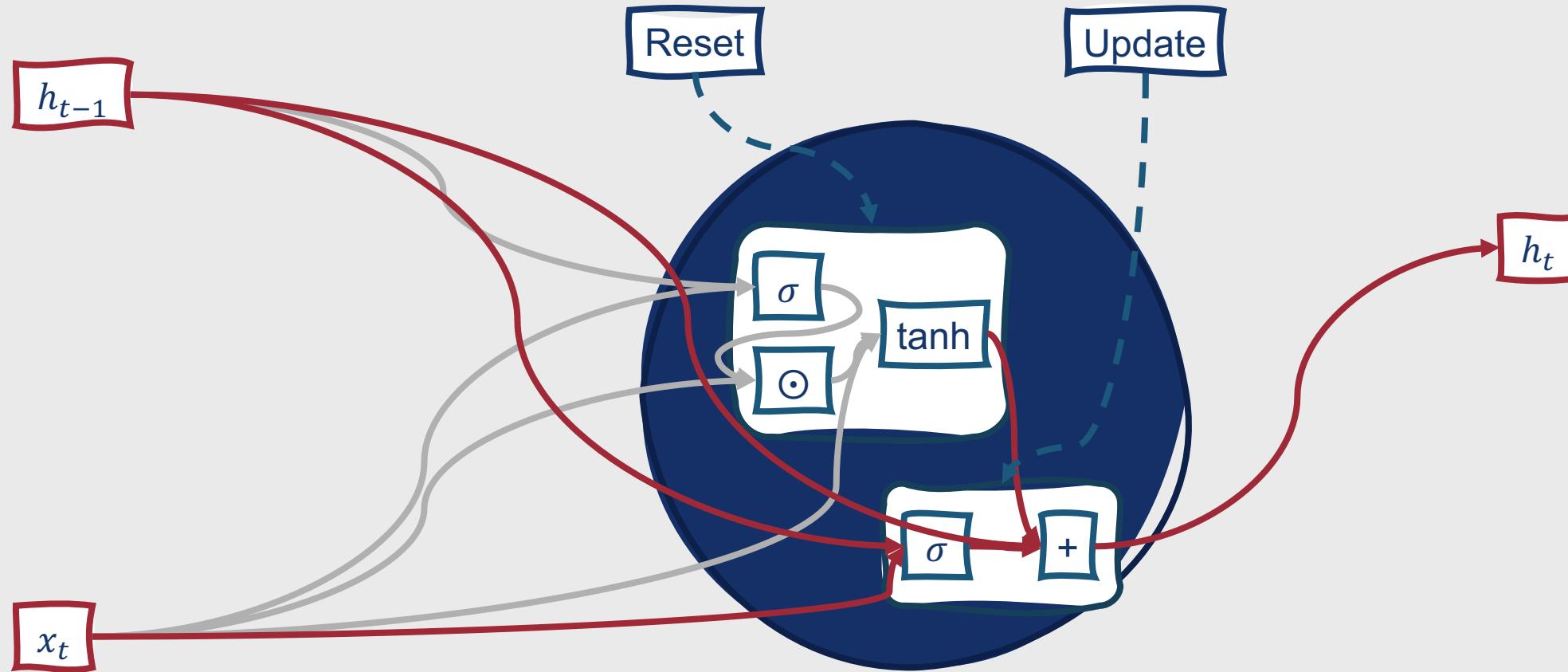
# What does this process look like in a single GRU unit?



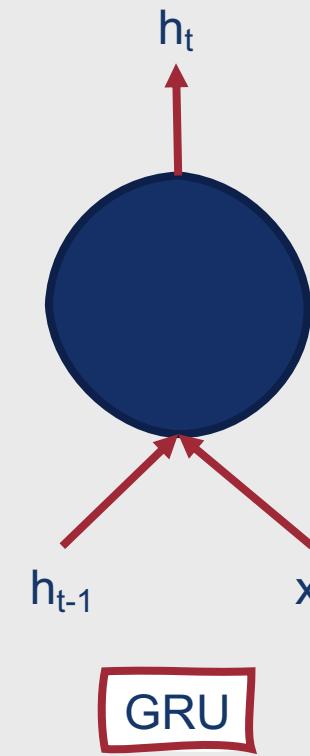
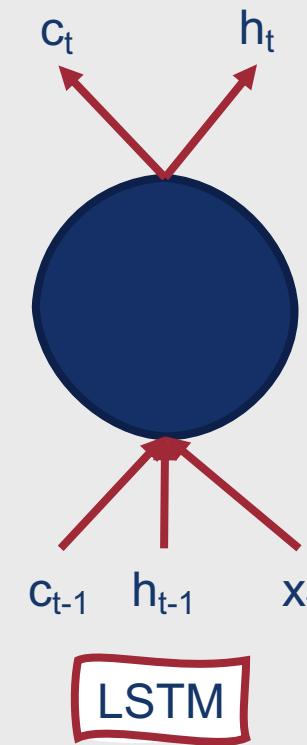
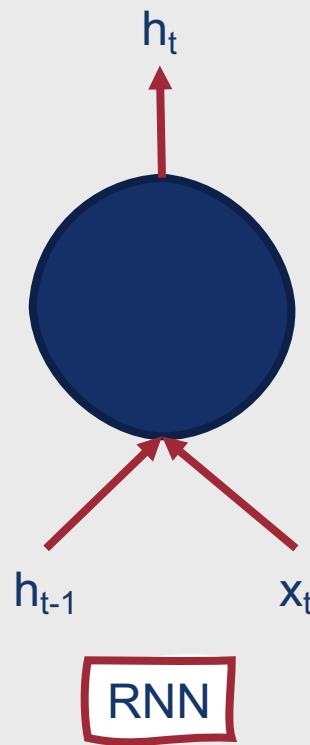
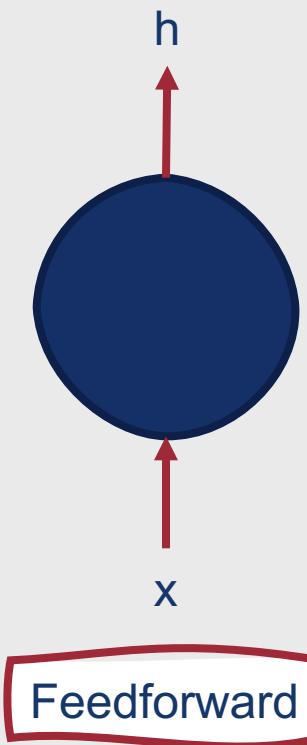
# What does this process look like in a single GRU unit?



# What does this process look like in a single GRU unit?



# Overall, comparing inputs and outputs for some different types of neural units....



# When to use LSTMs vs. GRUs?

## Why use GRUs instead of LSTMs?

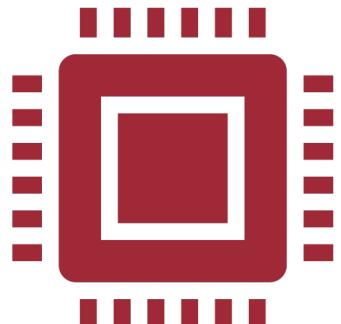
- **Computational efficiency:** Good for scenarios in which you need to train your model quickly and don't have access to high-performance computing resources

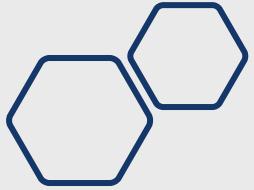
## Why use LSTMs instead of GRUs?

- **Performance:** LSTMs generally outperform GRUs at the same tasks

## Advanced RNNs are a powerful tool, but they are not without their limitations.

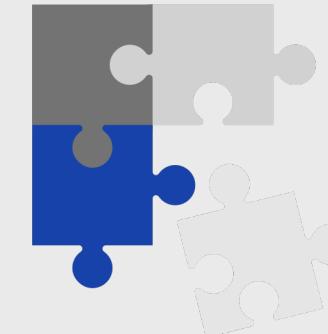
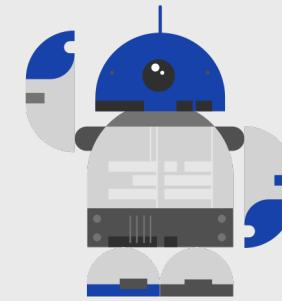
- Remaining challenges:
  - Even with sophisticated architectures, processing long-distance dependencies through **many recurrences** can eventually lead to loss of valuable information
  - Sequential processing models cannot productively leverage **parallel resources**





# Transformers

- Get rid of recurrences entirely
- Closer to feedforward neural networks



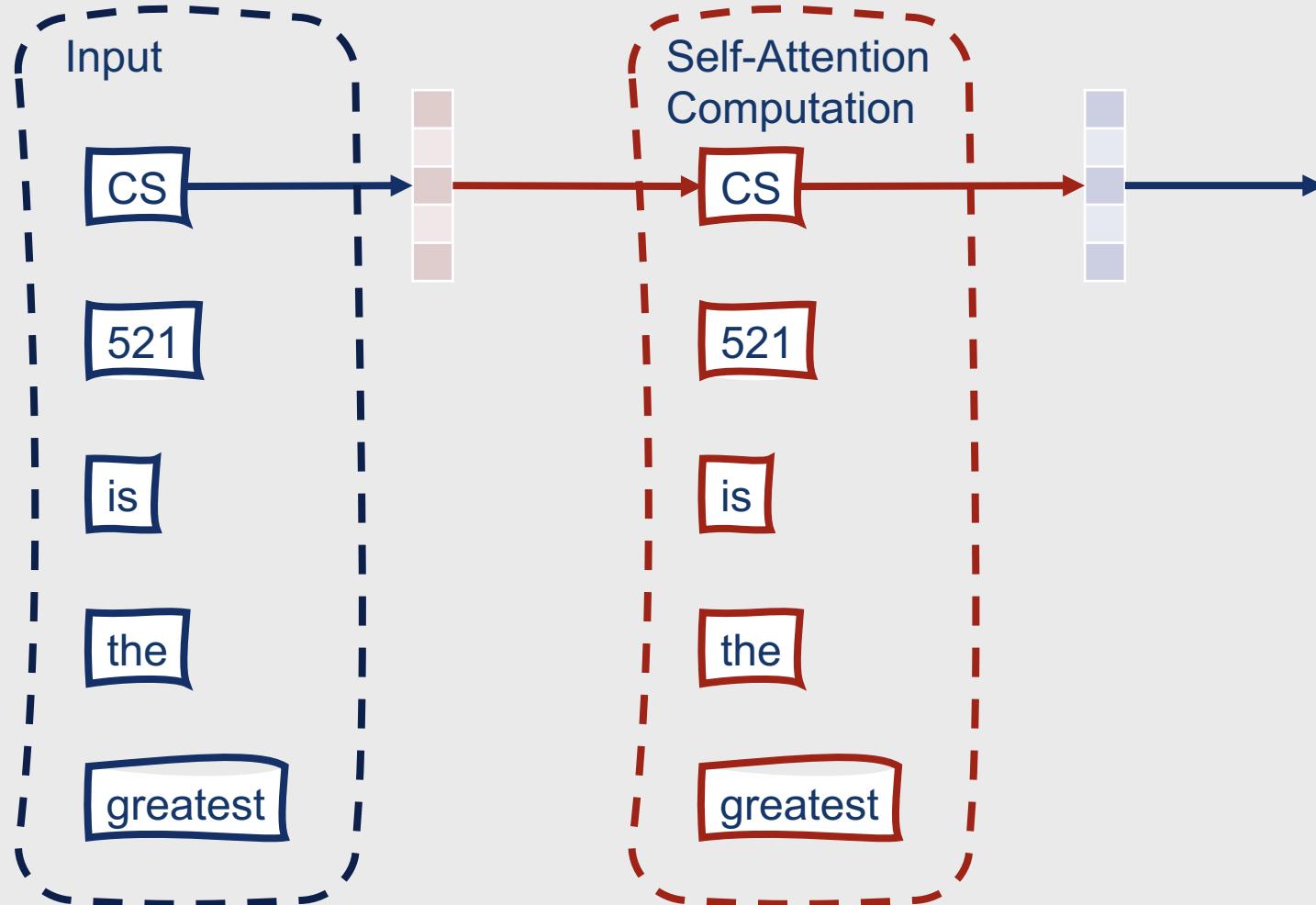
# How do Transformers work?

- Stacks of:
  - Linear layers
  - Feedforward layers
  - Self-attention layers
- Goal: Map sequences of input  $(x_1, \dots, x_n)$  to sequences of output  $(y_1, \dots, y_n)$  of the same length

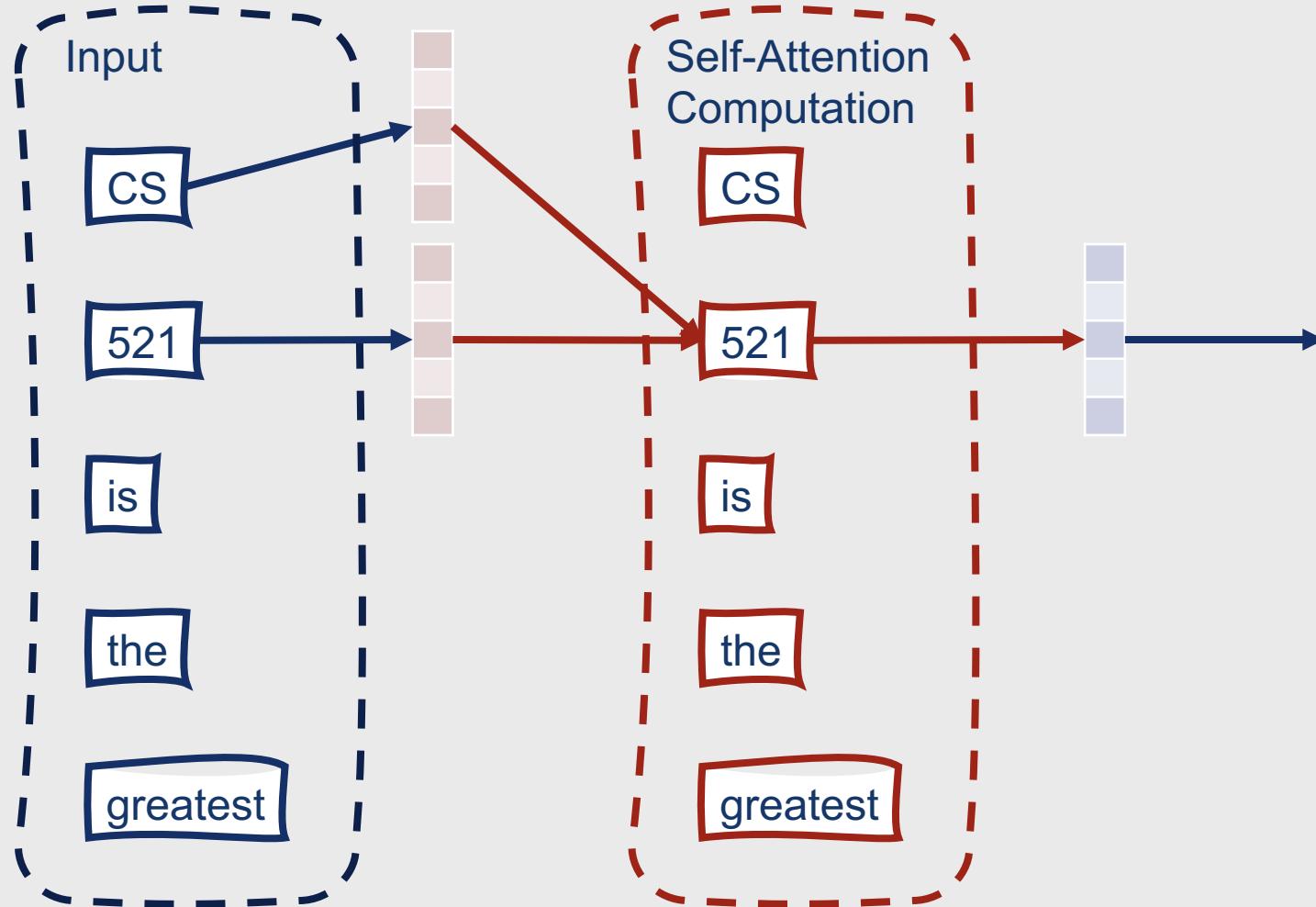
# Self-Attention

- Allows us to consider context in absence of recurrent connections
- For a given element in a sequence, determines which other element(s) up to that point are most relevant to it
  - Each computation is independent of other computations → easy parallelization
  - Each computation only considers elements up to that point in the sequence → easy language modeling
- Ultimately maps a sequence of inputs to an equal-length sequence of outputs

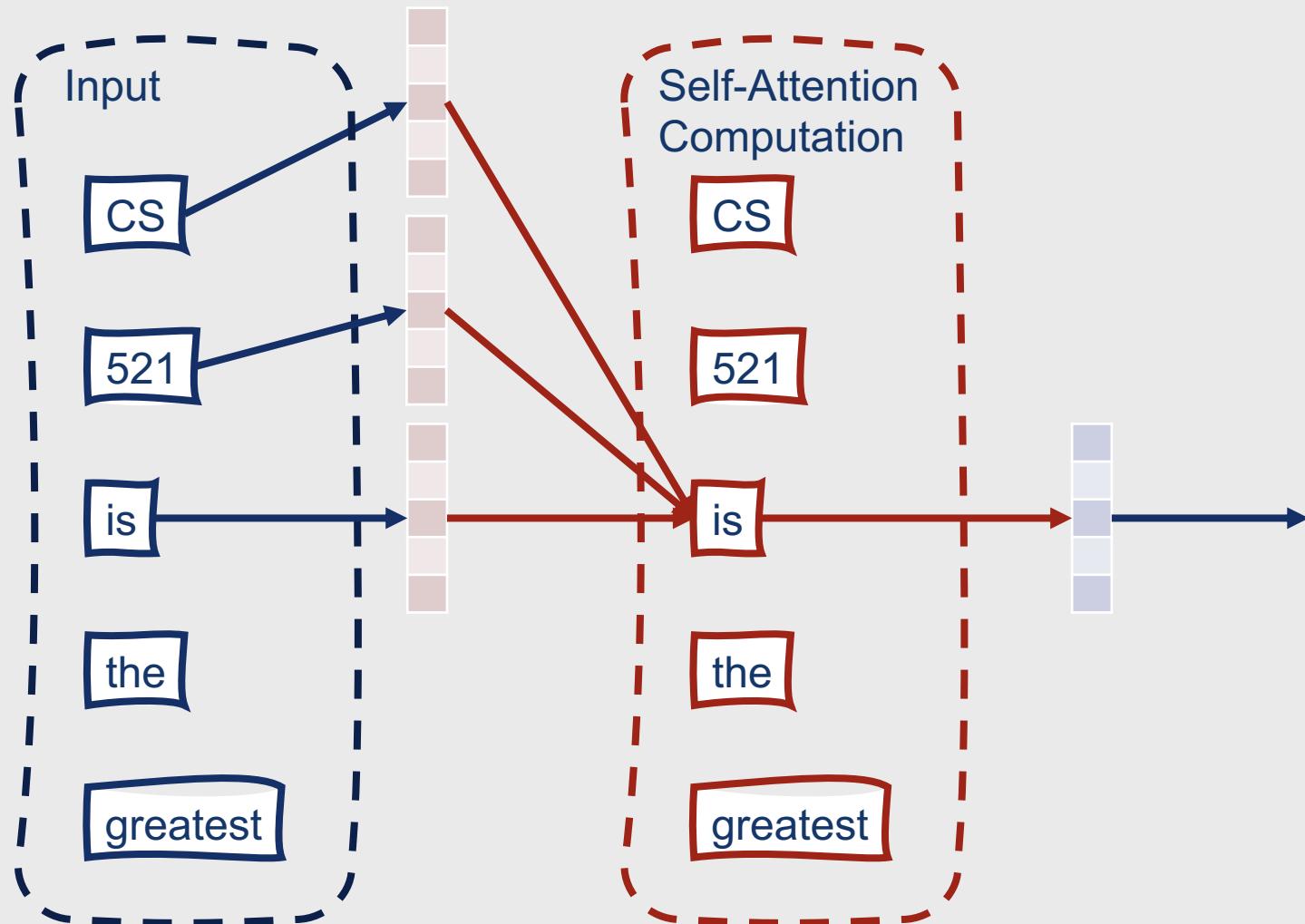
# Self-Attention



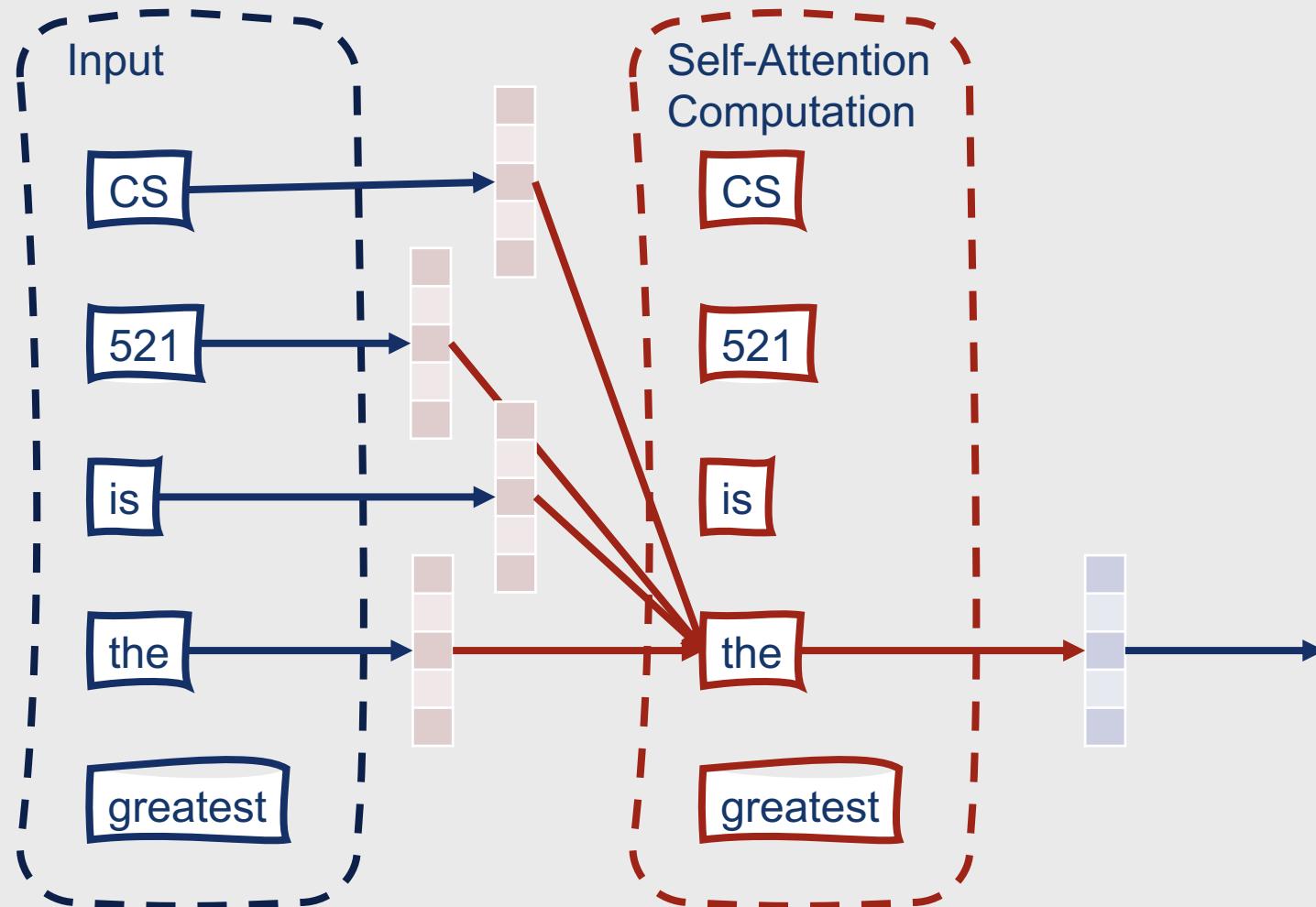
# Self-Attention



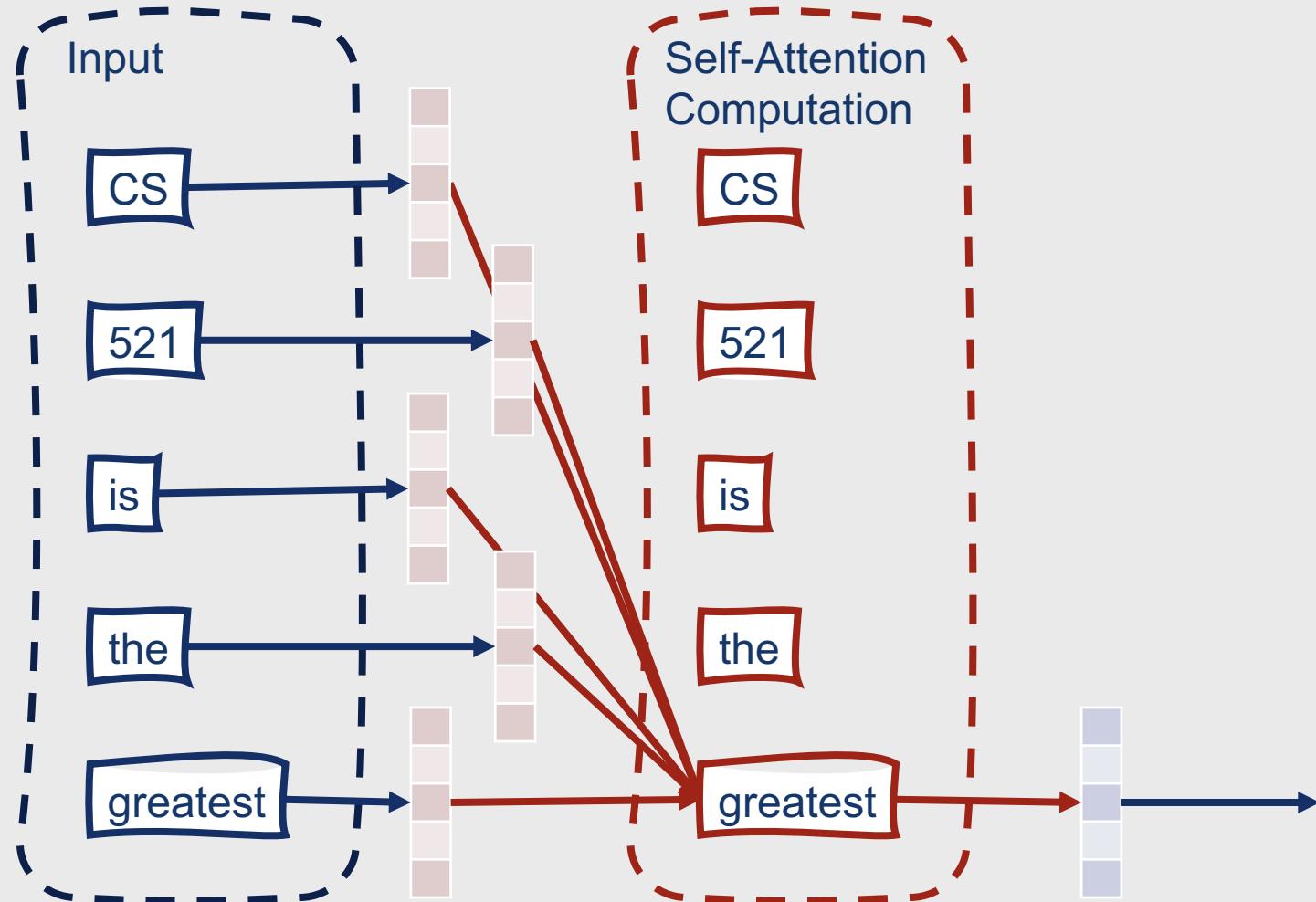
# Self-Attention



# Self-Attention



# Self-Attention



# Computing Self-Attention

- Simplest method:
  - Take the dot product between a given input element  $x_i$  and each input element  $(x_1, \dots, x_i)$  up until that point
    - $\text{score}(x_i, x_j) = x_i \cdot x_j$
  - Apply softmax normalization to create a vector of weights,  $\alpha_i$ , indicating proportional relevance of each sequence element to the current focus of attention,  $x_i$ 
    - $\alpha_{ij} = \text{softmax}(\text{score}(x_i, x_j)) \forall j \leq i = \frac{e^{\text{score}(x_i, x_j)}}{\sum_{k=1}^i e^{\text{score}(x_i, x_k)}} \forall j \leq i$
  - Take the sum of inputs thus far weighted by  $\alpha_i$  to produce an output  $y_i$ 
    - $y_i = \sum_{j \leq i} \alpha_{ij} x_j$

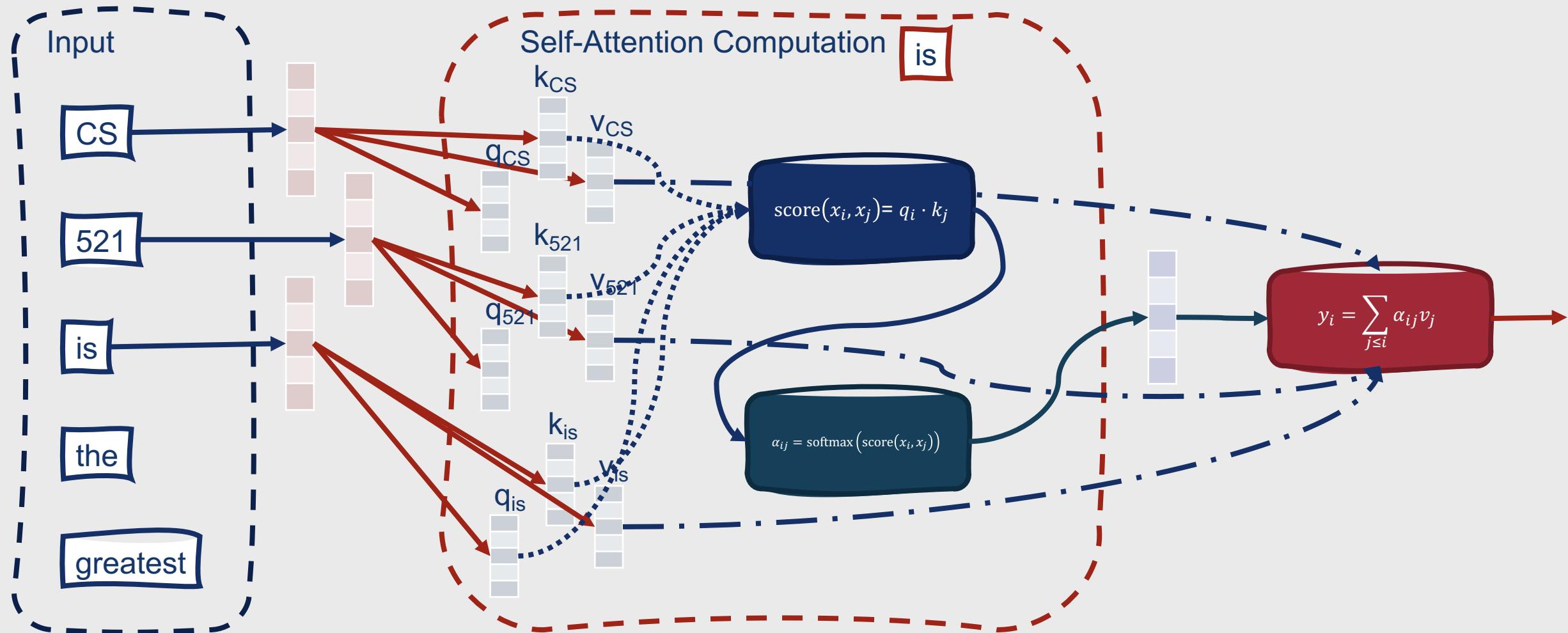
# How do Transformers learn?

- Continually updating weight matrices applied to inputs
- Weight matrices are learned for each of three roles when computing self-attention:
  - **Query:** The focus of attention when it is being compared to inputs up until that point,  $W^Q$
  - **Key:** An input that is being compared to the focus of attention,  $W^K$
  - **Value:** A value being used to compute the output for the current focus of attention,  $W^V$

# Training Transformers

- Weight matrices are applied to inputs in the context of their respective roles
  - $q_i = W^Q x_i$
  - $k_i = W^K x_i$
  - $v_i = W^V x_i$
- Then, we can update our equations for computing self-attention so that these roles are reflected in them:
  - $\text{score}(x_i, x_j) = q_i \cdot k_j$
  - $\alpha_{ij} = \text{softmax}(\text{score}(x_i, x_j)) \forall j \leq i$
  - $y_i = \sum_{j \leq i} \alpha_{ij} v_j$

# Self-Attention

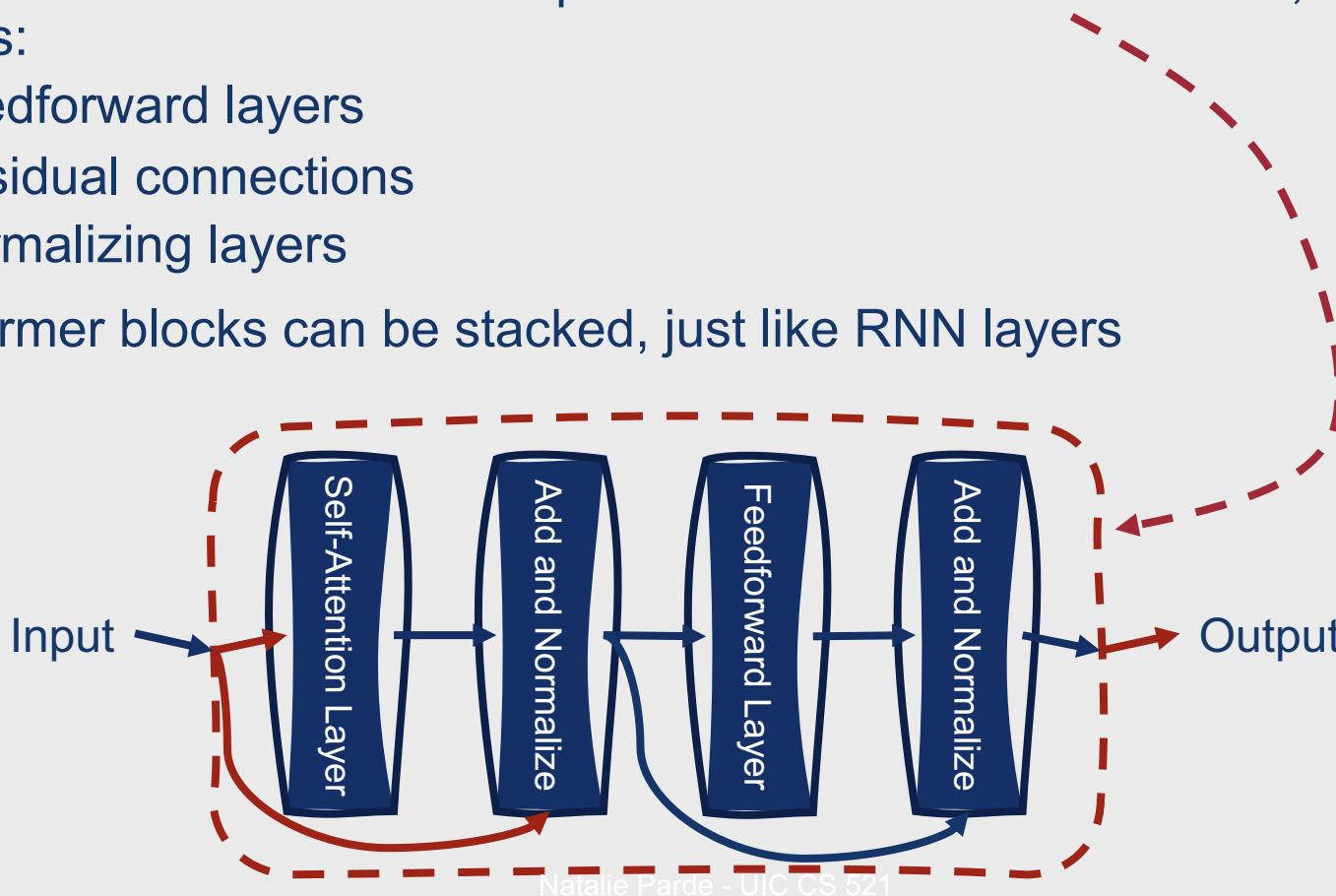


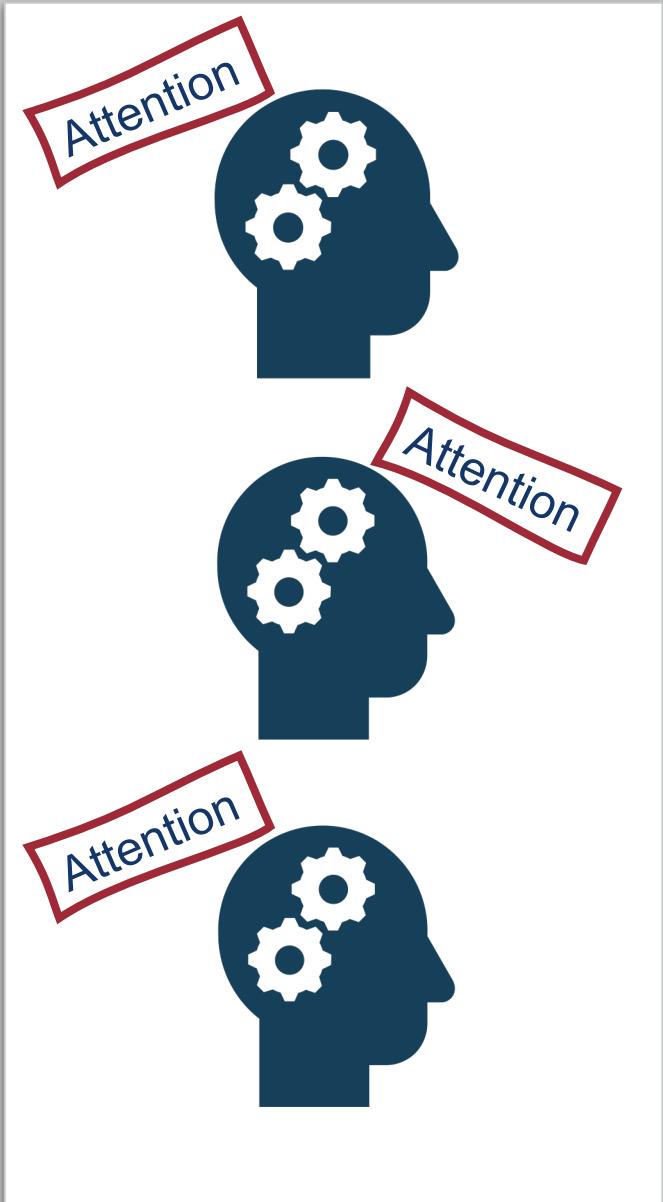
# Practical Considerations

- Combining a dot product with an exponential (as in softmax) may lead to arbitrarily large values
- It is common to scale the scoring function based on the dimensionality of the key (and query) vectors,  $d_k$ 
  - $\text{score}(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$
- Each  $y_i$  is computed independently, so we can parallelize computations using efficient matrix multiplication routines where  $X$  is a matrix containing all input embeddings
  - $Q = W^Q X$
  - $K = W^K X$
  - $V = W^V X$
  - $\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ 
    - Zero out the upper triangular portion of the comparison matrix in a language modeling setting to avoid including knowledge of future words!

# Transformer Blocks

- Self-attention is the central component of a **Transformer block**, which also includes:
  - Feedforward layers
  - Residual connections
  - Normalizing layers
- Transformer blocks can be stacked, just like RNN layers





# Multihead Attention

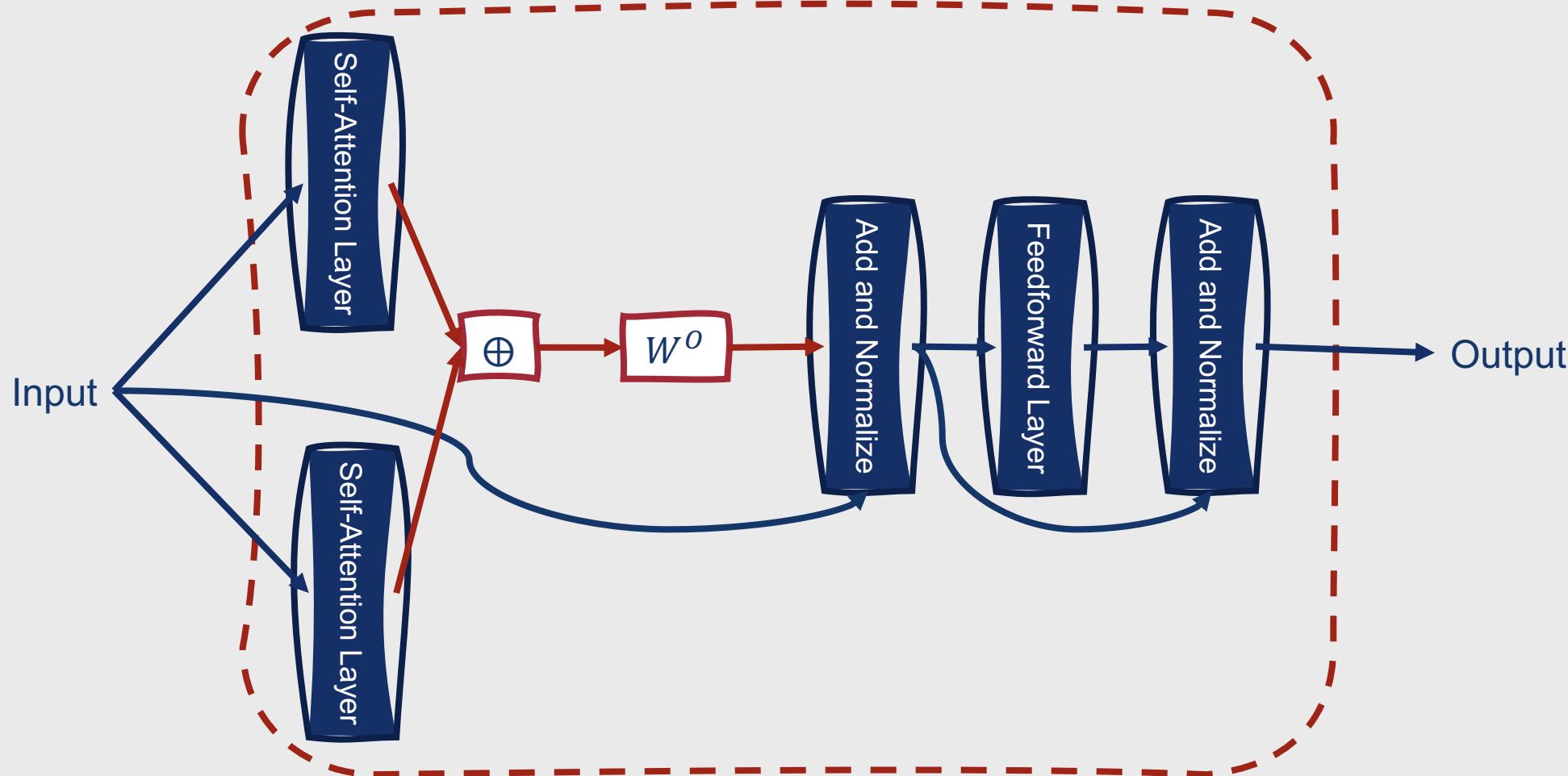
- Each self-attention layer represents a single **attention head**
- However, words can relate to one another in many different ways!
- **Multihead attention** places multiple attention heads in parallel in the Transformer model
  - Since each attention head has its own set of weights, each one can learn different aspects of the relations between input elements at the same level of abstraction

# Computing Multihead Attention

---

- Each head in the self-attention layer is parameterized with its own weights
  - $Q = W_i^Q X$
  - $K = W_i^K X$
  - $V = W_i^V X$
- The output of a multihead attention layer with  $n$  heads comprises  $n$  vectors of equal length
- These heads are concatenated and then reduced to the original input/output dimensionality
  - $\text{head}_i = \text{SelfAttention}(W_i^Q X, W_i^K X, W_i^V X)$
  - $\text{MultiheadAttention}(Q, K, V) = W^O(\text{head}_1 \oplus \text{head}_2 \oplus \dots \oplus \text{head}_n)$

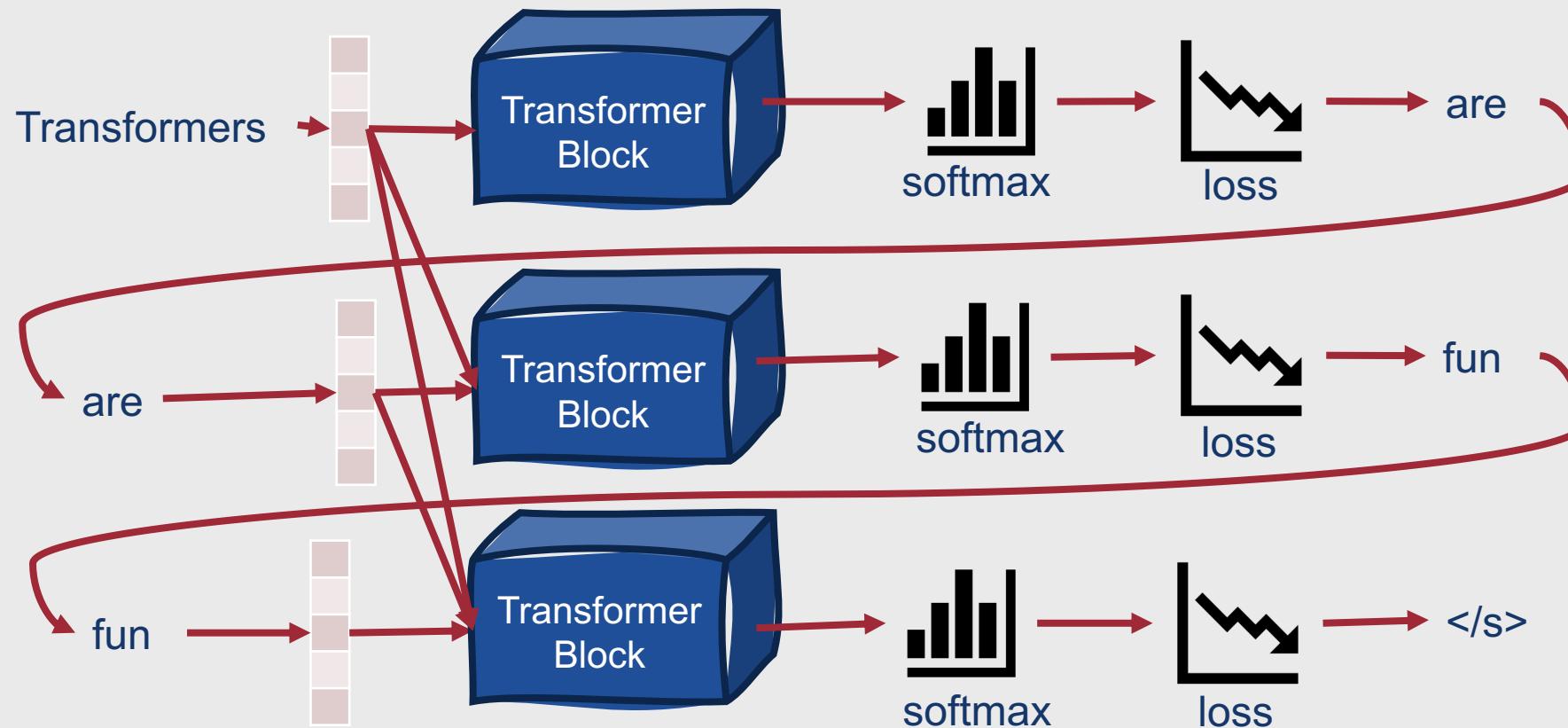
# Multihead Attention



# Positional Embeddings

- Since Transformers don't make use of recurrent connections, they instead employ separate **positional embeddings** to encode positionality
  - Randomly initialize an embedding for each input position
  - Update weights during the training process
  - Input embedding with positional information = word embedding + positional embedding
- Static functions mapping positions to vectors can be used as an alternative

# Transformers as Autoregressive Language Models



# Summary: Deep Learning Architectures for Sequence Processing

- Review: **Feedforward neural networks** are comprised of interconnected layers of computing units
- Neural networks are trained used **backpropagation**
- **Convolutional neural networks** were originally designed for image processing, but can be useful for learning phrases and fundamental structural components
- **Recurrent neural networks** consider temporal sequence
  - **LSTMs**, **GRUs**, and **BiLSTMs** are all variations of the “vanilla” RNN model
- **Transformers** use self-attention to learn which components of an input are important for processing one another