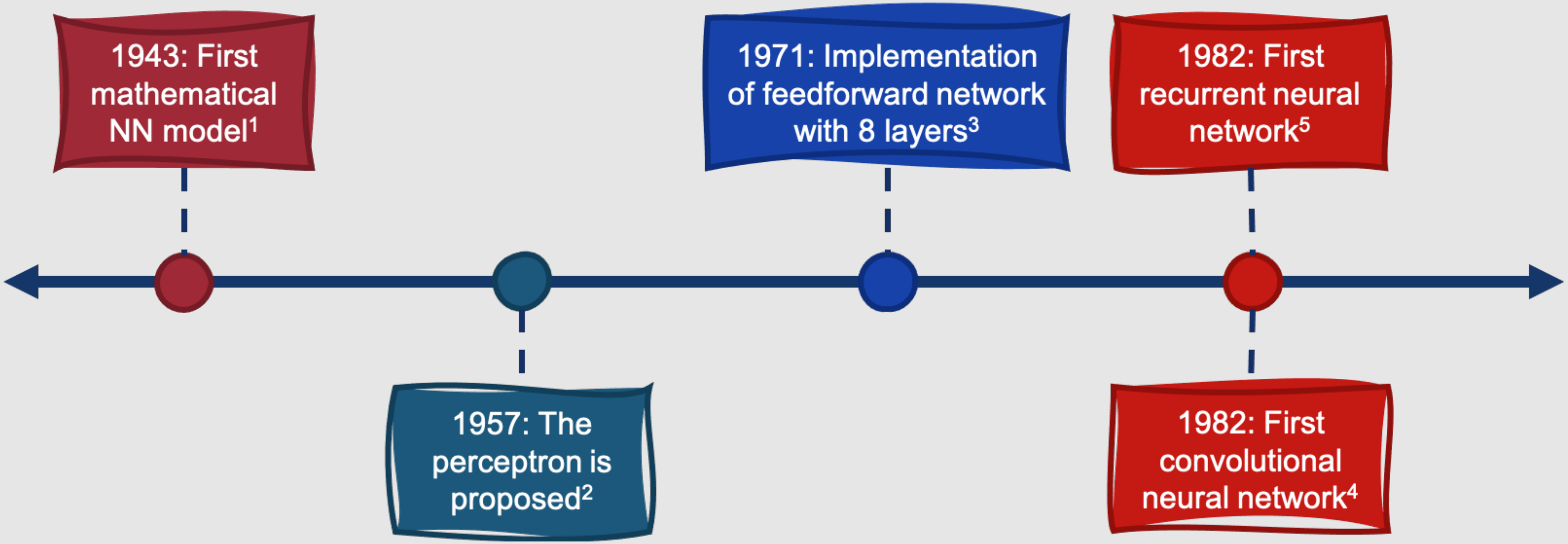
Deep Learning Architectures for Sequence Processing

Natalie Parde

UIC CS 521

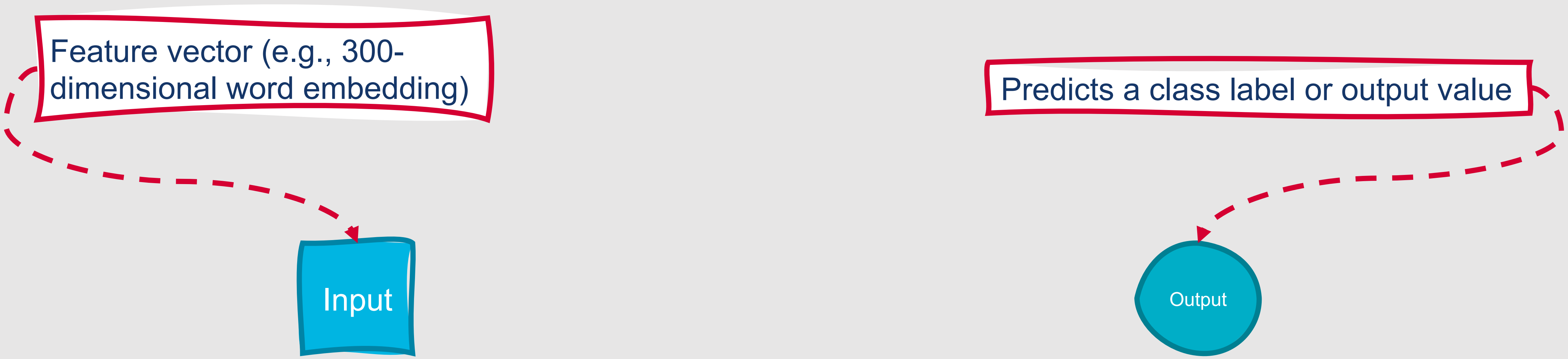
Review: Neural Networks Basics


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

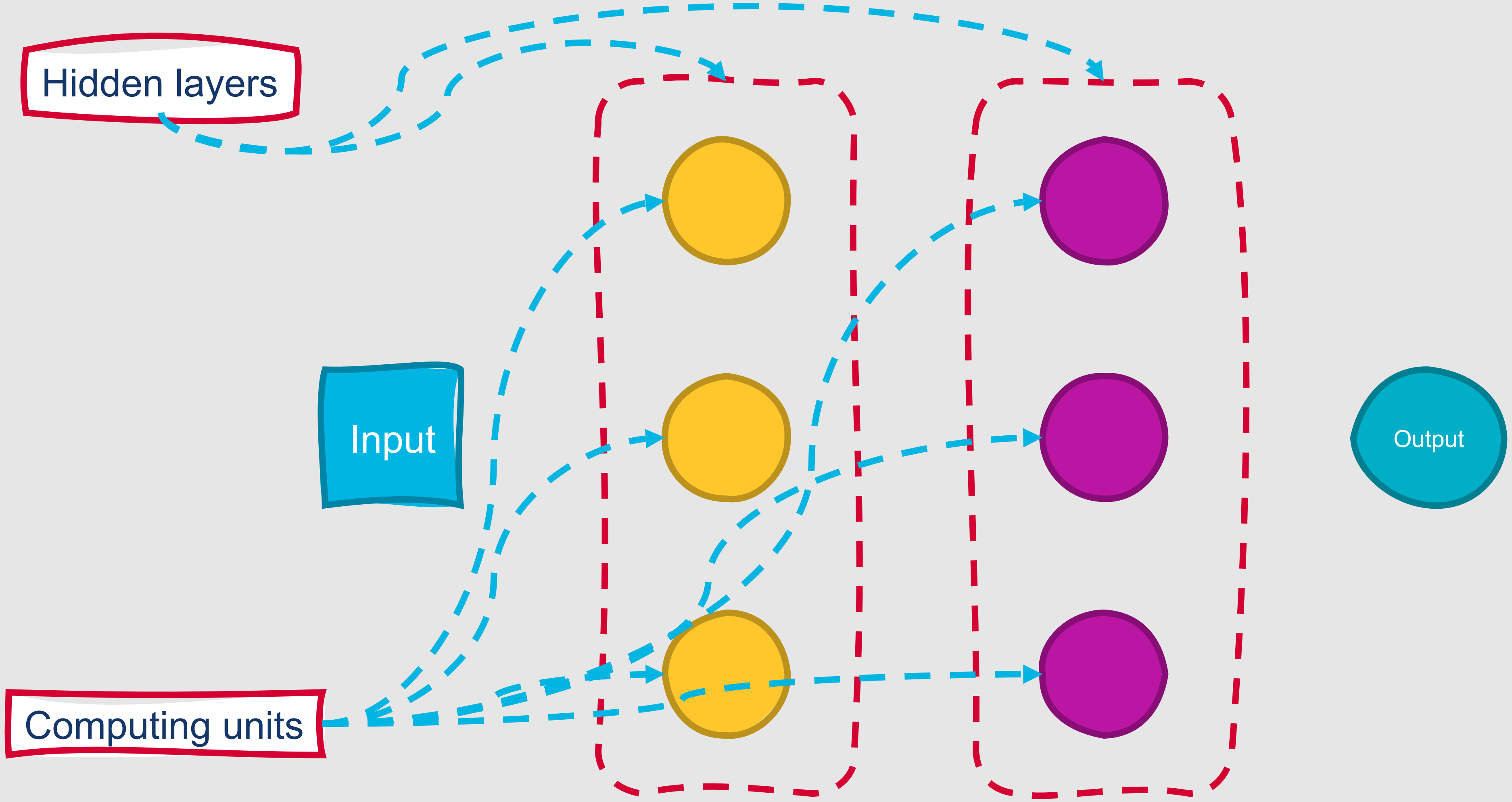
Feedforward Neural Networks

- 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

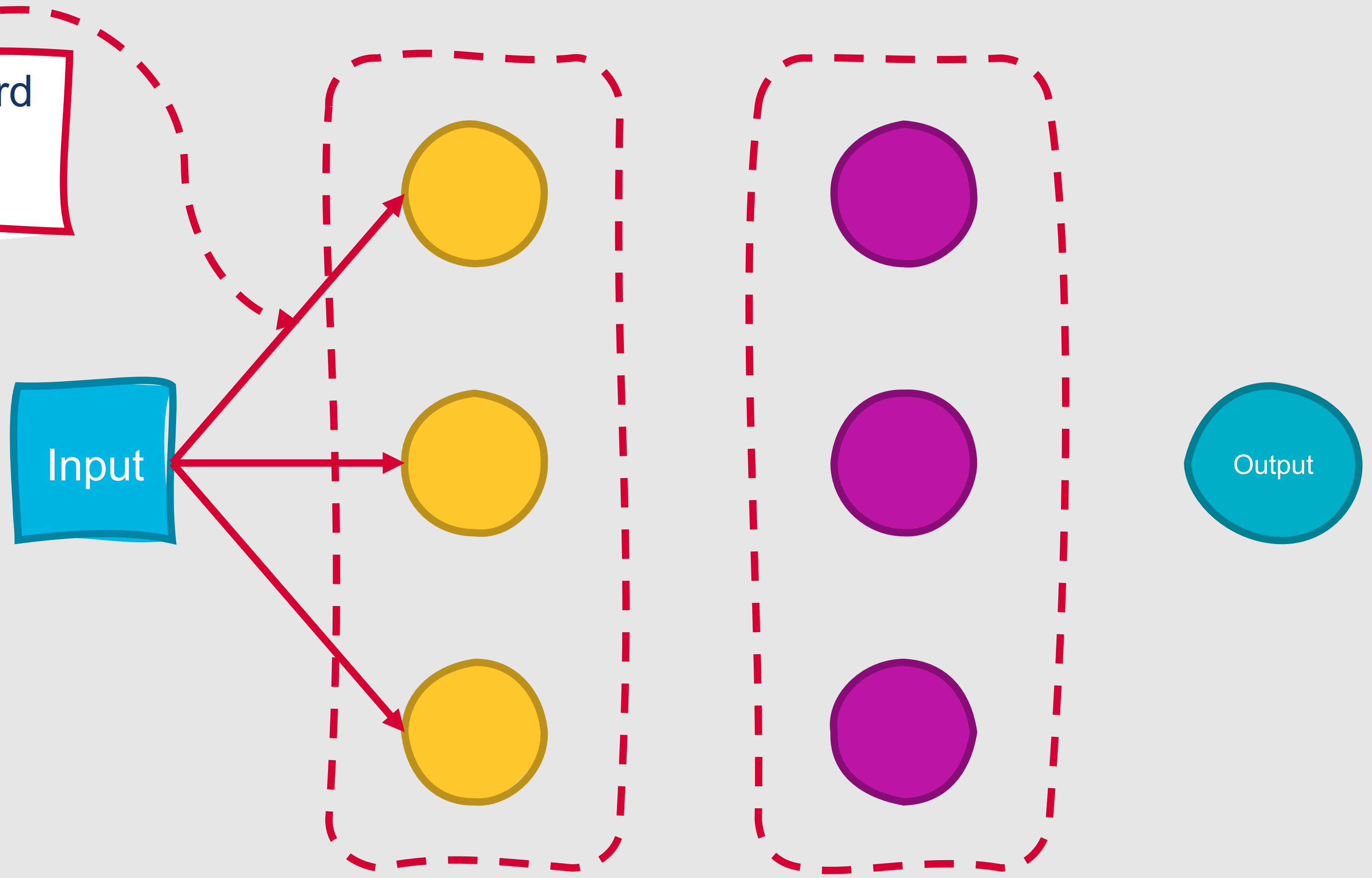

Feature vector (e.g., 300-dimensional word embedding)
Predicts a class label or output value
Input
Output

Feedforward Neural Networks



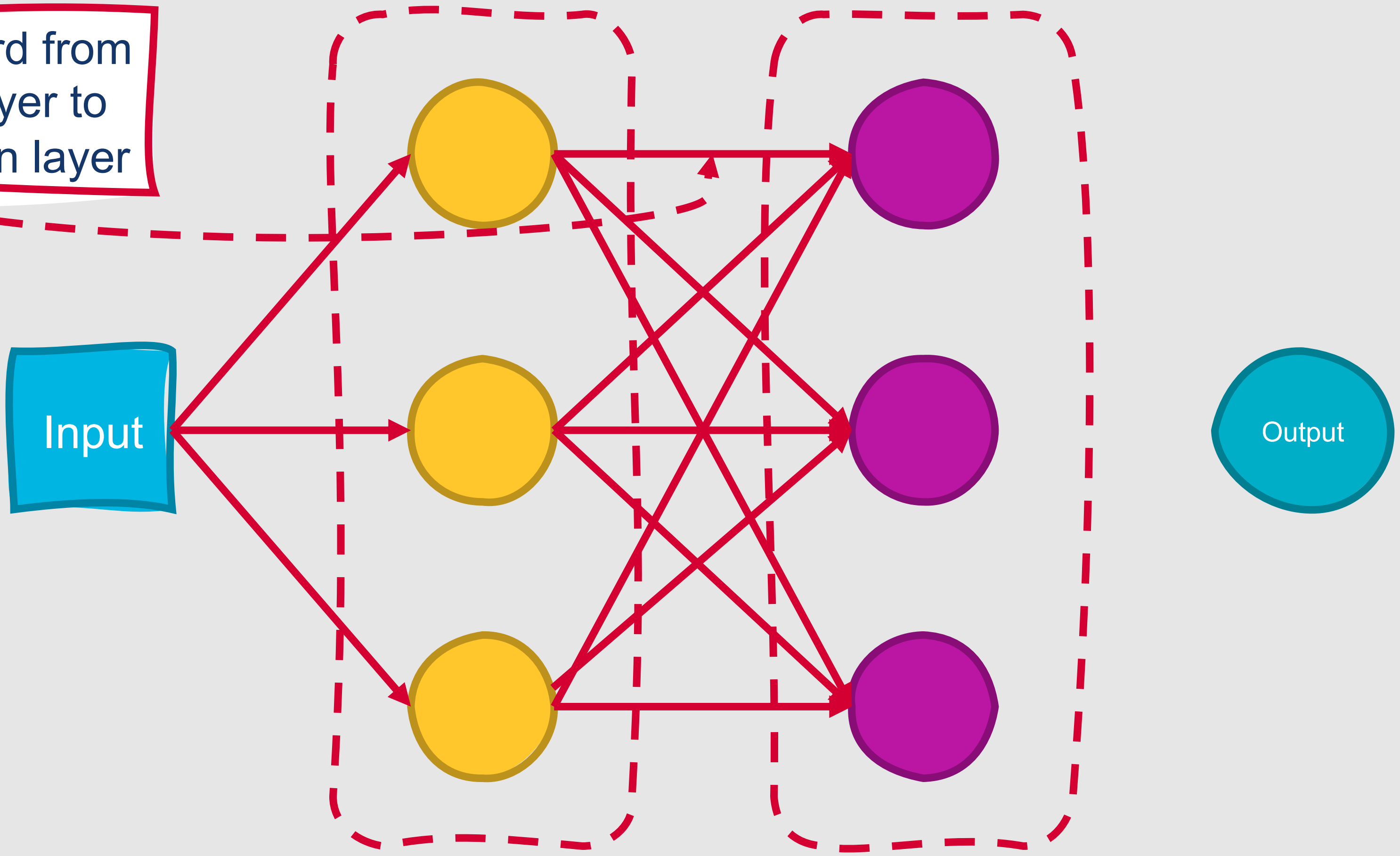
Feedforward Neural Networks

Data is fed forward from input to the first hidden layer



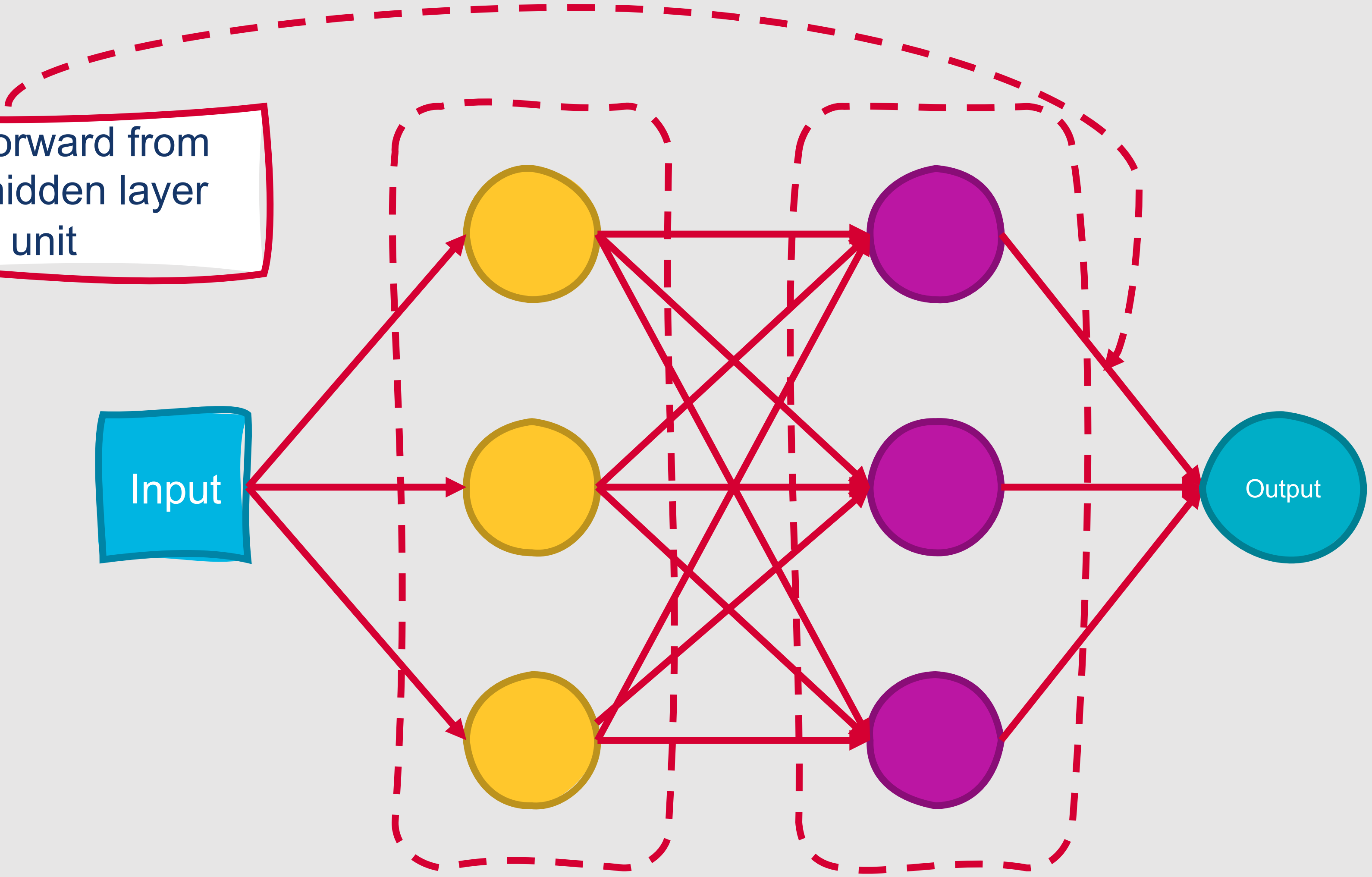
Feedforward Neural Networks

Data is fed forward from the first hidden layer to the second hidden layer

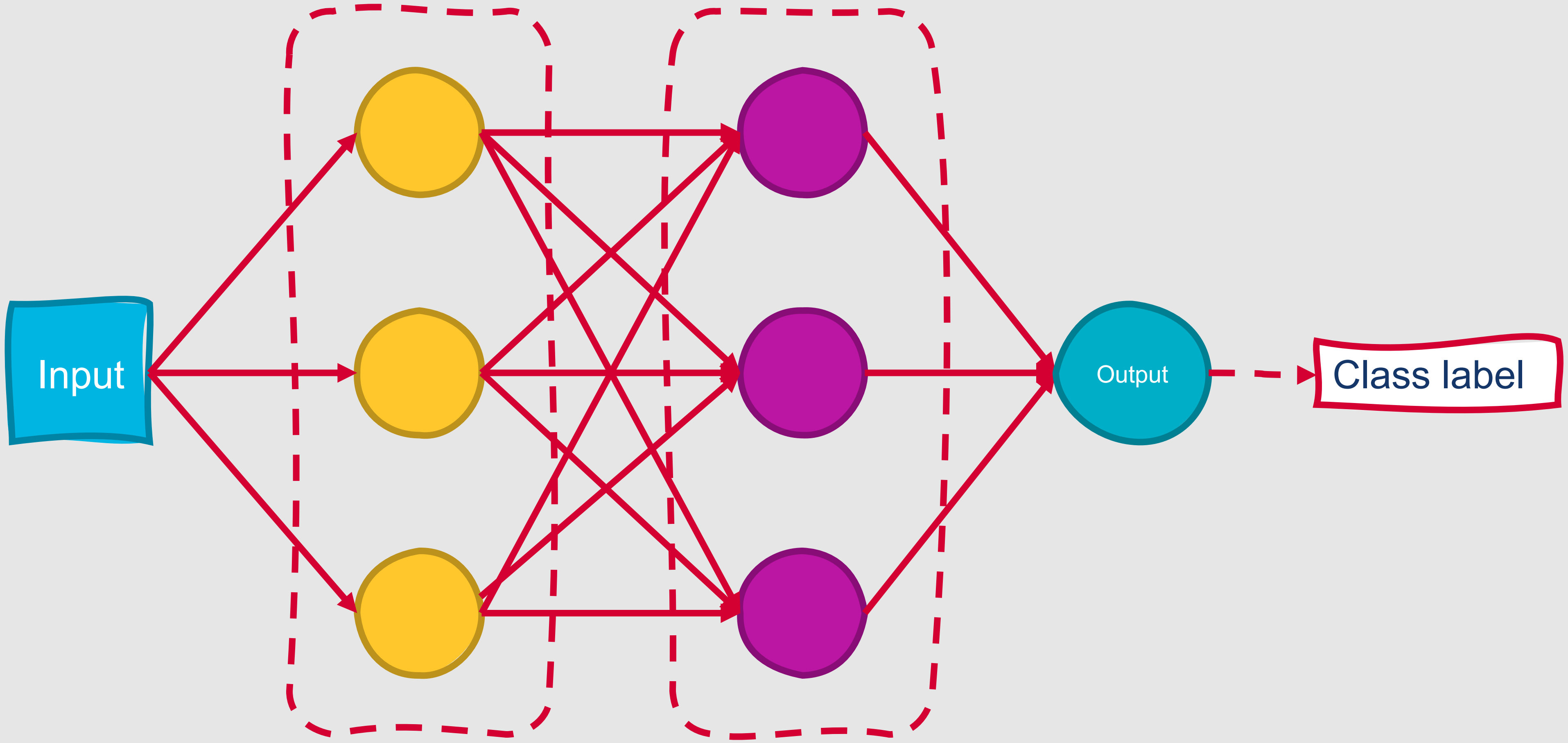


Feedforward Neural Networks

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

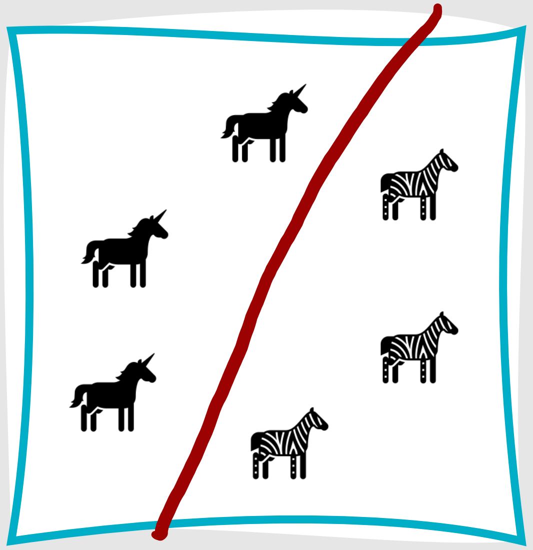


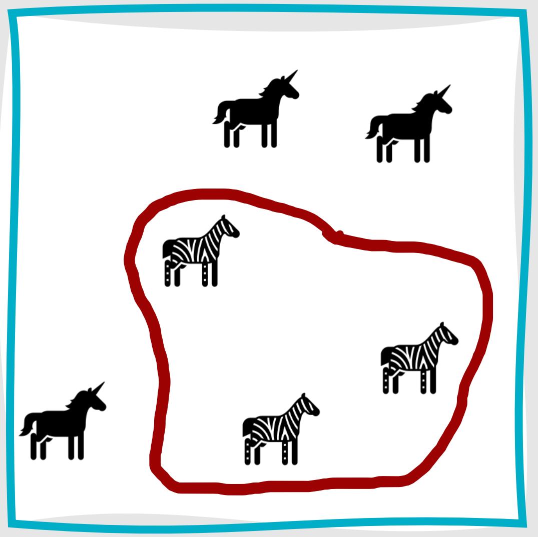
Feedforward Neural Networks



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

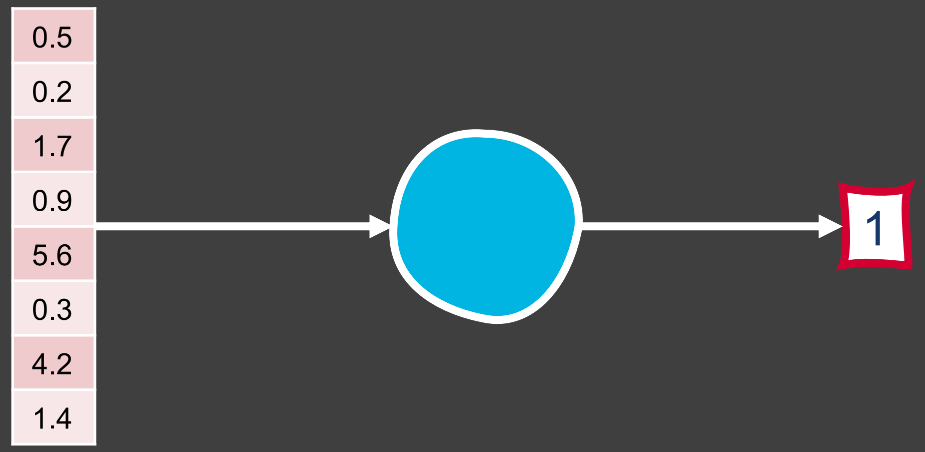
- Traditional classification algorithms usually assume that data is **linearly separable**
- Neural networks are better equipped to learn complex, **nonlinear** separations between data classes





Building Blocks for Neural Networks

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



Computational Units

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

Computational Units

- The weighted sum of inputs computes a **linear function** of x
- We pass this sum through one of many possible nonlinear functions, commonly referred to as **activations**
- The output of a computation unit is thus the **activation value** for the unit, y
  - $y = f(z) = f(w \cdot x + b)$

There are many different activation functions!

exponential linear unit (elu)

softmax

scaled exponential linear unit (selu)

softplus

softsign

rectified linear unit (relu)

hyperbolic tangent (tanh)

sigmoid

There are many different activation functions!

exponential linear unit (elu)

softmax

scaled exponential linear unit (selu)

softplus

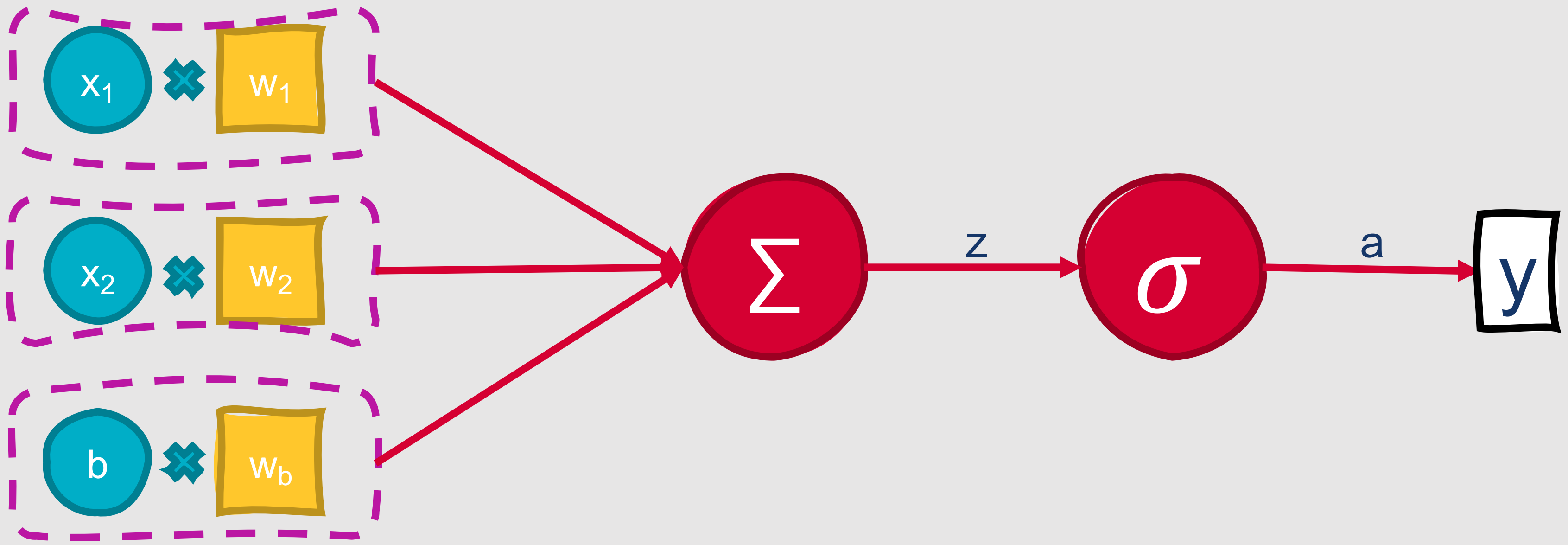
softsign

rectified linear unit (relu)

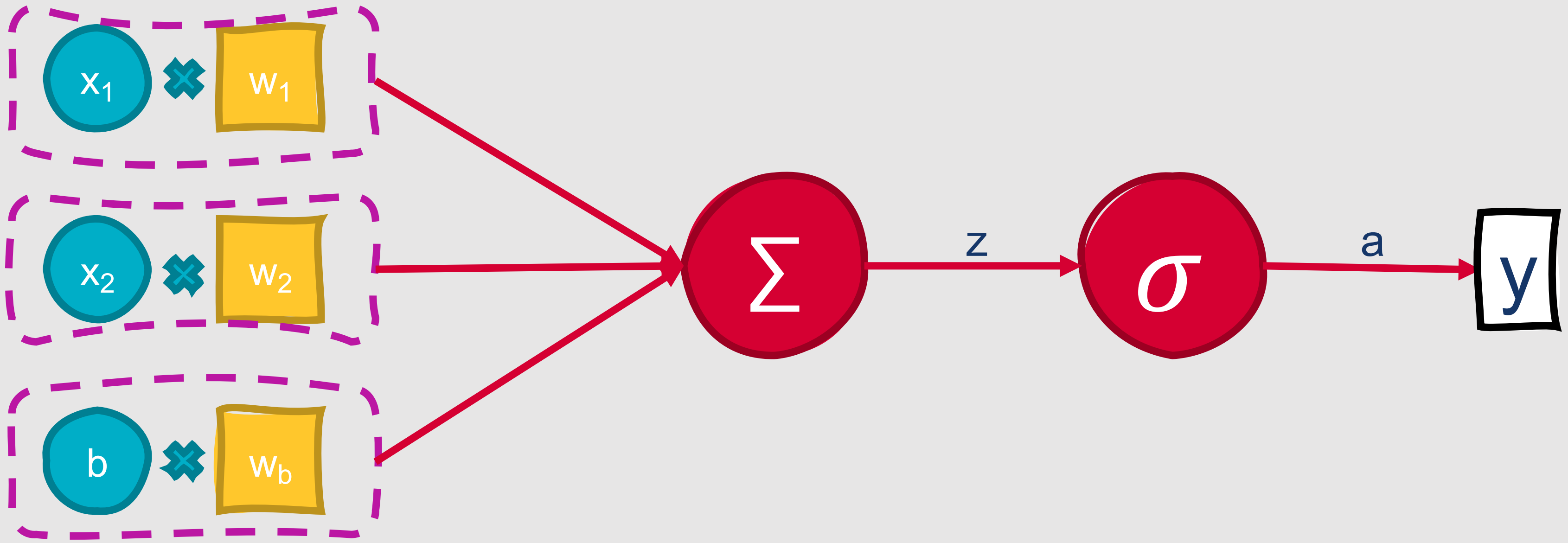
hyperbolic tangent (tanh)

sigmoid

Computational Unit with Sigmoid Activation



Example: Computational Unit with Sigmoid Activation

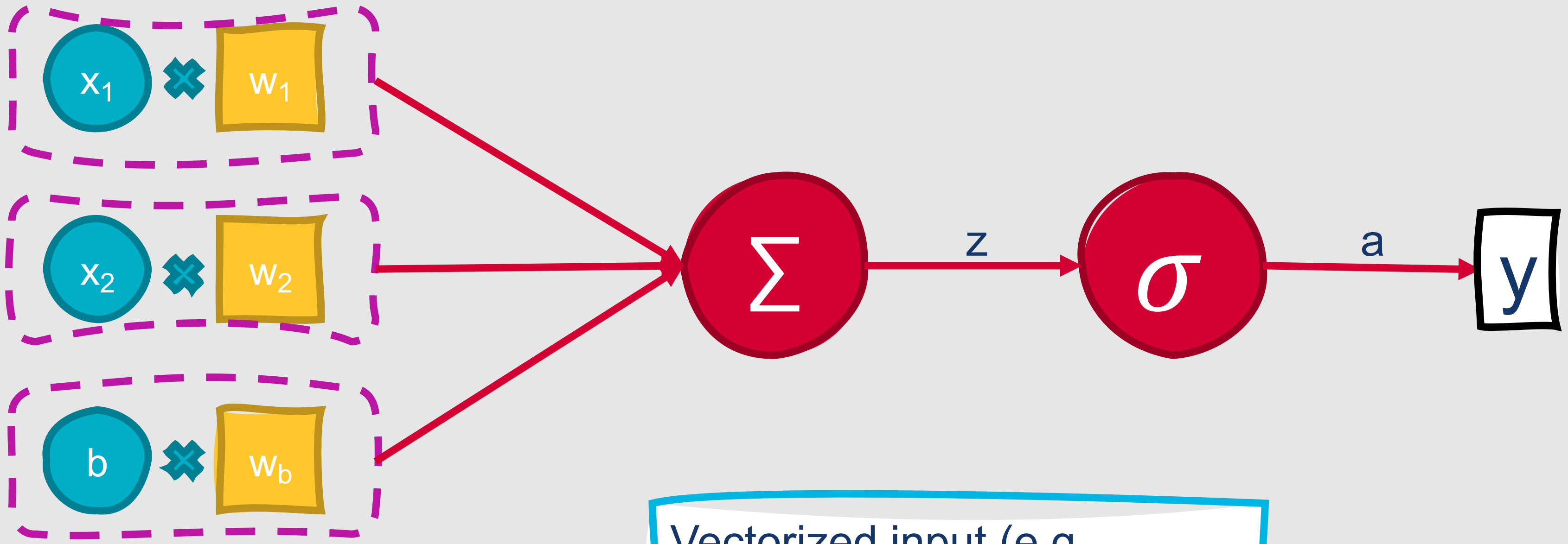


Input: “beautiful brutalist architecture”

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

Bias: 1.0

Example: Computational Unit with Sigmoid Activation

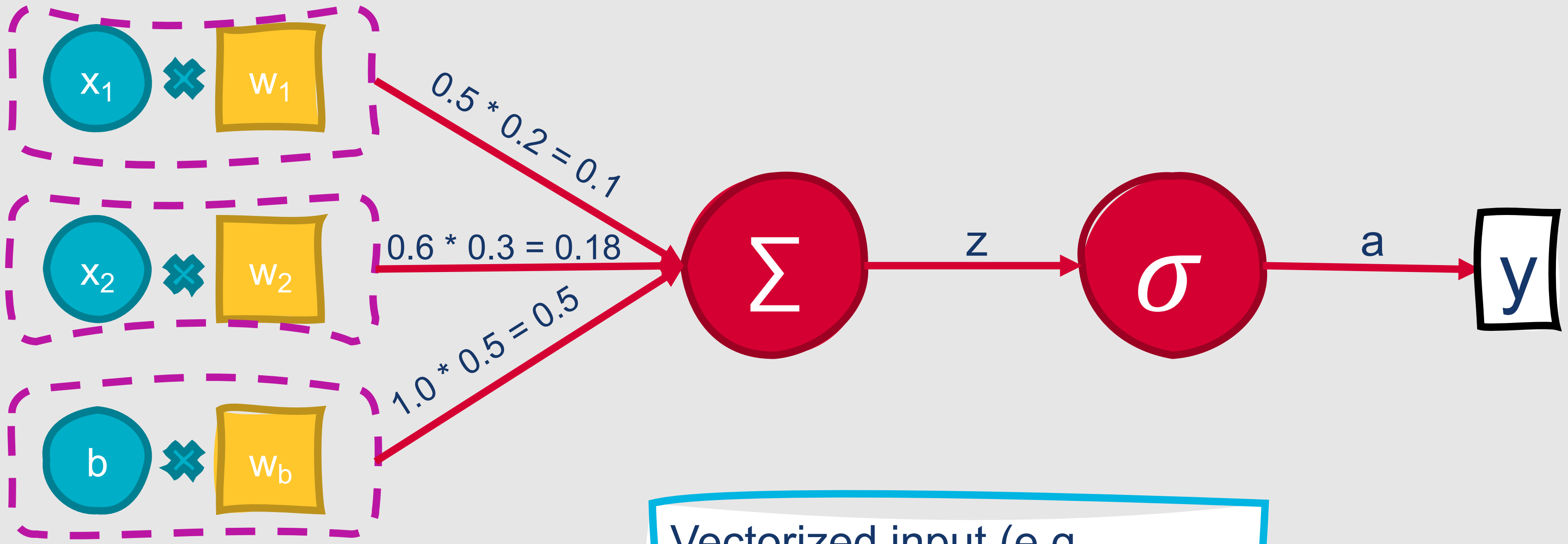


Input: “beautiful brutalist architecture” → Vectorized input (e.g., averaged *n*-dimensional embeddings for “beautiful,” “brutalist,” and “architecture”) → [0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with Sigmoid Activation

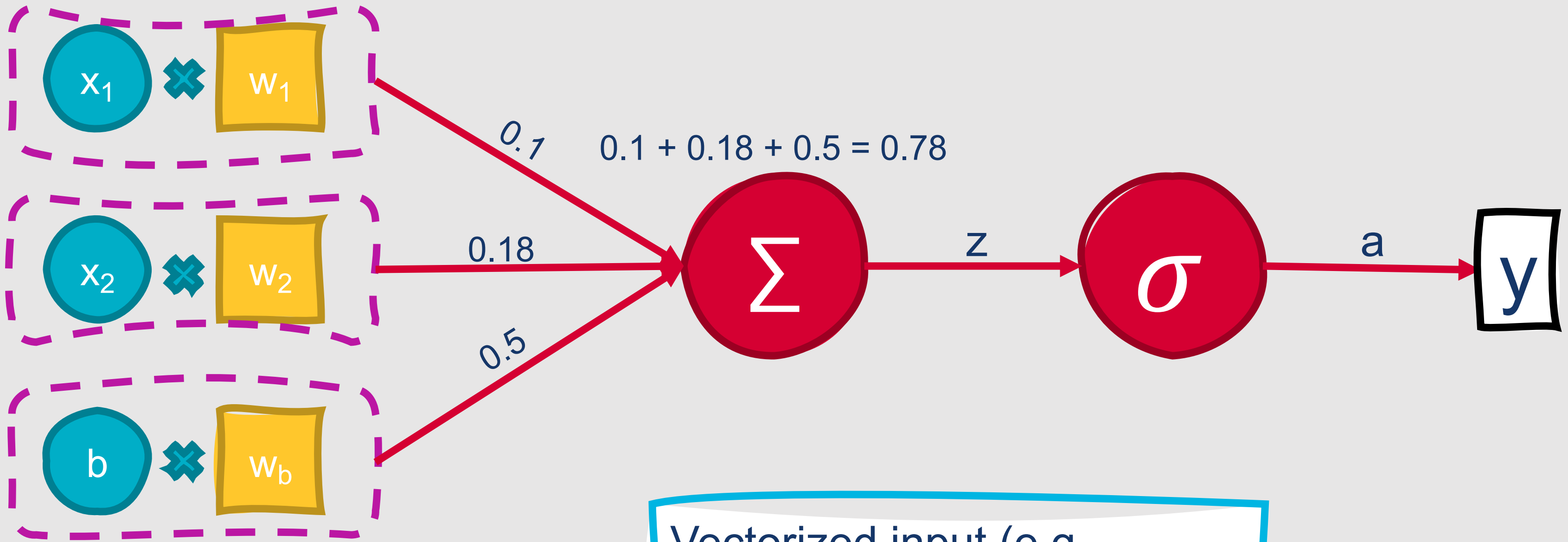


Input: "beautiful brutalist architecture" → Vectorized input (e.g., averaged *n*-dimensional embeddings for "beautiful," "brutalist," and "architecture") → [0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with Sigmoid Activation

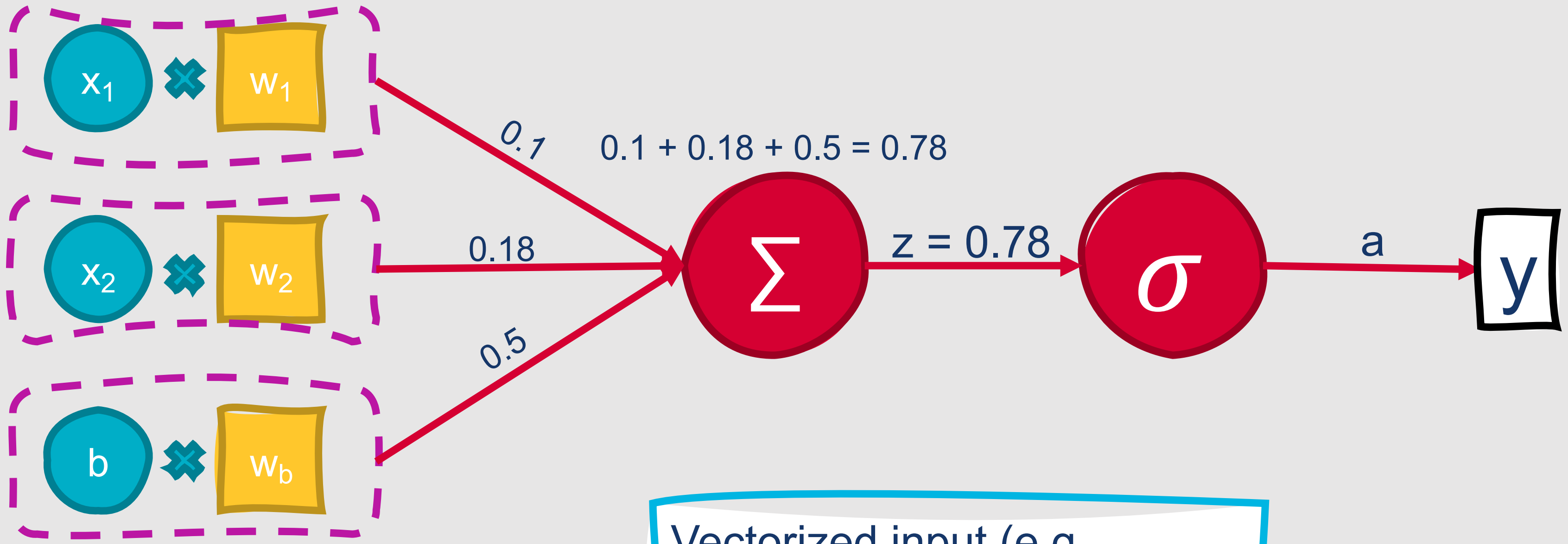


Input: “beautiful brutalist architecture” → Vectorized input (e.g., averaged *n*-dimensional embeddings for “beautiful,” “brutalist,” and “architecture”) → [0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with Sigmoid Activation

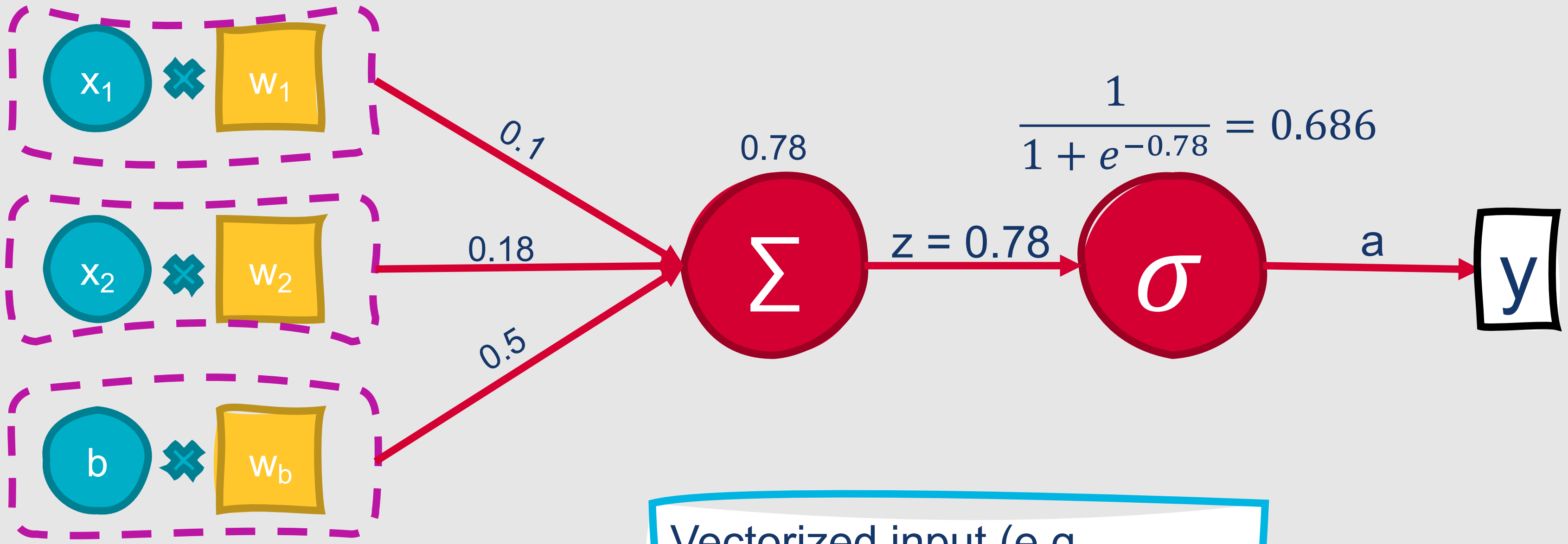


Input: "beautiful brutalist architecture" → Vectorized input (e.g., averaged *n*-dimensional embeddings for "beautiful," "brutalist," and "architecture") → [0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with Sigmoid Activation

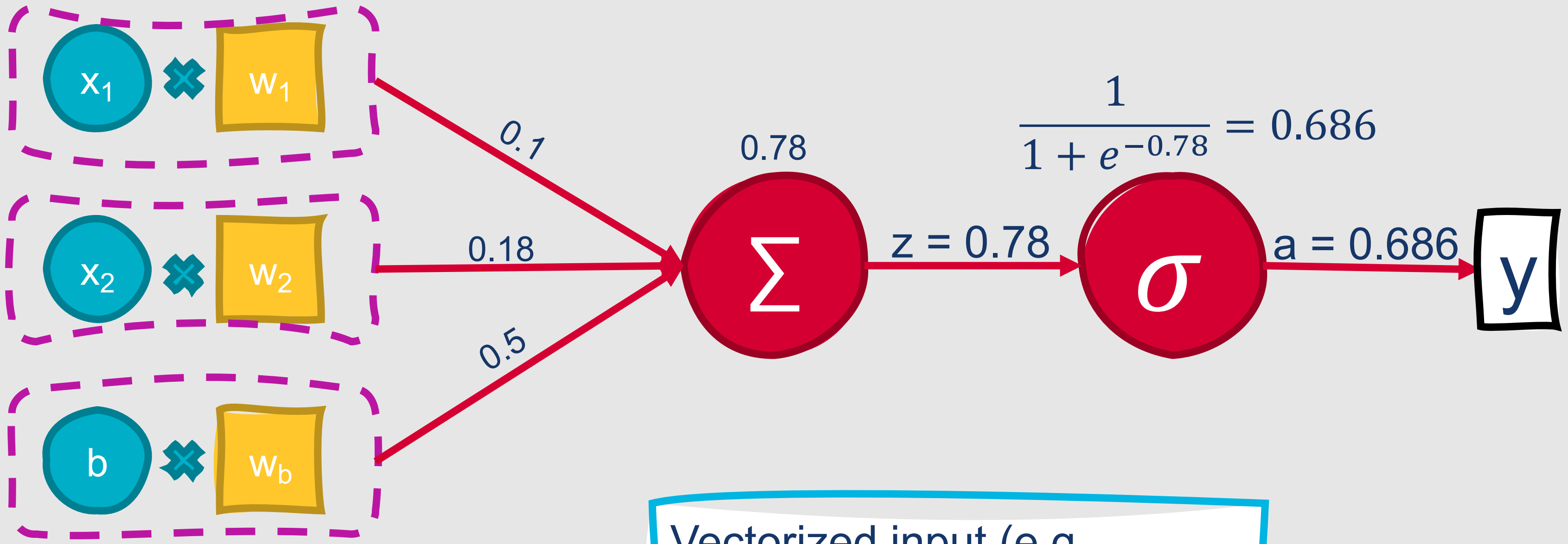


Input: "beautiful brutalist architecture" → Vectorized input (e.g., averaged *n*-dimensional embeddings for "beautiful," "brutalist," and "architecture") → [0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with Sigmoid Activation



Input: “beautiful brutalist architecture”

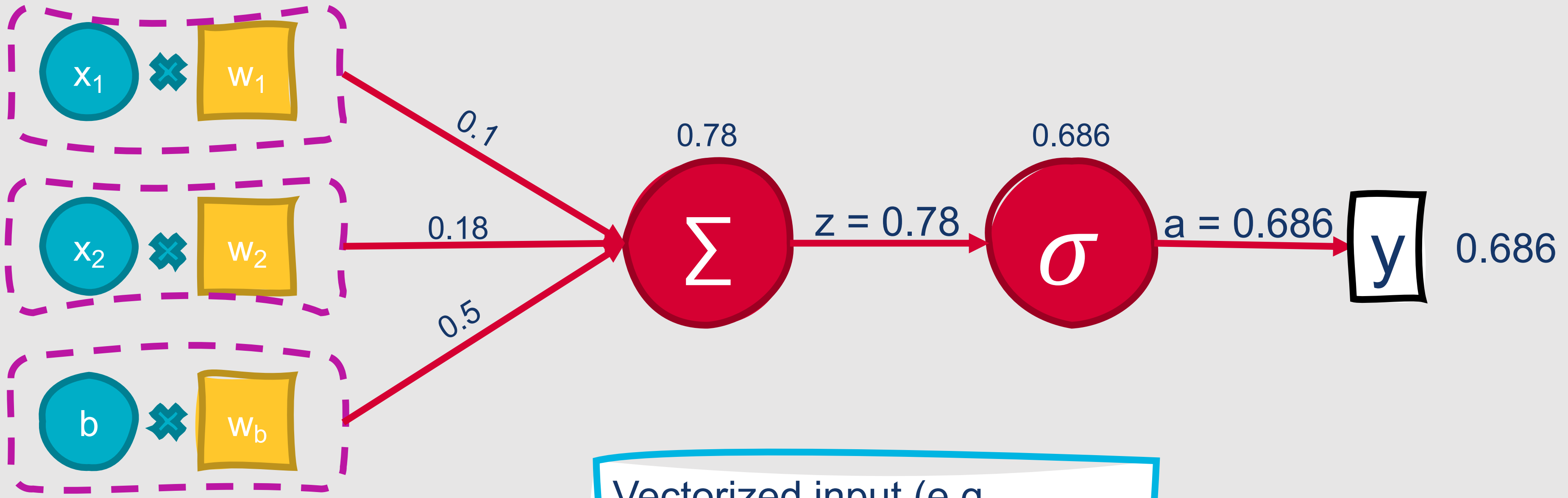
Vectorized input (e.g., averaged *n*-dimensional embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

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

Bias: 1.0

Example: Computational Unit with Sigmoid Activation



Input: “beautiful brutalist architecture”

Vectorized input (e.g., averaged *n*-dimensional embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

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

Bias: 1.0

Particularly Popular Activation Functions

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

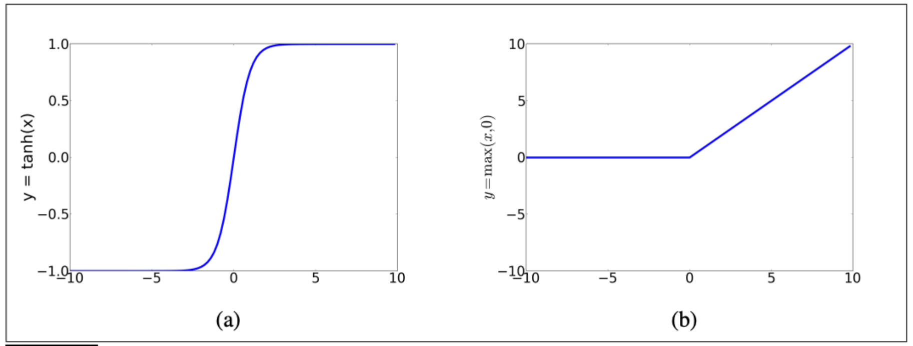


Figure 7.3 The tanh and ReLU activation functions.

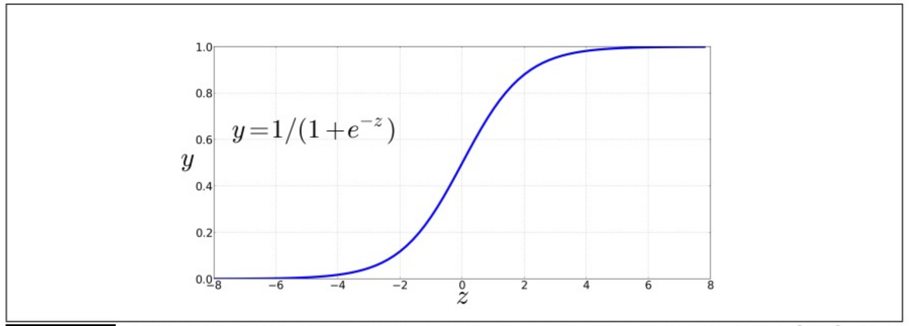
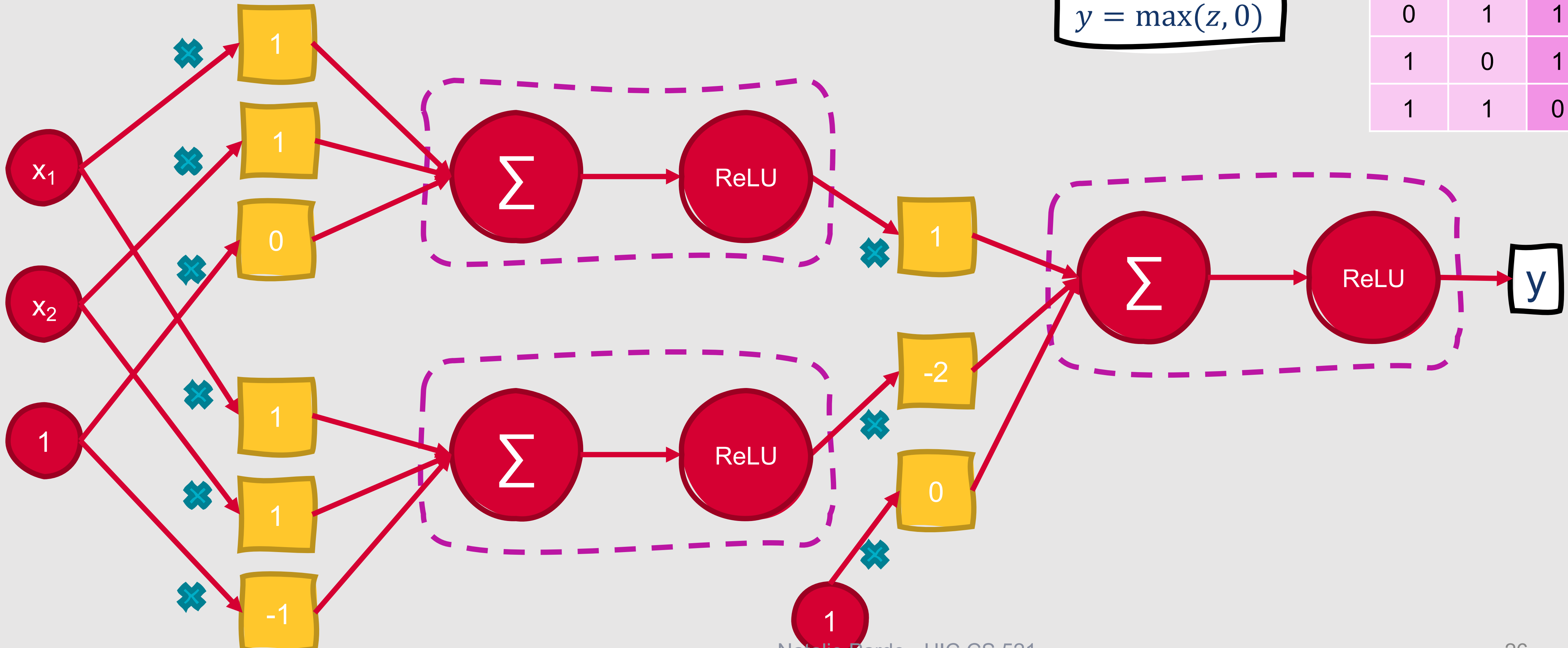


Figure 7.1 The sigmoid function takes a real value and maps it to the range $[0,1]$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

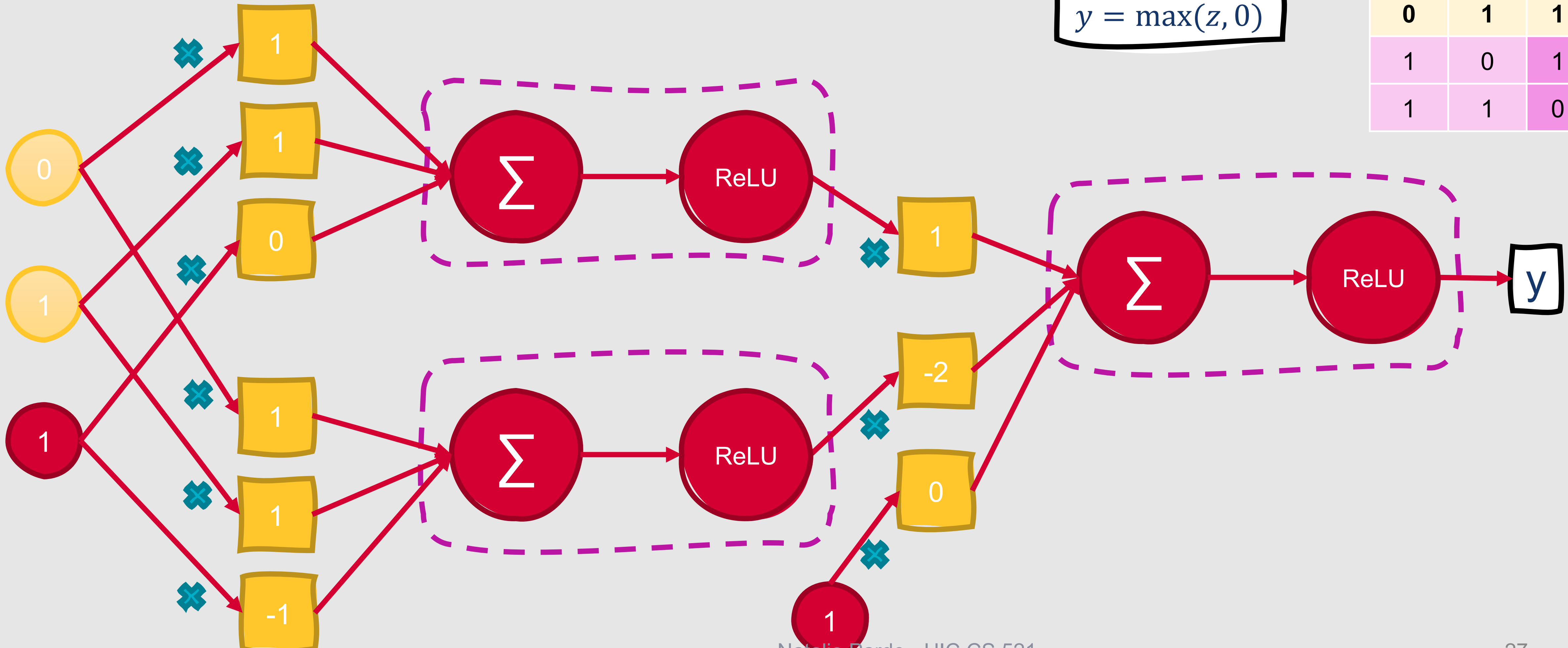
Combining Computational Units



$y = \max(z, 0)$

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

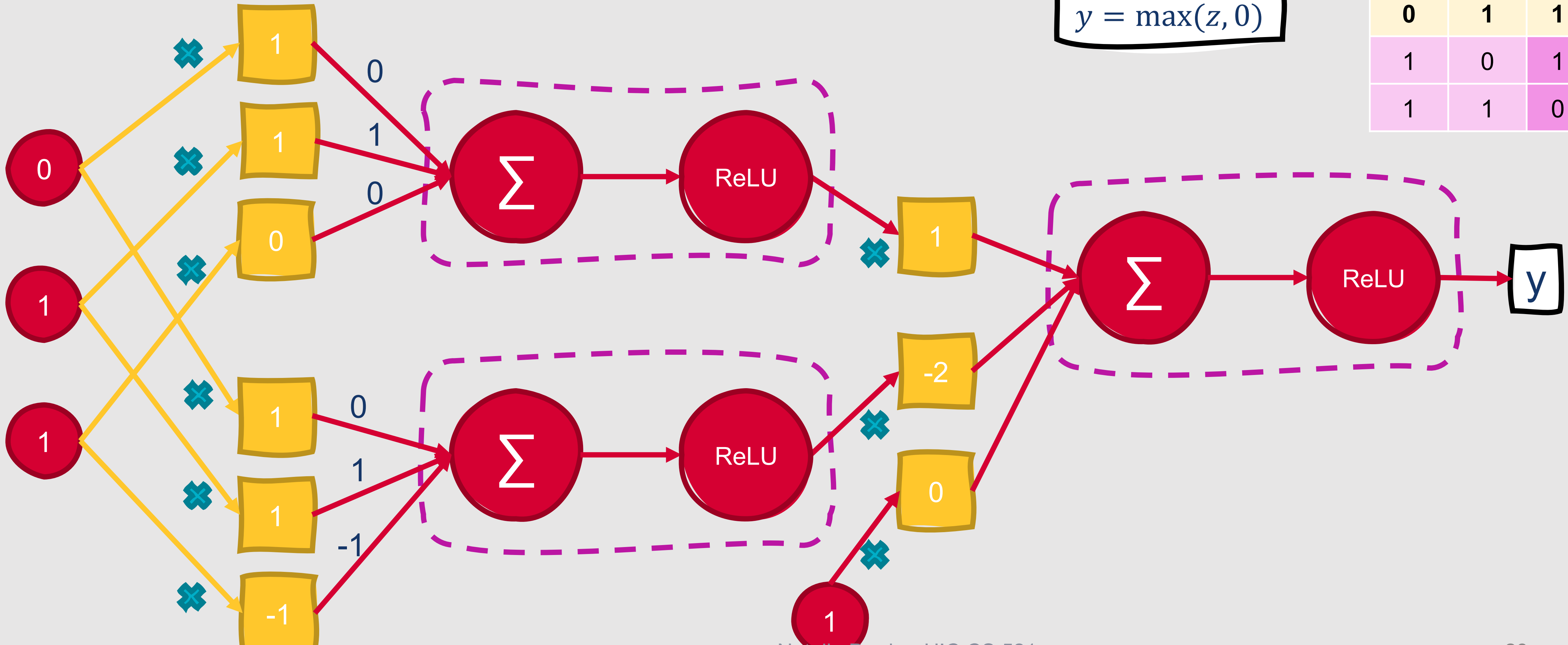
Combining Computational Units



$y = \max(z, 0)$

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

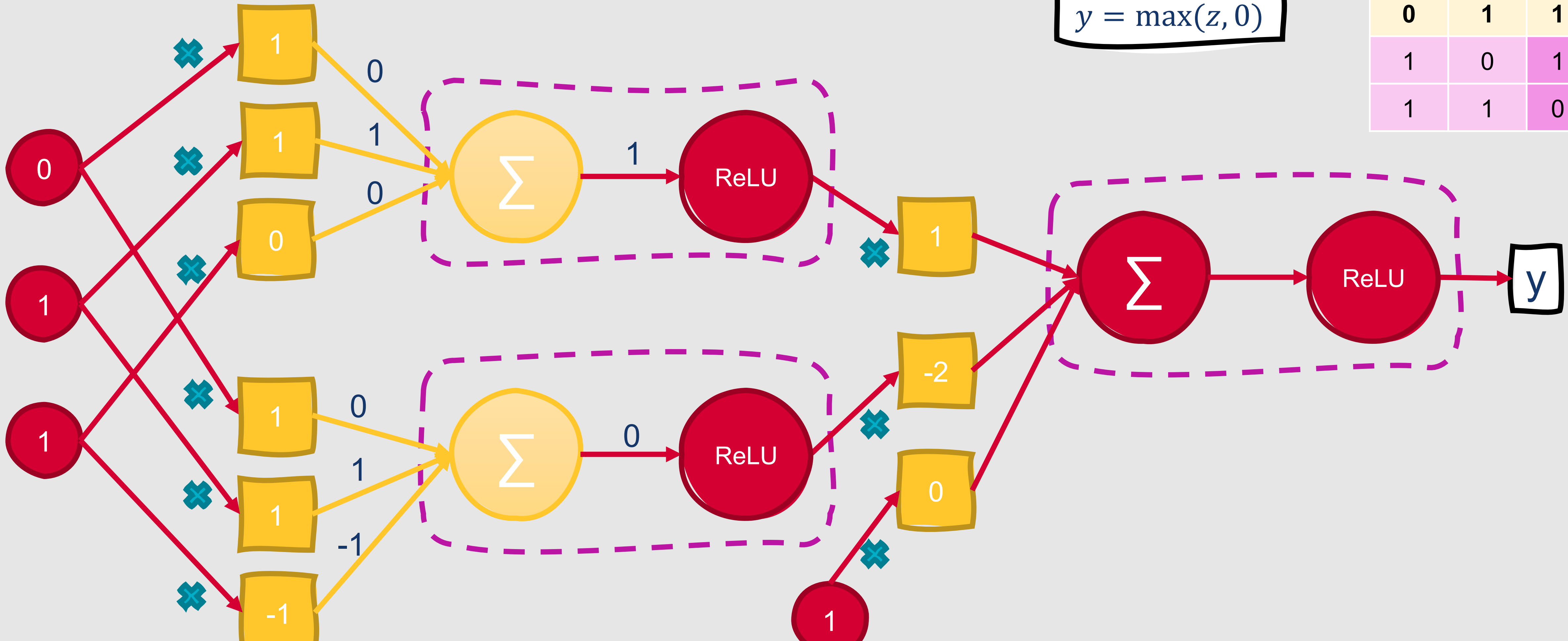
Combining Computational Units



$y = \max(z, 0)$

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

Combining Computational Units



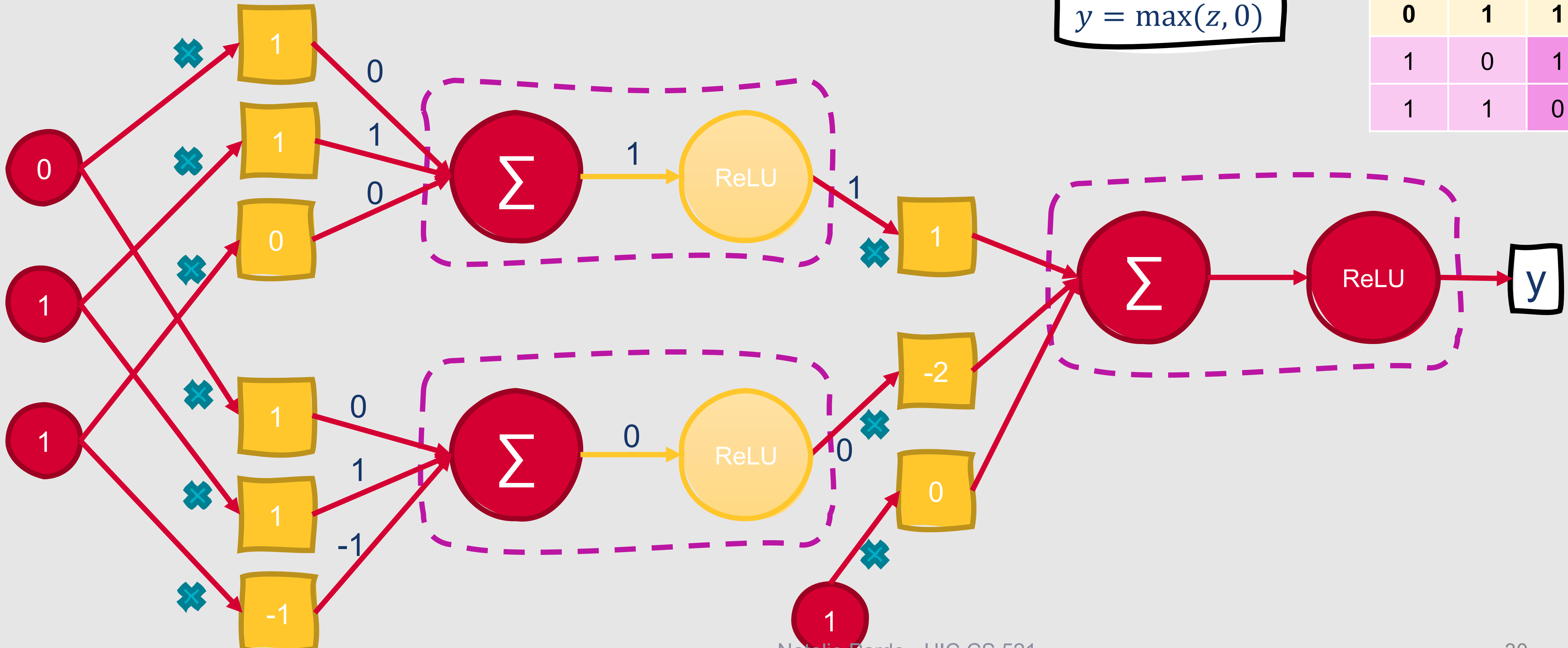
$y = \max(z, 0)$

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

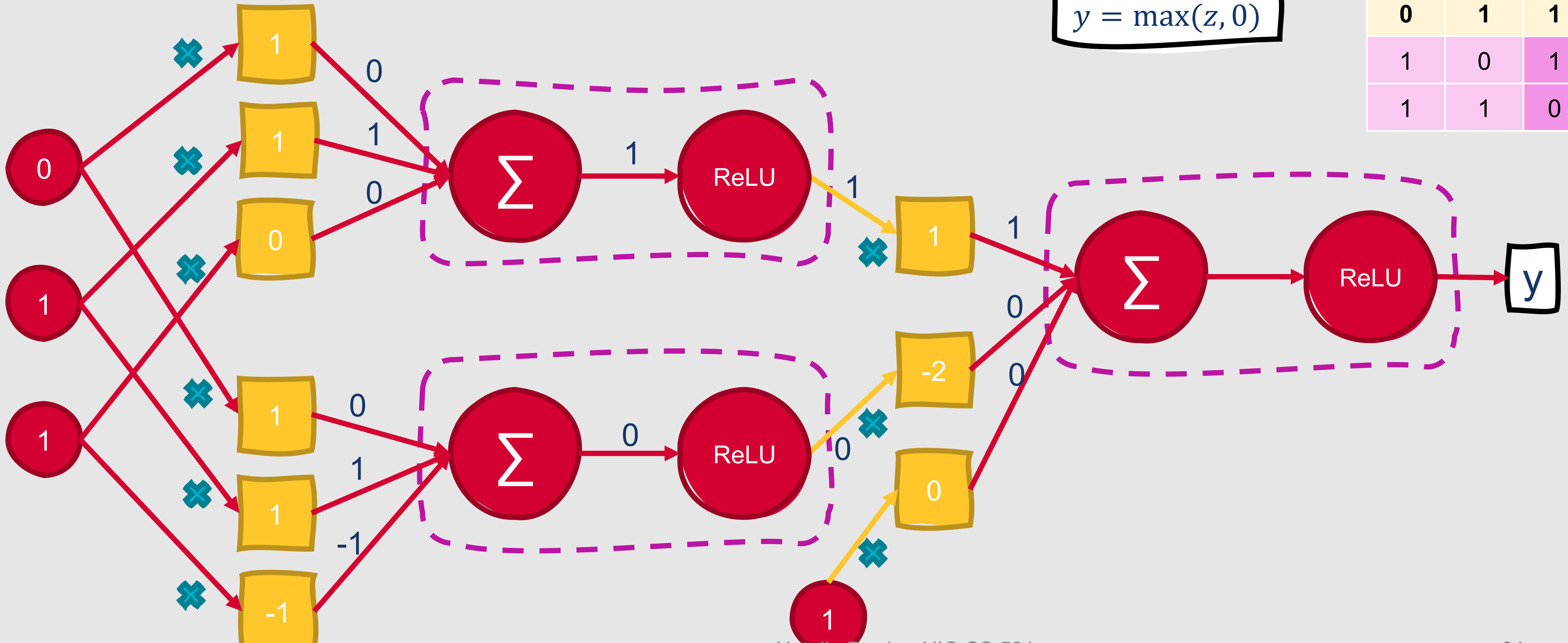
Combining Computational Units

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

$y = \max(z, 0)$



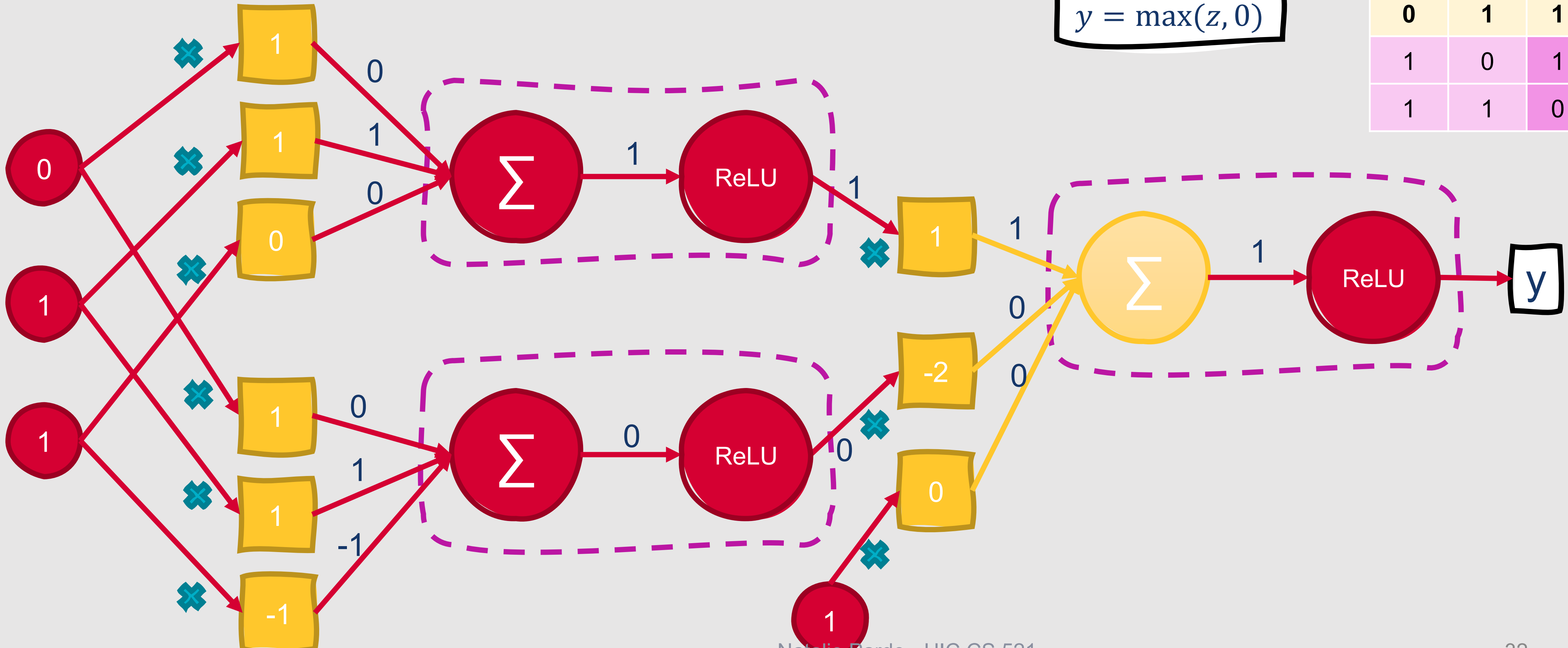
Combining Computational Units



$y = \max(z, 0)$

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

Combining Computational Units



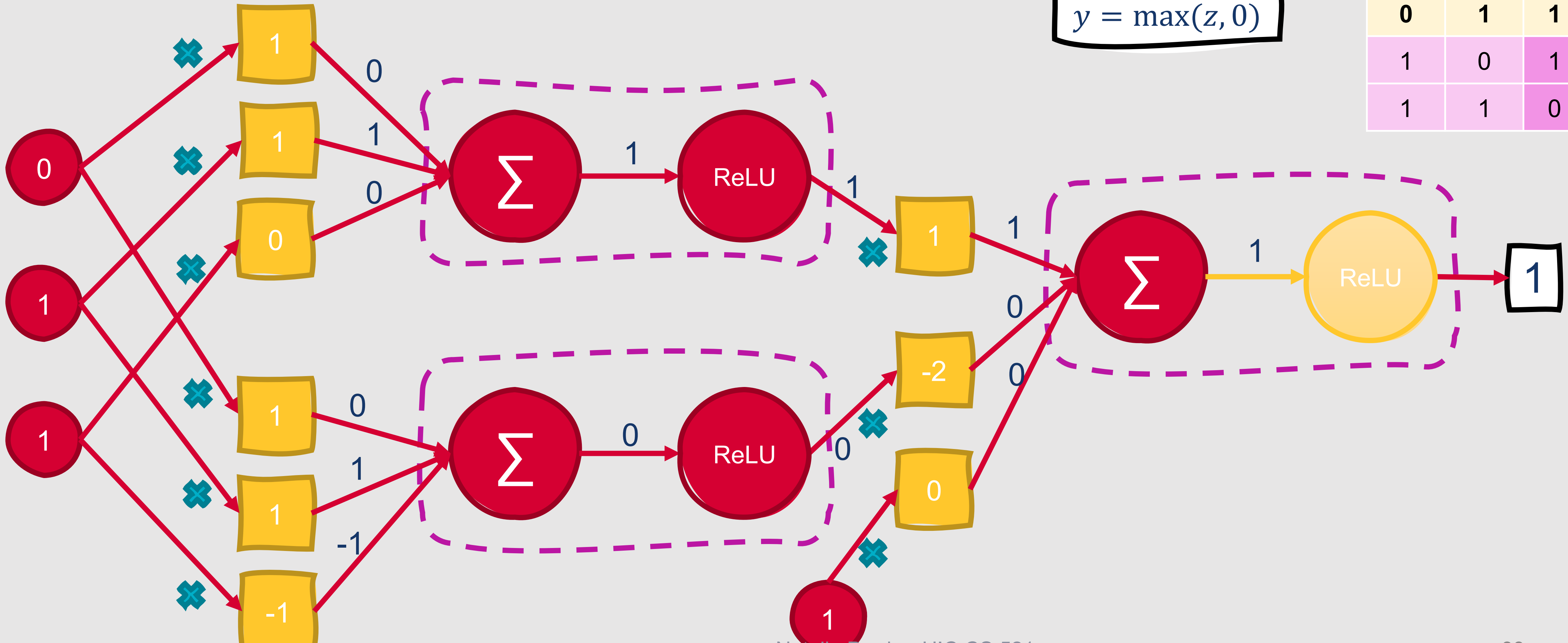
$y = \max(z, 0)$

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

Combining Computational Units

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

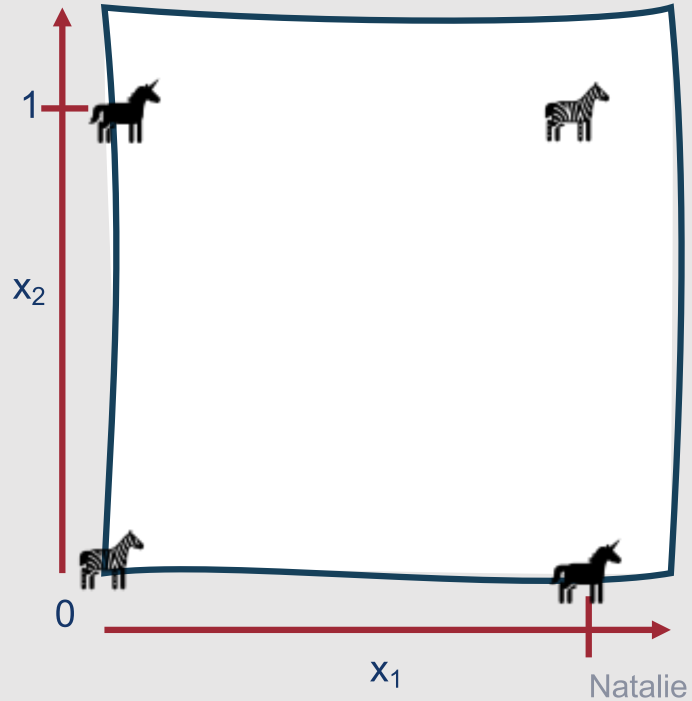
$y = \max(z, 0)$



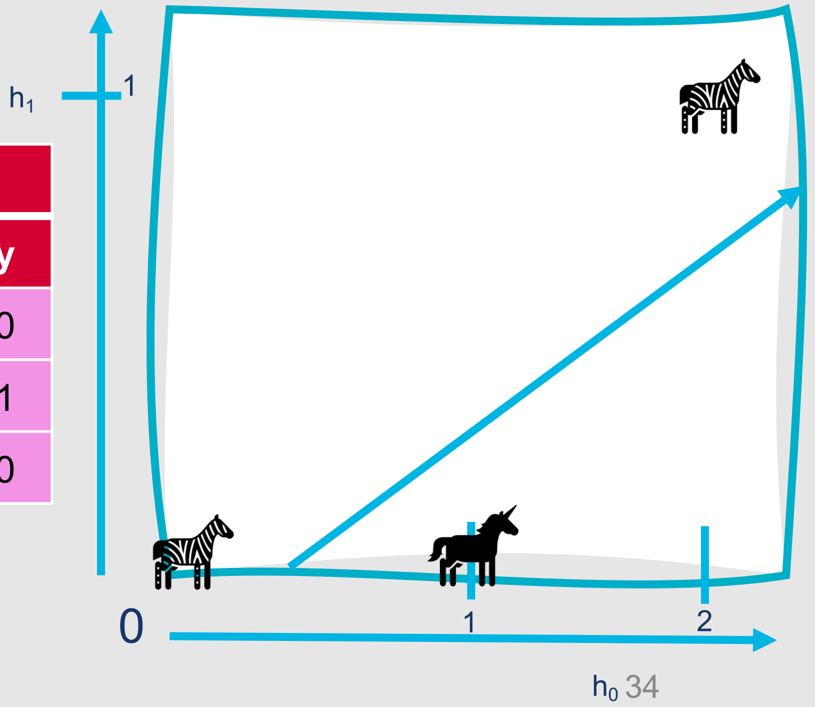
Why does this work?

- When computational units are combined, the outputs from each successive layer provide **new representations** for the input that can better separate the data into the target classes

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



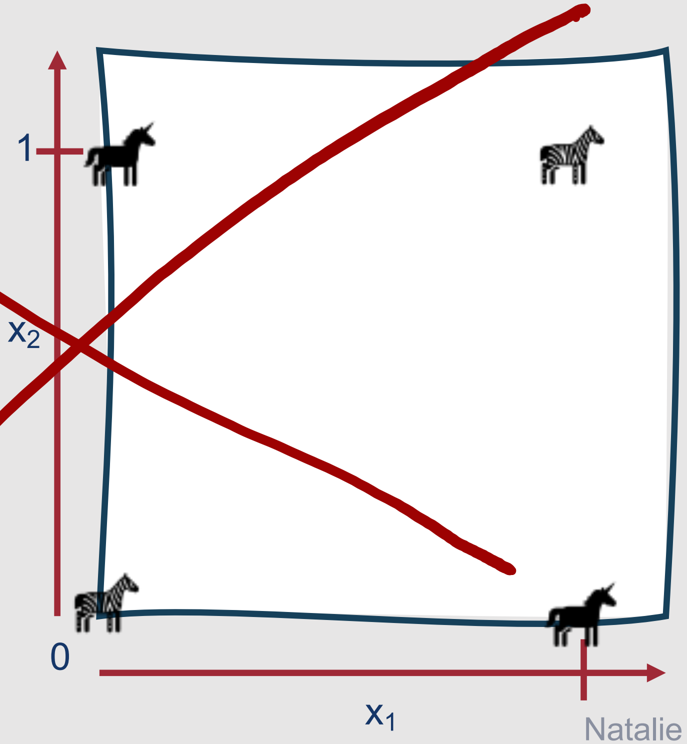
XOR		
h0	**h1**	**y**
0	0	0
1	0	1
2	1	0



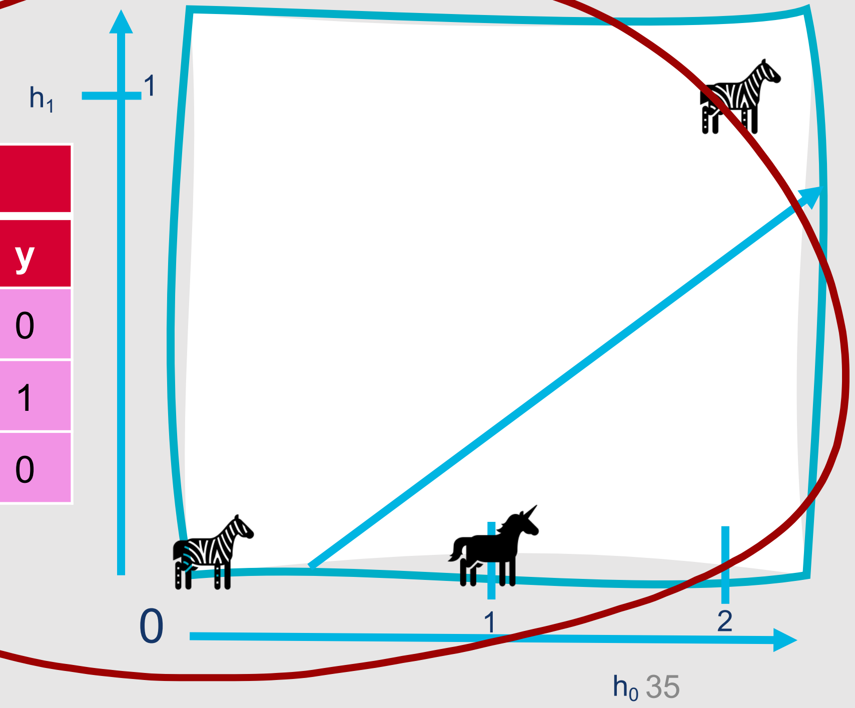
Why does this work?

- When computational units are combined, the outputs from each successive layer provide **new representations** for the input that can better separate the data into the target classes

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



Formalizing Feedforward Neural Networks

- Let $W^{[n]}$ be the weight matrix for layer n, $\mathbf{b}^{[n]}$ be the bias vector for layer n, and so forth
- Let $g(\cdot)$ be an activation function, such as:
    - ReLU
    - tanh
    - softmax
- 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 $\mathbf{a}^{[0]}$

Formalizing Feedforward Neural Networks

- We can represent a two-layer network as:
  - $z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$
  - $a^{[1]} = g^{[1]}\left(z^{[1]}\right)$
  - $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
  - $a^{[2]} = g^{[2]}(z^{[2]})$
  - $y' = a^{[2]}$
- 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
- Final layers will usually use softmax (for multinomial classification) or sigmoid (for binary classification) activations

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

Cross-entropy loss

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

How do we train neural networks?

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

???

Recall….

- When we train a logistic regression classifier, we can compute the gradient of our loss function by just taking its derivative:
  - $\frac{\partial L_{CE}(w,b)}{\partial w_j} = (\hat{y} - y)x_j = \underbrace{(\sigma(w \cdot x + b) - y)}_{\text{Difference between true and estimated } y} x_j$

Difference between true and estimated *y*

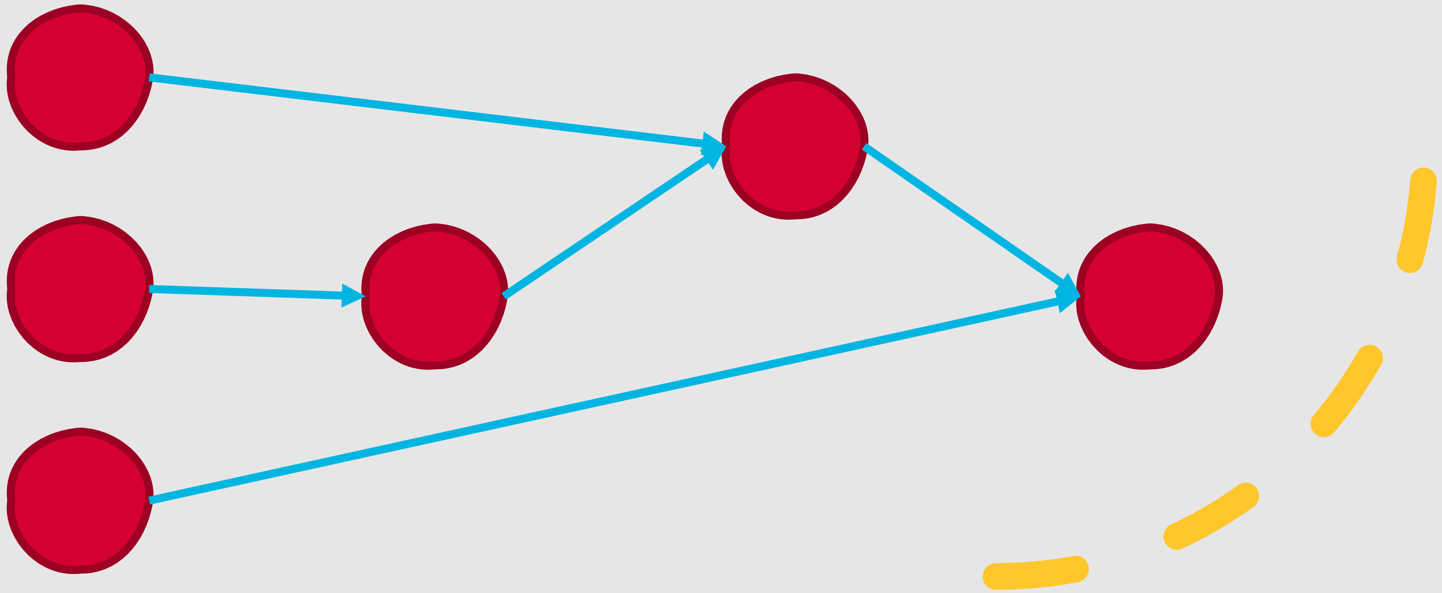
Corresponding input observation

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

- Why?
  - Taking the derivative of the loss function only provides the gradient for the final 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** in the context of neural networks
- Frames neural networks as **computation graphs**
  - Representations of interconnected mathematical operations
  - **Nodes** = Operations
  - **Directed edges** = connections between output/input of nodes



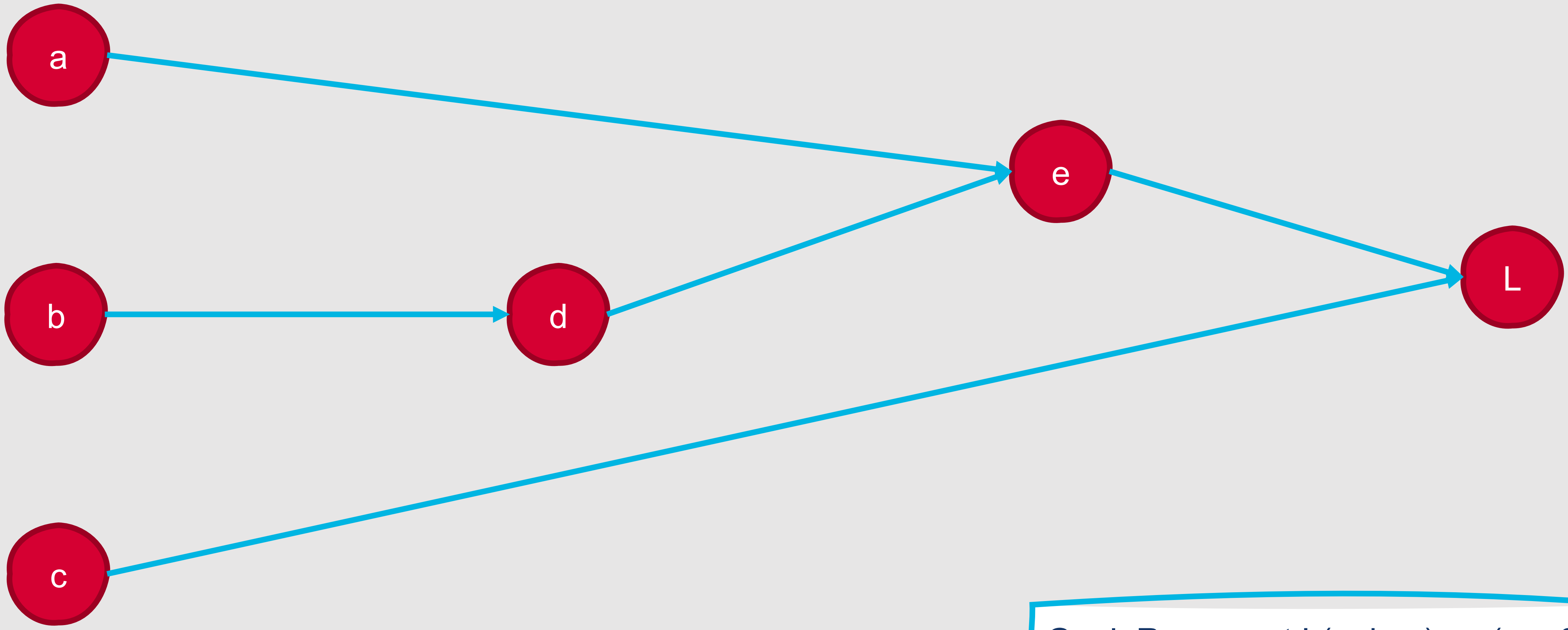
There are two different ways that we can pass information through our neural network computation graphs.

- **Forward pass**
  - Apply operations in the direction of the final layer
  - Pass the output of one computation as the input to the next
- **Backward pass**
  - Compute partial derivatives in the opposite direction of the final layer
  - Multiply them by the partial derivatives passed down from the previous step

Example: Forward Pass

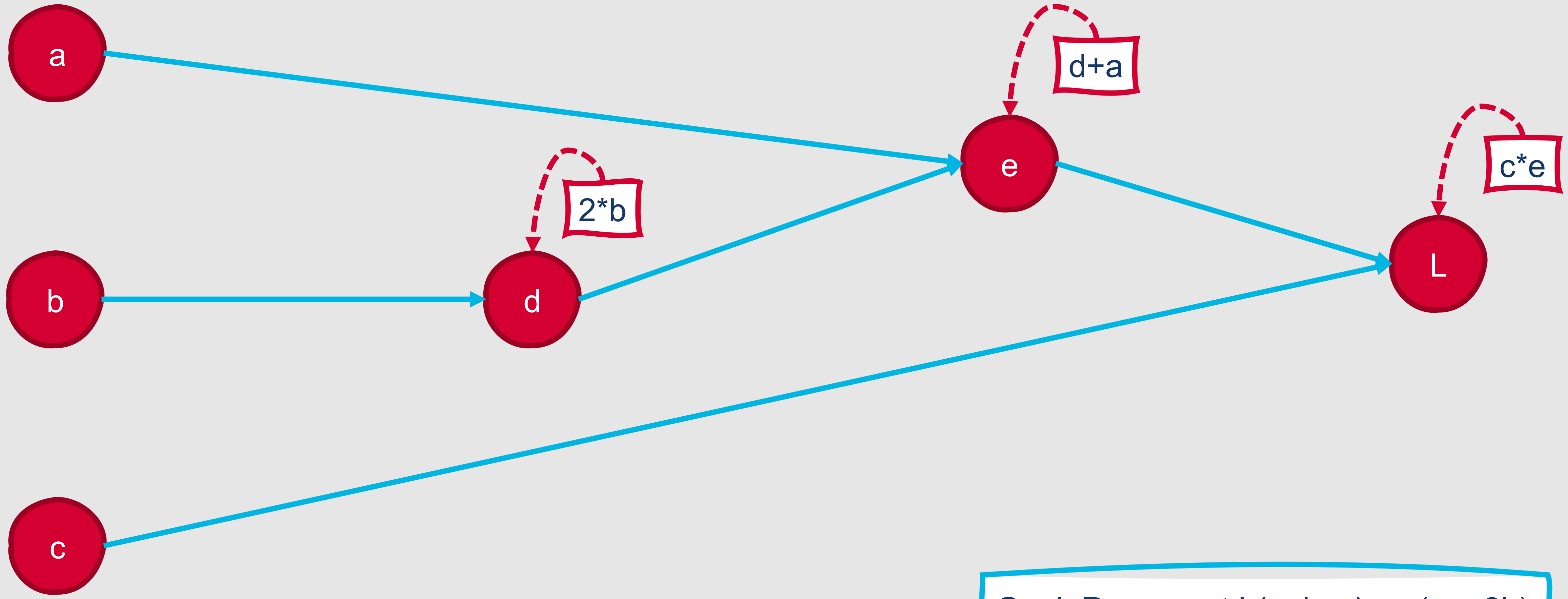
Goal: Represent $L(a, b, c) = c(a + 2b)$

Example: Forward Pass



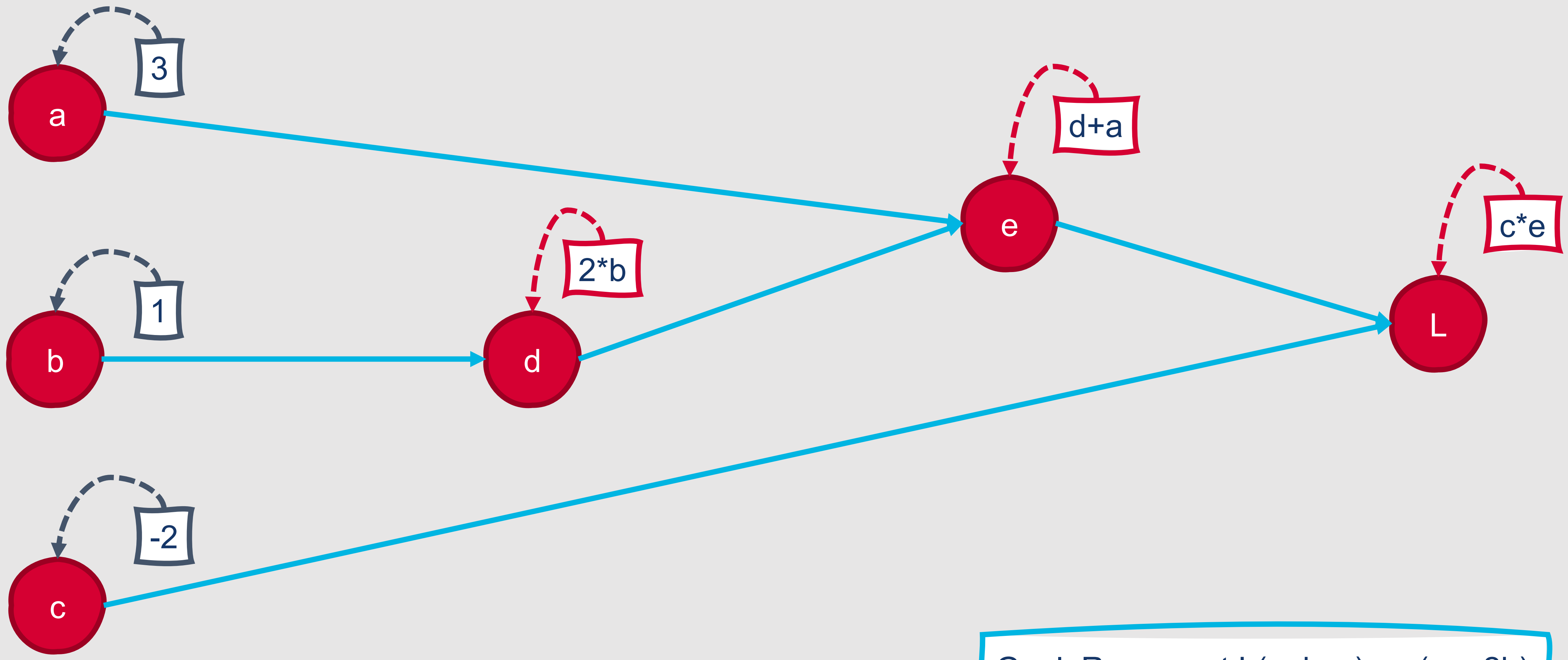
Goal: Represent L(a, b, c) = c(a + 2b)

Example: Forward Pass



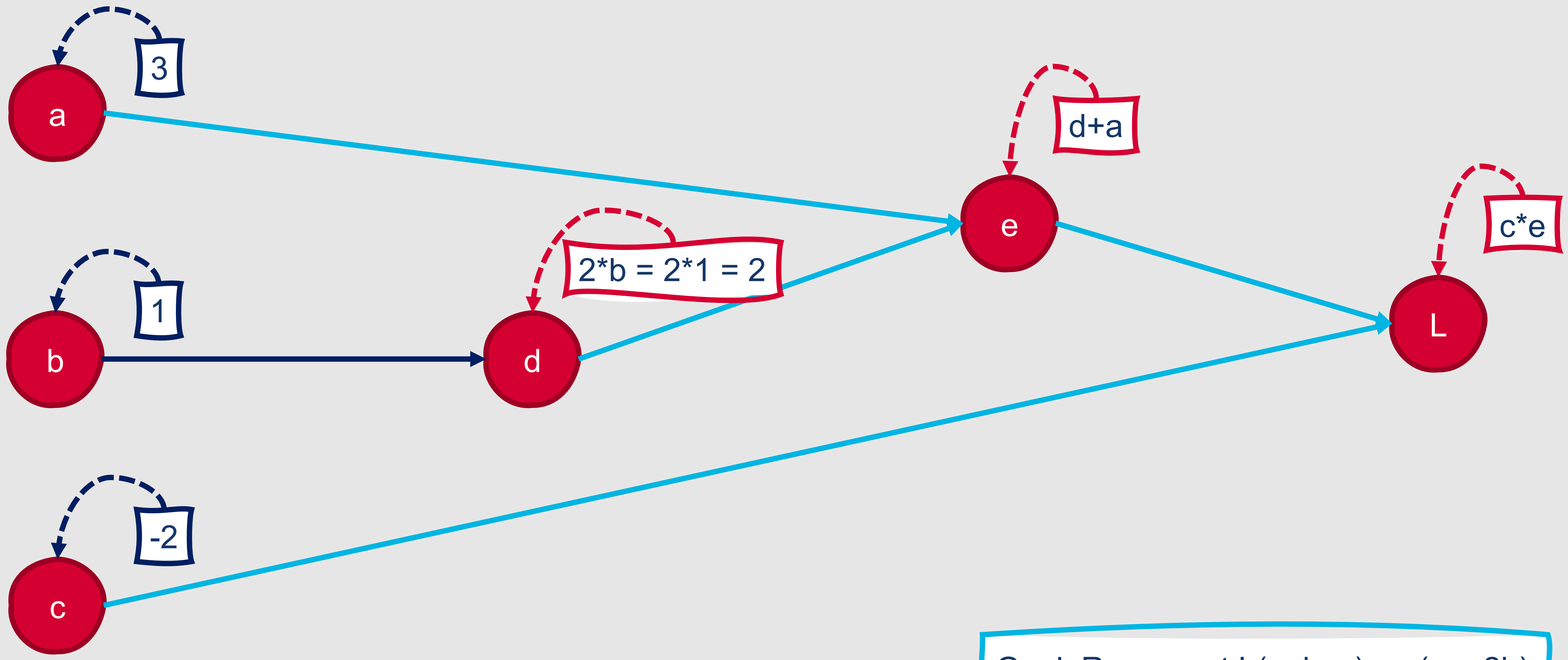
Goal: Represent L(a, b, c) = c(a + 2b)

Example: Forward Pass



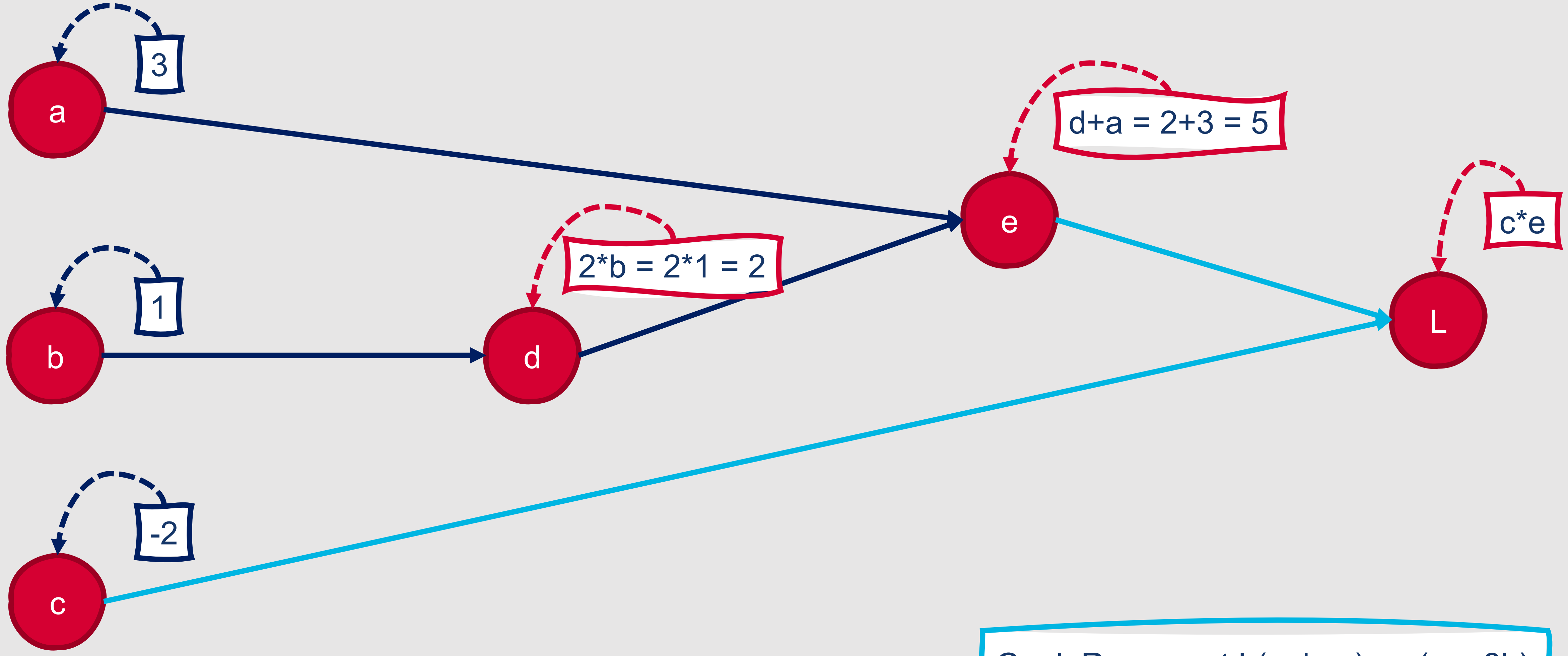
Goal: Represent L(a, b, c) = c(a + 2b)

Example: Forward Pass



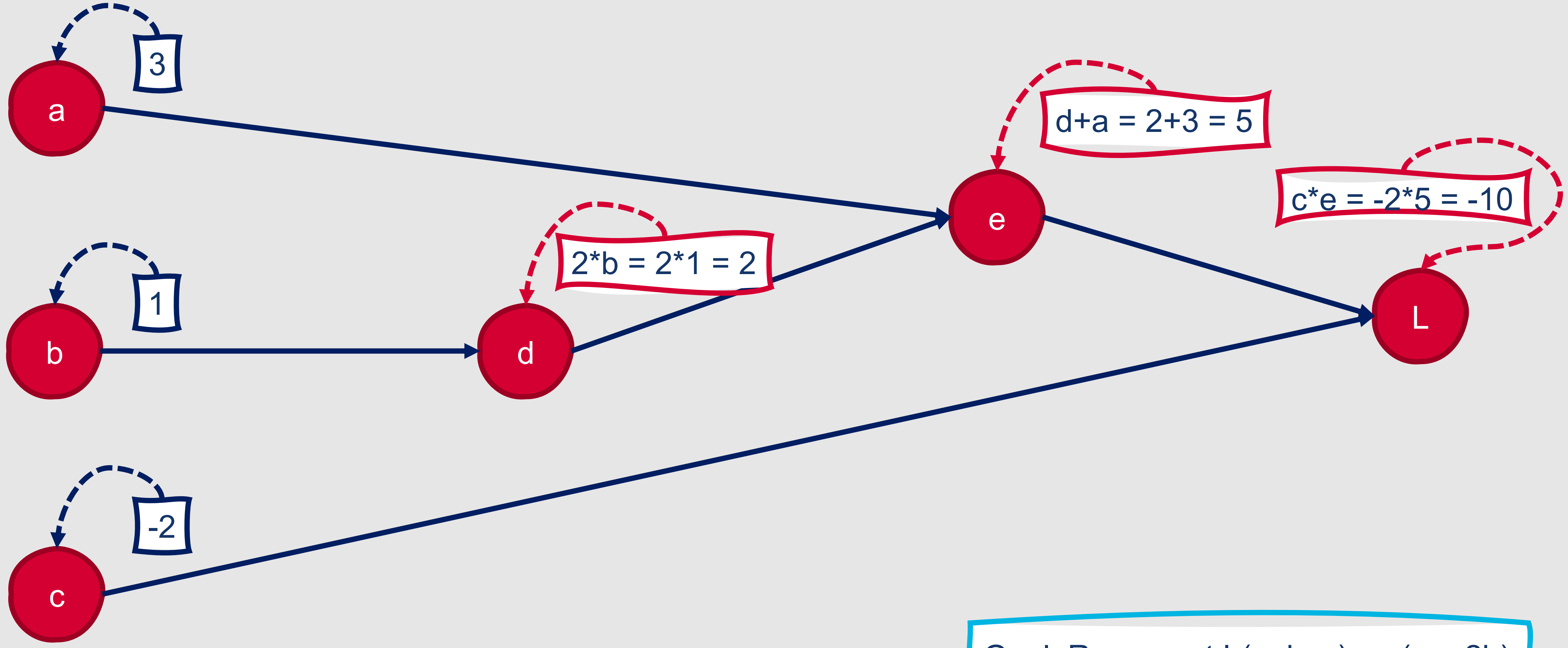
Goal: Represent L(a, b, c) = c(a + 2b)

Example: Forward Pass



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

Example: Forward Pass



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

To perform a 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
    - $\frac{df}{dx} = \frac{du}{dv} * \frac{dv}{dx}$

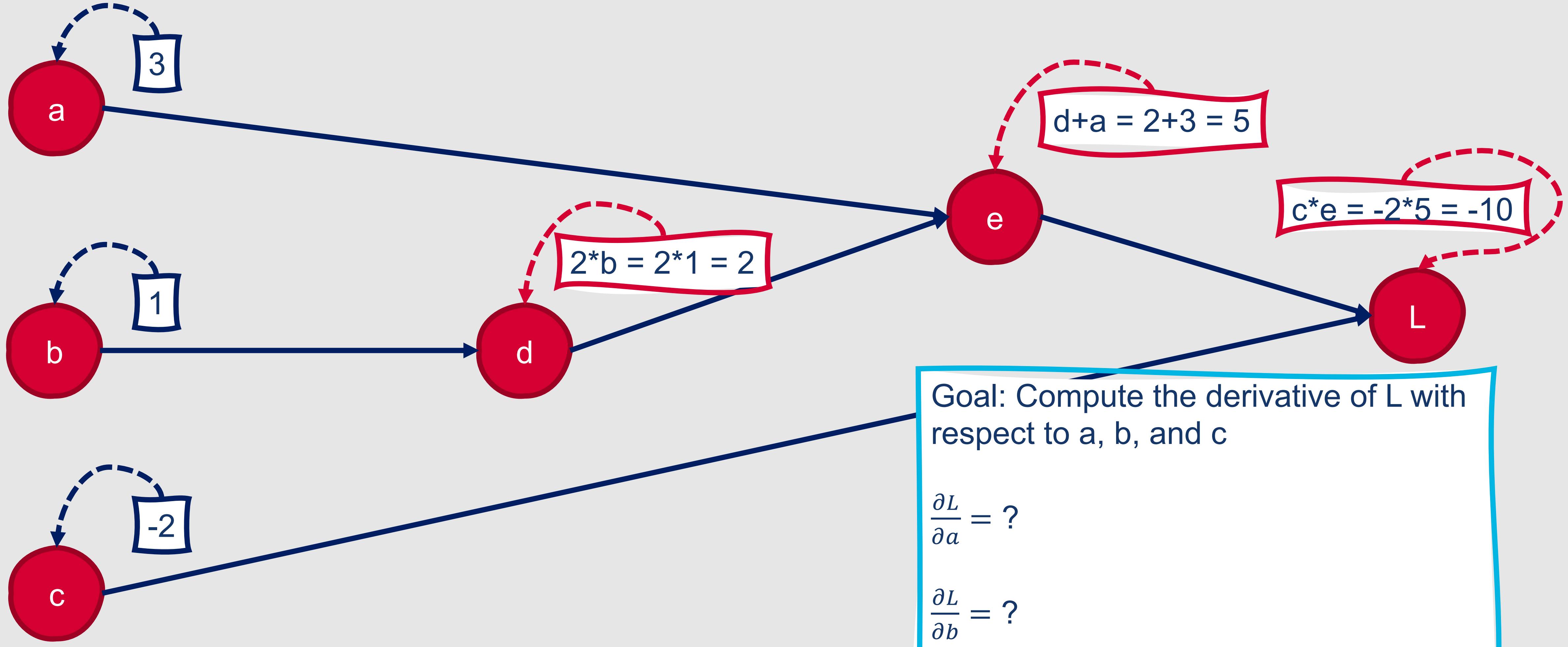
Derivatives of popular activation functions:

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

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

In theory, $\frac{\partial \text{ReLU}(0)}{\partial z}$ is undefined! In practice, by convention we set $\frac{\partial \text{ReLU}(0)}{\partial z} = 0$.

Example: Backward Pass



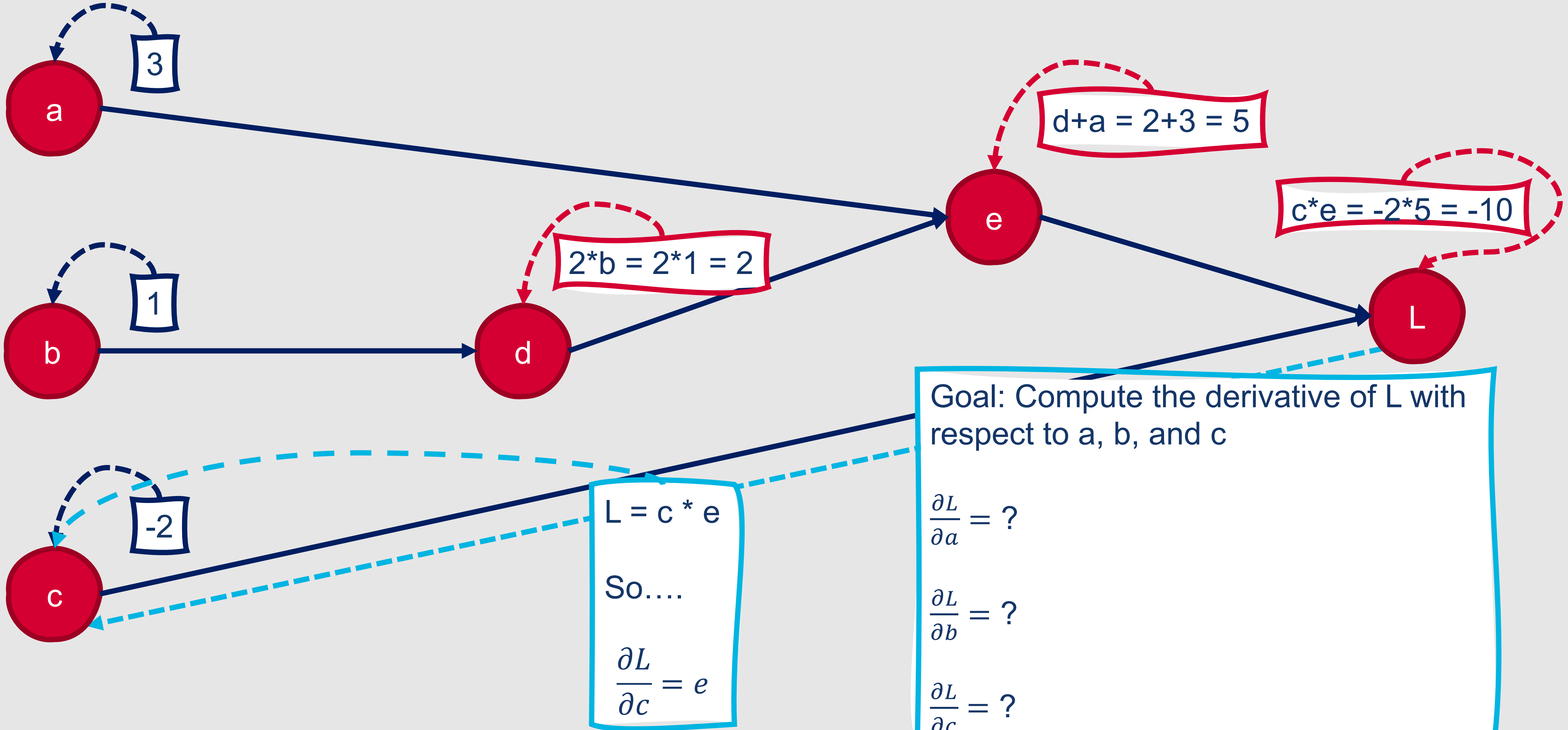
Goal: Compute the derivative of L with respect to a, b, and c

$\frac{\partial L}{\partial a} = ?$

$\frac{\partial L}{\partial b} = ?$

$\frac{\partial L}{\partial c} = ?$

Example: Backward Pass



L = c * e

So….

$$\frac{\partial L}{\partial c} = e$$

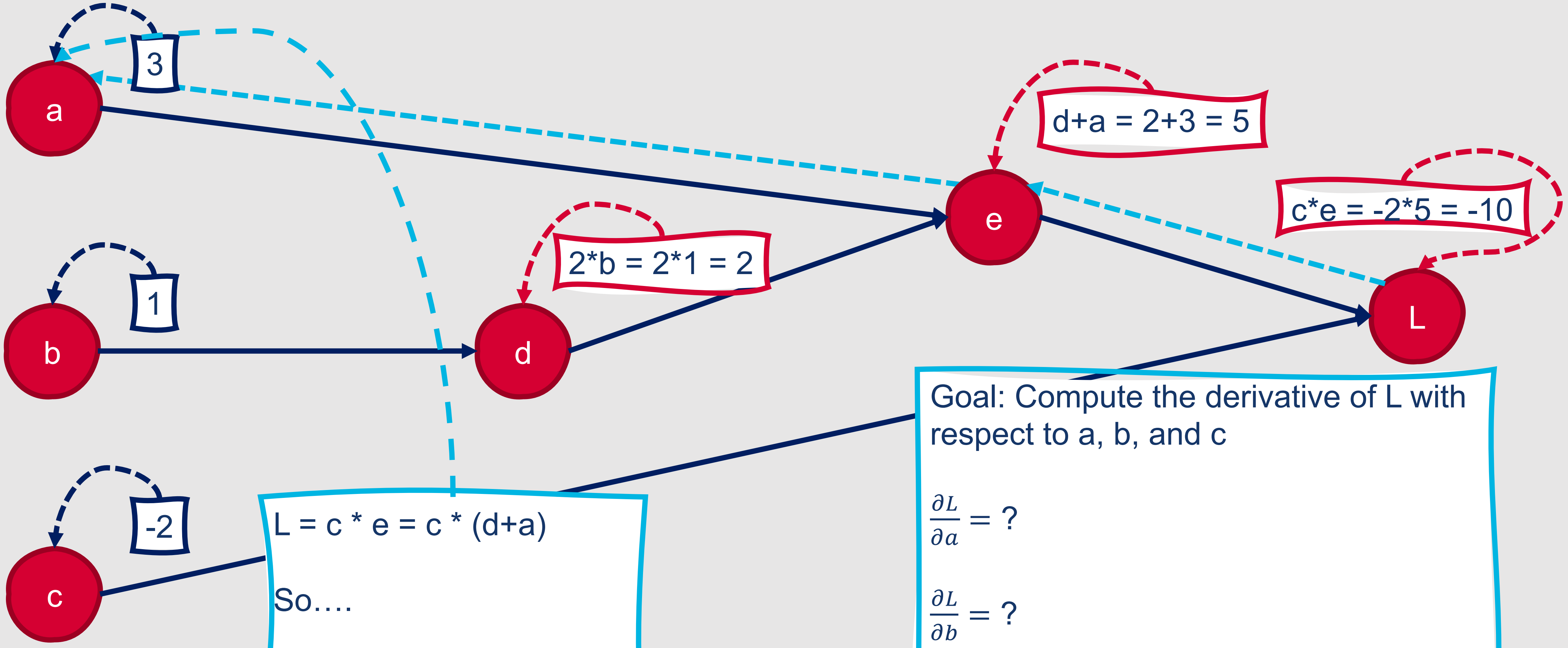
Goal: Compute the derivative of L with respect to a, b, and c

$$\frac{\partial L}{\partial a} = ?$$

$$\frac{\partial L}{\partial b} = ?$$

$$\frac{\partial L}{\partial c} = ?$$

Example: Backward Pass



L = c * e = c * (d+a)

So….

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} = c * 1 = c$$

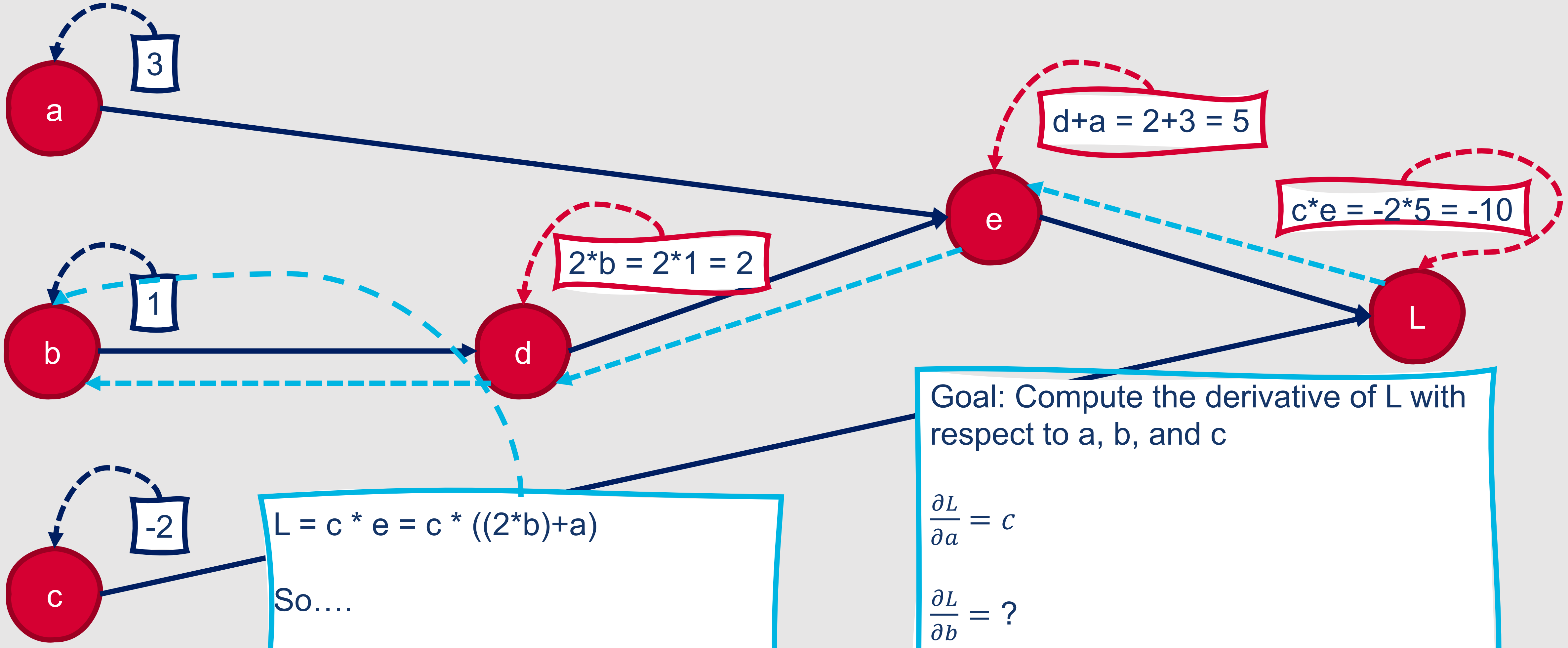
Goal: Compute the derivative of L with respect to a, b, and c

$$\frac{\partial L}{\partial a} = ?$$

$$\frac{\partial L}{\partial b} = ?$$

$$\frac{\partial L}{\partial c} = e$$

Example: Backward Pass



L = c * e = c * ((2*b)+a)

So….

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = c * 1 * 2 = 2 * c$$

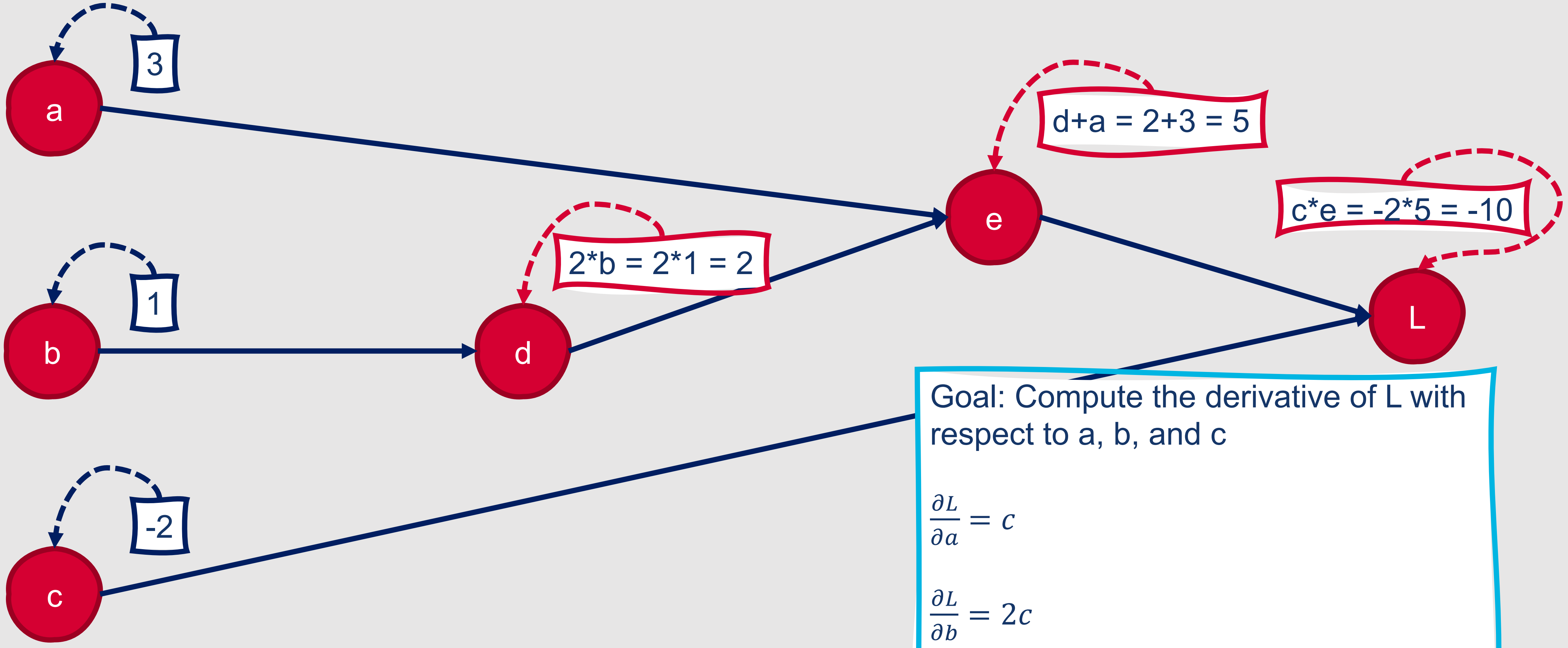
Goal: Compute the derivative of L with respect to a, b, and c

$$\frac{\partial L}{\partial a} = c$$

$$\frac{\partial L}{\partial b} = ?$$

$$\frac{\partial L}{\partial c} = e$$

Example: Backward Pass



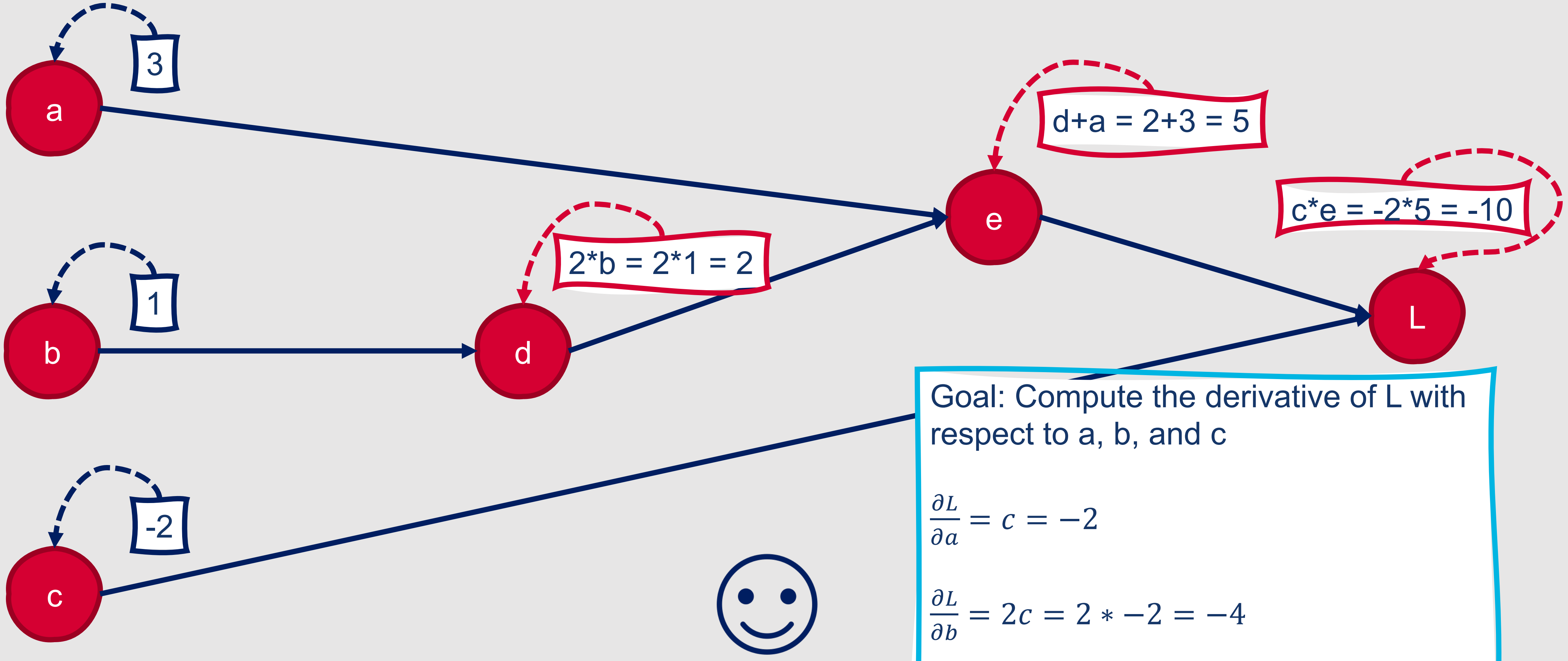
Goal: Compute the derivative of L with respect to a, b, and c

$$\frac{\partial L}{\partial a} = c$$

$$\frac{\partial L}{\partial b} = 2c$$

$$\frac{\partial L}{\partial c} = e$$

Example: Backward Pass



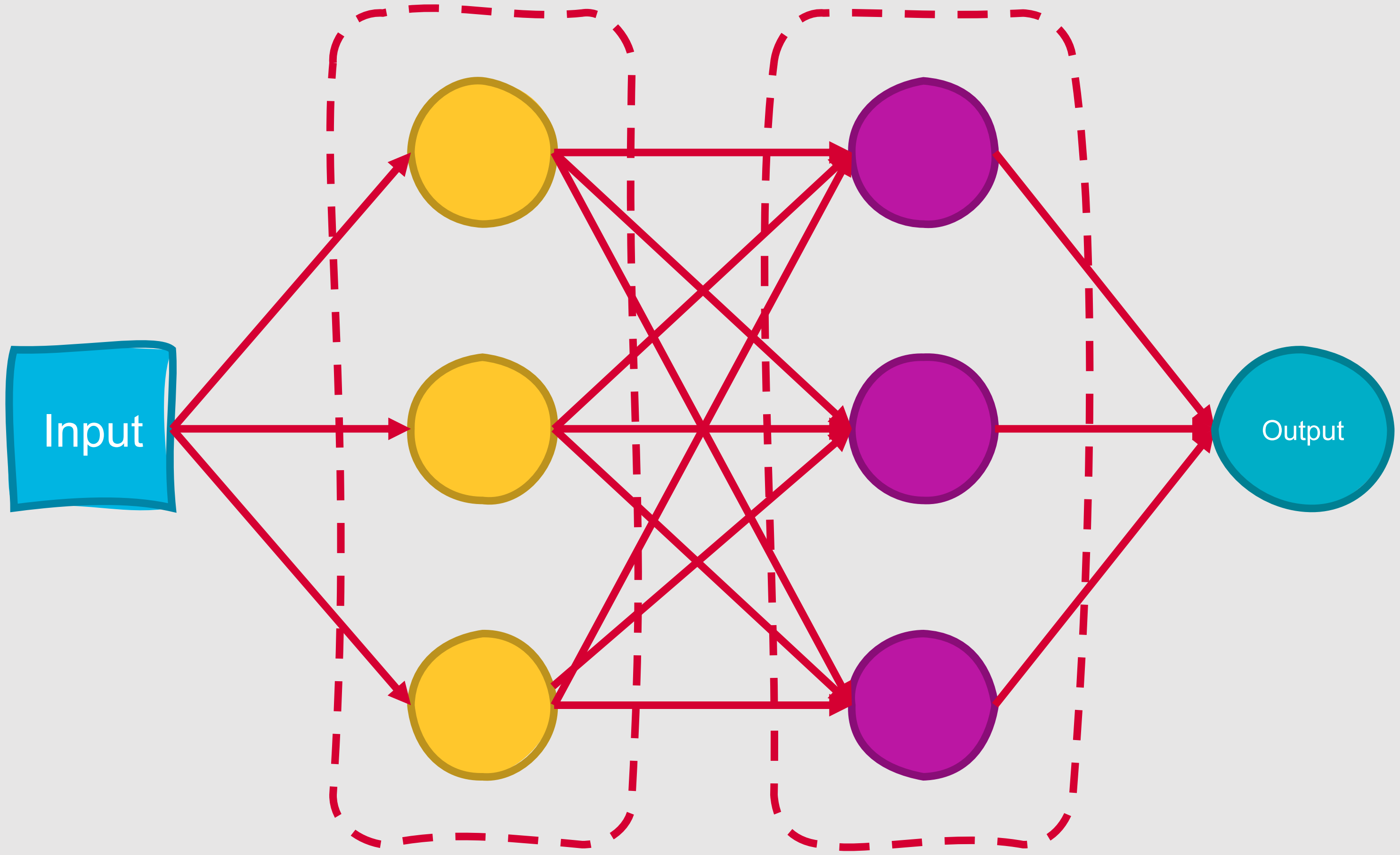
Goal: Compute the derivative of L with respect to a, b, and c

$$\frac{\partial L}{\partial a} = c = -2$$

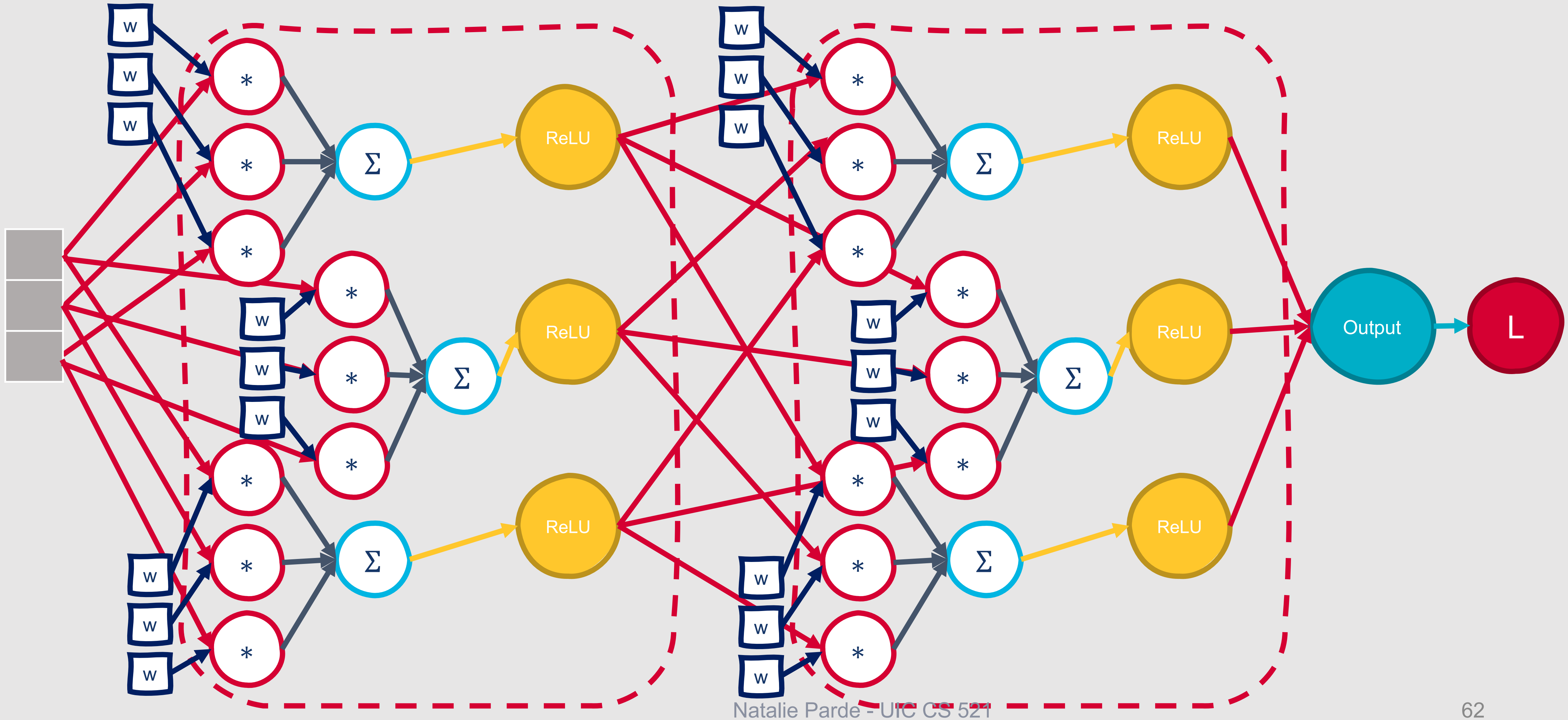
$$\frac{\partial L}{\partial b} = 2c = 2 * -2 = -4$$

$$\frac{\partial L}{\partial c} = e = 5$$

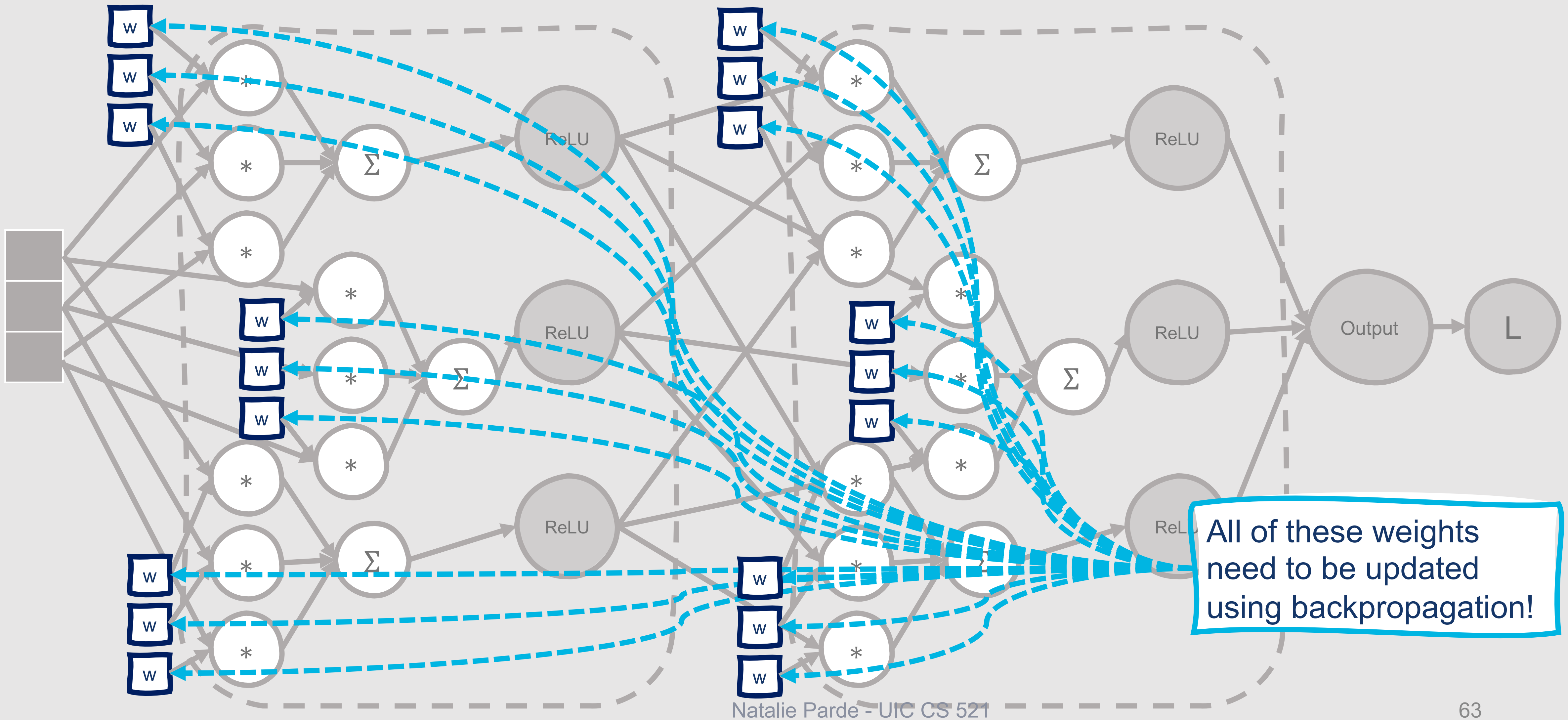
Computation graphs for neural networks involve numerous interconnected units.



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



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



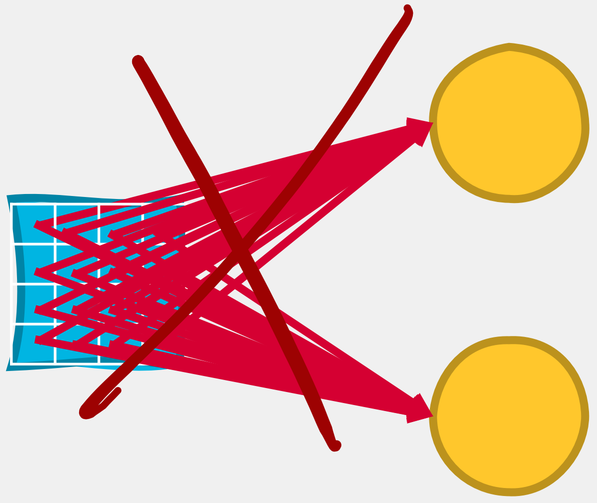
All of these weights need to be updated using backpropagation!

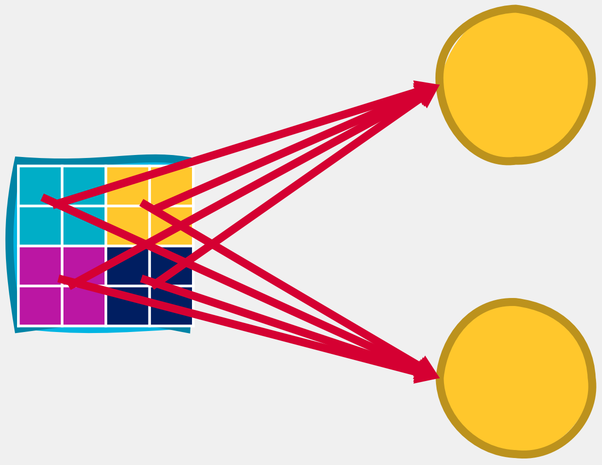
Convolutional Neural Networks

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

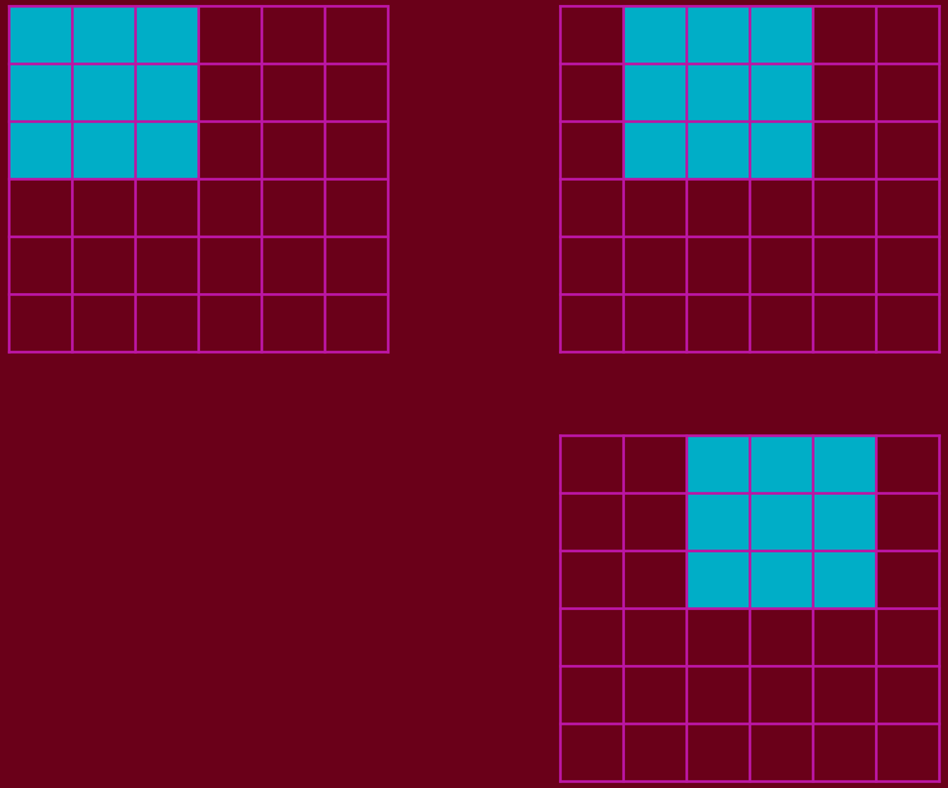
What are convolutional layers?

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



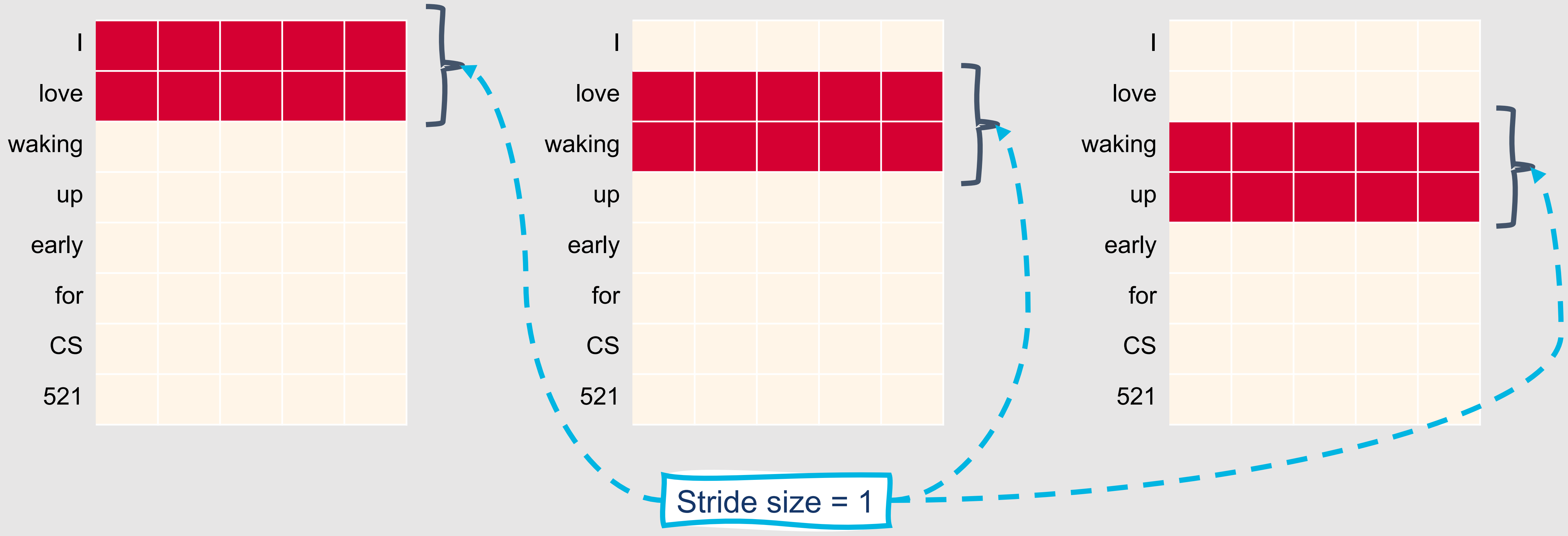


- First layer(s): low-level features
  - Color, gradient orientation
  - N-grams
- Higher layer(s): high-level features
  - Objects
  - Phrases

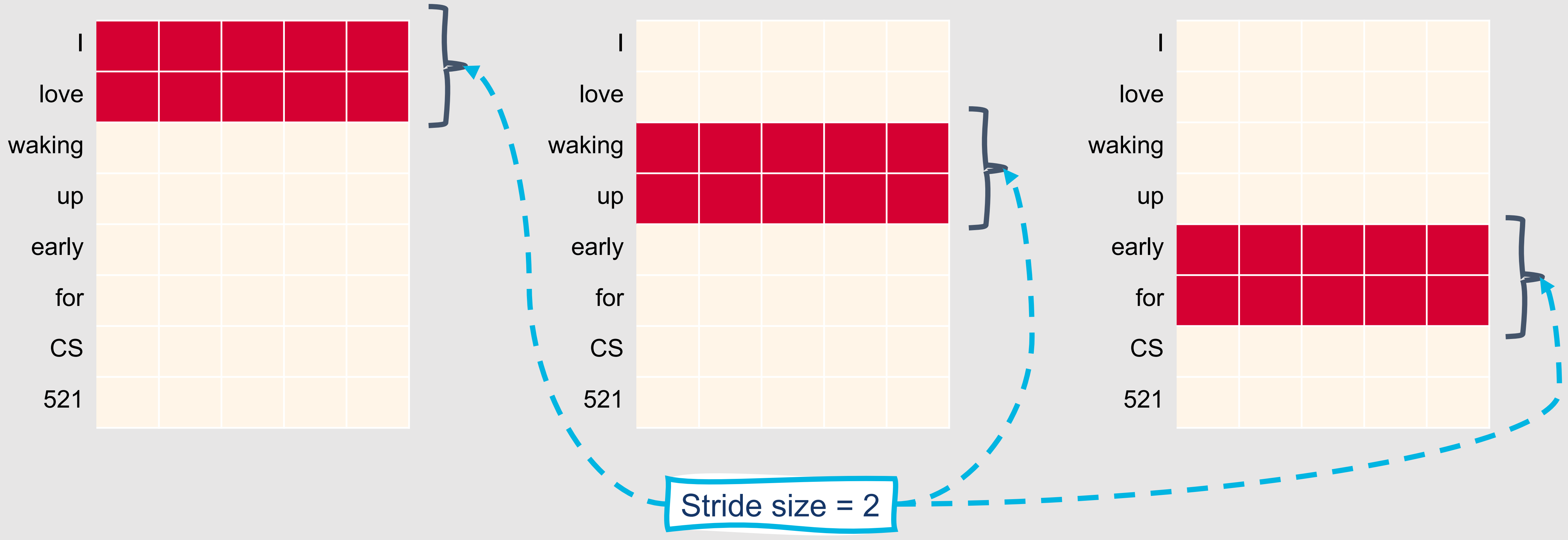


Convolutional Layers

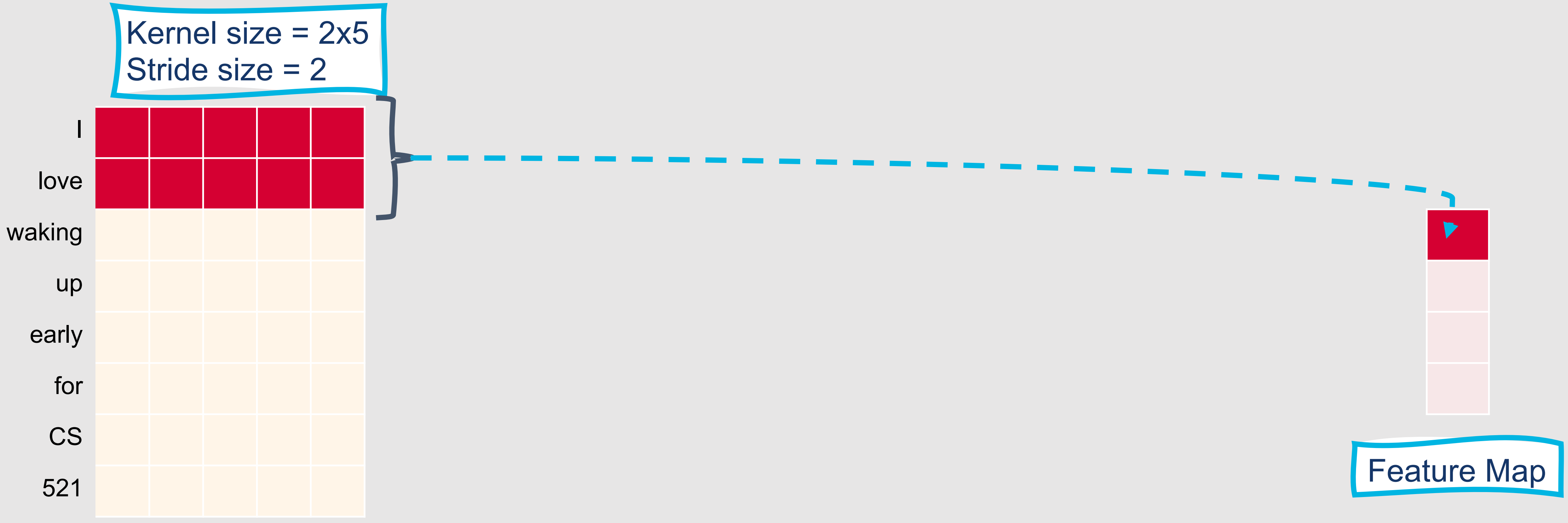
In NLP, convolutions are typically performed on entire rows of an input matrix, where each row corresponds to a word.



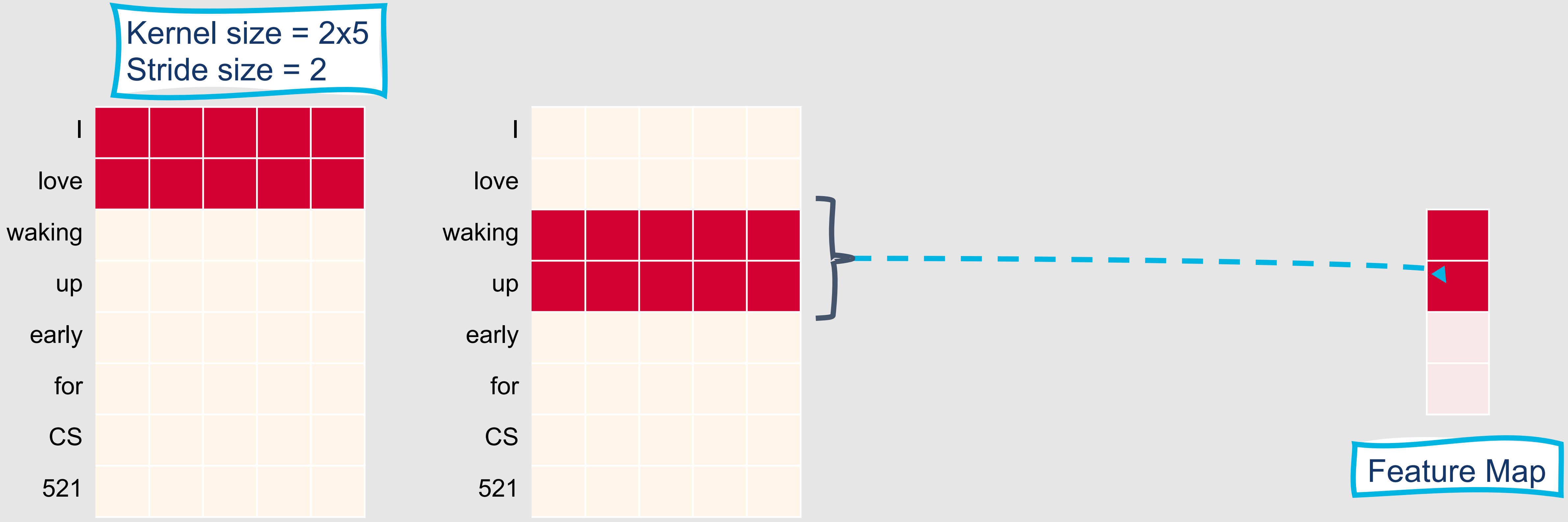
In NLP, convolutions are typically performed on entire rows of an input matrix, where each row corresponds to a word.



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

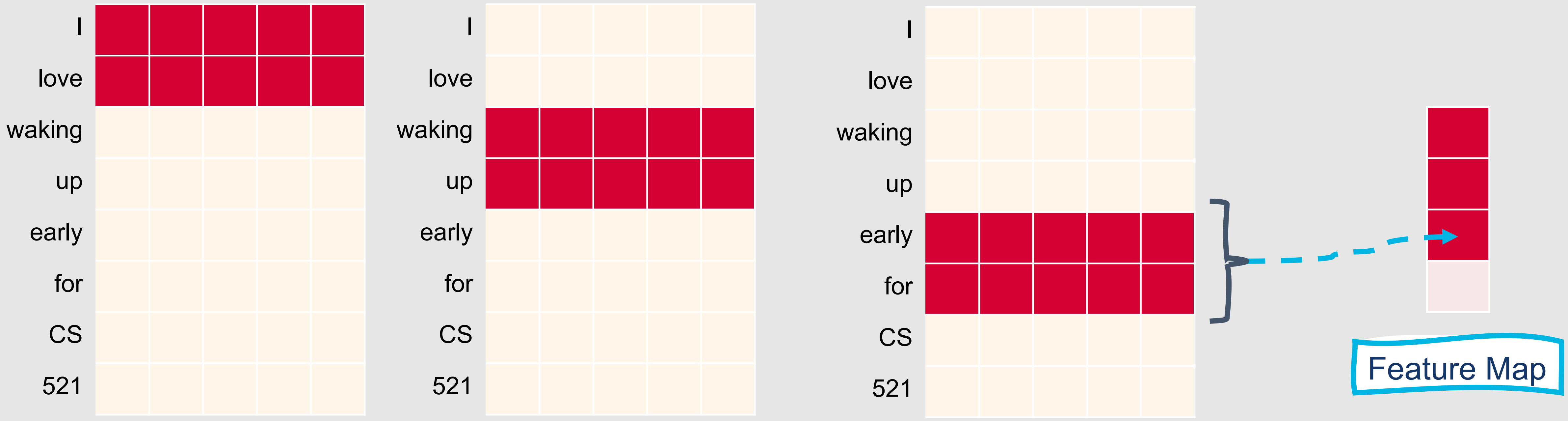


After applying a convolution with specific region (kernel) and stride sizes to an input matrix, we end up with a feature map.



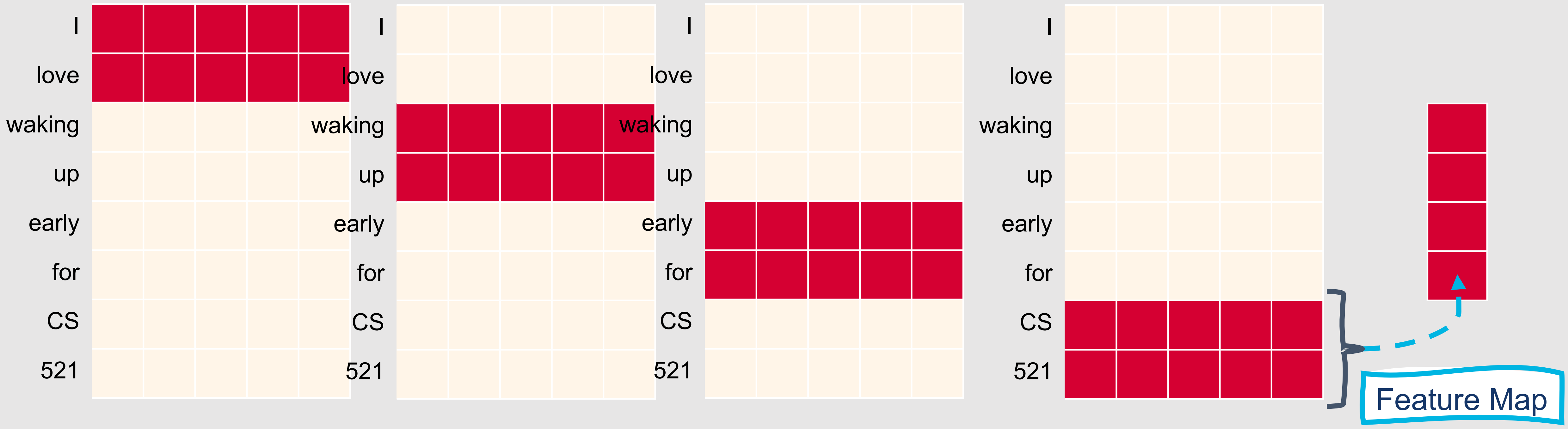
After applying a convolution with specific region (kernel) and stride sizes to an input matrix, we end up with a feature map.

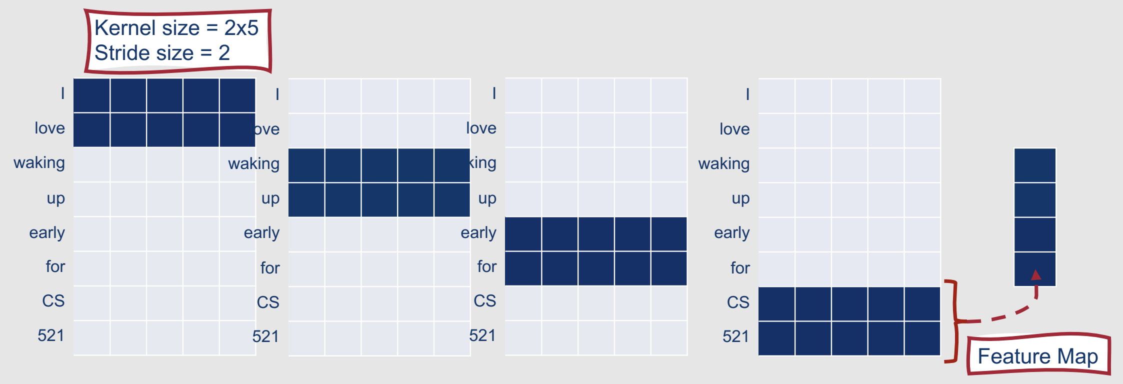
Kernel size = 2x5
Stride size = 2

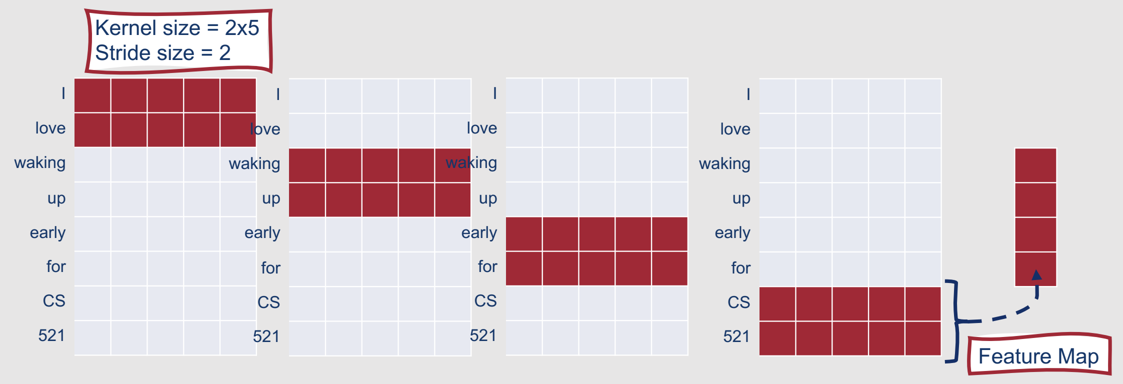


After applying a convolution with specific region (kernel) and stride sizes to an input matrix, we end up with a feature map.

Kernel size = 2x5
Stride size = 2







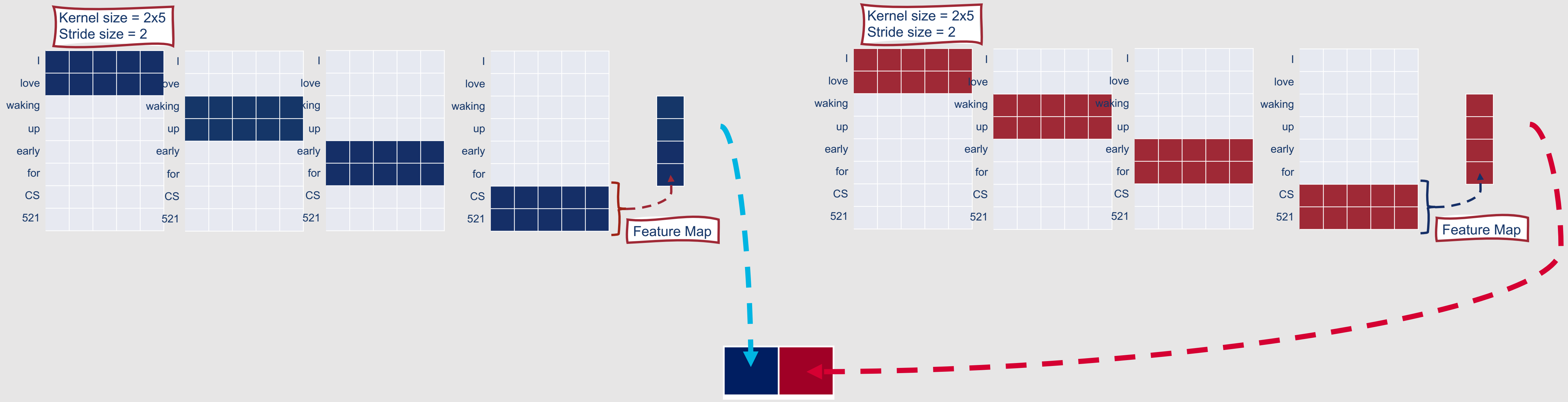
It’s common to extract multiple different feature maps from the same input.

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

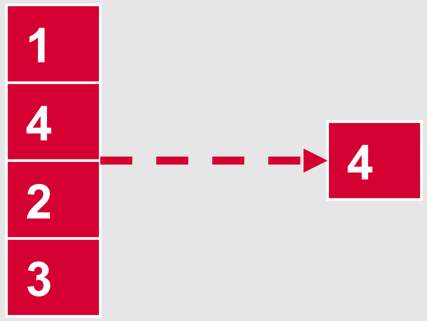
- Why use pooling layers?
  - Further increase **efficiency**
  - Improve the model's ability to be **invariant to small changes**

Pooling Layers



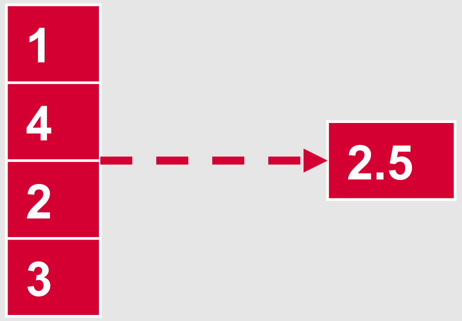
Common Techniques for Pooling

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

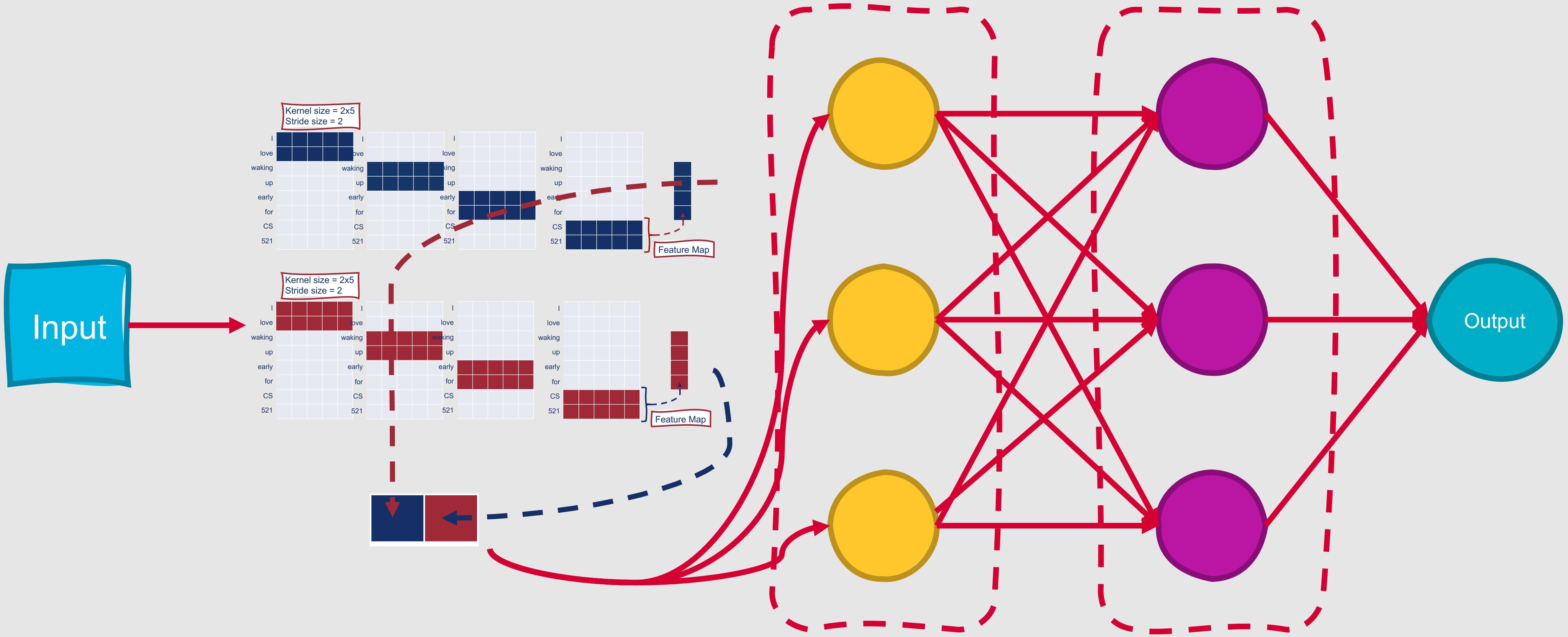


Common Techniques for Pooling

- **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 passed along as input to the rest of the network.

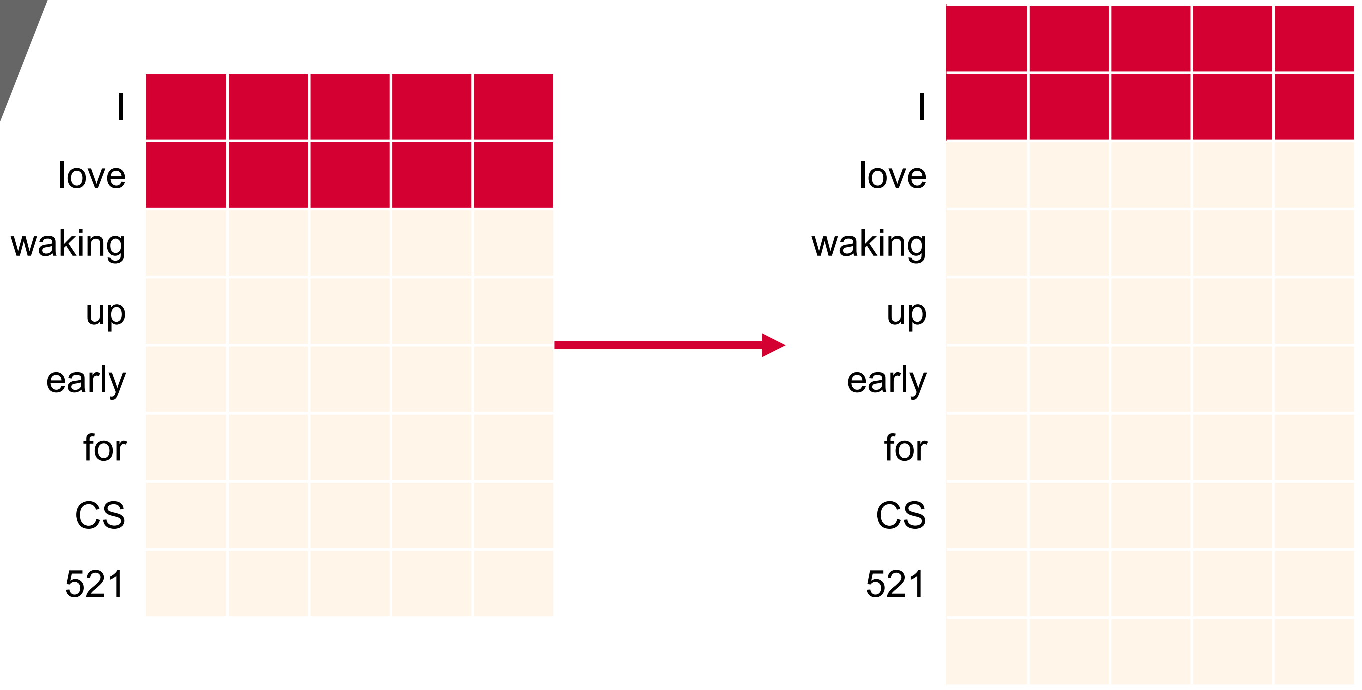


Convolutional neural network architectures can vary greatly!

- 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.

Why use CNNs for NLP tasks at all?

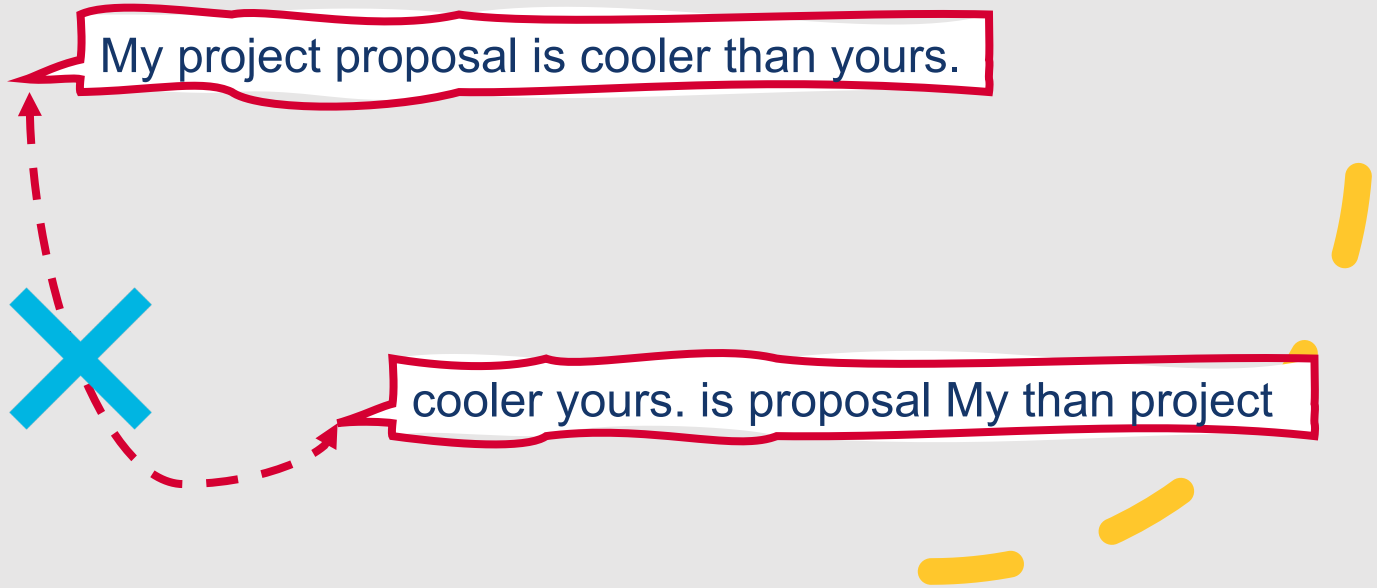
- Traditionally for image classification!
- However, offer unique advantages for NLP tasks:
  - CNNs inherently extract meaningful local structures from input
  - In NLP → implicitly-learned, useful n-grams!

Summary: Feedforward Neural Networks and CNNs

- **Neural networks** are built from interconnected layers of computing units
- To train the weights across the entire network, loss values are passed backward to earlier layers using **backpropagation**
- Model design and training procedures for neural networks can be optimized by tuning **hyperparameters**
- One more specialized neural network architecture is the **convolutional neural network**
- CNNs can be an efficient way to capture **local structure** in numerous types of input data

Language is inherently temporal.

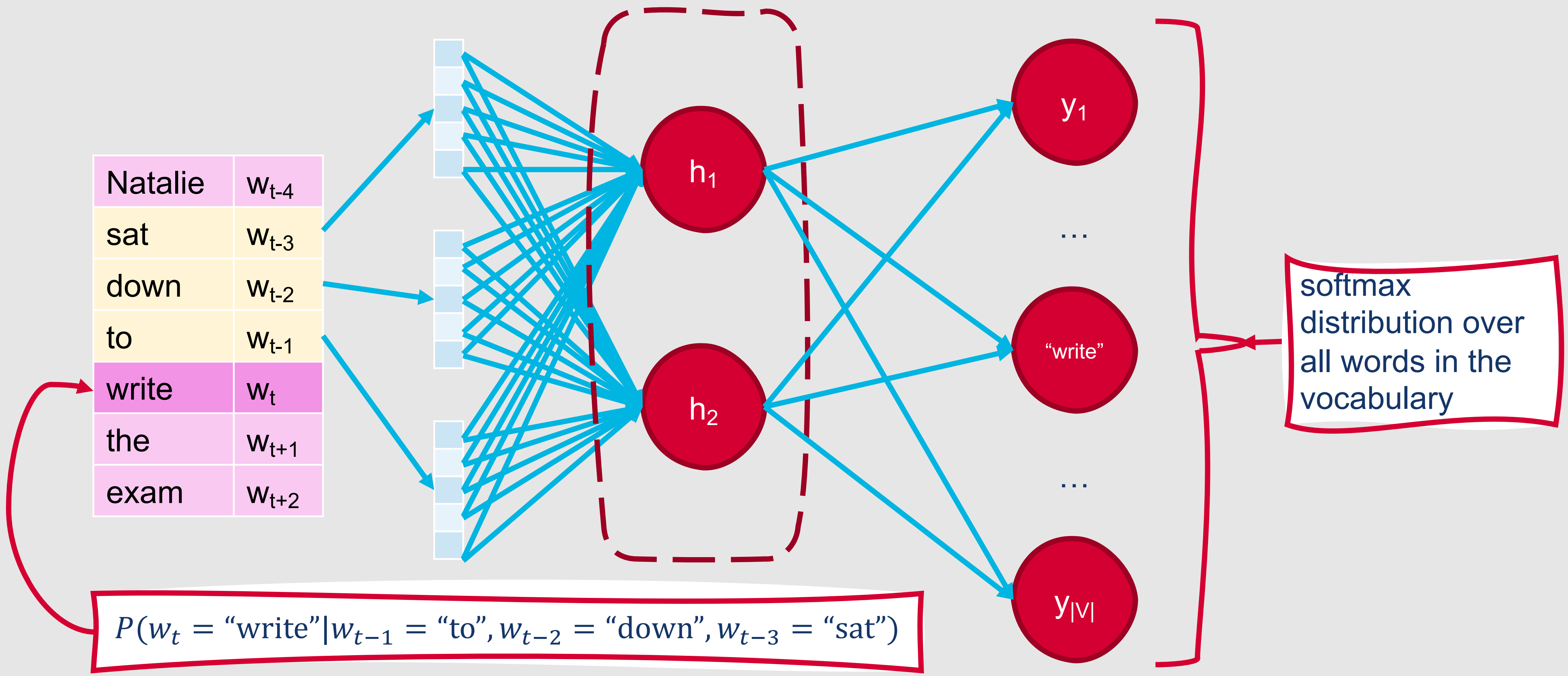
- **Continuous input streams** of **indefinite length** that unfold over **time**
- Even clear from the metaphors we use to describe language:
  - Conversation flow
  - News feed
  - Twitter stream


My project proposal is cooler than yours.
cooler yours. is proposal My than project

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

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

Sliding Window Approach

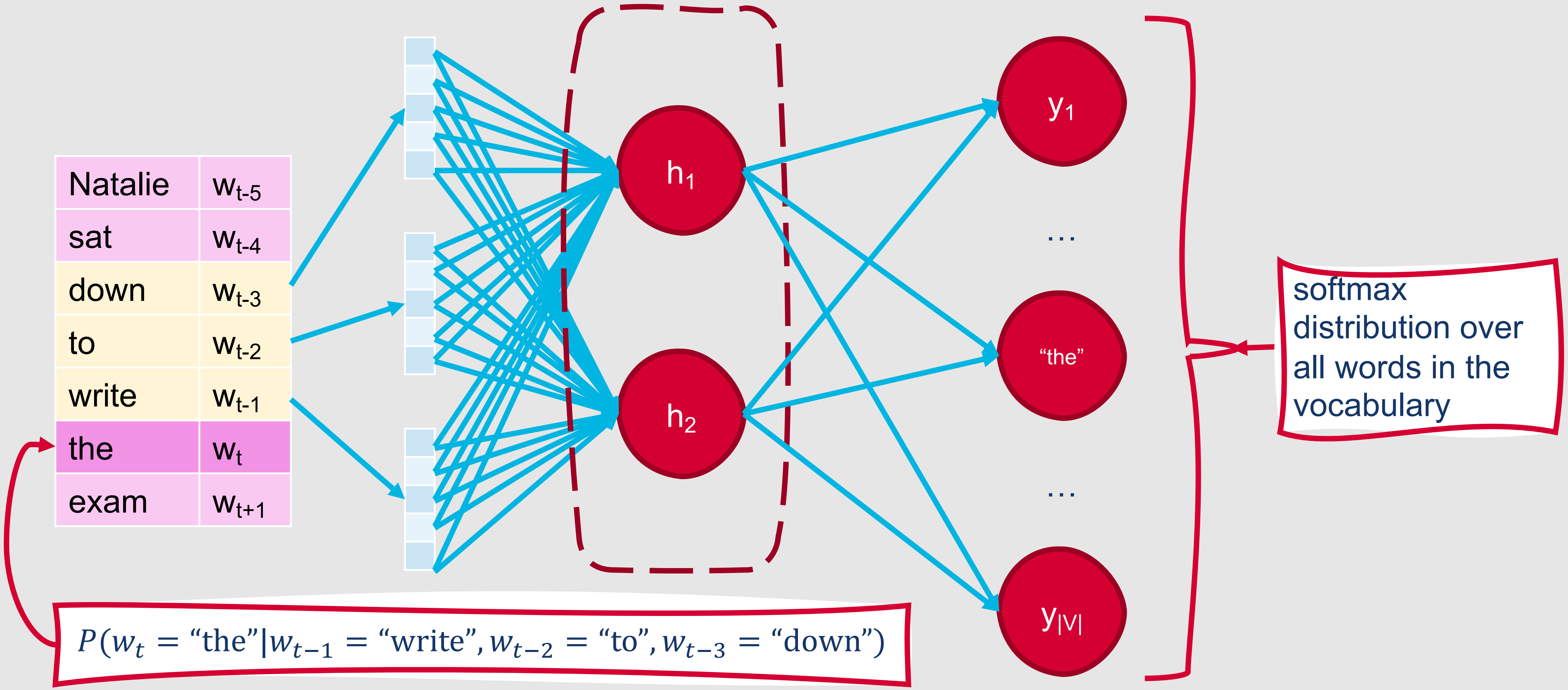


Natalie	w_{t-4}
sat	w_{t-3}
down	w_{t-2}
to	w_{t-1}
write	w_t
the	w_{t+1}
exam	w_{t+2}

softmax distribution over all words in the vocabulary

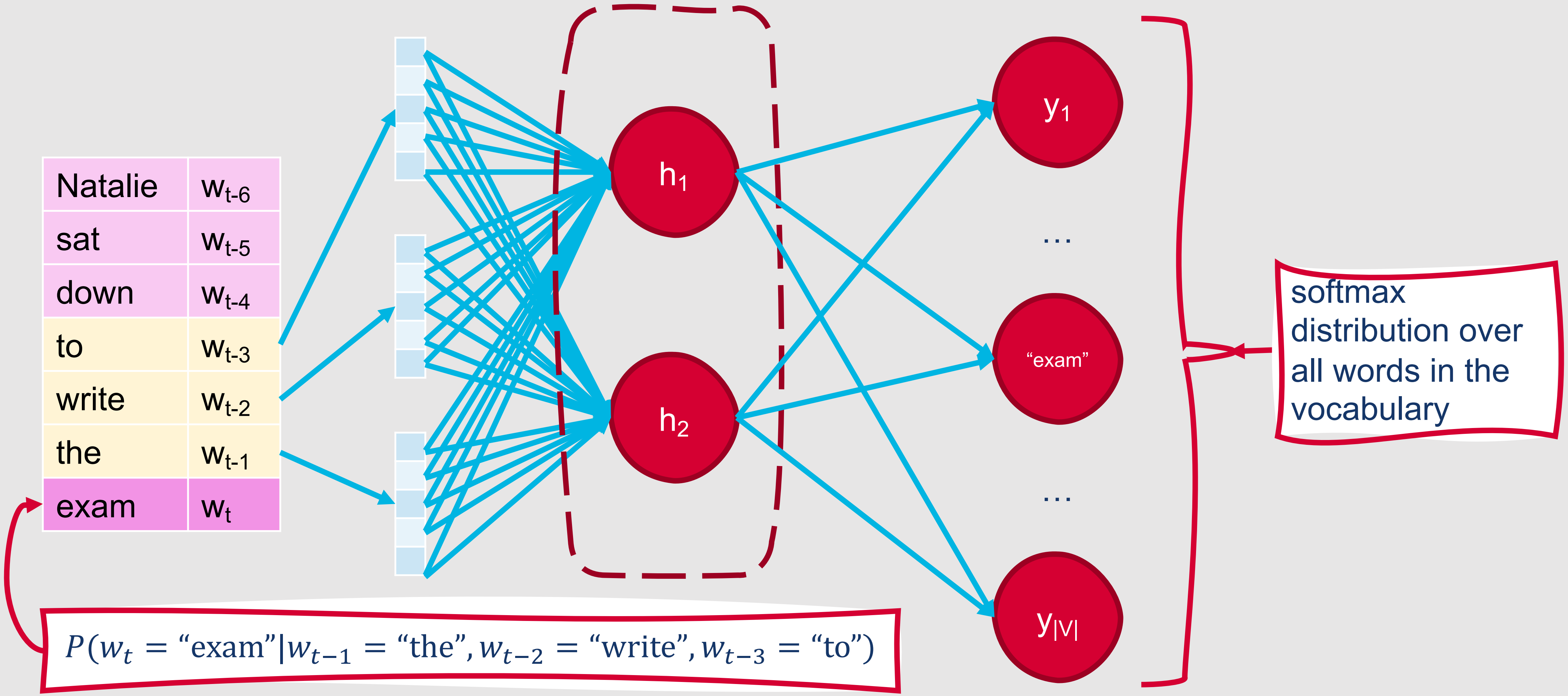
$P(w_t = \text{"write"} | w_{t-1} = \text{"to"}, w_{t-2} = \text{"down"}, w_{t-3} = \text{"sat"})$

Sliding Window Approach



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

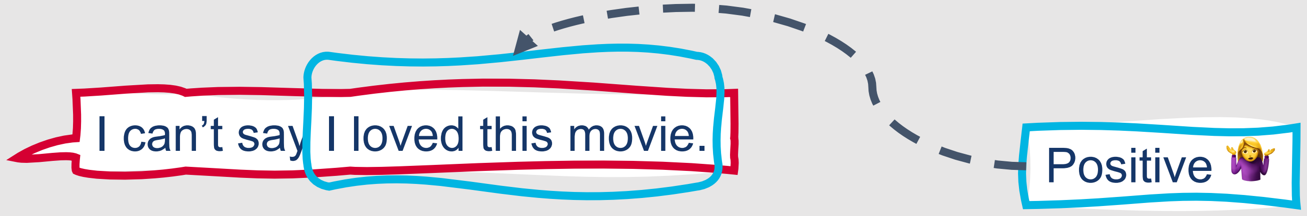
Sliding Window Approach



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

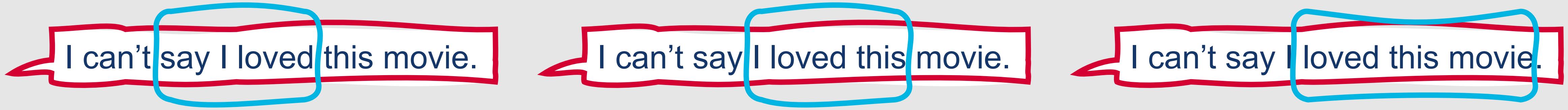
However, this method has some limitations.

- **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

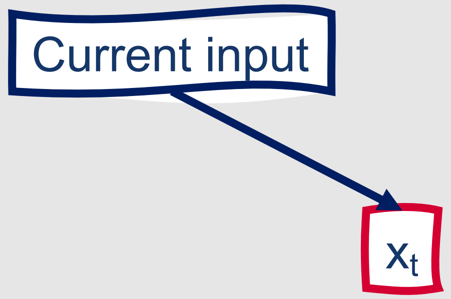
- Contain cycles within their connections, where the value of a unit is dependent upon outputs from previous timesteps

Can accept variable length inputs without the use of fixed-size windows

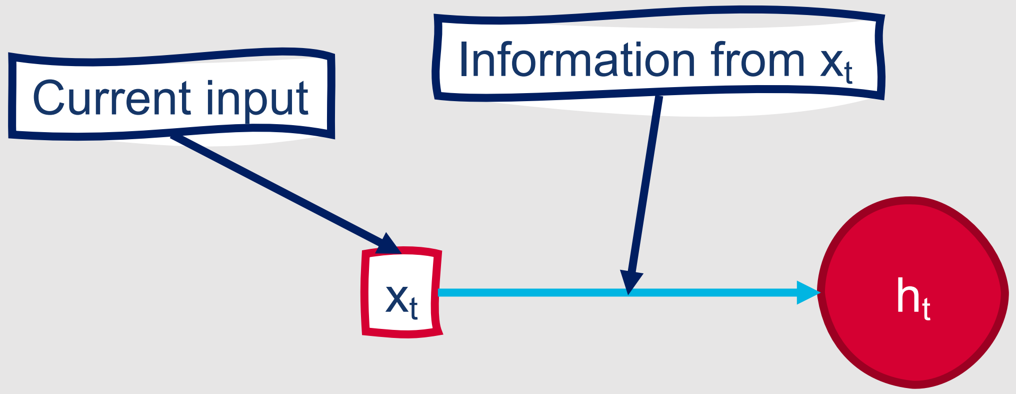
Many varieties exist

- "Vanilla" RNNs
- Long short-term memory networks (LSTMs)
- Gated recurrent units (GRUs)

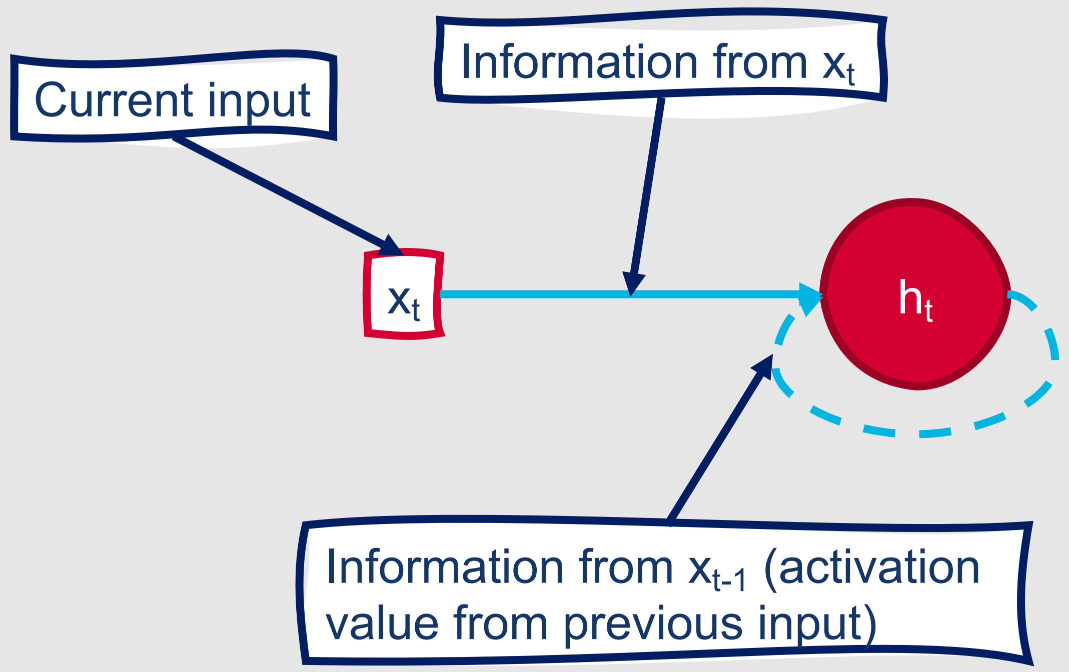
Vanilla RNN Unit



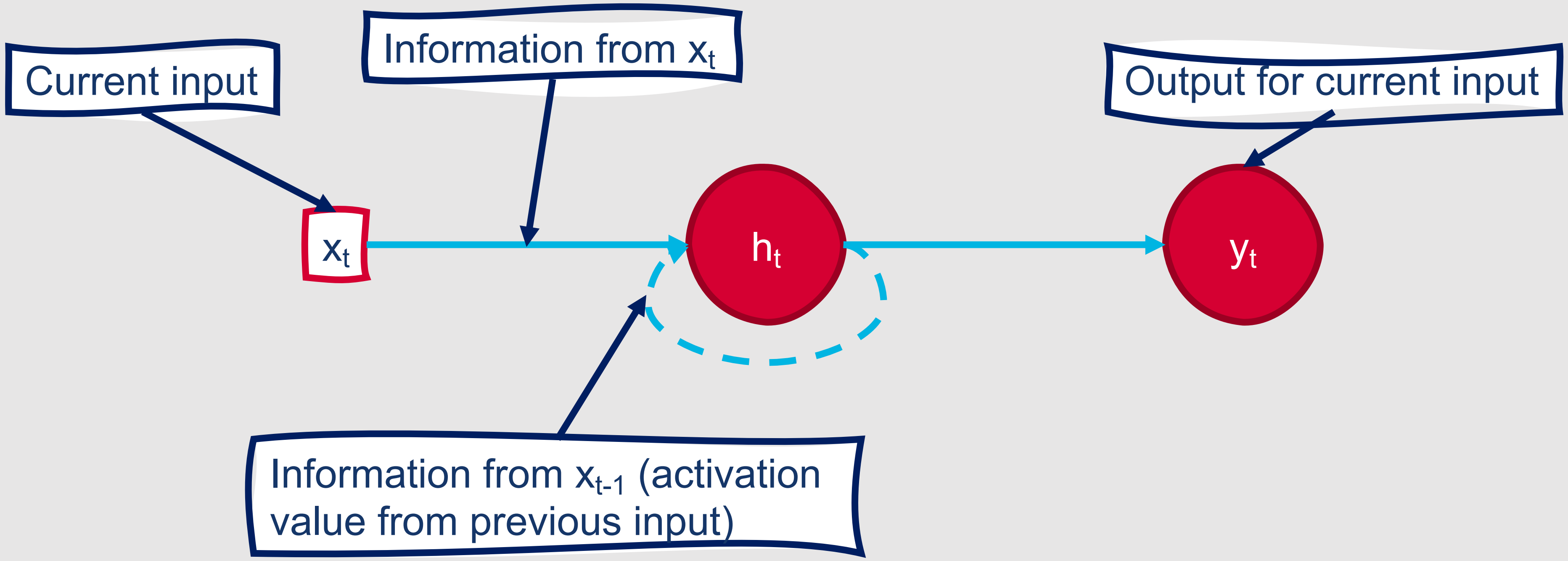
Vanilla RNN Unit



Vanilla RNN Unit


Current input
Information from x_t
x_t
h_t
Information from x_{t-1} (activation value from previous input)

Vanilla RNN Unit



Although they are more complex, computation units in RNNs 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

Biggest 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

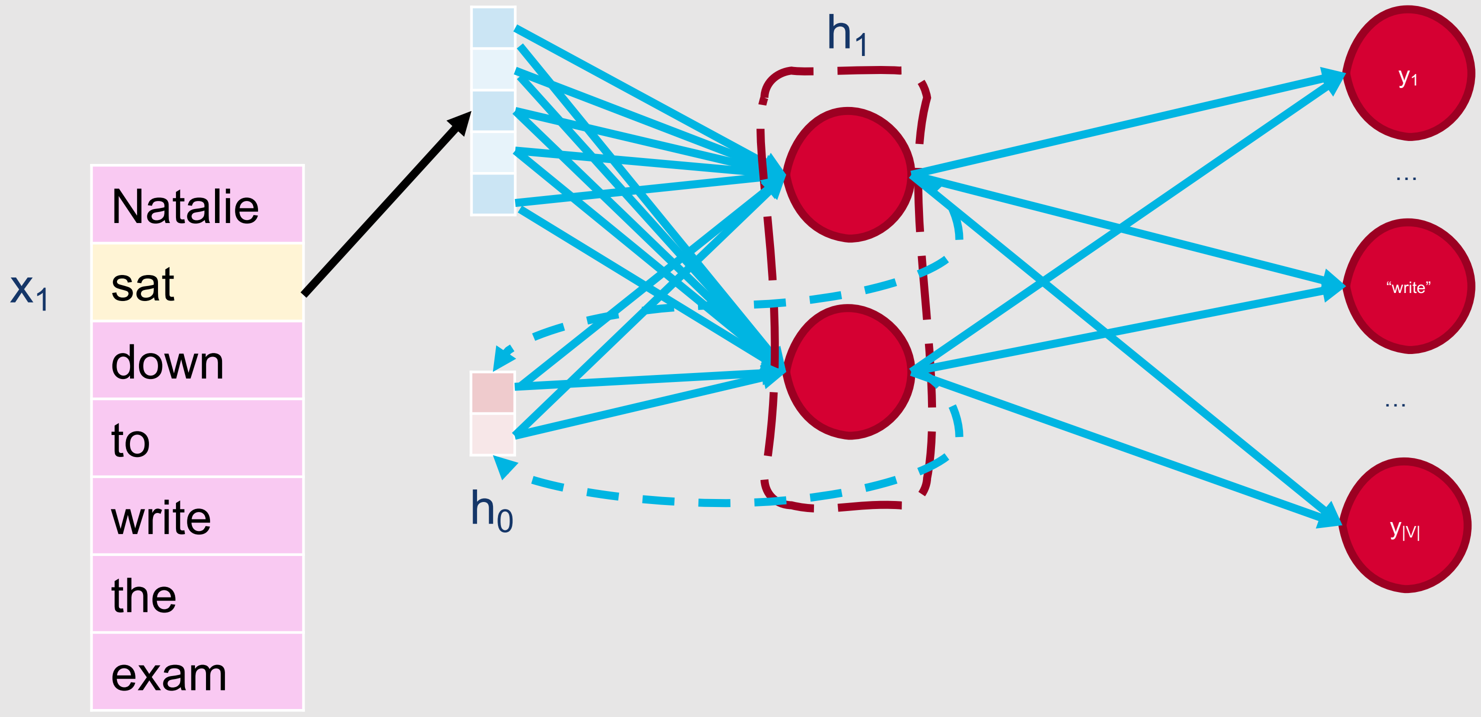
- Recall the basic set of equations for a feedforward neural network:
  - $\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$
  - $\mathbf{z} = V\mathbf{h}$
  - $y = \text{softmax}(\mathbf{z})$
- Just add (weights X activation values from previous timestep) product to the current (weights X inputs) product
  - $\mathbf{h_t} = \sigma(W\mathbf{x}_t + U\mathbf{h}_{t-1} + \mathbf{b})$
  - $\mathbf{z_t} = V\mathbf{h}_t$
  - $y_t = \text{softmax}(\mathbf{z})$
- *W*, *U*, and *V* are shared across all timesteps

Formal Algorithm

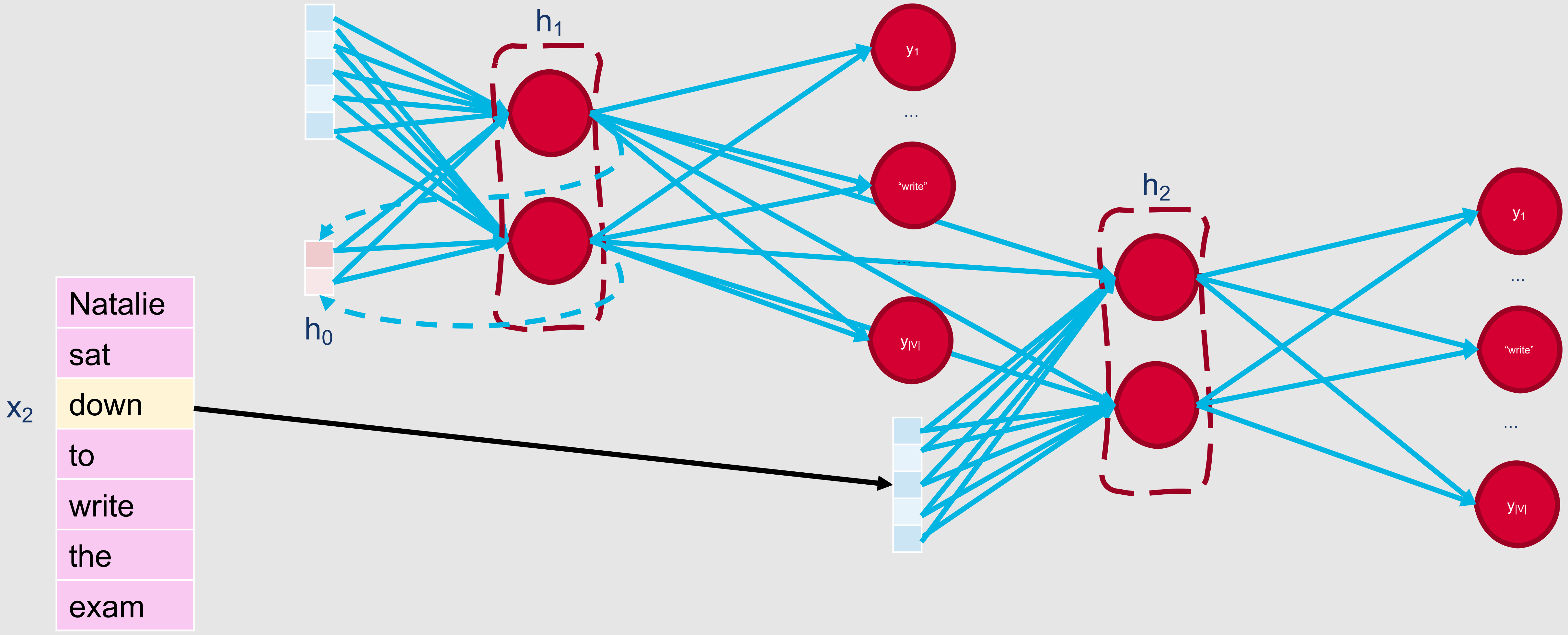
```
$h_0$ ← 0  # Initialize activations from the hidden layer to 0

# Iterate through each input element in temporal order
for i ← 1 to length(x) do:
    $\mathbf{h_i} \leftarrow g(U\mathbf{h_{i-1}} + W\mathbf{x_i} + \mathbf{b})$   # Bias vector is optional
    $y_i \leftarrow f(V\mathbf{h_i})$
```

New values for h and y are calculated with each time step!

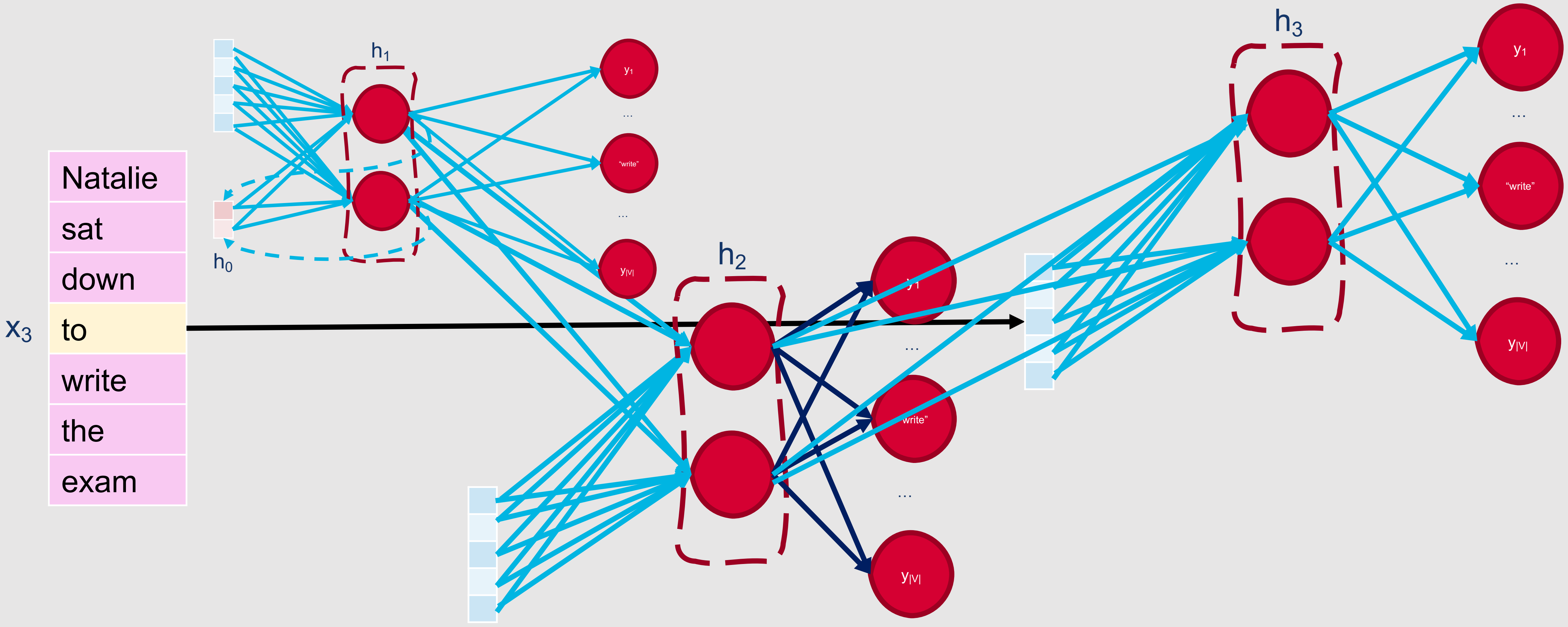
Earlier Example: RNN Edition



Earlier Example: RNN Edition



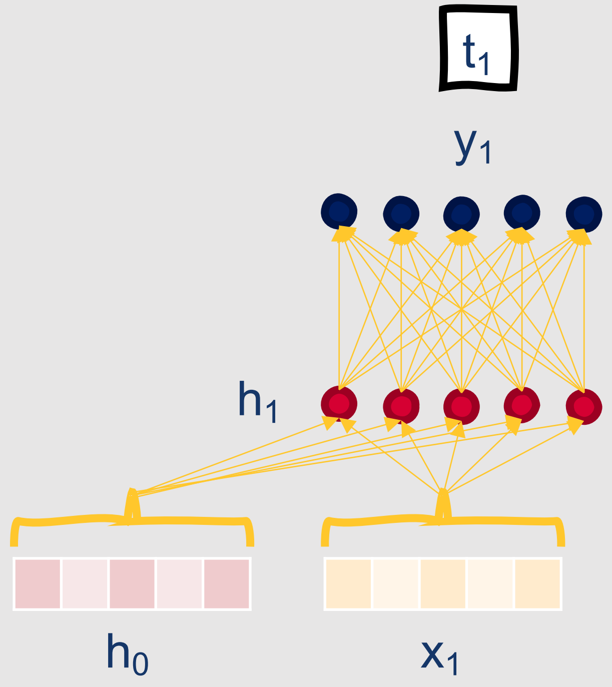
Earlier Example: RNN Edition



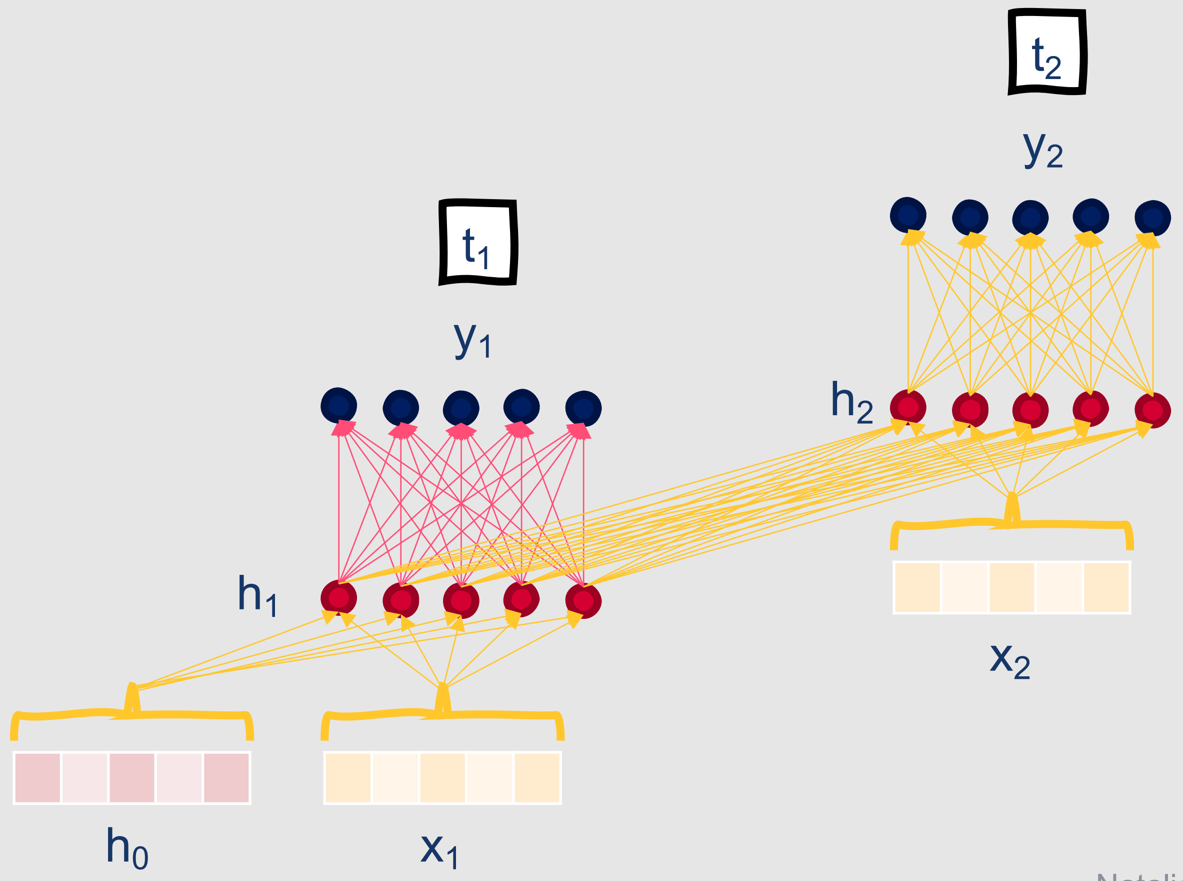
Training RNNs

- 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*
- For forward inference:
  - Compute h_t and y_t at **each step in time**
  - Compute the loss at **each step in time**
- For backward inference:
  - Process the sequence in reverse
  - Compute the required error gradients at **each step backward in time**

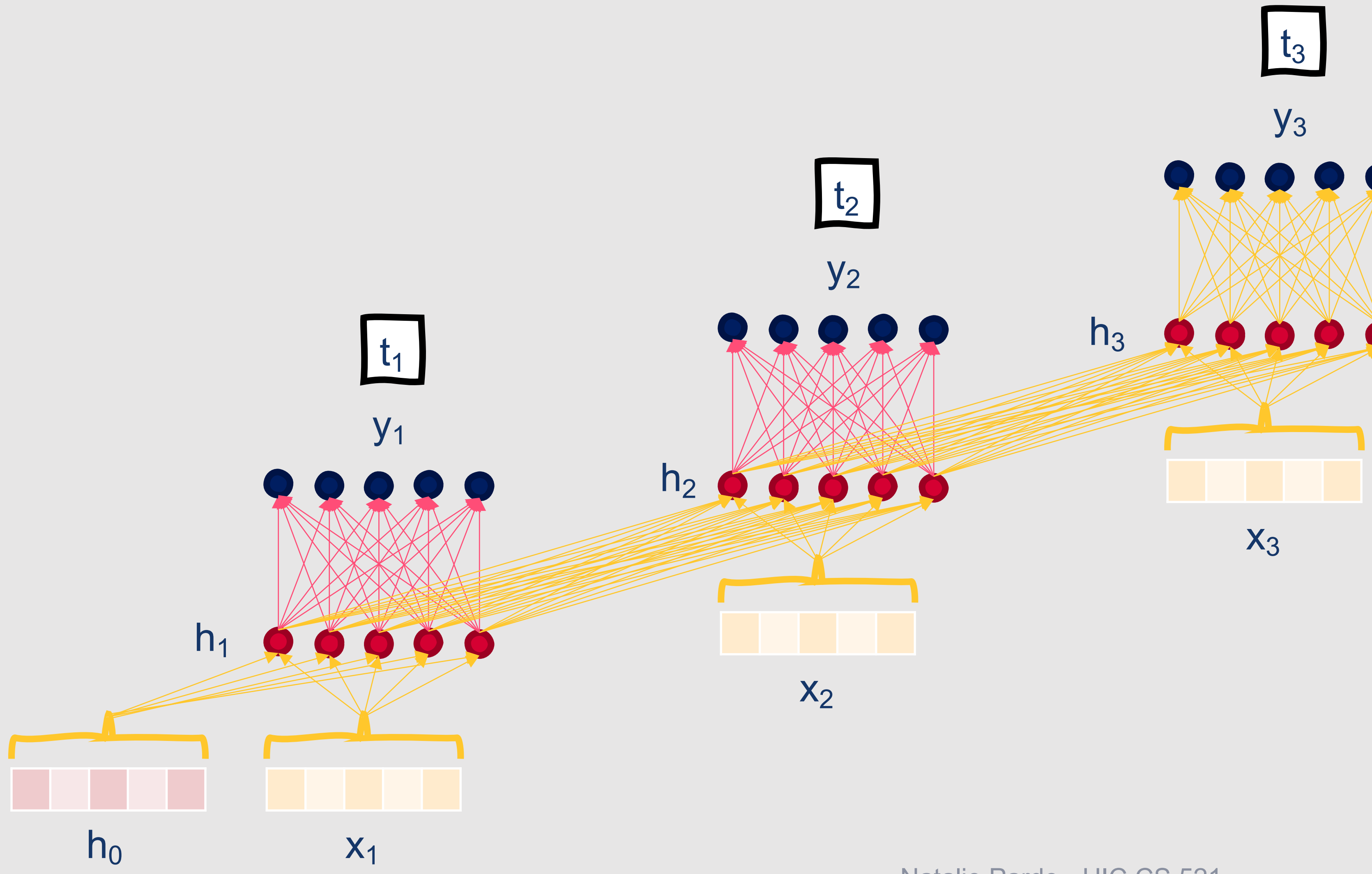
Forward Pass



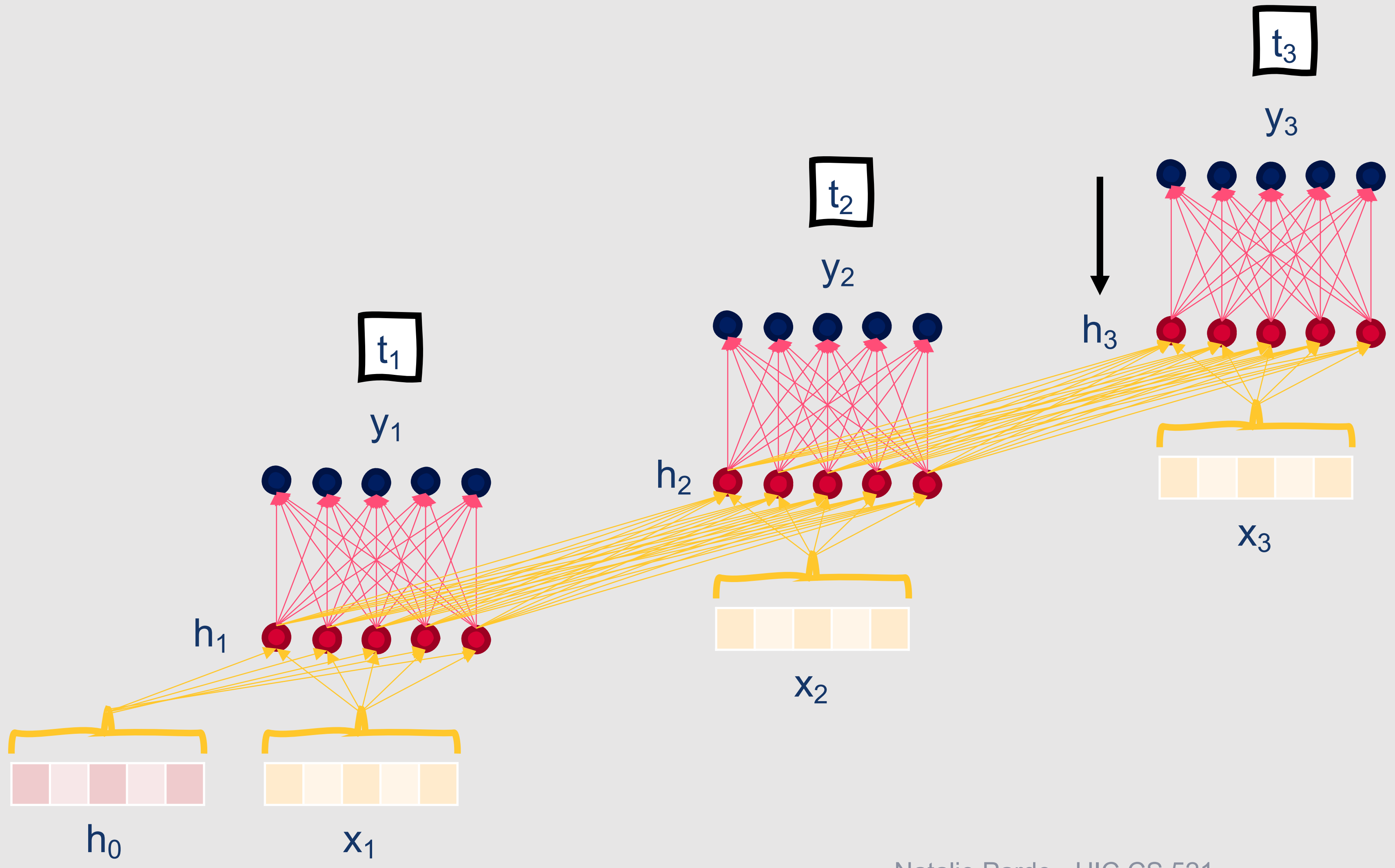
Forward Pass



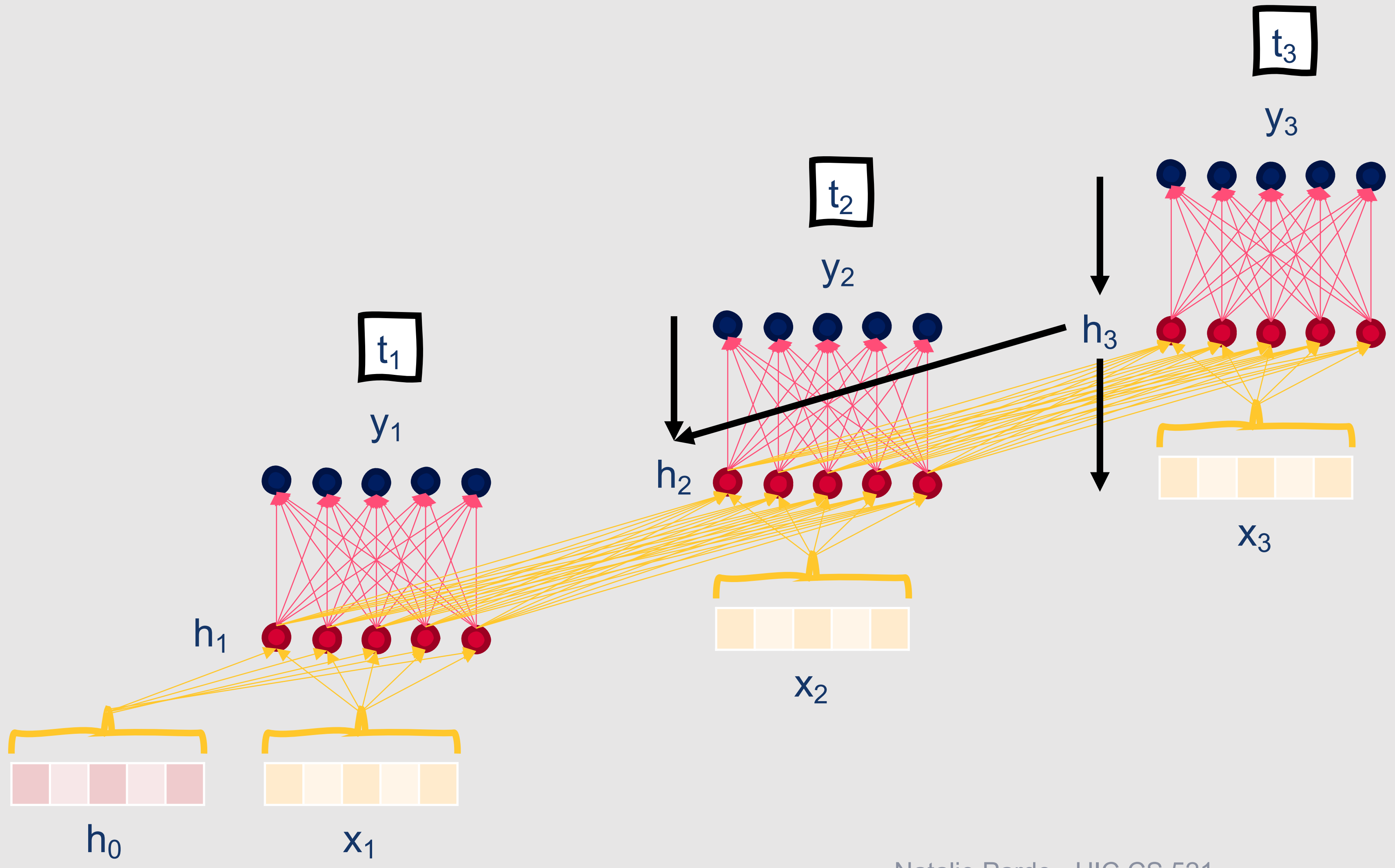
Forward Pass



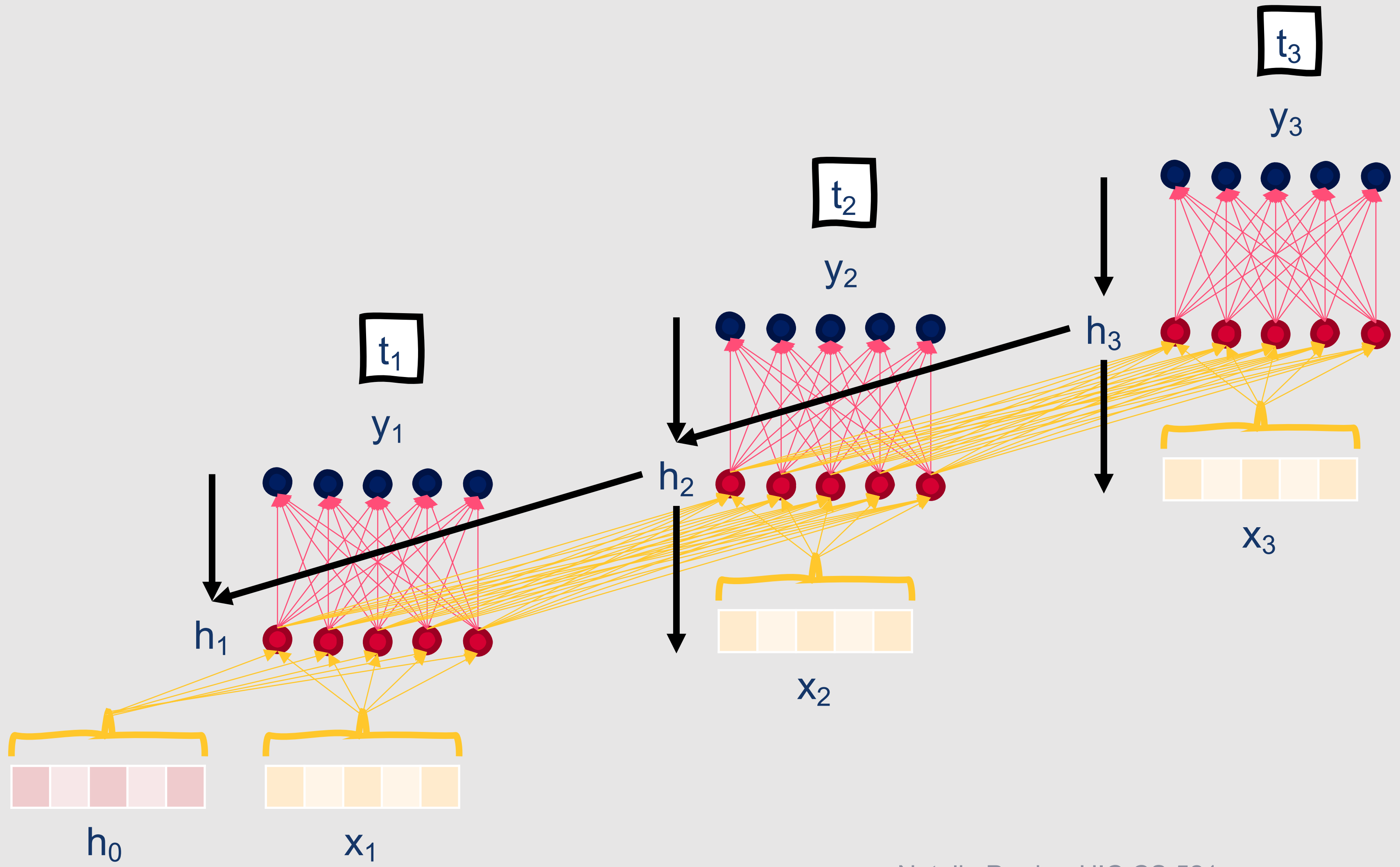
Backward Pass



Backward Pass

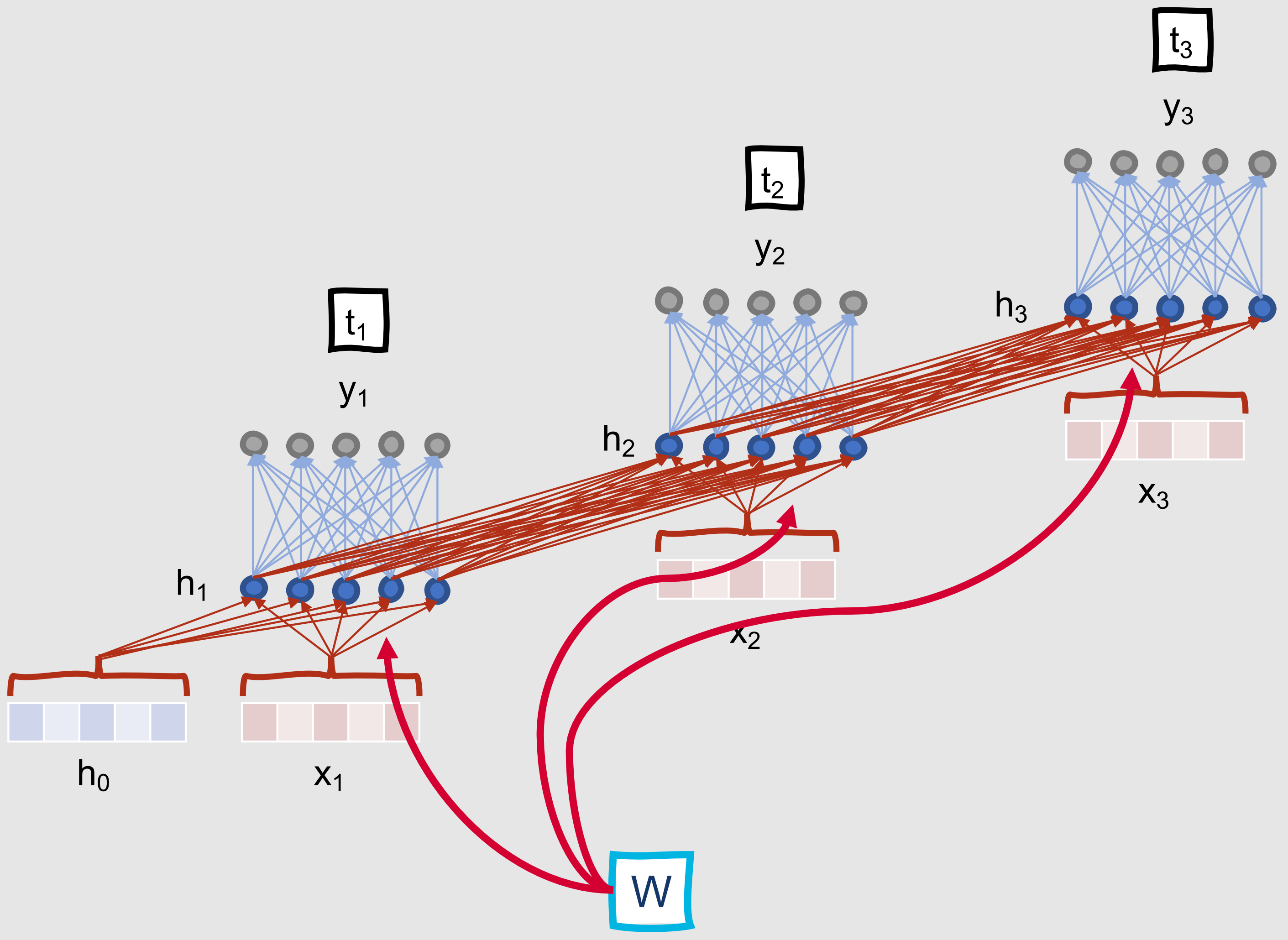


Backward Pass



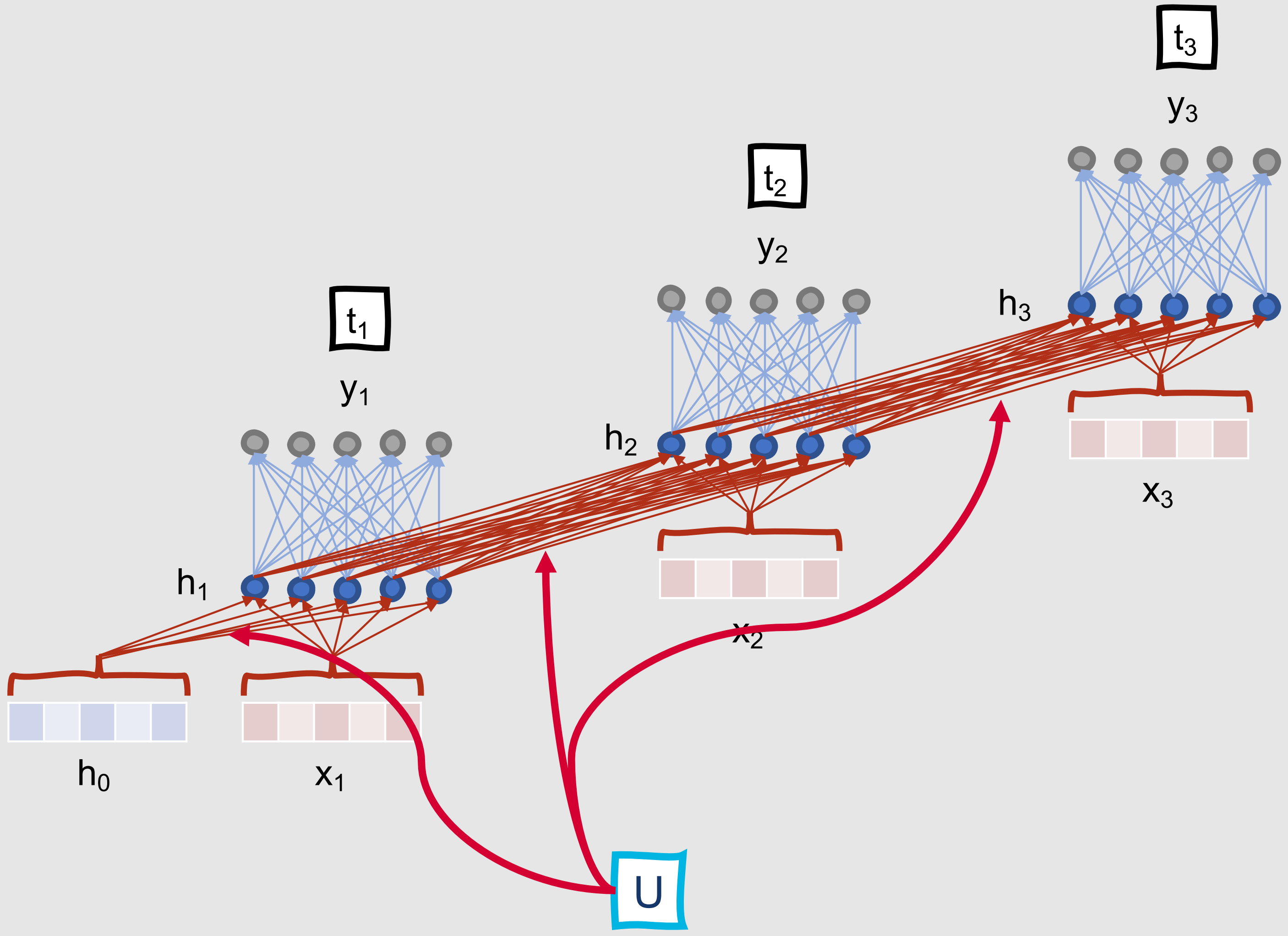
Updated Backpropagation Equations

- 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



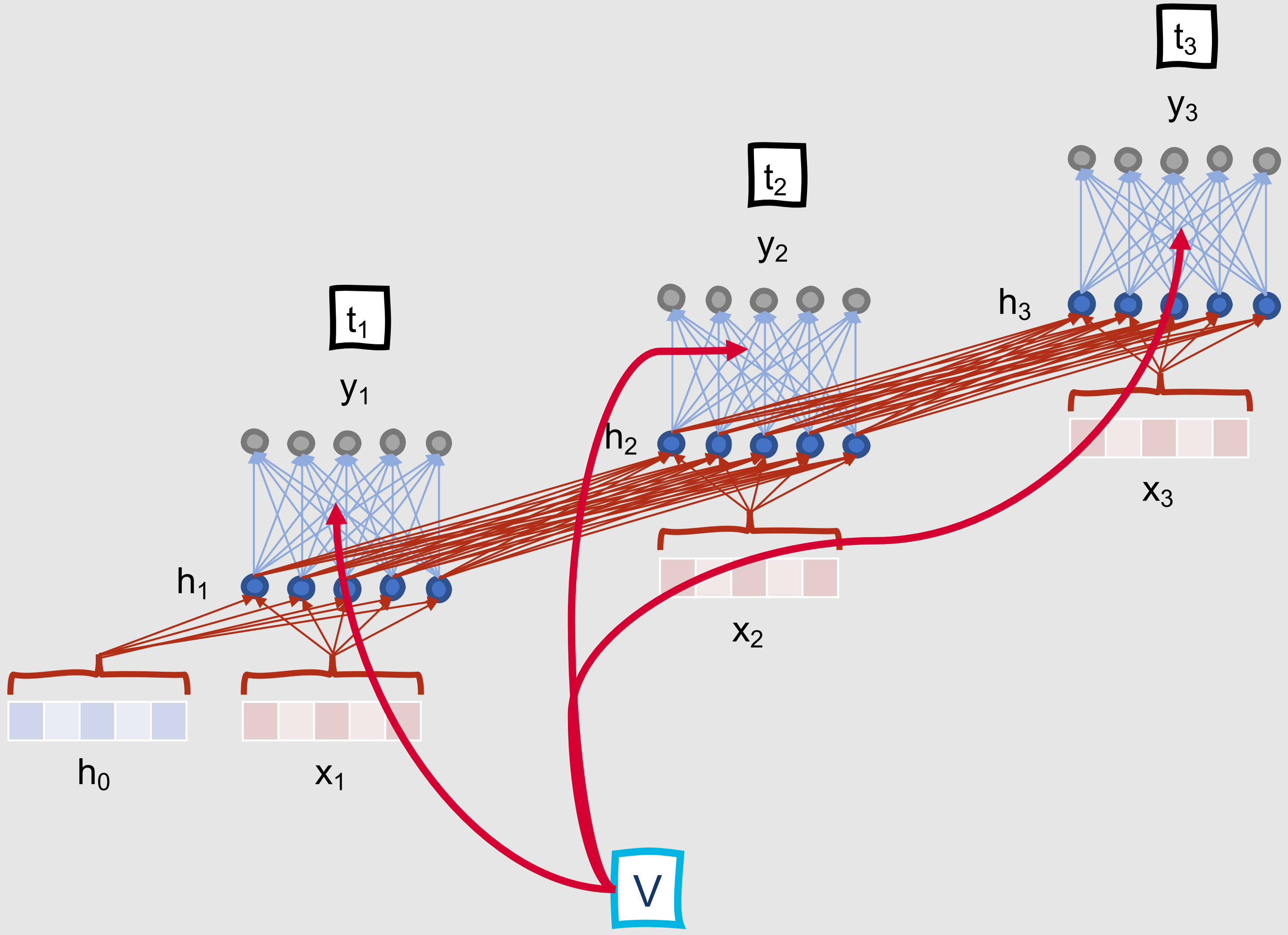
Updated Backpropagation Equations

- 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



Weight Update Equations

- Updating the weights for *V* works no differently from feedforward networks
- When updating the other weights, remember that a hidden layer, δ_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*
  - $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial a}\frac{\partial a}{\partial W} = \delta_h x_t$
  - $\frac{\partial L}{\partial U} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial a}\frac{\partial a}{\partial U} = \delta_h h_{t-1}$
- Backpropagate the error from δ_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*!

Earlier language models (e.g., n-gram language models or those using feedforward neural networks) attempted 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** address many of these challenges

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

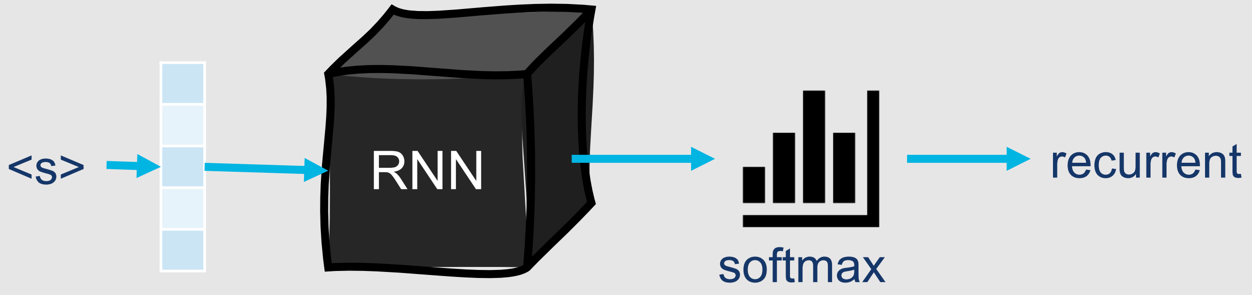
Generation with Neural Language Models

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

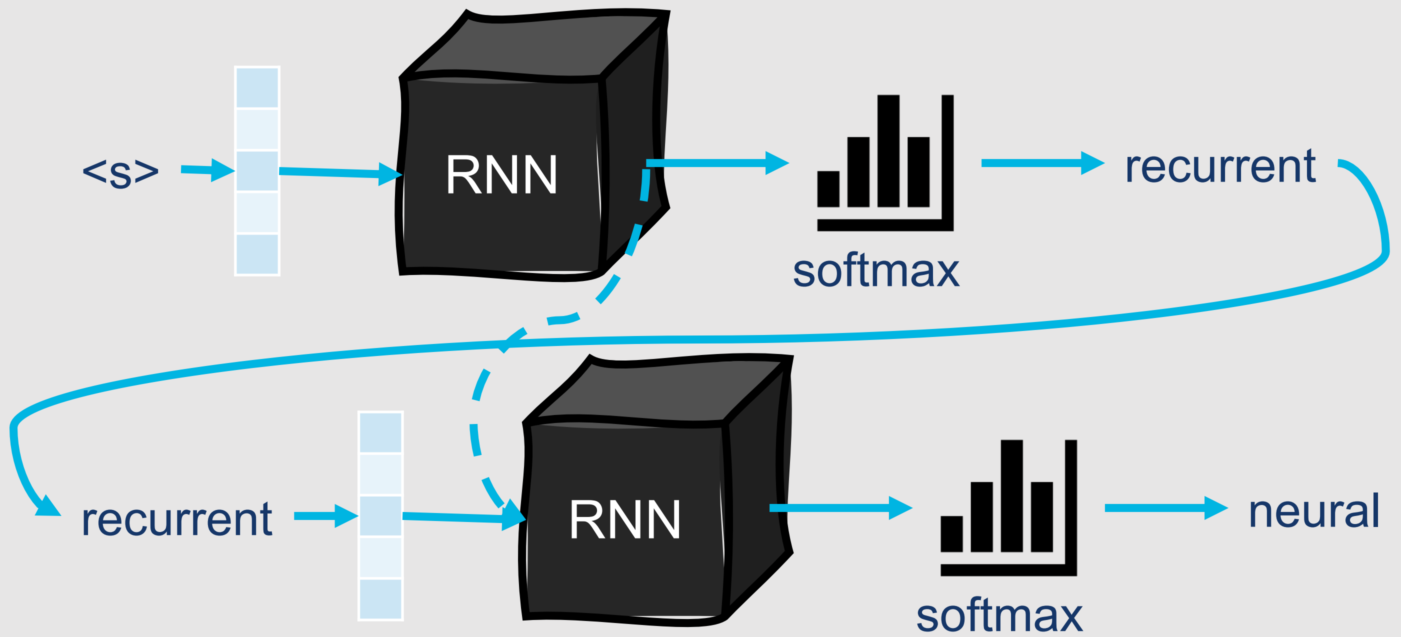
Autoregressive Generation

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

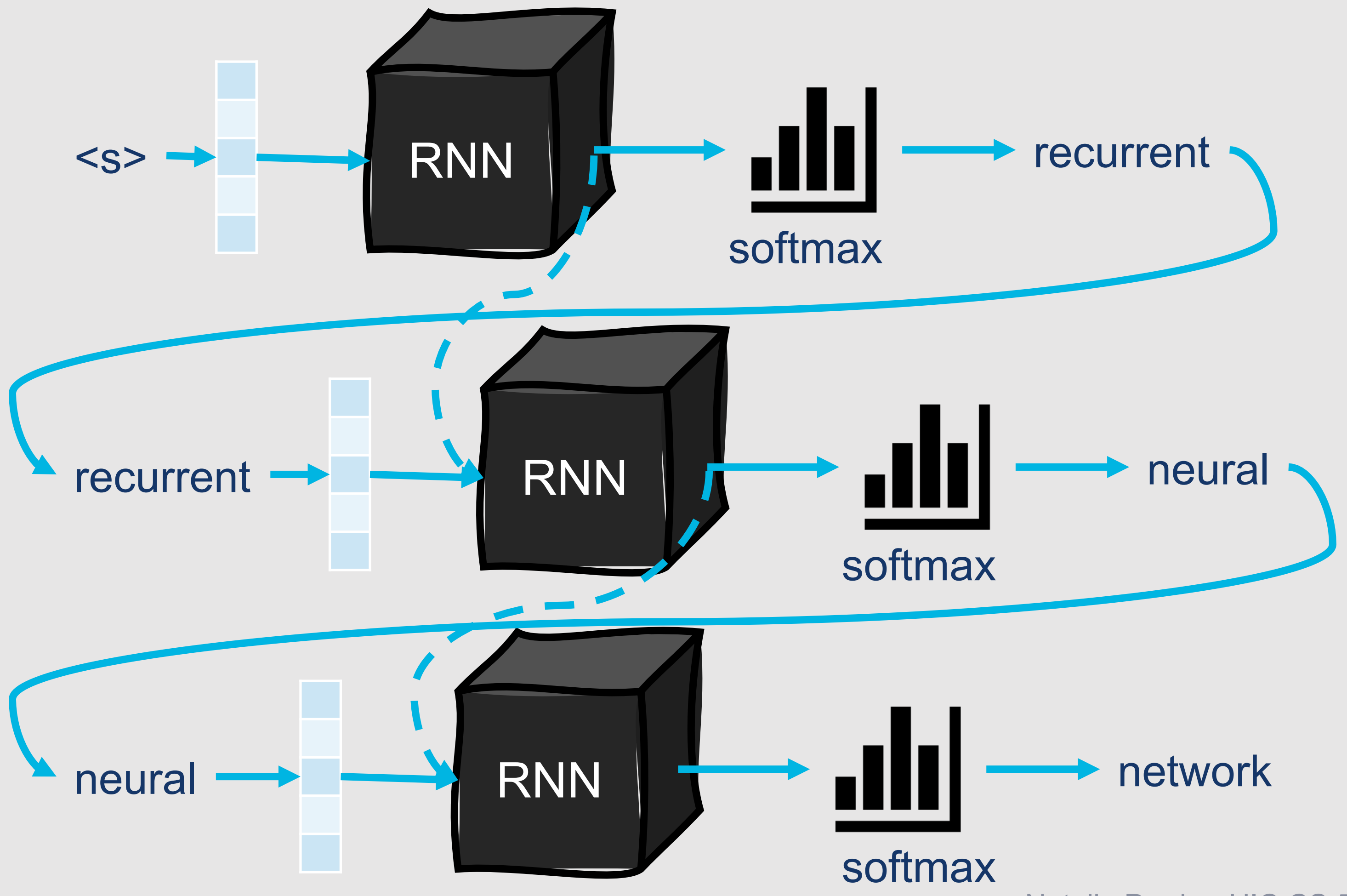
Autoregressive Generation



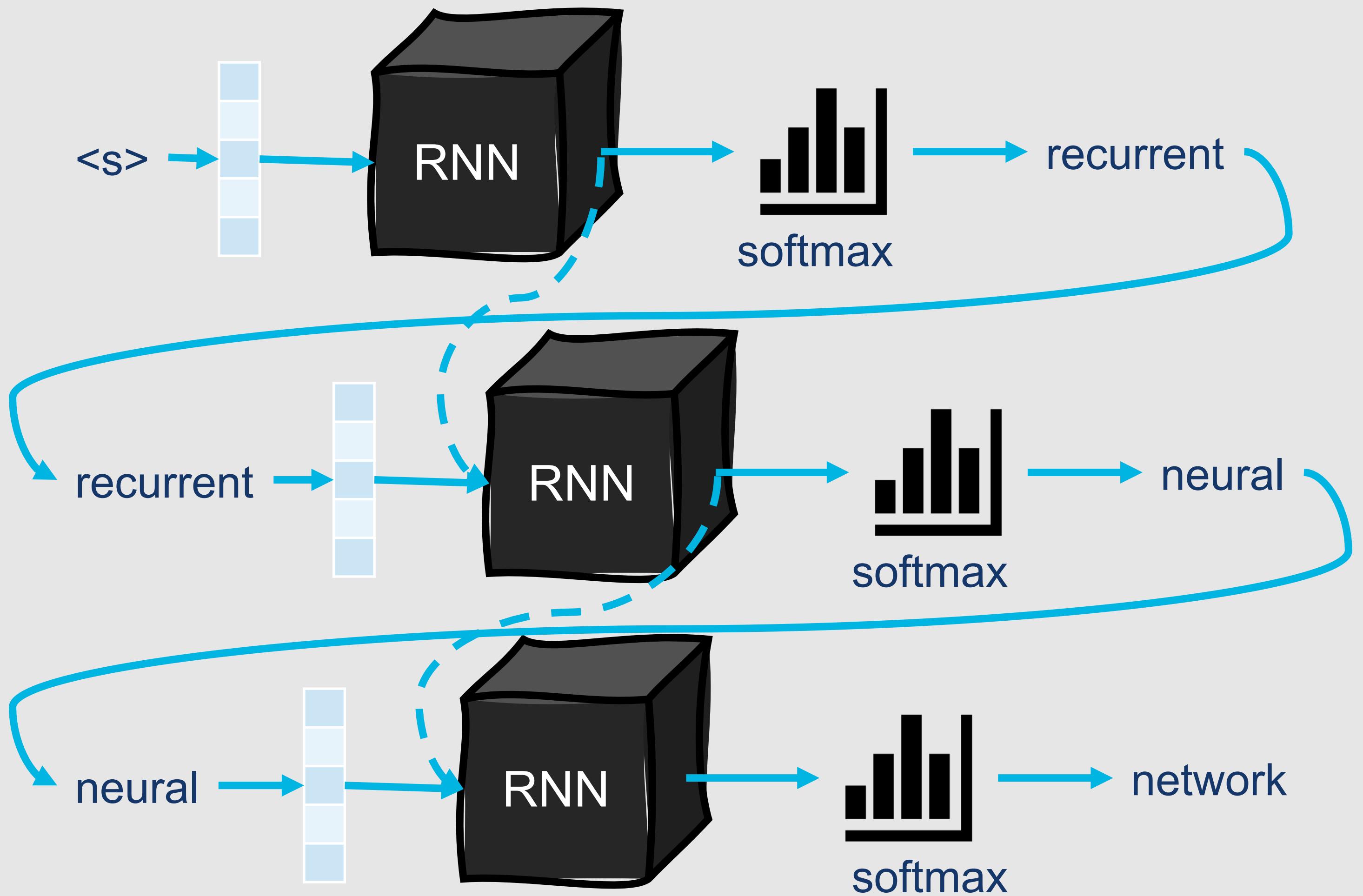
Autoregressive Generation



Autoregressive Generation



Autoregressive Generation



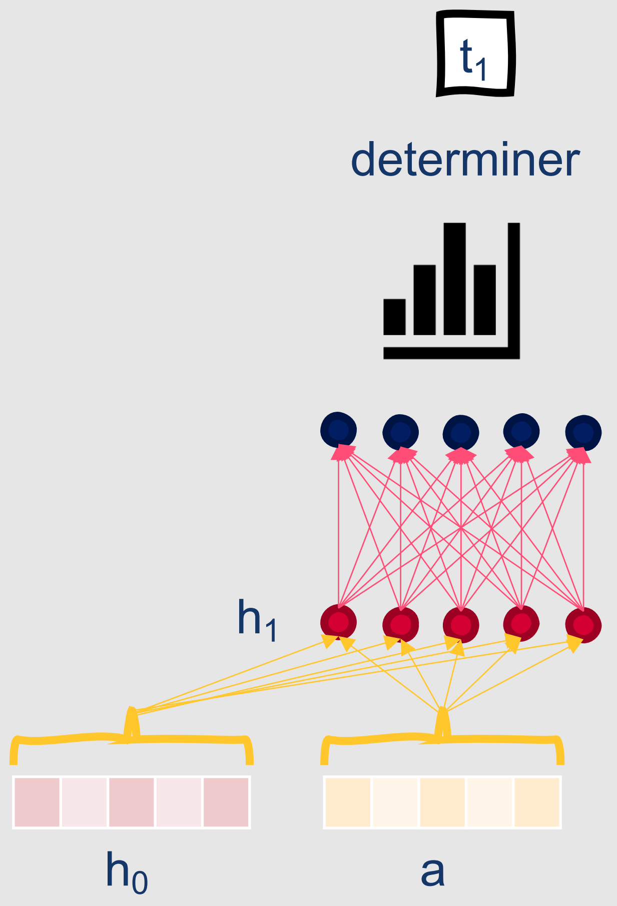
Key to successful autoregressive generation?

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

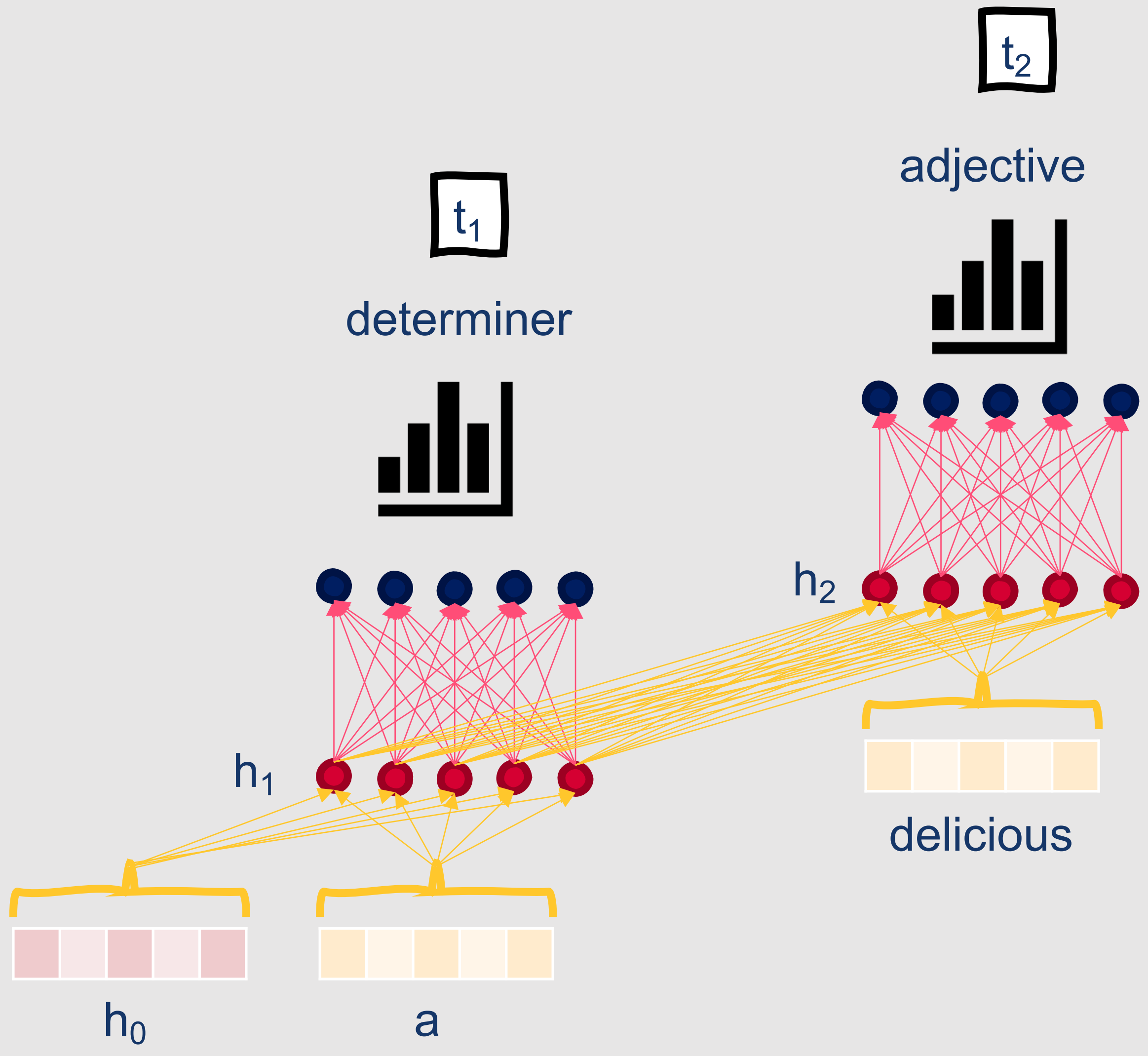
Sequence processing models like RNNs are also useful for many classification problems.

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

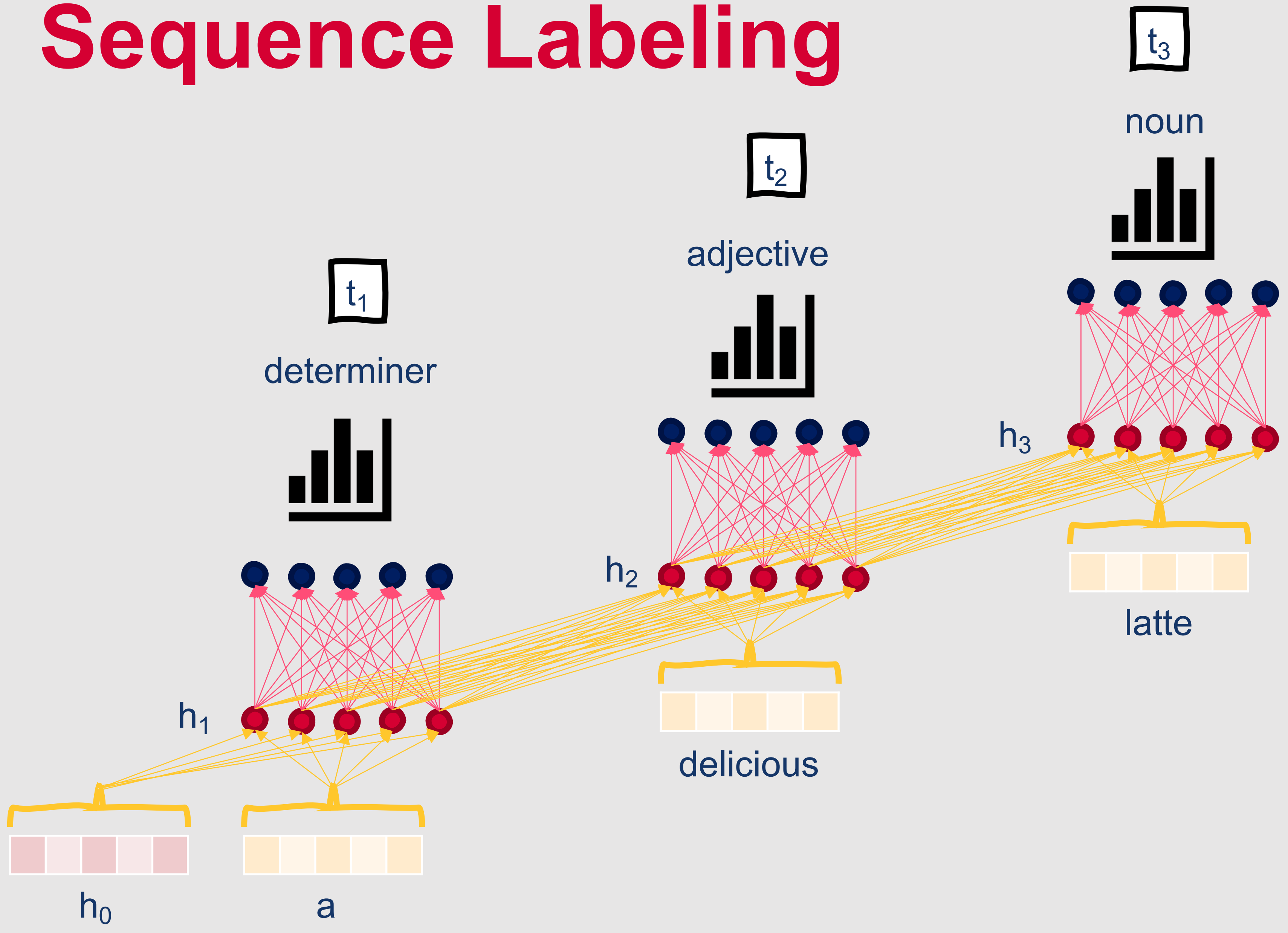
Sequence Labeling



Sequence Labeling



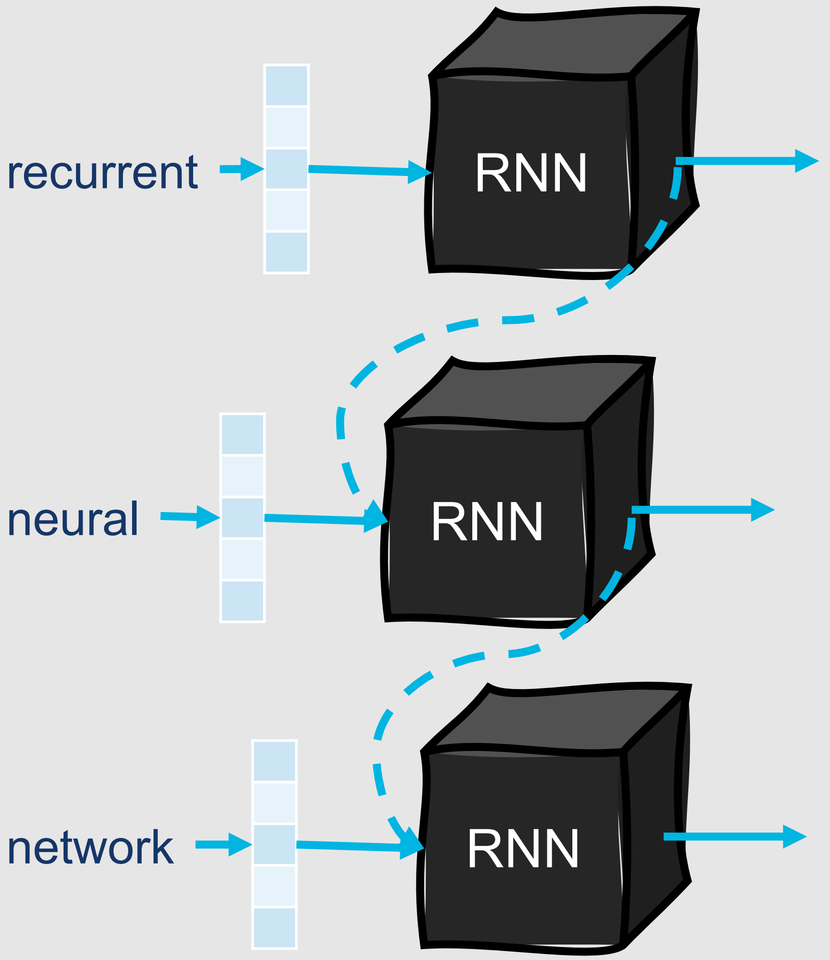
Sequence Labeling



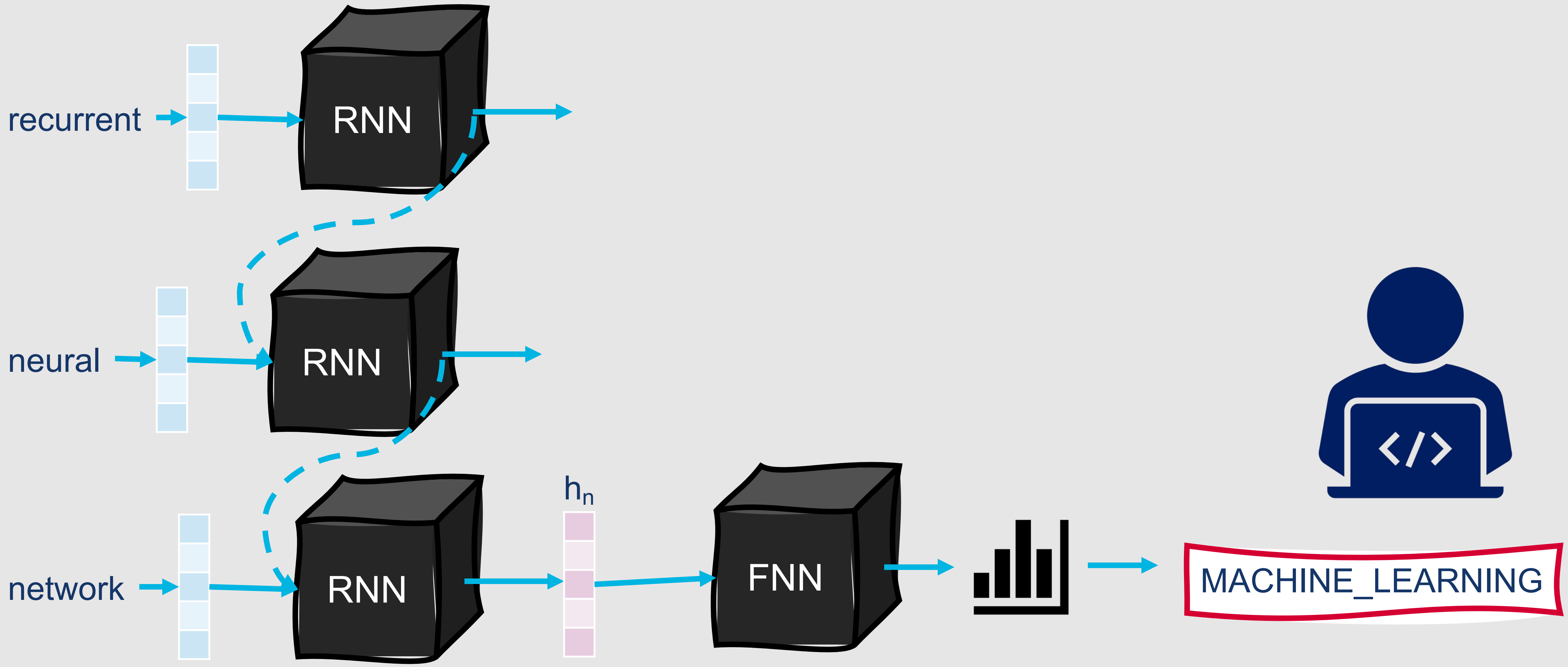
How to use RNNs for sequence classification?

1. Pass the sequence through an RNN one word at a time, as usual
2. Assume that the hidden layer for the final word, $\mathbf{h}_n$, acts as a compressed representation of the entire sequence
3. Use $\mathbf{h}_n$ as input to a subsequent feedforward neural network
4. Choose a class via softmax over all the possible classes

Sequence Classification



Sequence Classification

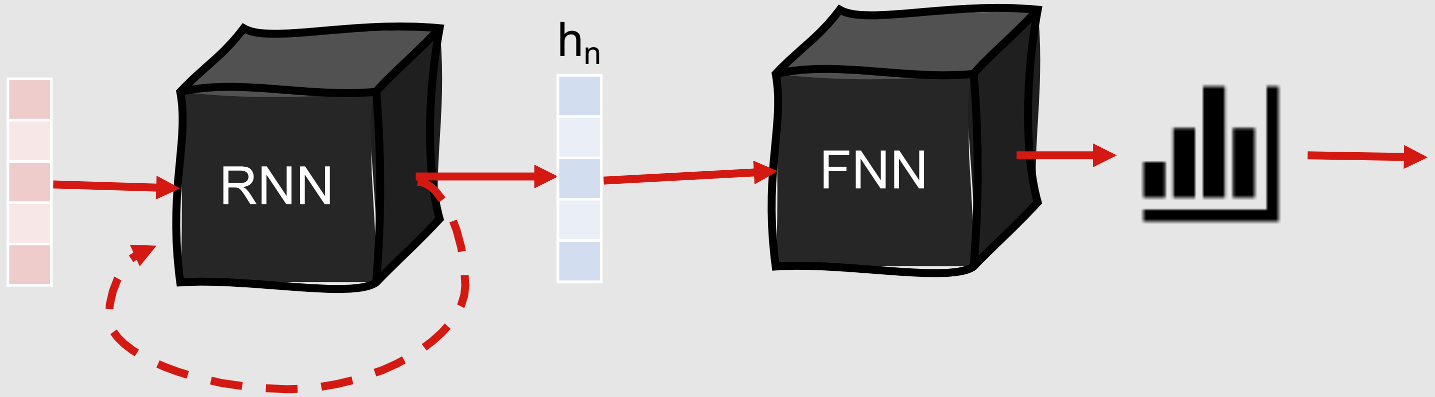


Notes about Sequence Classification

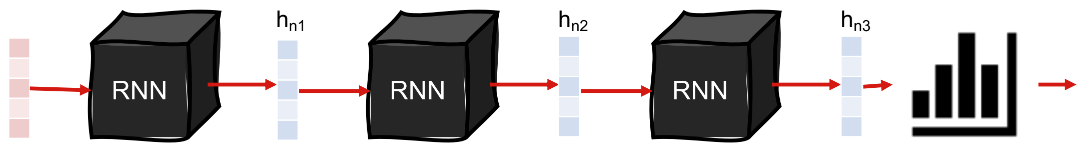
- Loss function is based entirely on the final classification task (not with intermediate outputs)
- Errors are still backpropagated all the way through the RNN
- The process of adjusting weights the entire way through the network based on the loss from a downstream application is often referred to as **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



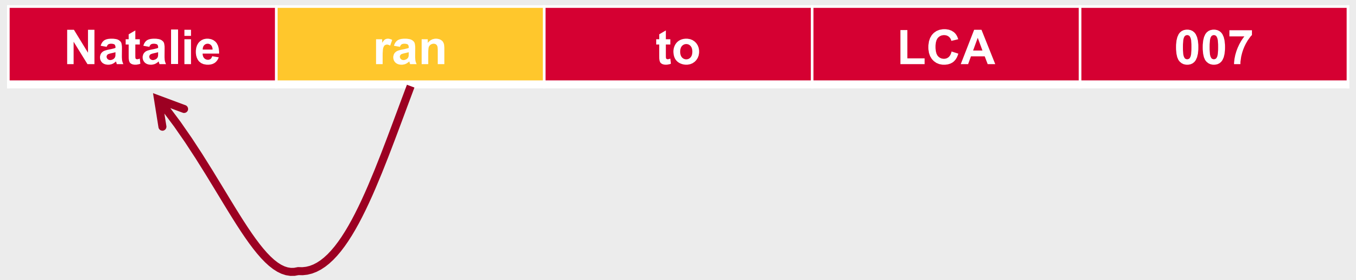
Stacked RNNs



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

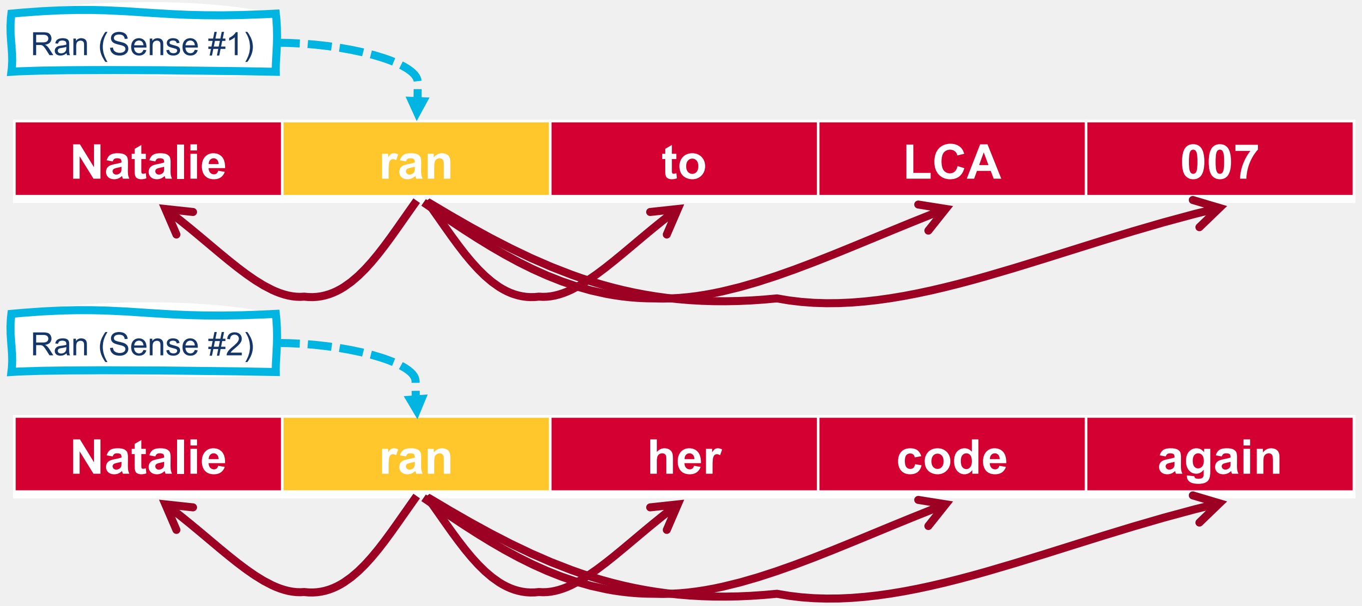
Bidirectional RNNs

- Standard 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*



Bidirectional RNNs

- However, in many cases the context after the current timestep could be useful as well!
- In many applications we also have access to the entire input sequence anyway



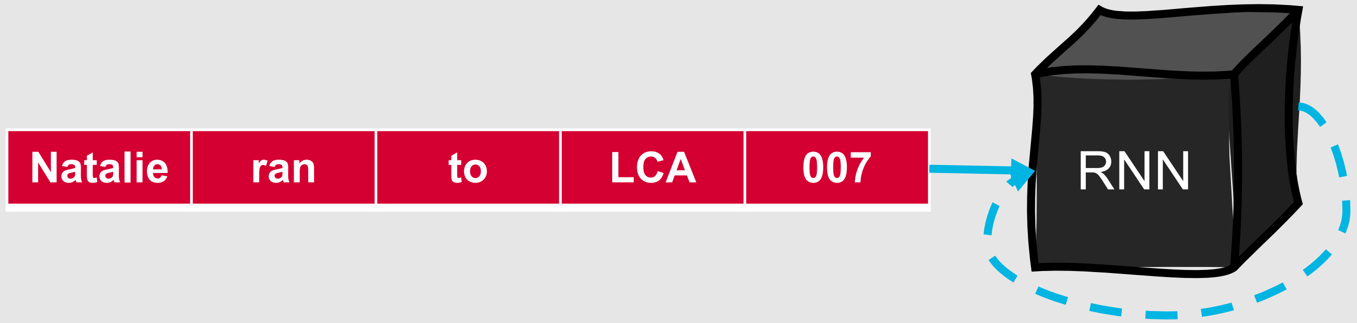
Bidirectional RNNs

- How can we make use of information both before and after the current timestep?
  - 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

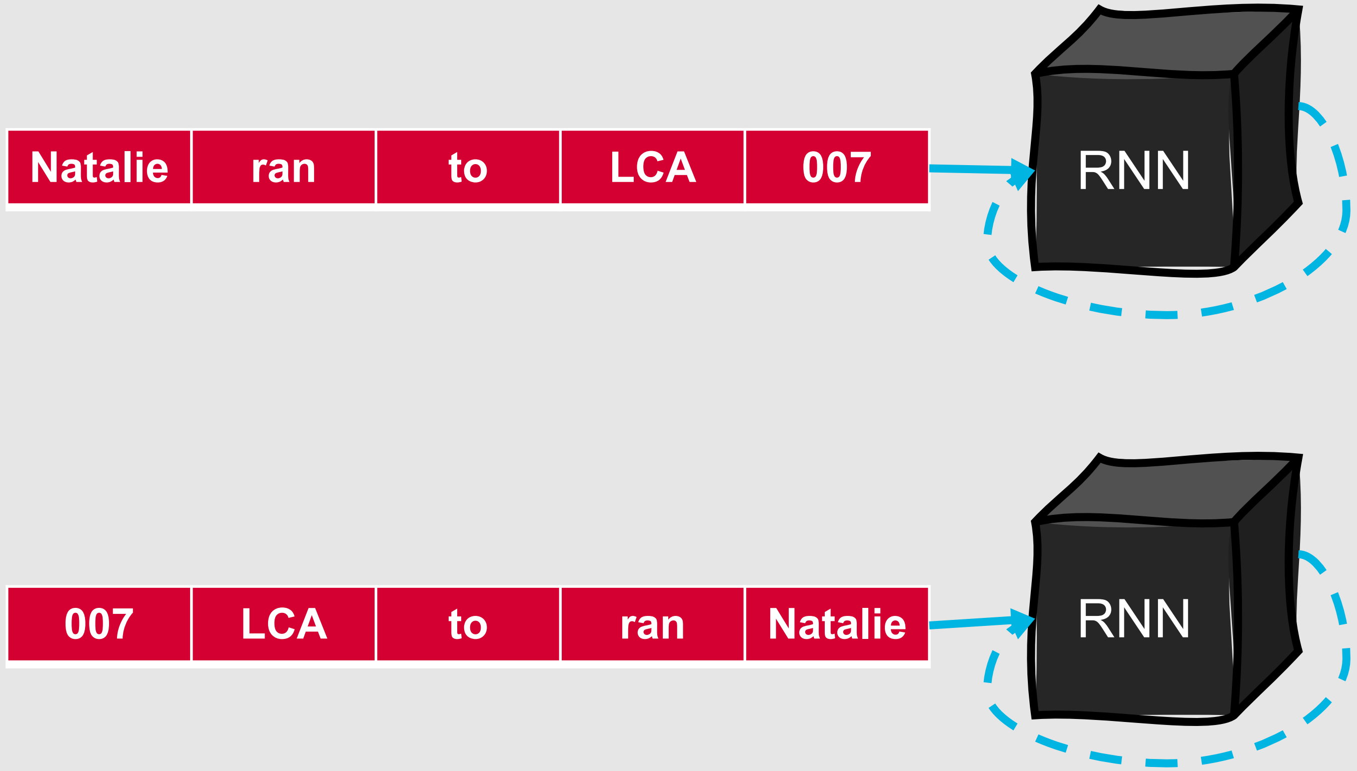
Bidirectional RNNs

- 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 prior and future 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.

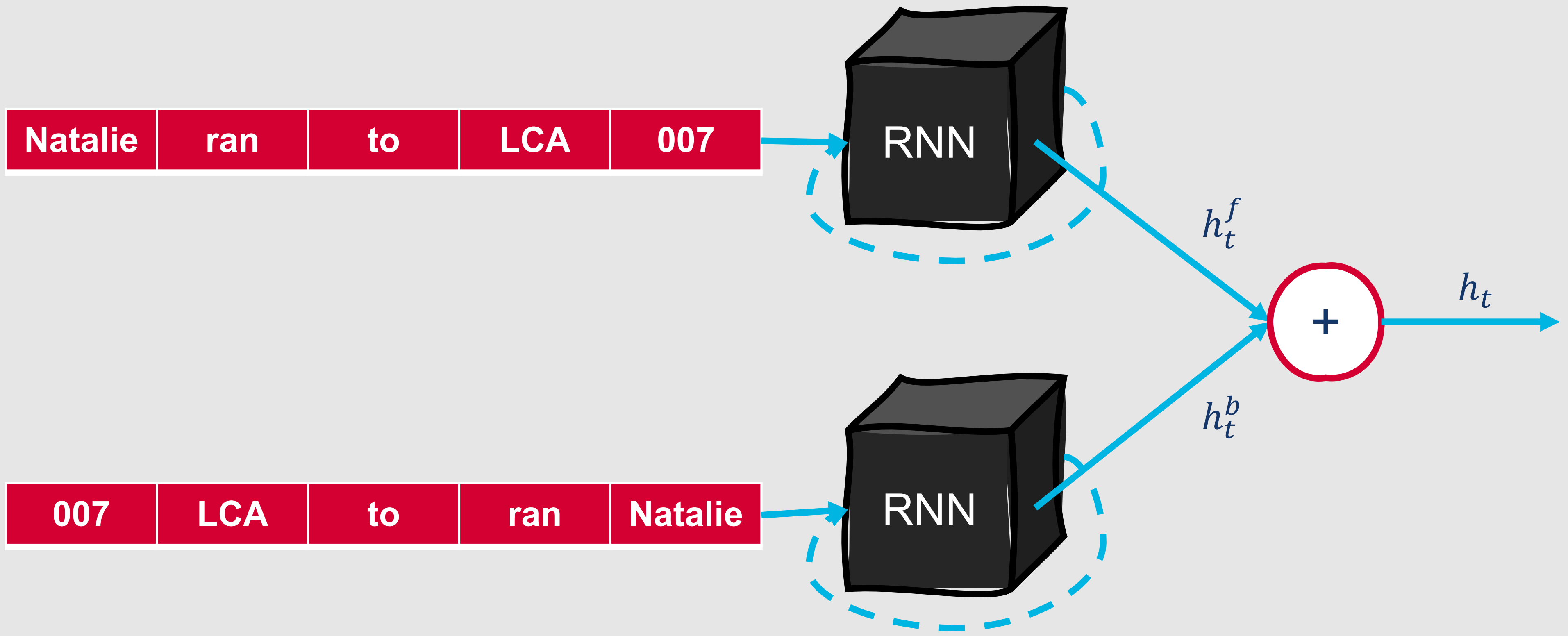
Bidirectional RNNs



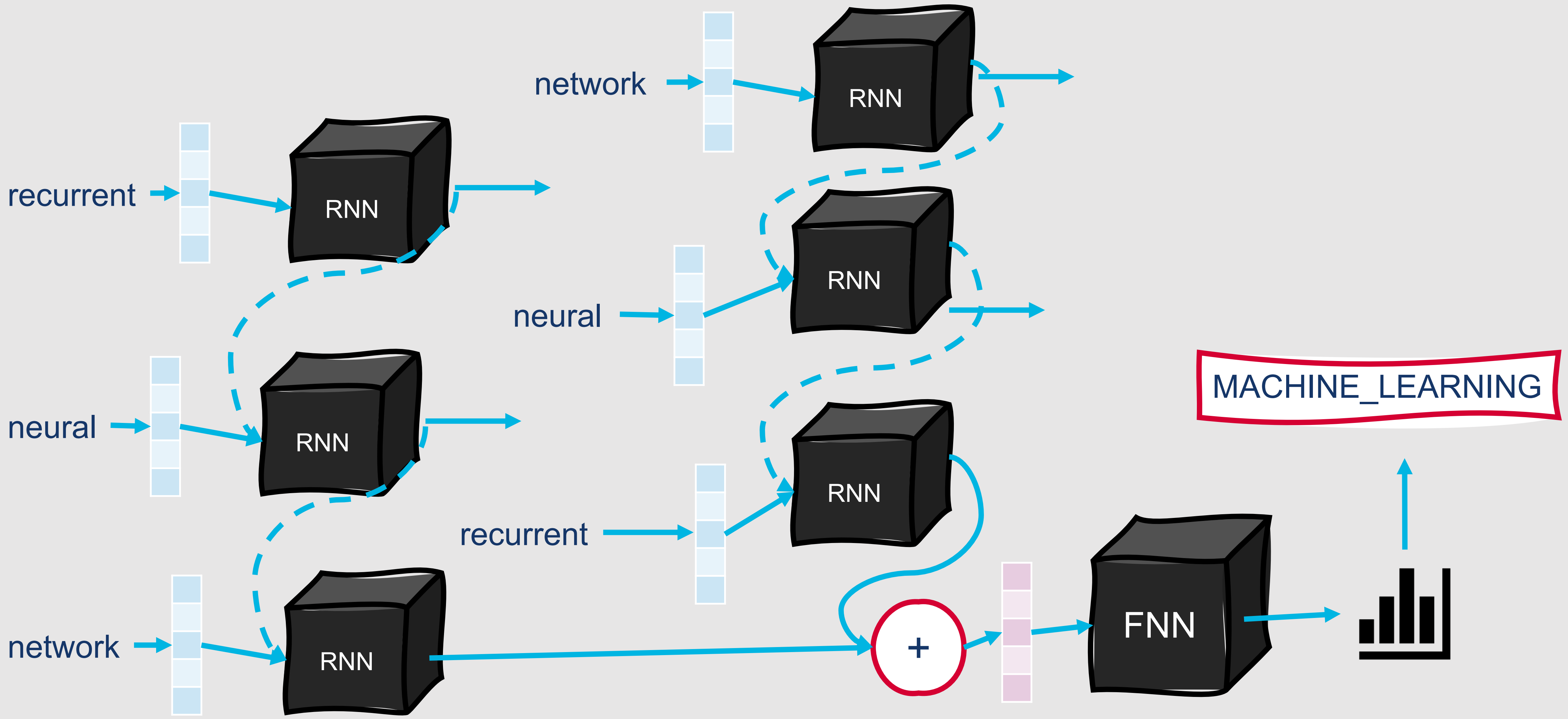
Bidirectional RNNs



Bidirectional RNNs

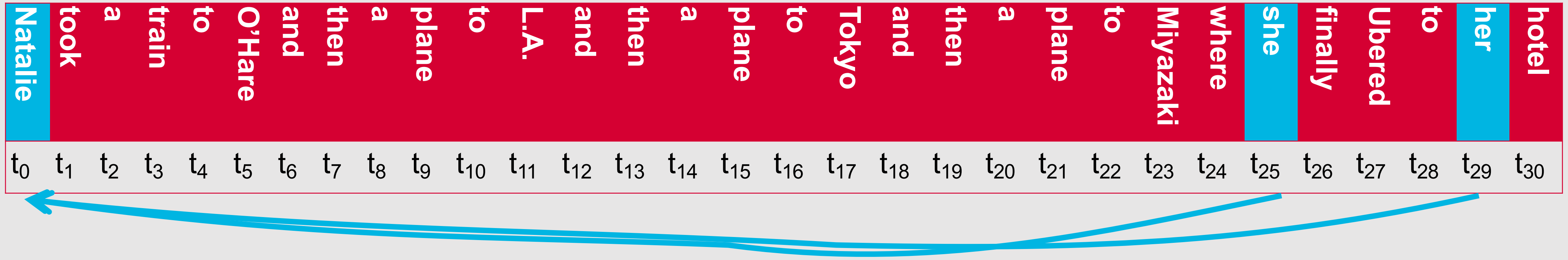


Sequence Classification with a Bidirectional RNN



Despite the advantages we've seen compared to feedforward neural networks, RNNs may still struggle with managing context.

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


Natalie
took
a
train
to
O'Hare
and
then
a
plane
to
L.A.
and
then
a
plane
to
Tokyo
and
then
a
plane
to
Miyazaki
where
she
finally
Ubered
to
her
hotel
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12} t_{13} t_{14} t_{15} t_{16} t_{17} t_{18} t_{19} t_{20} t_{21} t_{22} t_{23} t_{24} t_{25} t_{26} t_{27} t_{28} t_{29} t_{30}

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?

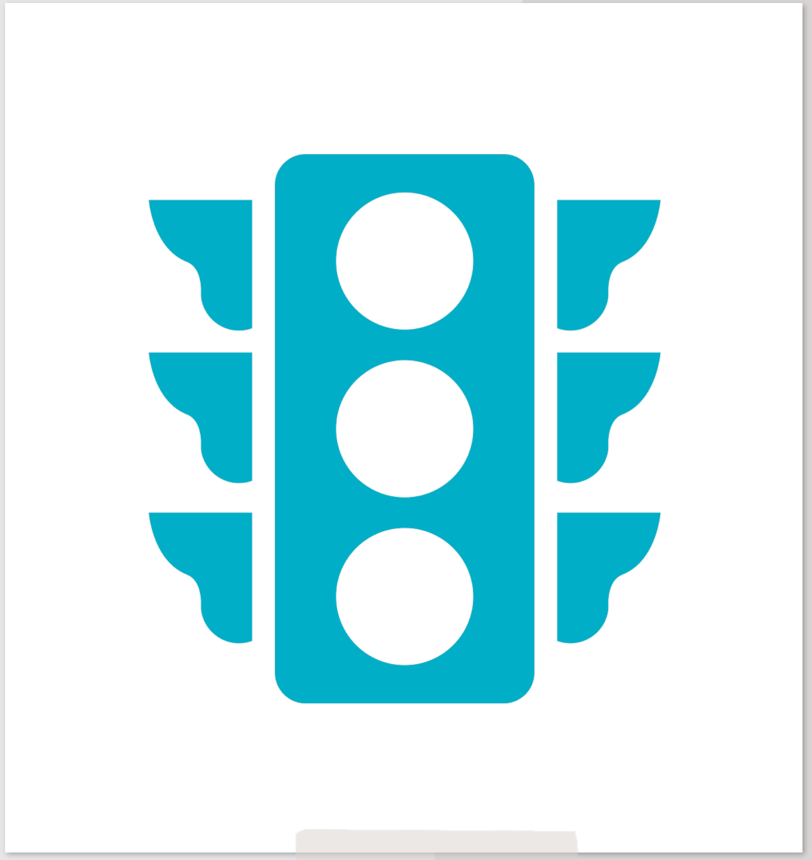
- Design more complex RNNs that learn to:
  - **Forget** information that is no longer needed
  - **Remember** information still required for future decisions

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**
- 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
  - $f_t = \sigma(\underbrace{U_f h_{t-1} + W_f x_t})$
  - $k_t = \underbrace{c_{t-1}} \odot f_t$

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

Context vector from the previous timestep

Add Gate

- Goal: Select the information to add to the current context
  - $g_t = \underbrace{\tanh(U_g h_{t-1} + W_g x_t)}$
  - $i_t = \sigma(\underbrace{U_i h_{t-1} + W_i x_t})$
  - $j_t = g_t \odot i_t$
  - $c_t = j_t + k_t$

Regular RNN computation

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

Add Gate

- Goal: Select the information to add to the current context
  - $g_t = \tanh(U_g h_{t-1} + W_g x_t)$
  - $i_t = \sigma(U_i h_{t-1} + W_i x_t)$
  - $j_t = \underbrace{g_t \odot i_t}_{\text{New information to be added}}$
  - $c_t = \underbrace{j_t + k_t}$

New information to be added

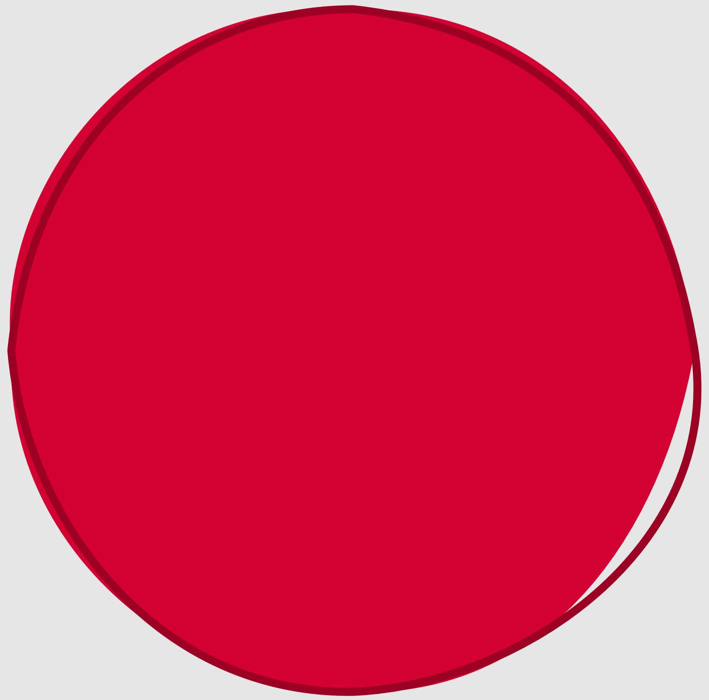
Updated context vector contains:
- New information to be added
- Existing information from context vector that was not removed by the forget gate

Output Gate

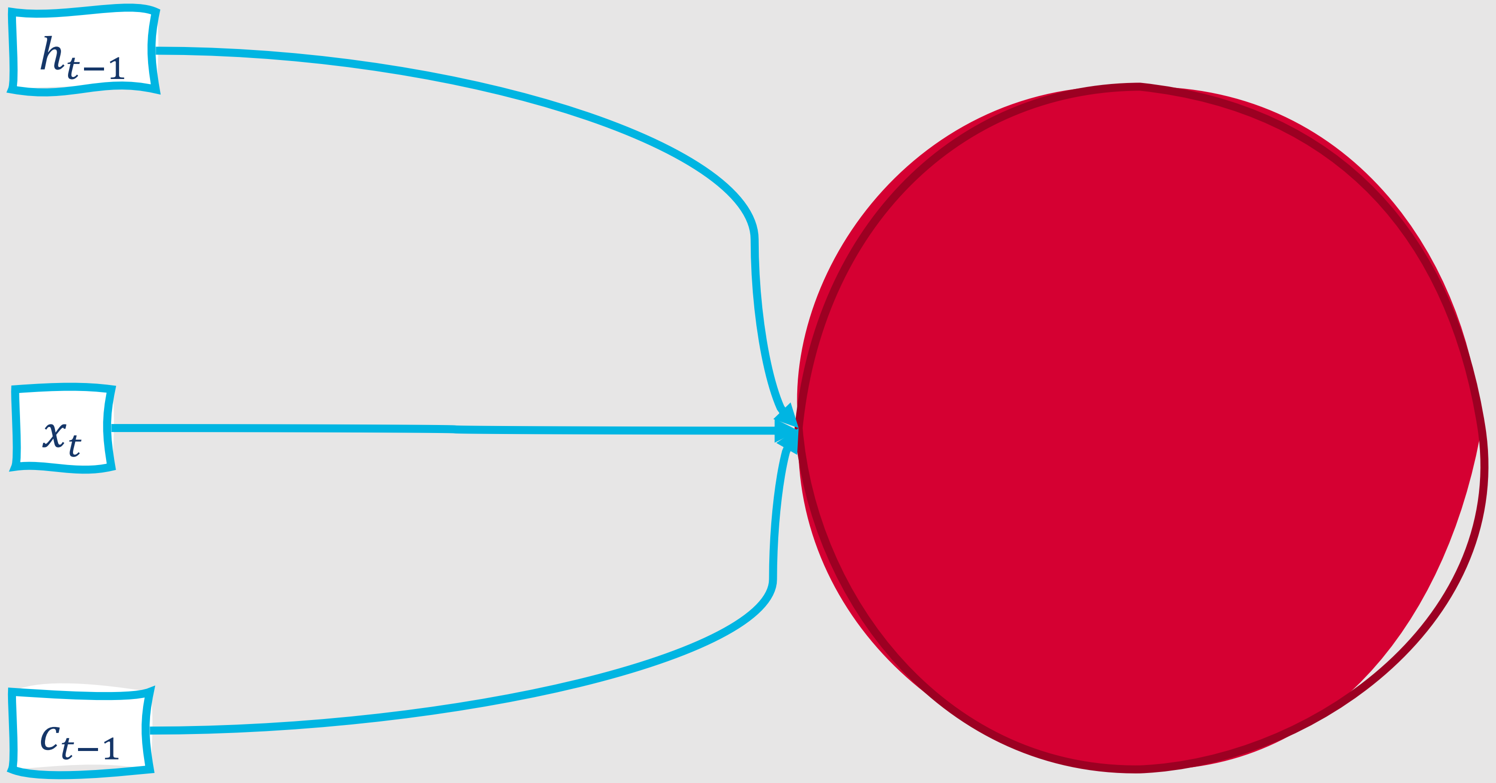
- Goal: Decide what information is required for the *current* hidden state
  - $o_t = \sigma(\underbrace{U_o h_{t-1} + W_o x_t})$
  - $h_t = \underbrace{o_t \odot \tanh(c_t)}_{\text{Updated hidden layer output}}$

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

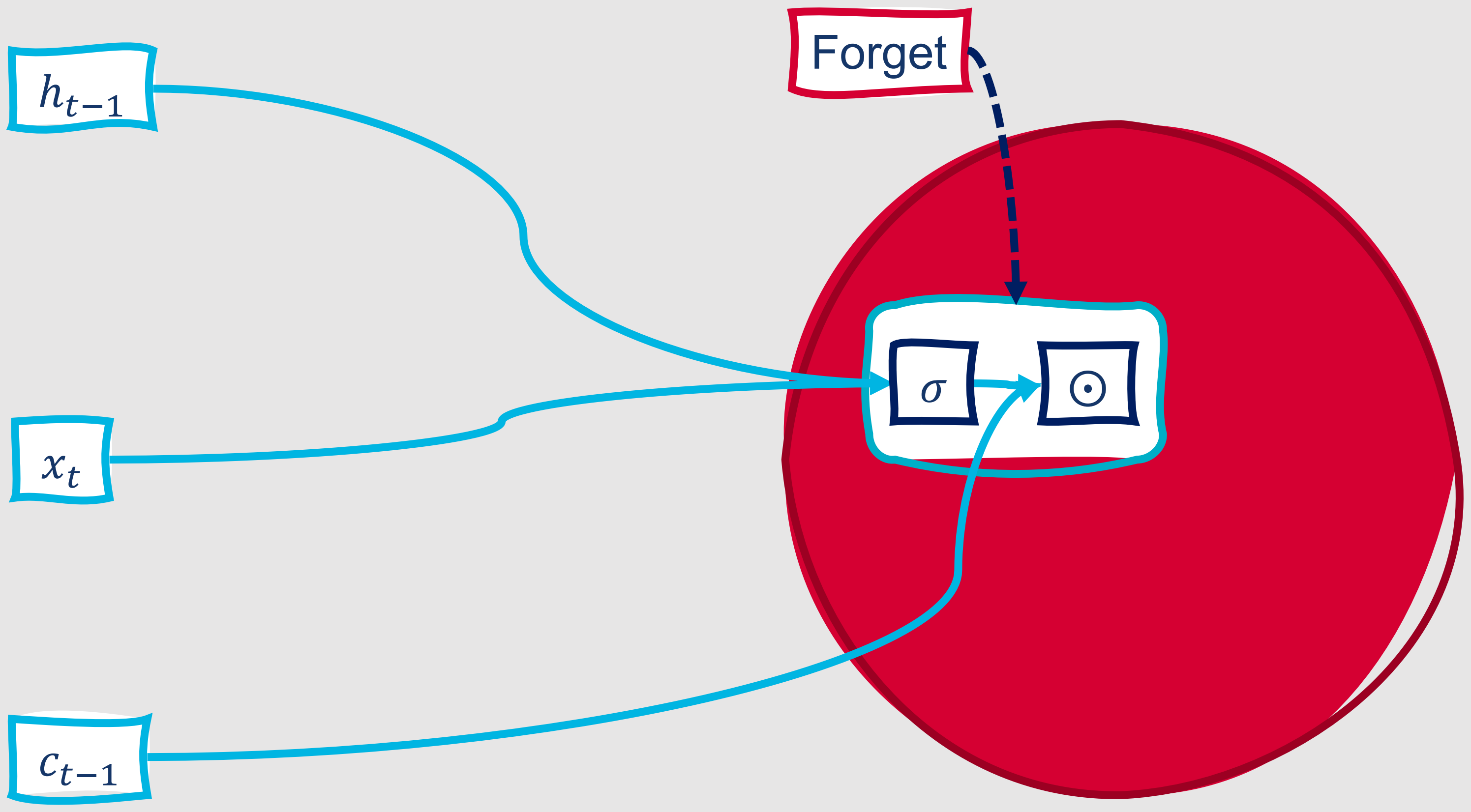
What does this process look like in a single LSTM unit?



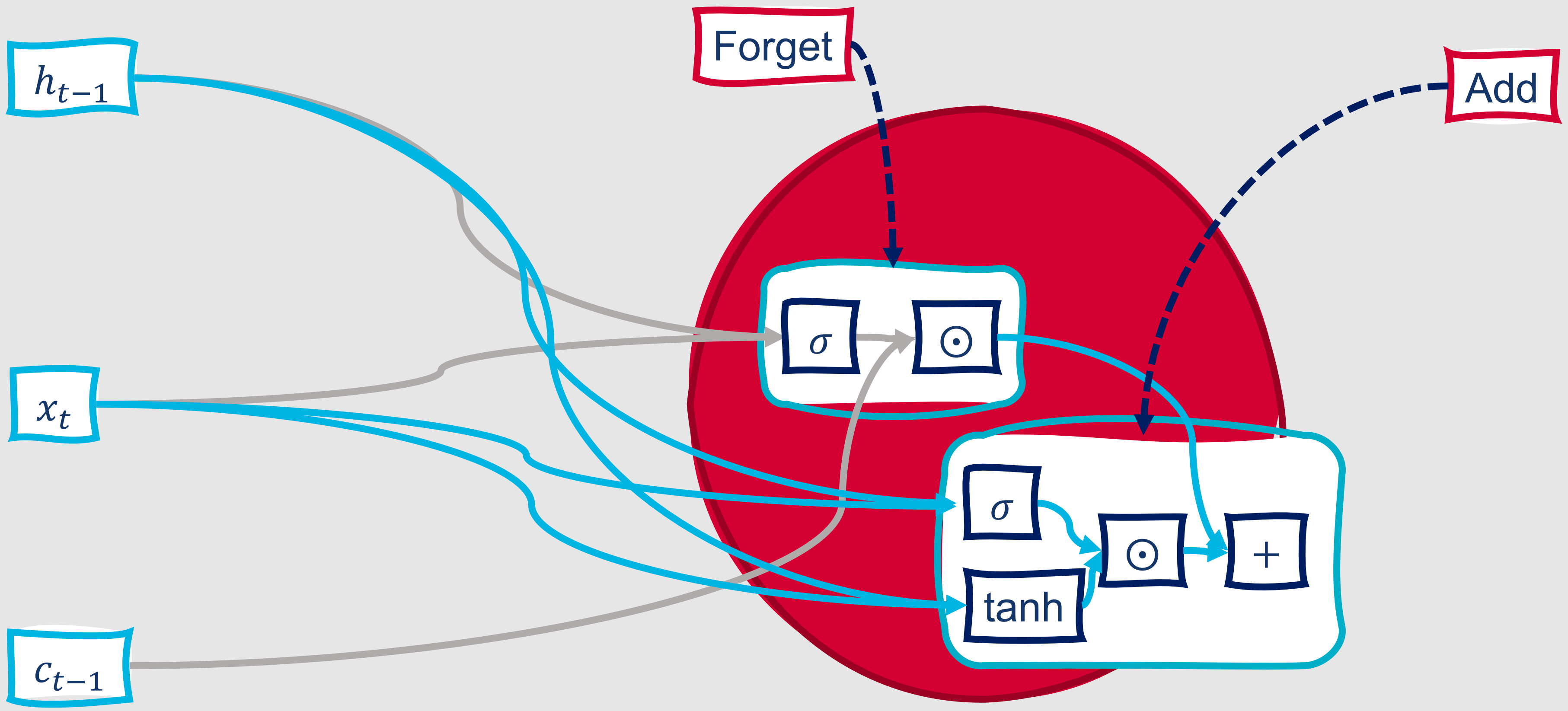
What does this process look like in a single LSTM unit?



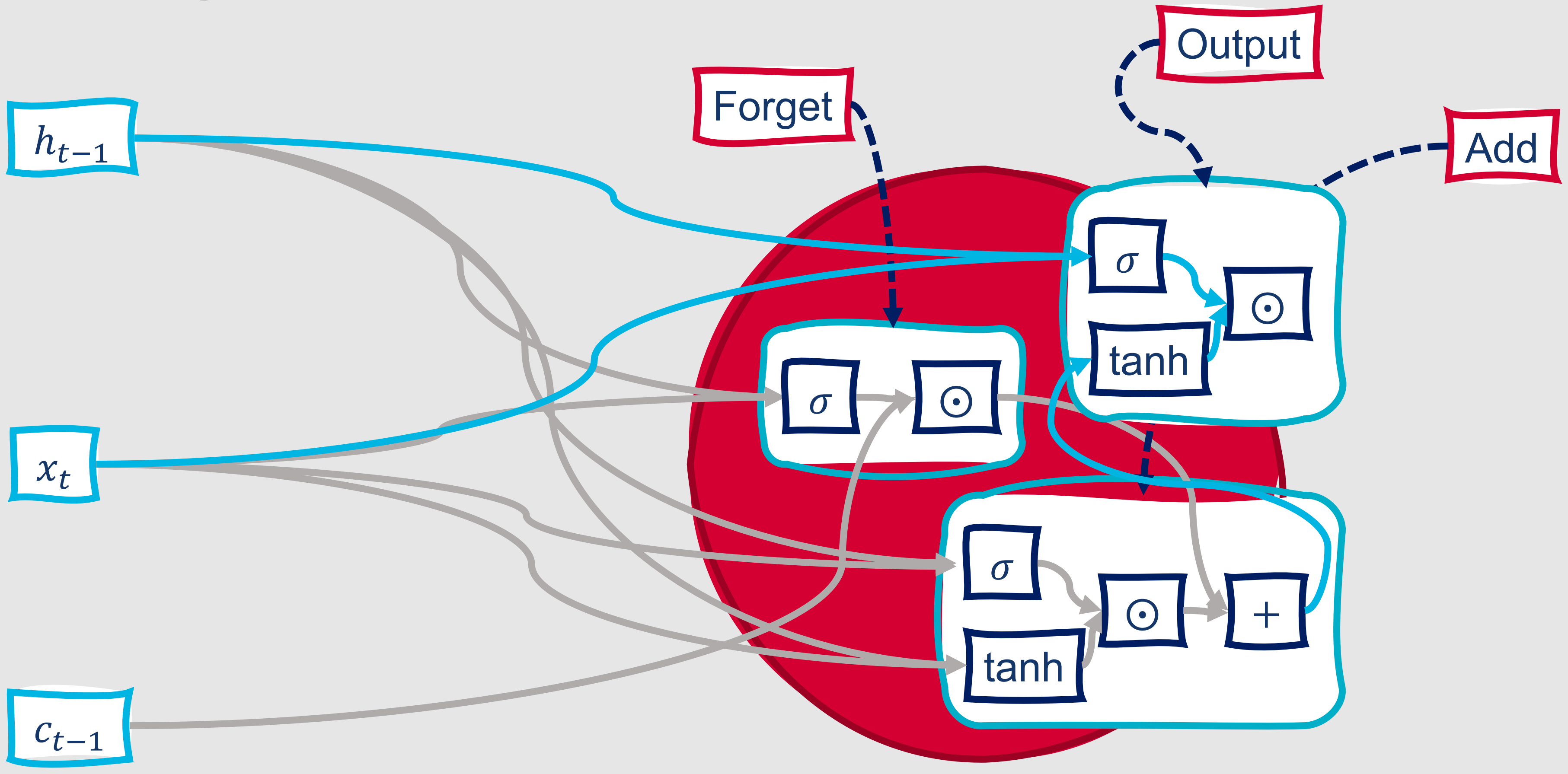
What does this process look like in a single LSTM unit?



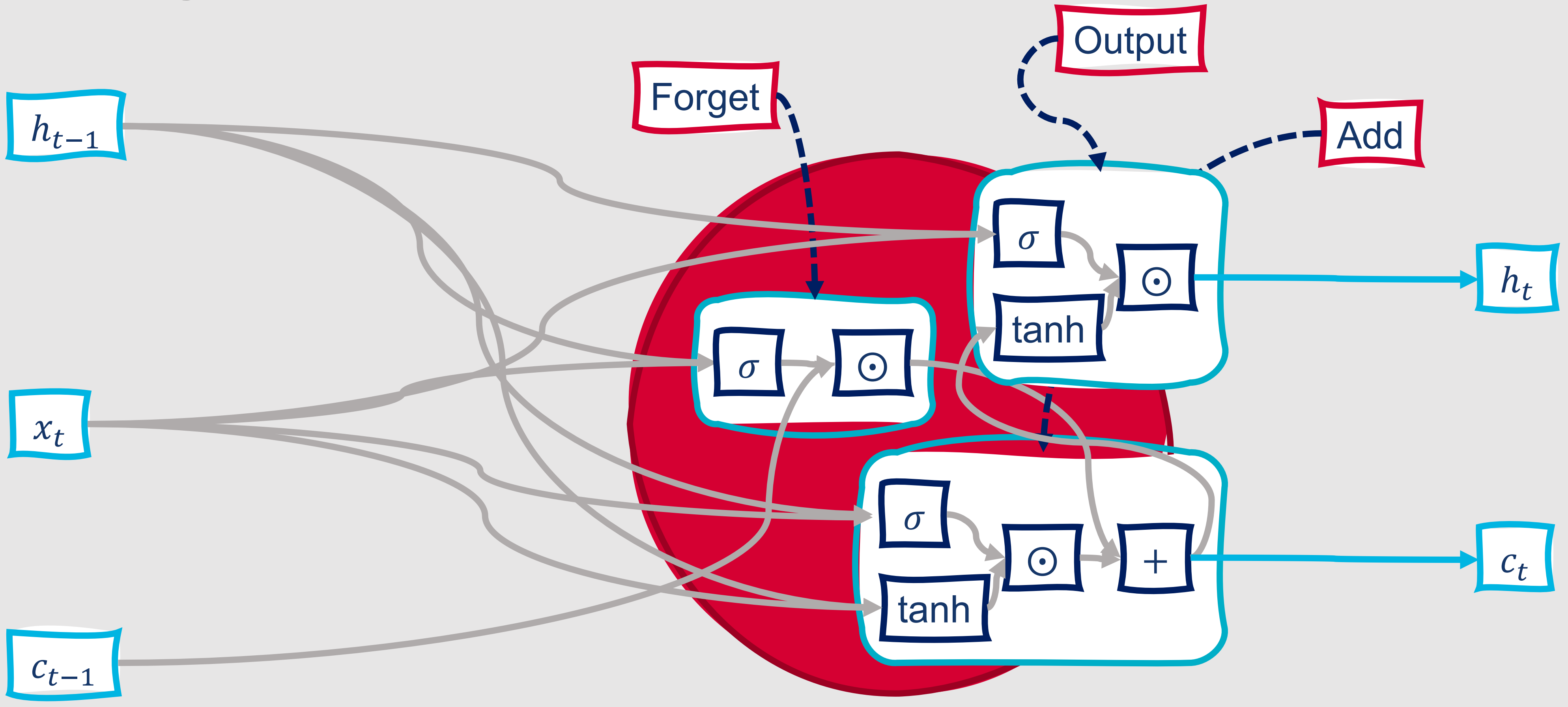
What does this process look like in a single LSTM unit?



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



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



Gated Recurrent Units (GRUs)

- Also manage the context that is passed through to the next timestep, but do so using 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
  - $r_t = \sigma(\underbrace{U_r h_{t-1} + W_r x_t})$
  - $\underbrace{\tilde{h}_t = \tanh(U(r_t \odot h_{t-1}) + W x_t)}_{\text{Intermediate representation for } h_t}$

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

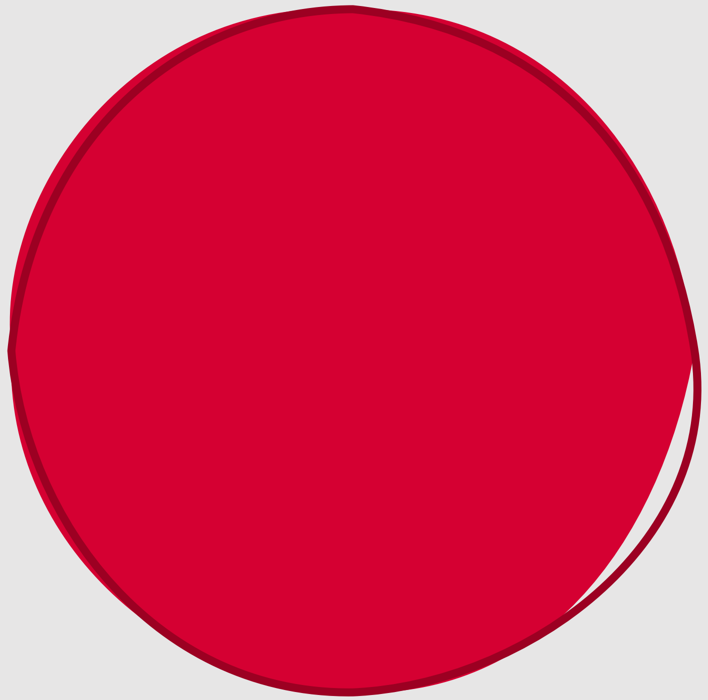
Update Gate

- Goal: Decide which aspects of the intermediate hidden state and which aspects of the previous hidden state need to be preserved for future use
  - $z_t = \sigma(\underbrace{U_z h_{t-1} + W_z x_t})$
  - $h_t = \underbrace{(1 - z_t)h_{t-1} + z_t \tilde{h}_t}$

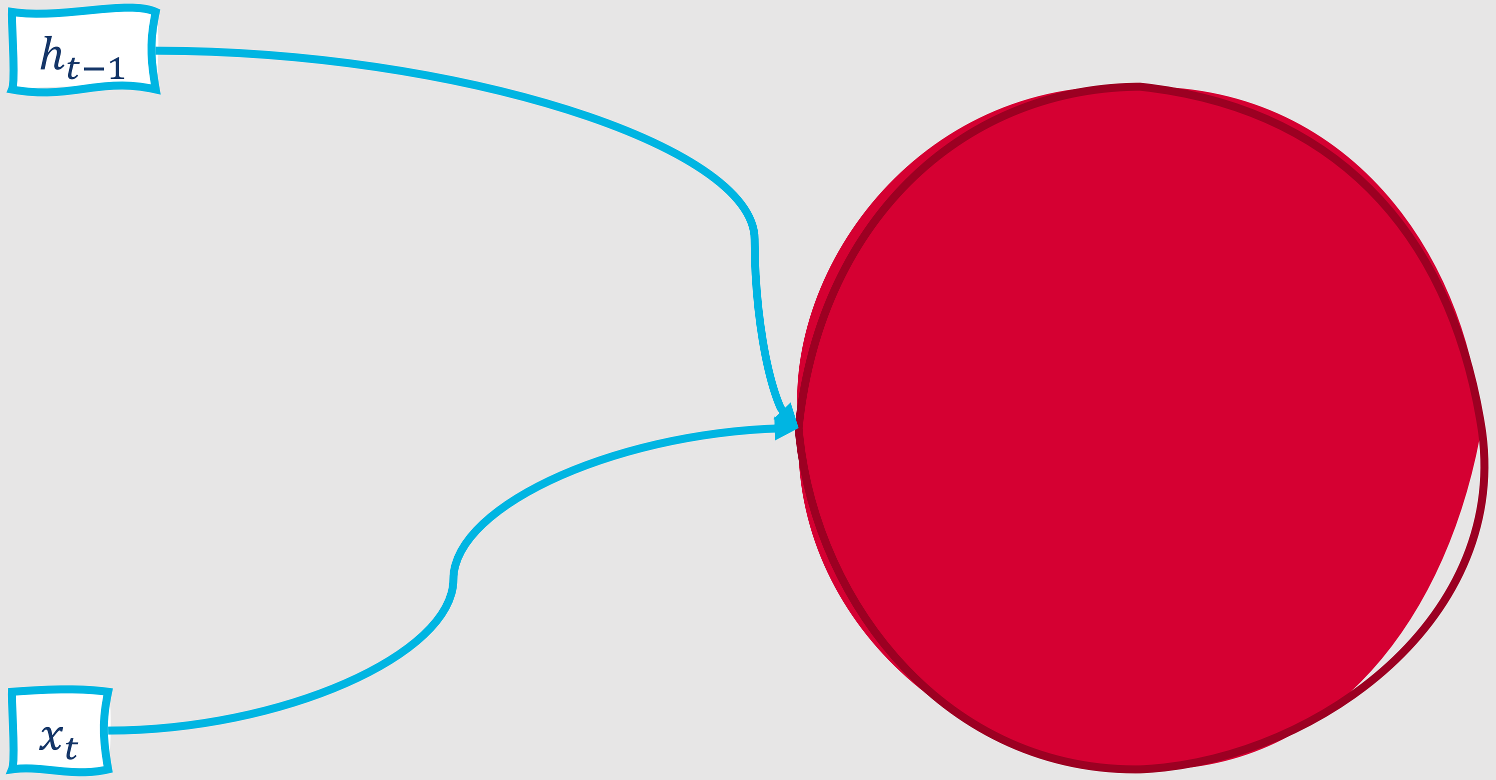
Weighted sum of:
- Hidden layer at the previous timestep
- Current input

Updated hidden layer output

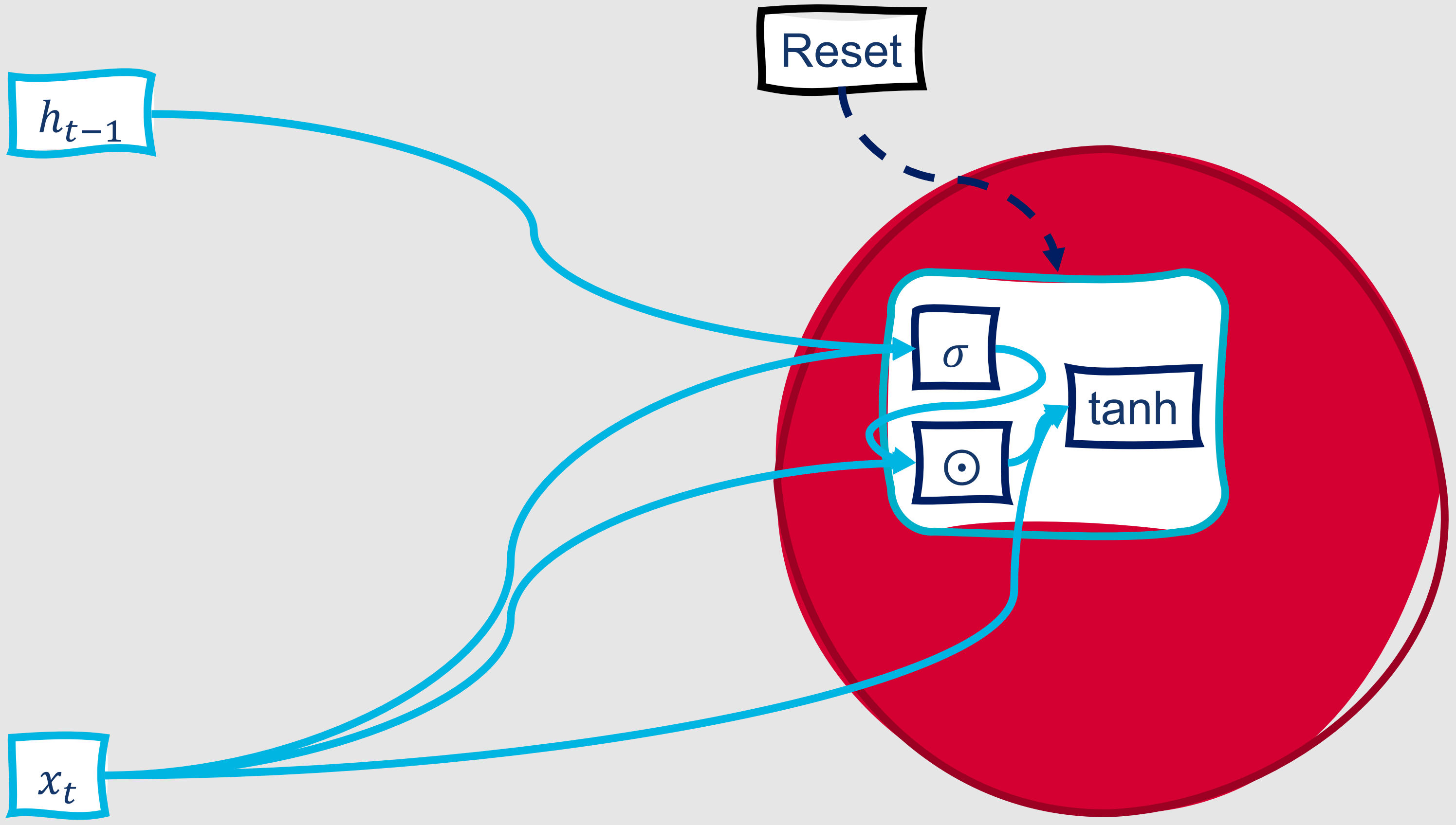
What does this process look like in a single GRU unit?



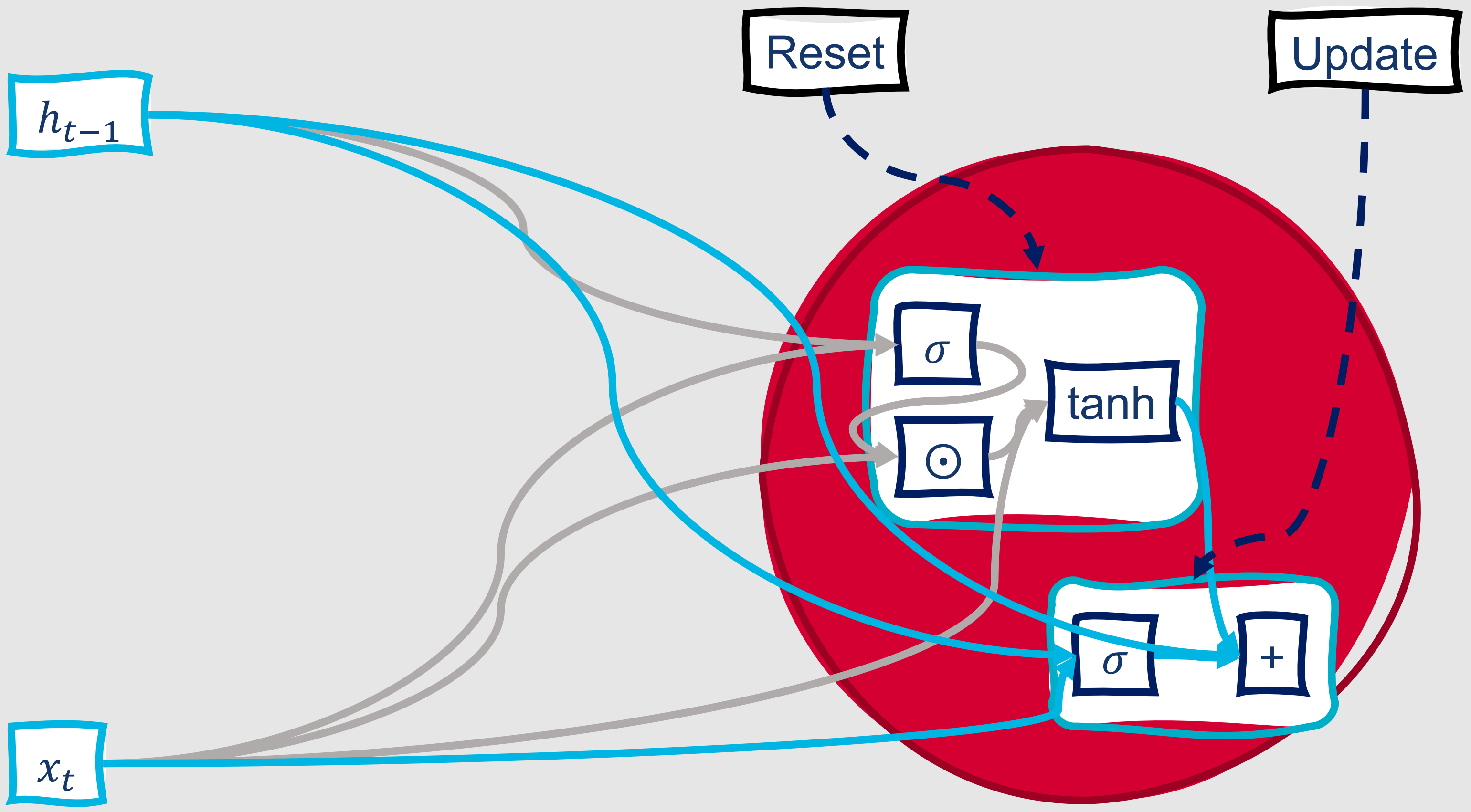
What does this process look like in a single GRU unit?



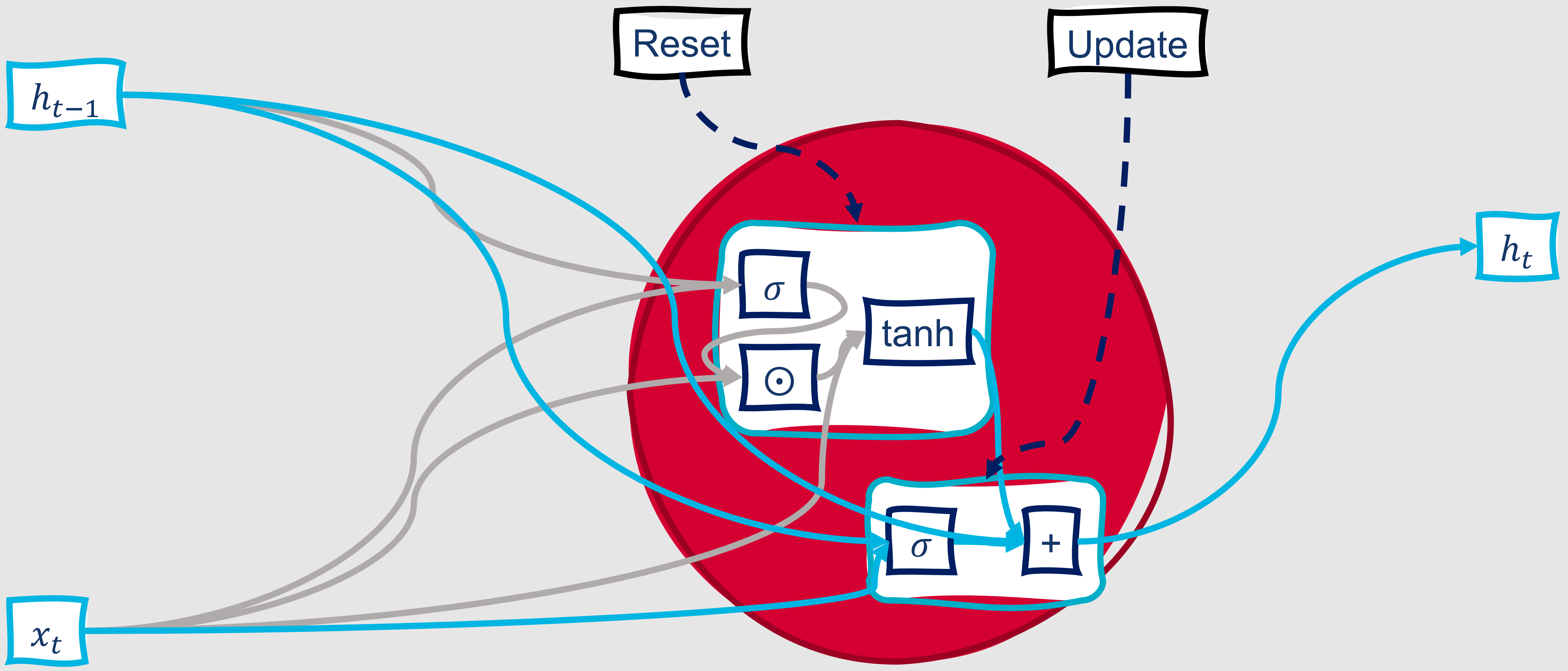
What does this process look like in a single GRU unit?



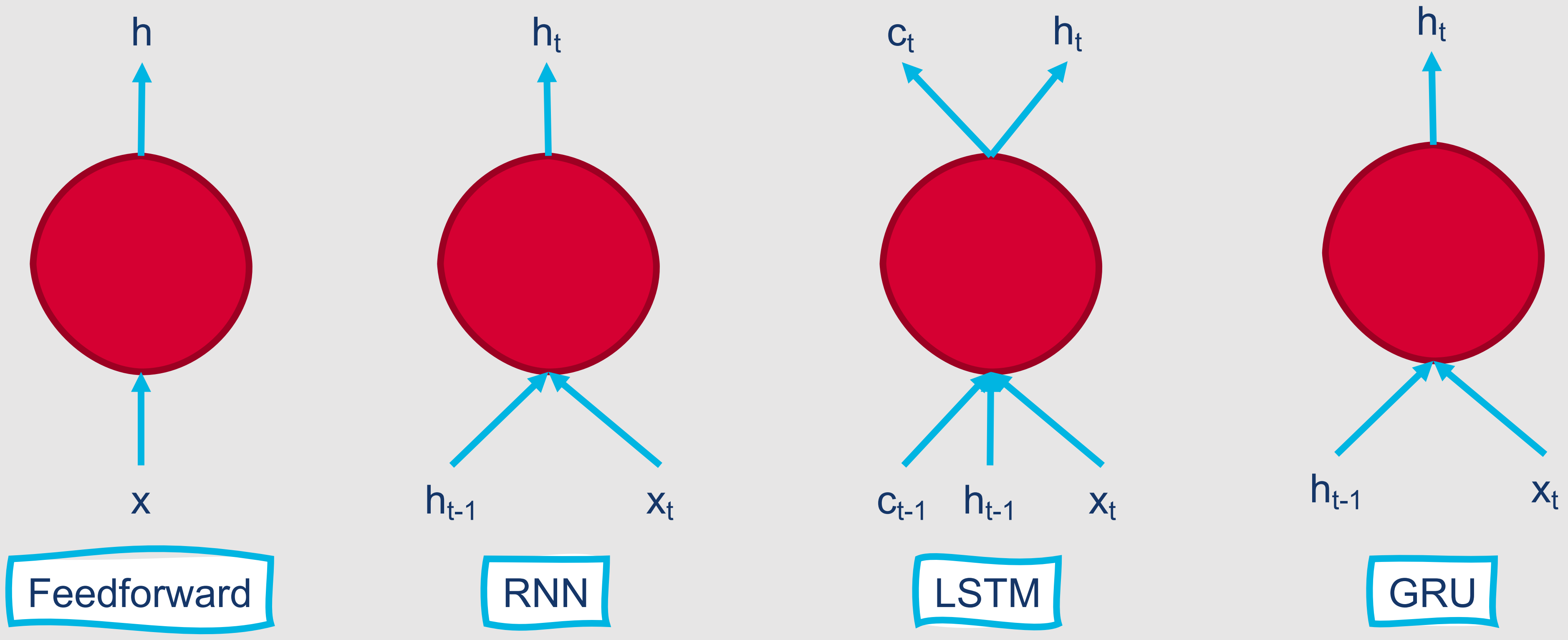
What does this process look like in a single GRU unit?



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

Summary: Recurrent Neural Networks

- **Recurrent neural networks** incorporate prior context into their design by leveraging weighted **temporal information** from the previous timestep
- RNNs can be **stacked** together, or they can be combined to form **bidirectional** models
- Two specialized RNN architectures designed to address weaknesses associated with long-range context are **long short-term memory networks** and **gated recurrent units**
- Both LSTMs and GRUs incorporate **gating mechanisms** to offer improved control over the flow of information between timesteps