Deep Learning for NLP

Natalie Parde

UIC CS 421

This Week's Topics

Tuesday

- Neural networks
- Computational units
- Combining layers of units

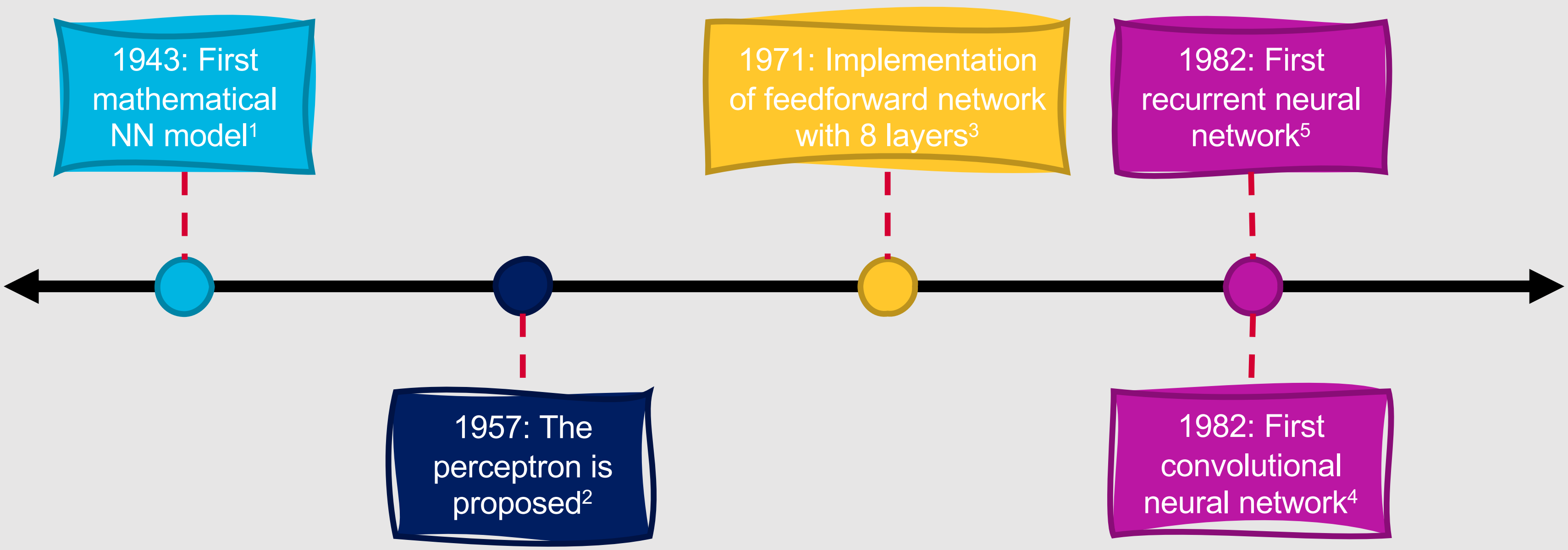
Thursday

- Backpropagation
- Neural language models
- Recurrent neural networks
- Other popular deep learning architectures

Now that we have more advanced word embeddings….

- We can incorporate these word embeddings in more sophisticated text classification models
- Extremely popular modern text classification model: **Neural network**
  - Classification models comprised of interconnected computing units, or **neurons**, (loosely!) mirroring the interconnected neurons in the human brain
- Neural networks are the force behind **deep learning**

Are neural networks new?



[1]McCulloch, W. S., and W. Pitts. "A logical calculus of the ideas immanent in nervous activity." *The bulletin of mathematical biophysics* 5.4 (1943): 115-133.

[2]Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory.

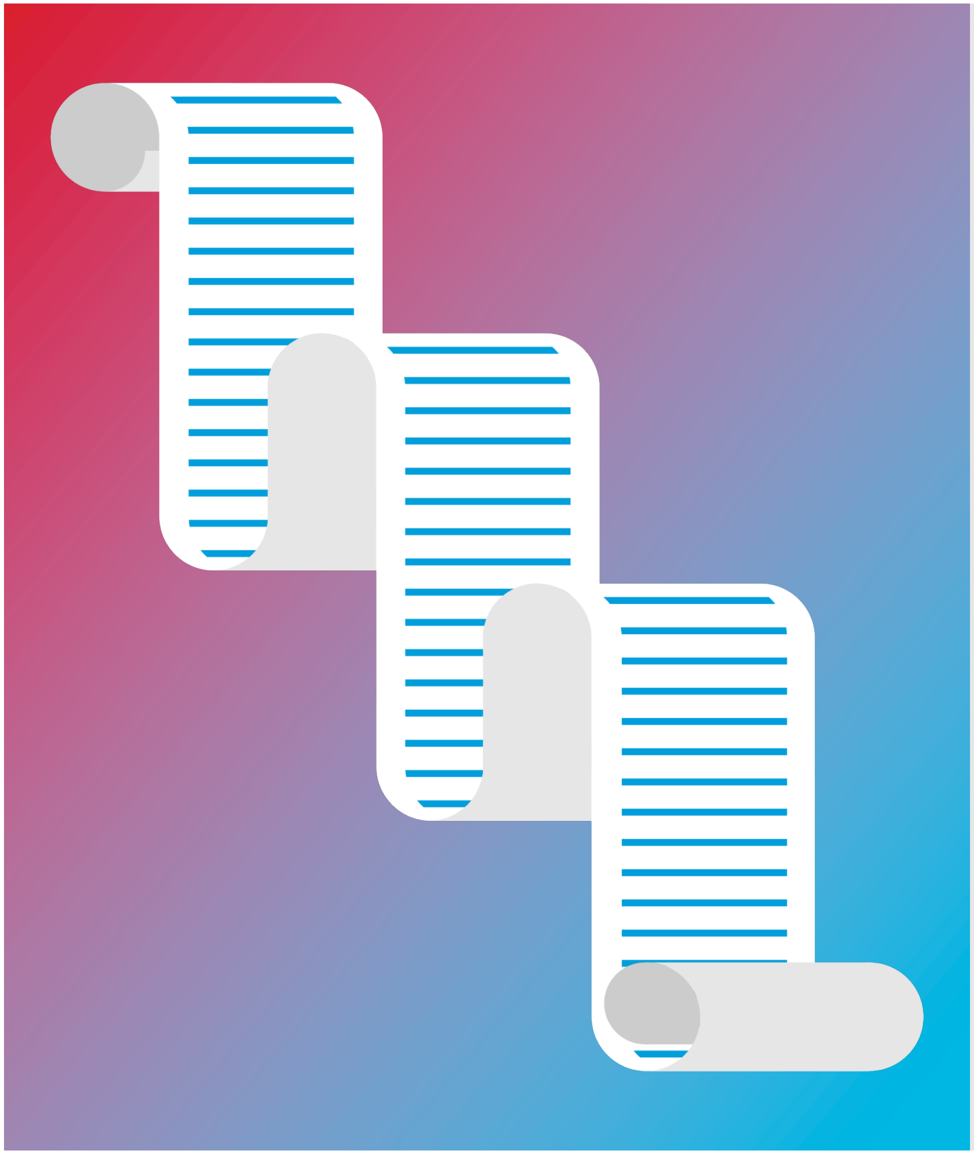
[3]Ivakhnenko, A. G. (1971). Polynomial theory of complex systems. *IEEE transactions on Systems, Man, and Cybernetics*, (4), 364-378.

[4]Fukushima, K., & Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets* (pp. 267-285). Springer, Berlin, Heidelberg.

[5]Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, *79*(8), 2554-2558.

Why haven't they been a big deal until recently then?

- Data
- Computing power

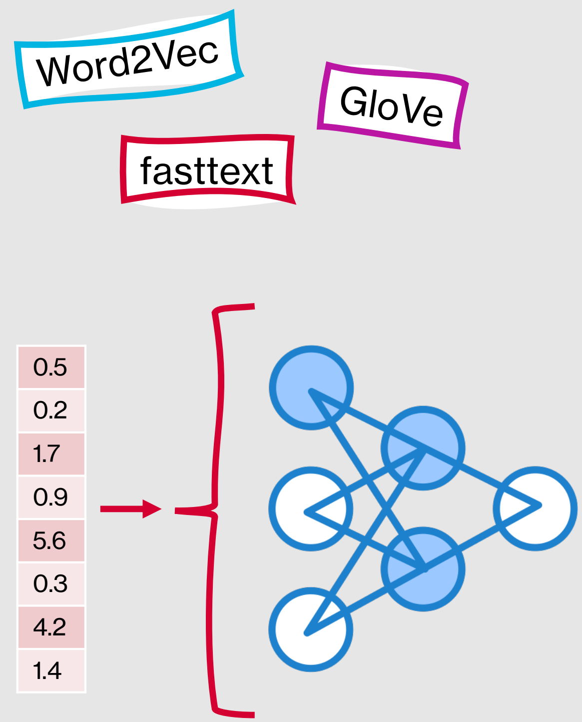


There are many types of neural networks!

- Feedforward neural networks
- Recurrent neural networks
- Convolutional neural networks
- Transformers
- ….

Common Themes across Deep Learning Approaches

- Input is typically a **dense vector representation**
  - In most cases, the dimensions within this representation do not correspond to specific, known attributes

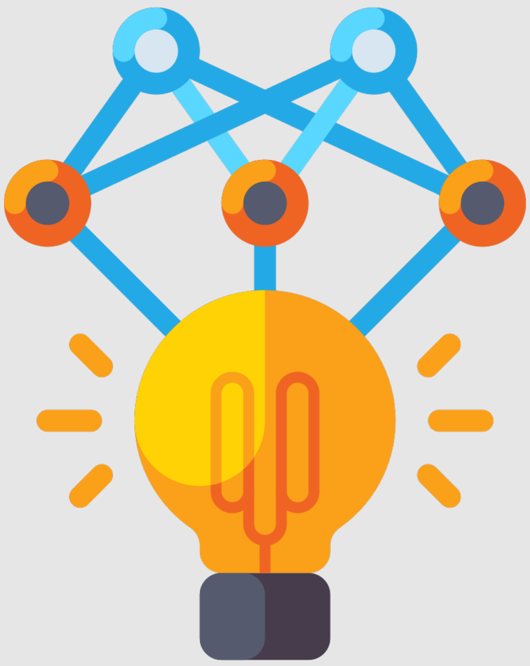


Common Themes across Deep Learning Approaches

- Input is typically a dense vector representation
  - In most cases, the dimensions within this representation do not correspond to specific, known attributes
- Structure of the deep learning model is determined at least partially by a **hyperparameter tuning process**
  - Many experiments will be run using different hyperparameter combinations to determine what leads to the best performance on the validation data

Common Themes across Deep Learning Approaches

- Input is typically a dense vector representation
  - In most cases, the dimensions within this representation do not correspond to specific, known attributes
- Structure of the deep learning model is determined at least partially by a hyperparameter tuning process
  - Many experiments will be run using different hyperparameter combinations to determine what leads to the best performance on the validation data
- Output is **task-dependent**
  - Can be a class label, a number, or a string of generated text



Common Themes across Deep Learning Approaches

- Input is typically a dense vector representation
  - In most cases, the dimensions within this representation do not correspond to specific, known attributes
- Structure of the deep learning model is determined at least partially by a hyperparameter tuning process
  - Many experiments will be run using different hyperparameter combinations to determine what leads to the best performance on the validation data
- Output is task-dependent
  - Can be a class label, a number, or a string of generated text
- Training can be performed **end-to-end**
  - The model is trained to predict the target output directly, rather than through pipelined components

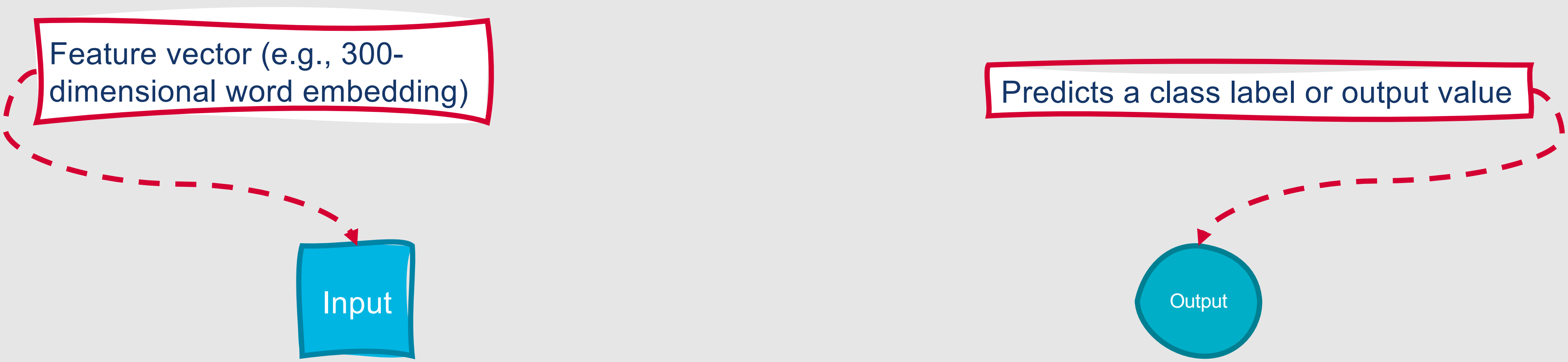
Despite these common themes, deep learning models are implemented in many different ways!

- They may vary in how they:
  - Handle prior context
  - Draw inferences from the data
  - Pass data between layers
- These variations make different kinds of deep learning models work better for different tasks

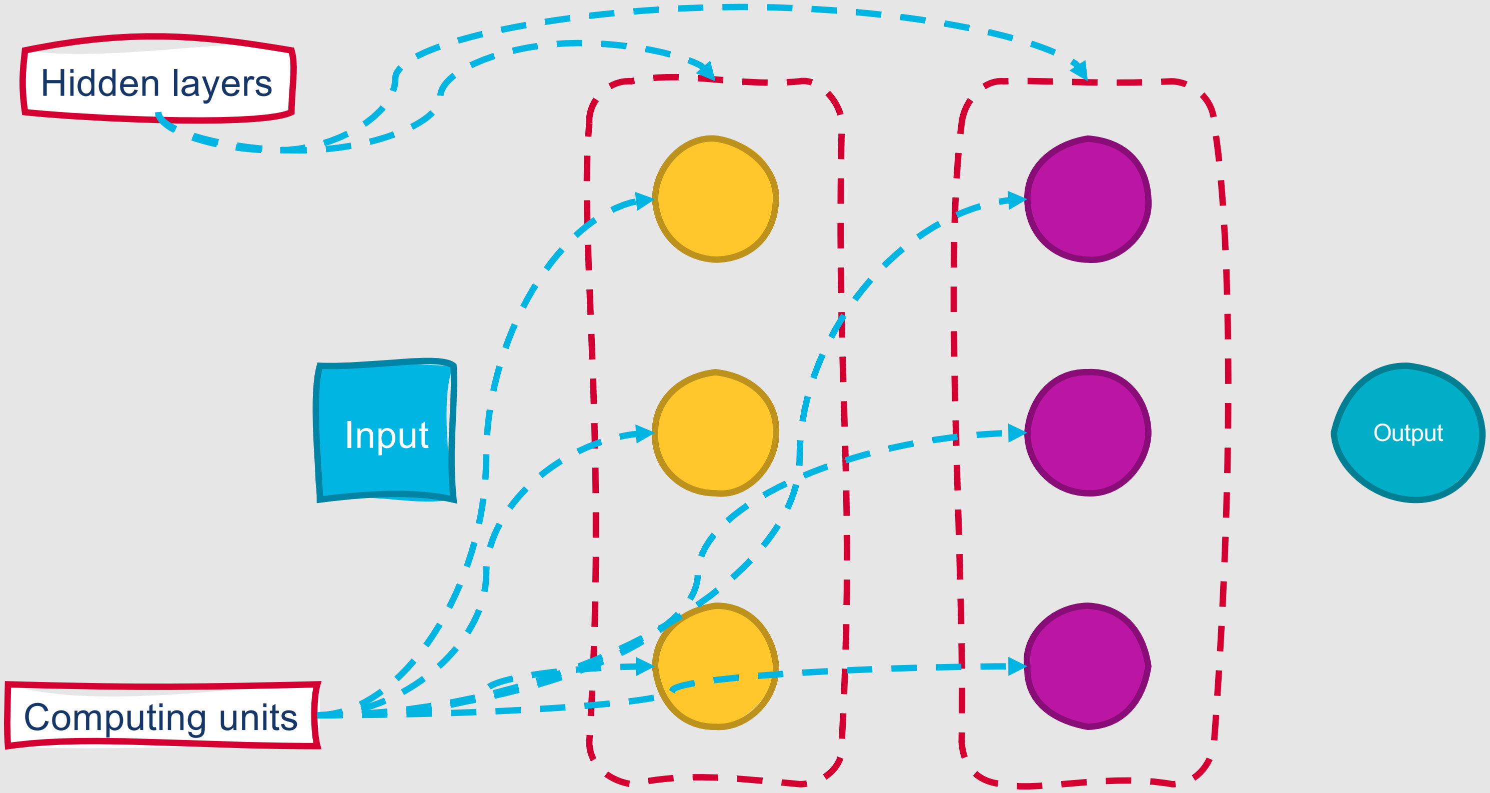
Feedforward Neural Networks

- 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

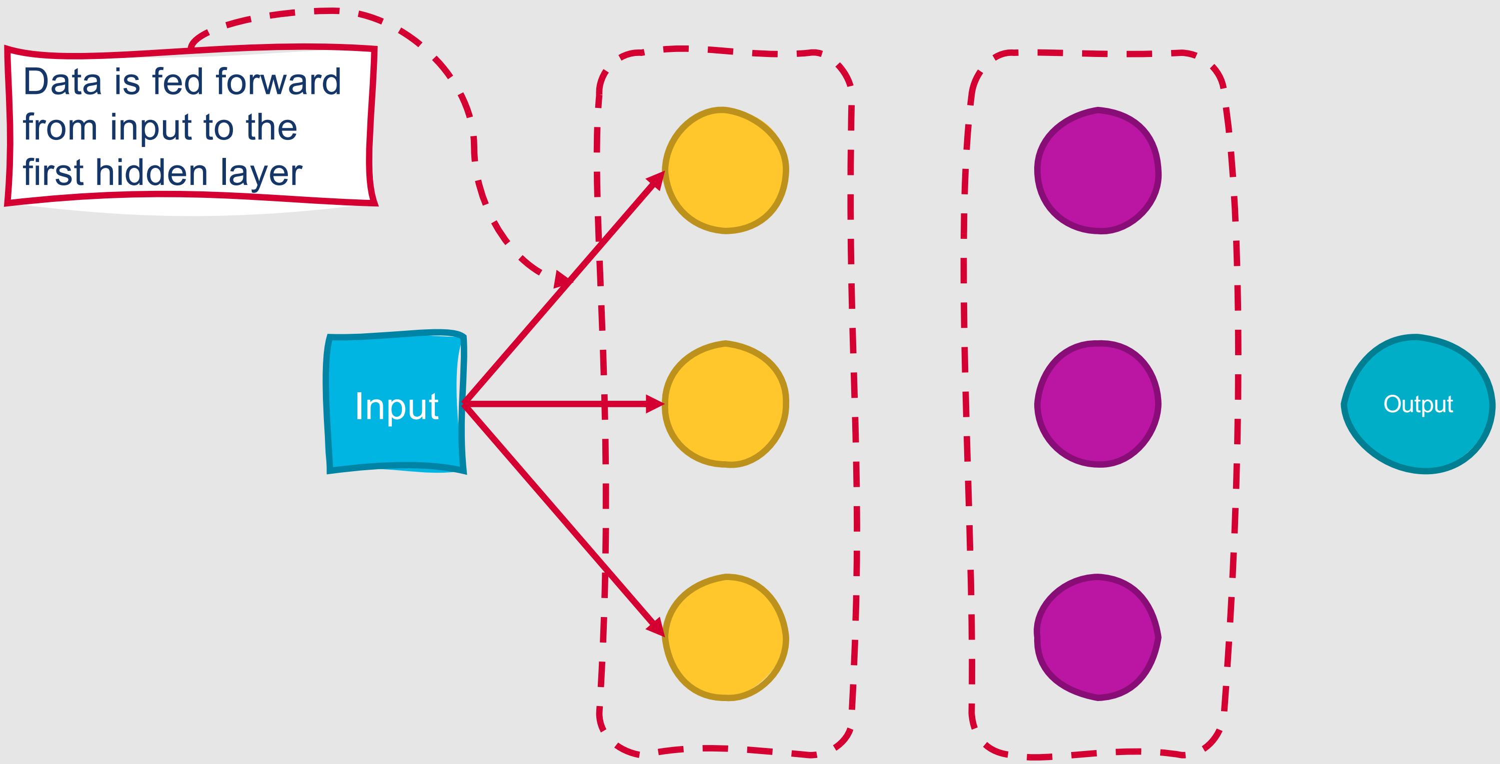
Feedforward Neural Networks



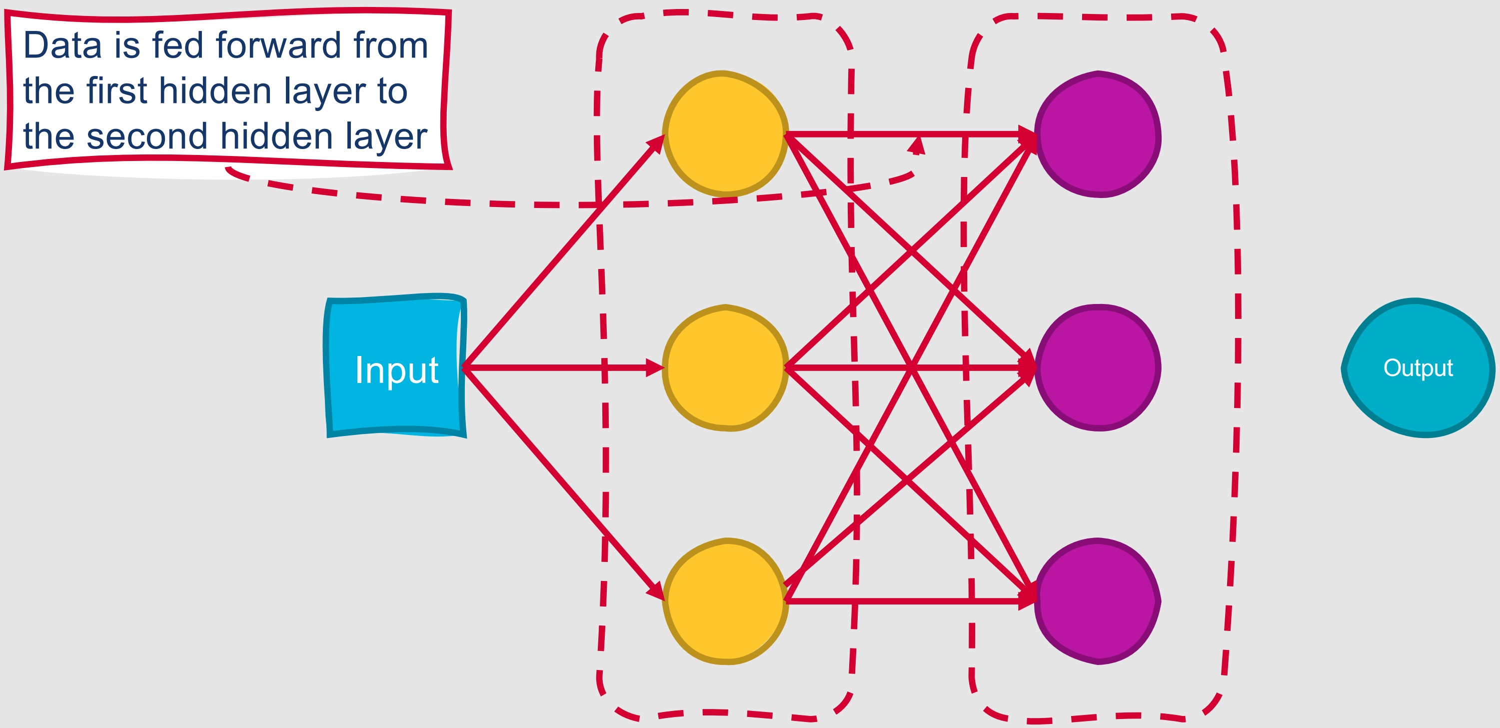
Feedforward Neural Networks



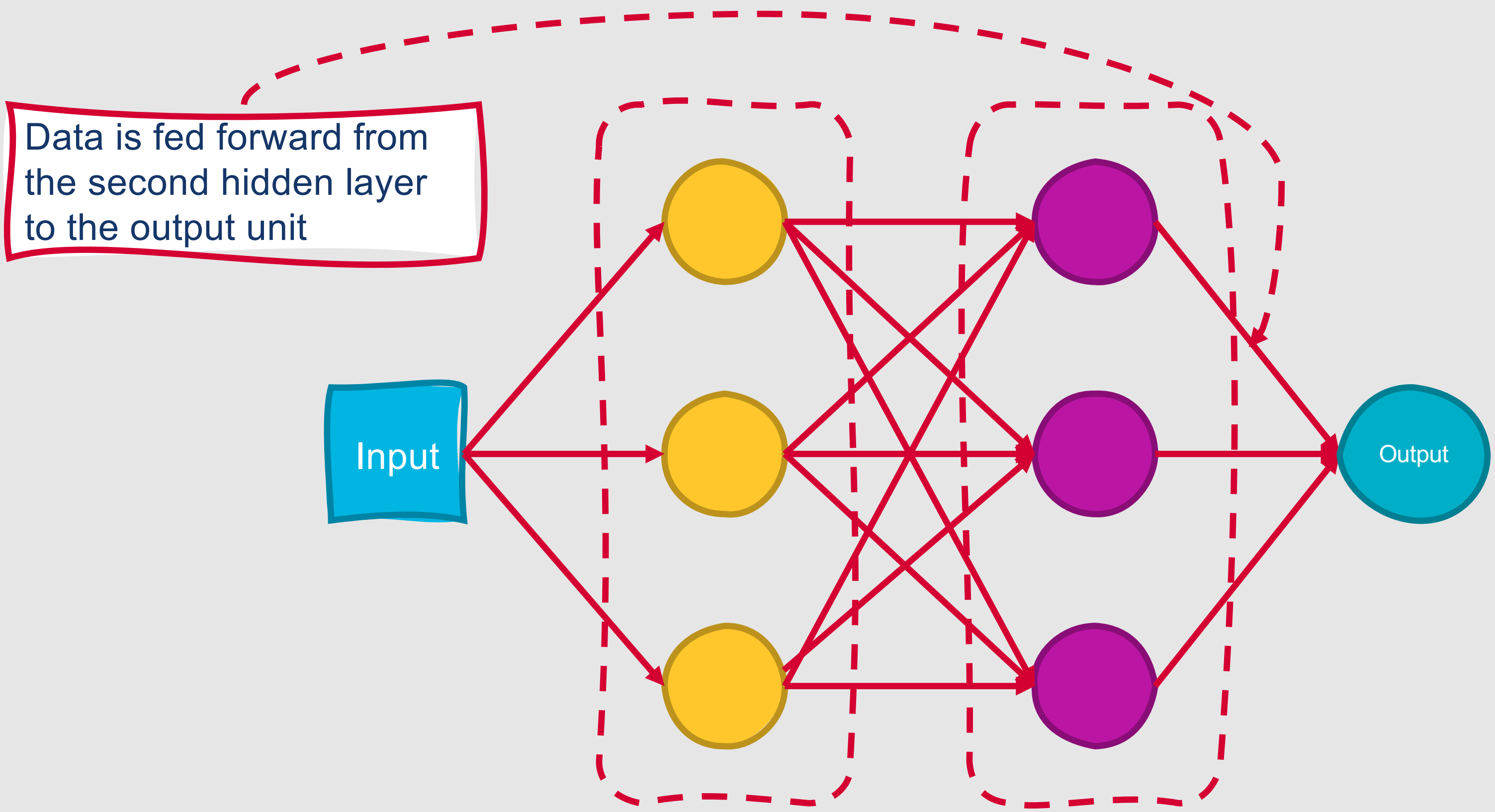
Feedforward Neural Networks



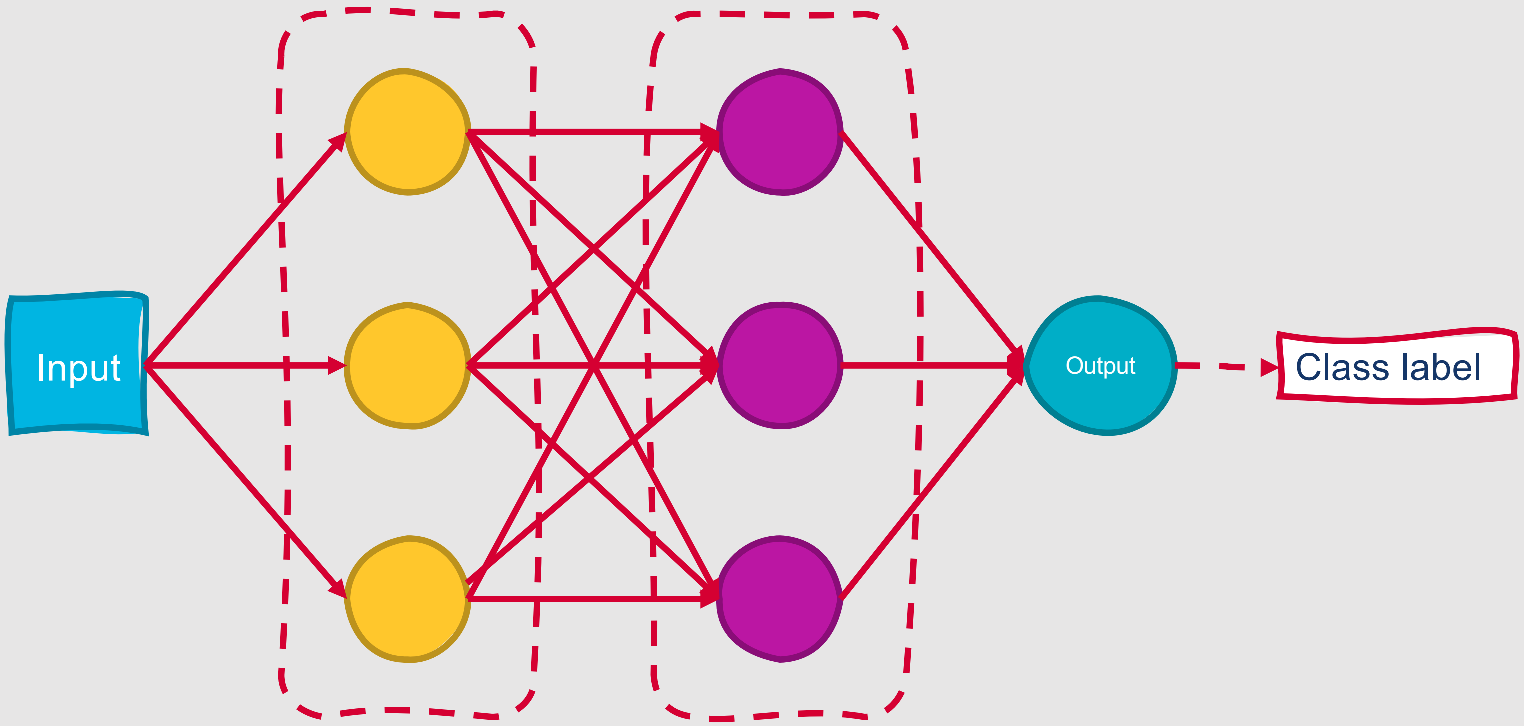
Feedforward Neural Networks



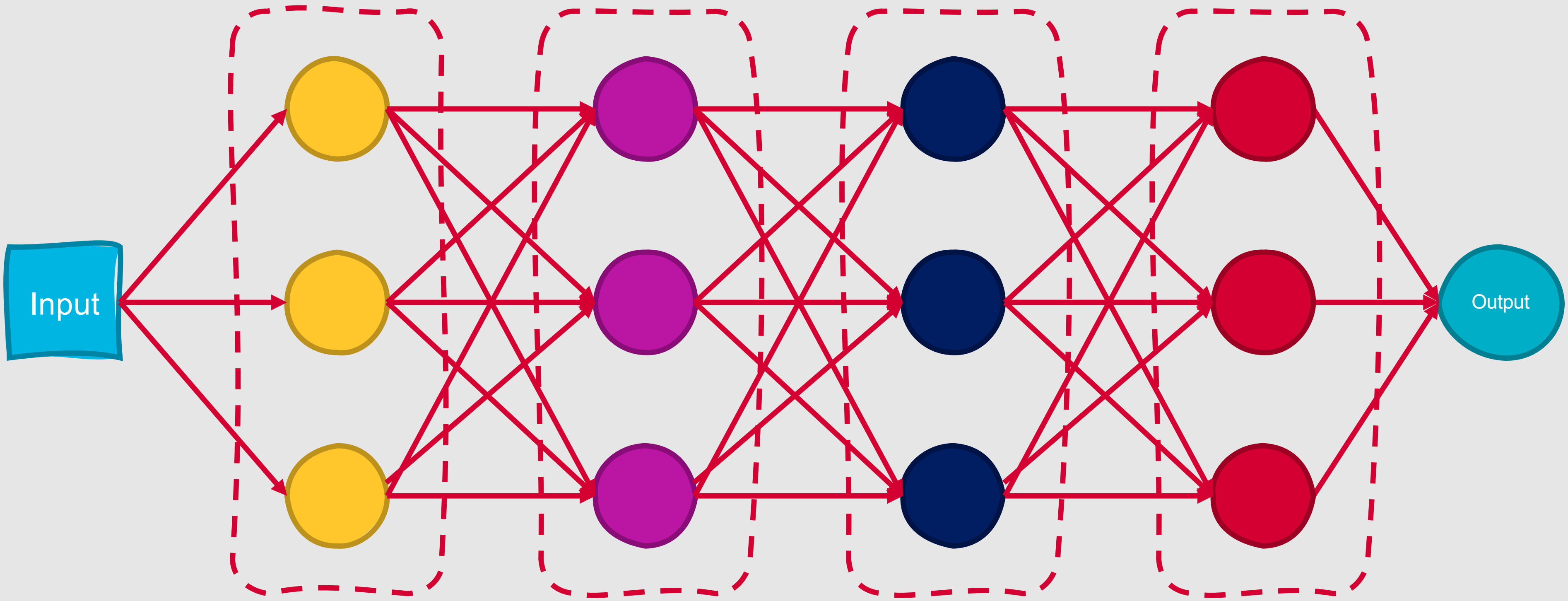
Feedforward Neural Networks



Feedforward Neural Networks

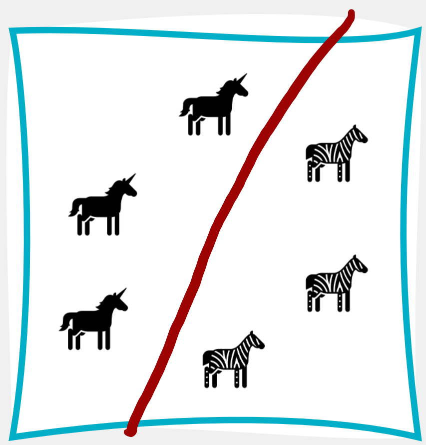


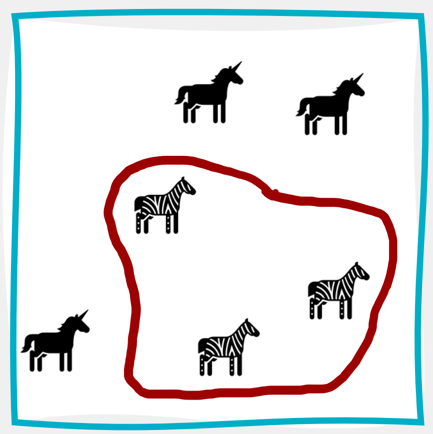
Any neural network architecture with hidden layers can be referred to as "deep learning," but this term often refers to networks with multiple hidden layers.



Neural networks tend to be more powerful than feature-based classifiers.

- Classification algorithms like naïve Bayes and logistic regression assume that data is **linearly separable**
- In contrast, neural networks learn **nonlinear** ways to separate the data





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

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

In general, deeper network → more data needed

Neural nets tend to work very well for large-scale problems, but not as well for small-scale problems

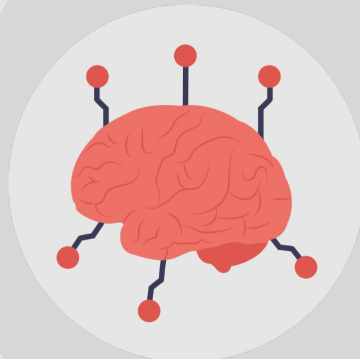
This Week's Topics

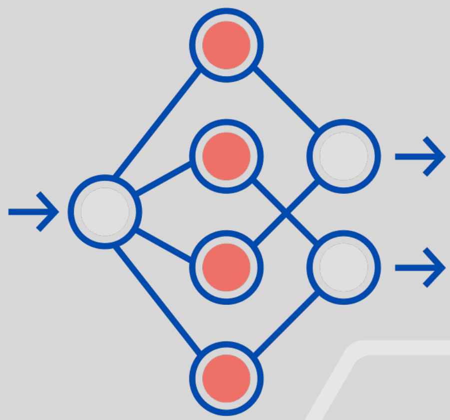
Tuesday

- Neural networks
- Computational units
- Combining layers of units

Thursday

- Backpropagation
- Neural language models
- Recurrent neural networks
- Other popular deep learning architectures

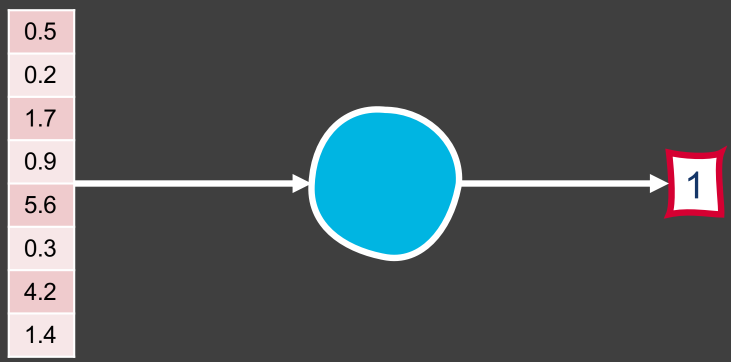




How do you build a neural network?

Building Blocks for Neural Networks

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

Sound familiar?

- This is exactly the same sort of weighted sum of inputs that we needed to find with logistic regression!
- Recall that we can also represent the weighted sum z using vector notation:
  - $\mathbf{z} = \mathbf{w} \cdot \mathbf{x} + b$

Computational Units

- Neural networks apply nonlinear functions referred to as **activations** to the weighted sum of inputs
- 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

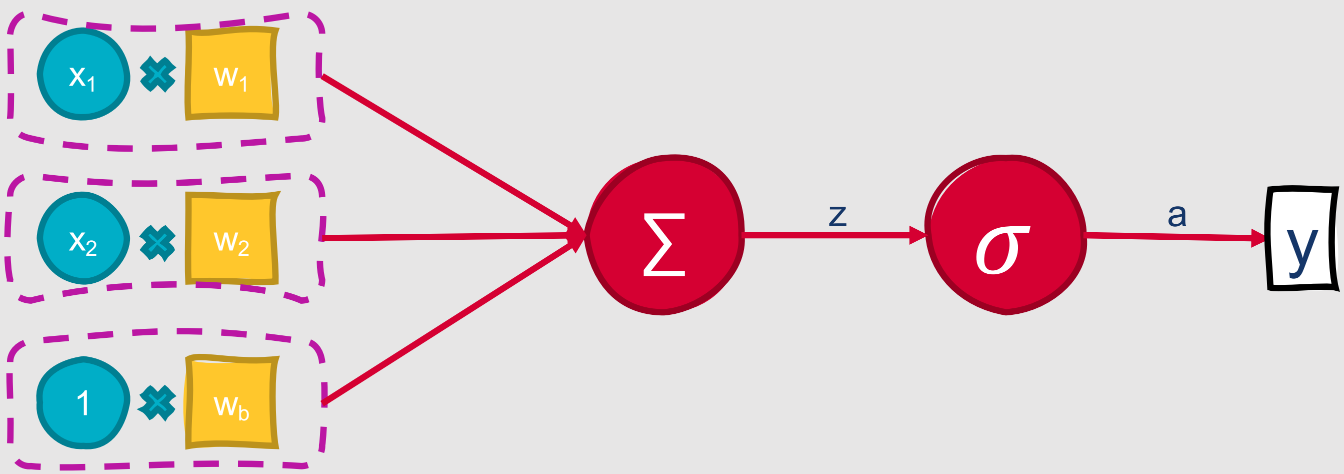
rectified linear unit (relu)

hyperbolic tangent (tanh)

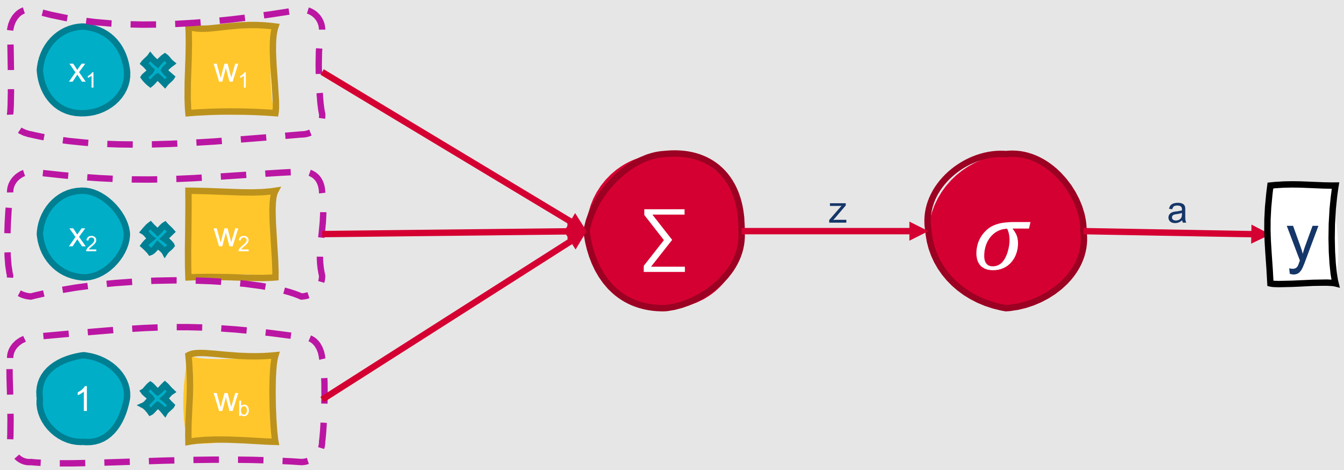
sigmoid

Exact same sigmoid function used with logistic regression

Computational Unit with Sigmoid Activation



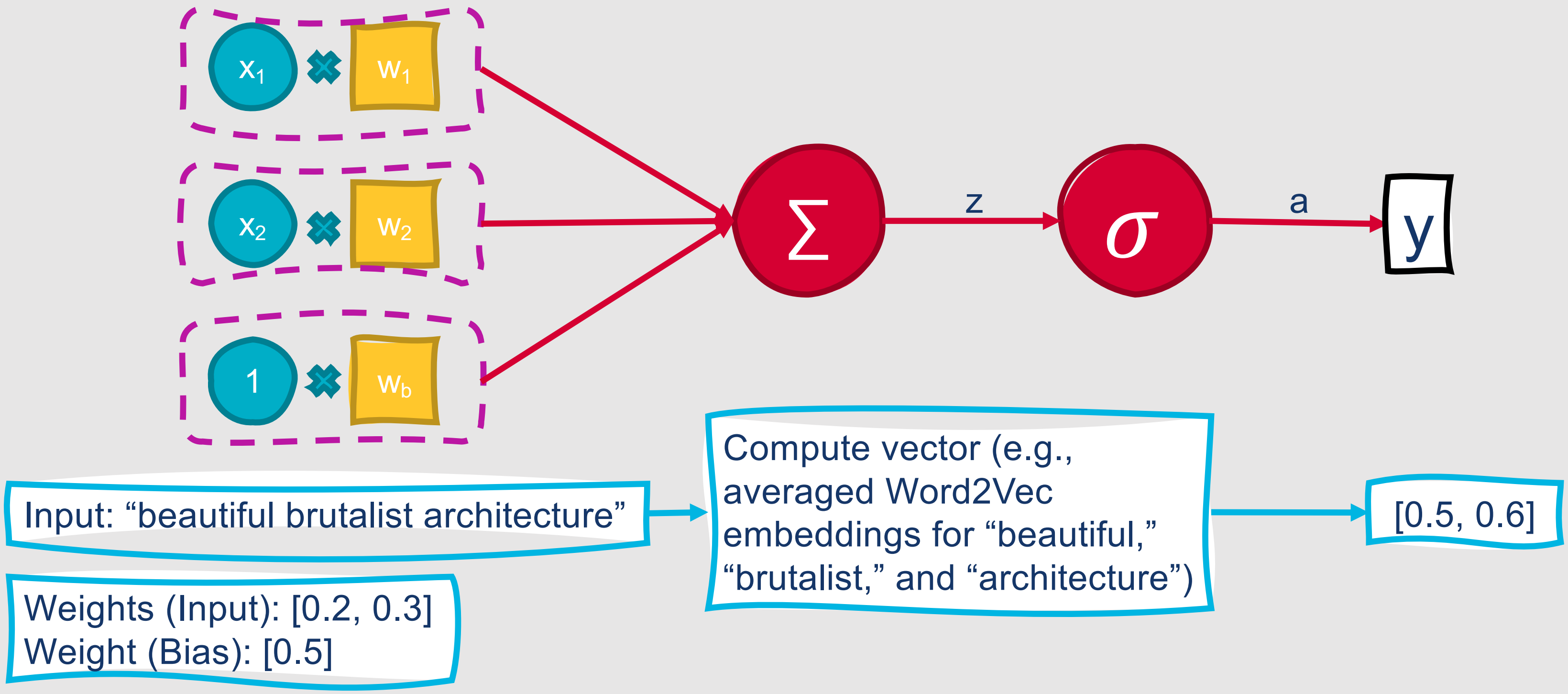
Example: Computational Unit with Sigmoid Activation



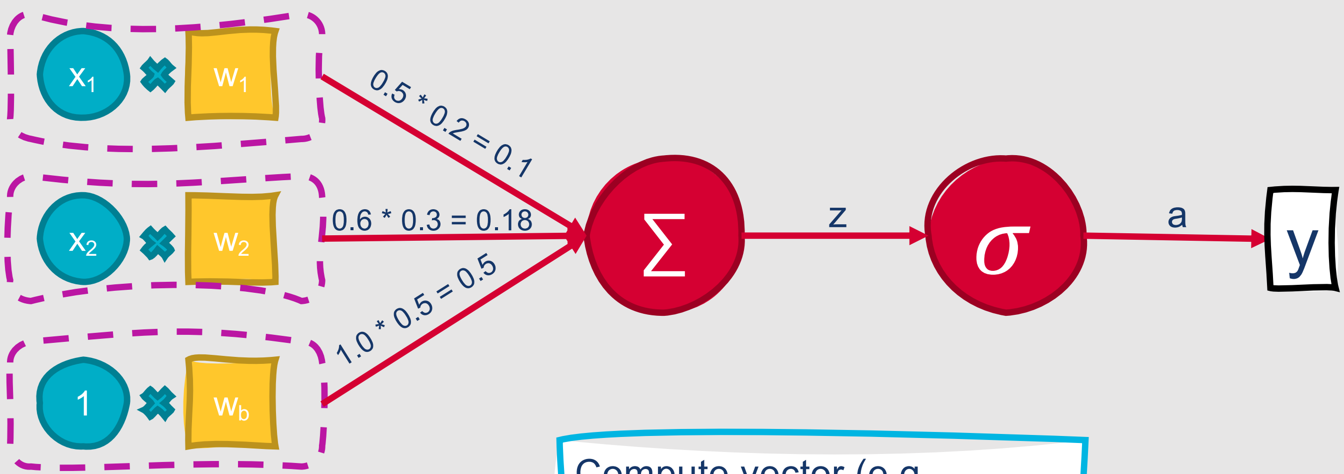
Input: "beautiful brutalist architecture"

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

Example: Computational Unit with Sigmoid Activation



Example: Computational Unit with Sigmoid Activation



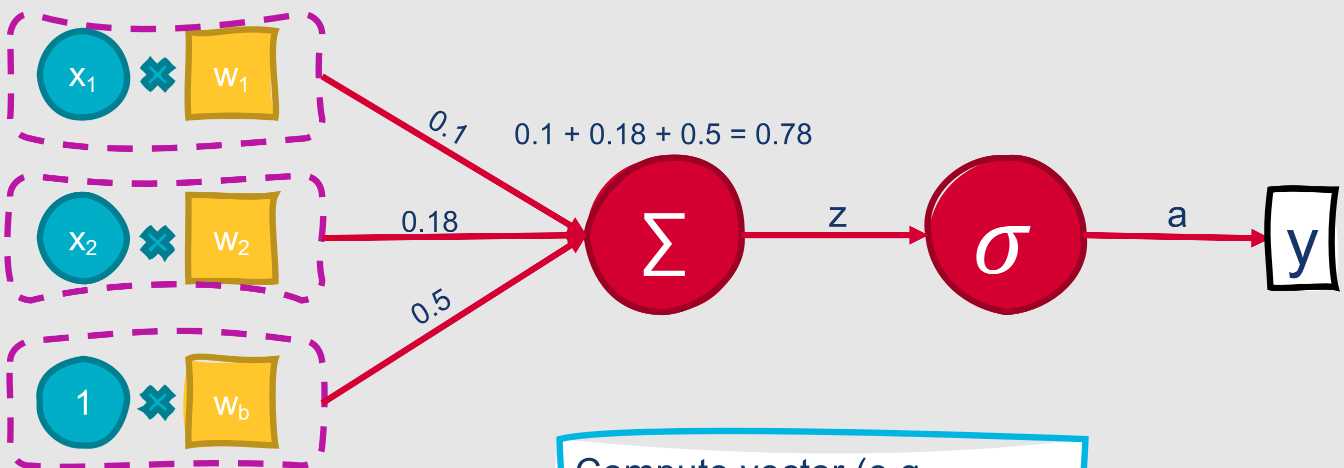
Input: “beautiful brutalist architecture”

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

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

Example: Computational Unit with Sigmoid Activation



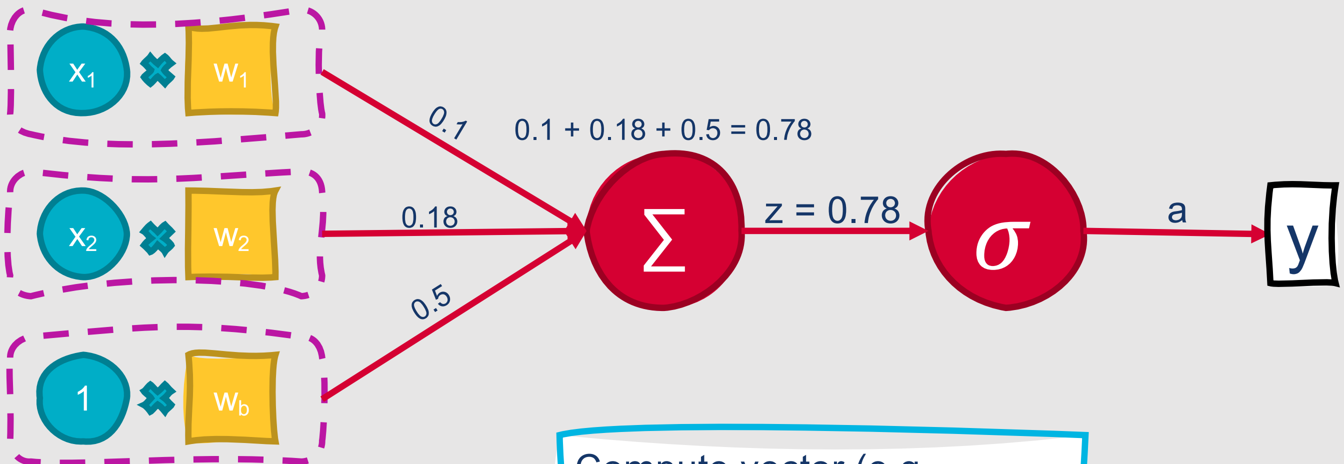
Input: “beautiful brutalist architecture”

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

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

Example: Computational Unit with Sigmoid Activation



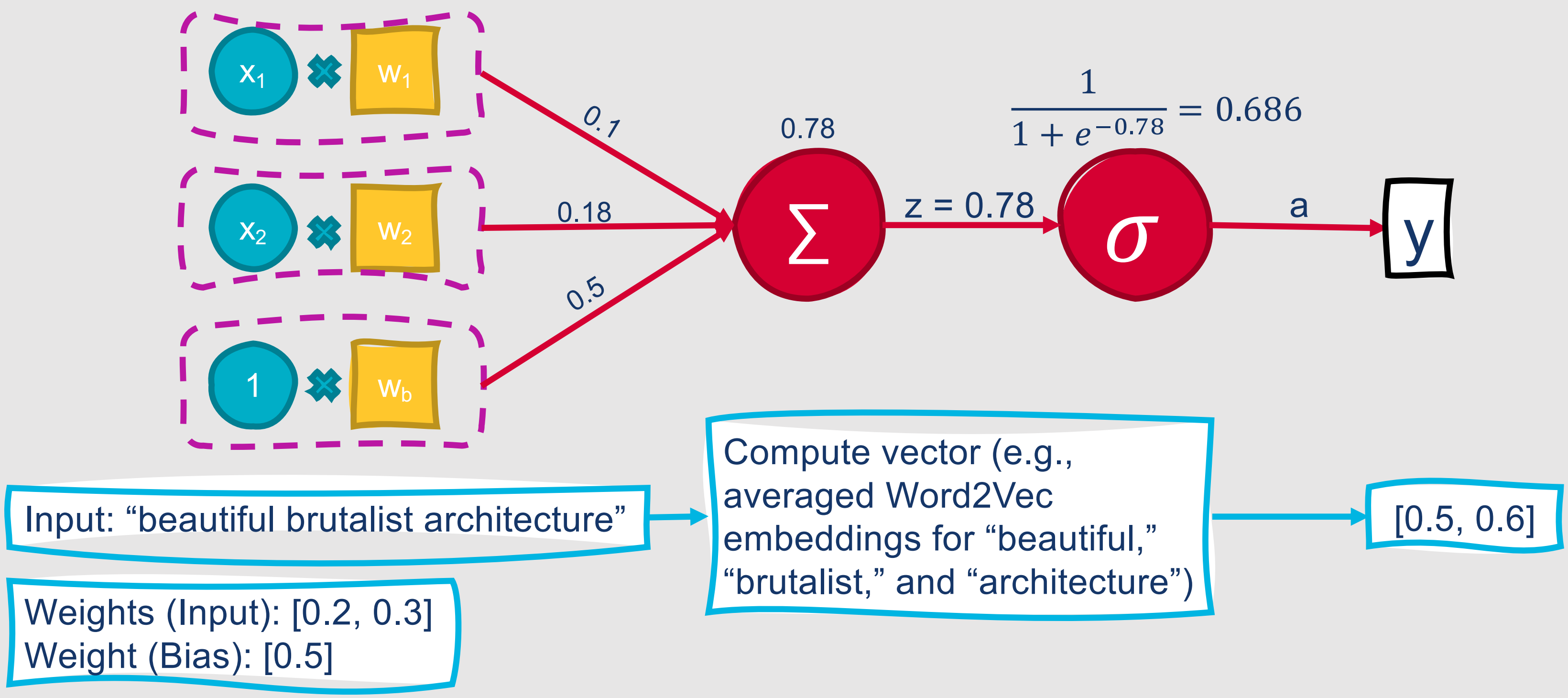
Input: “beautiful brutalist architecture”

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

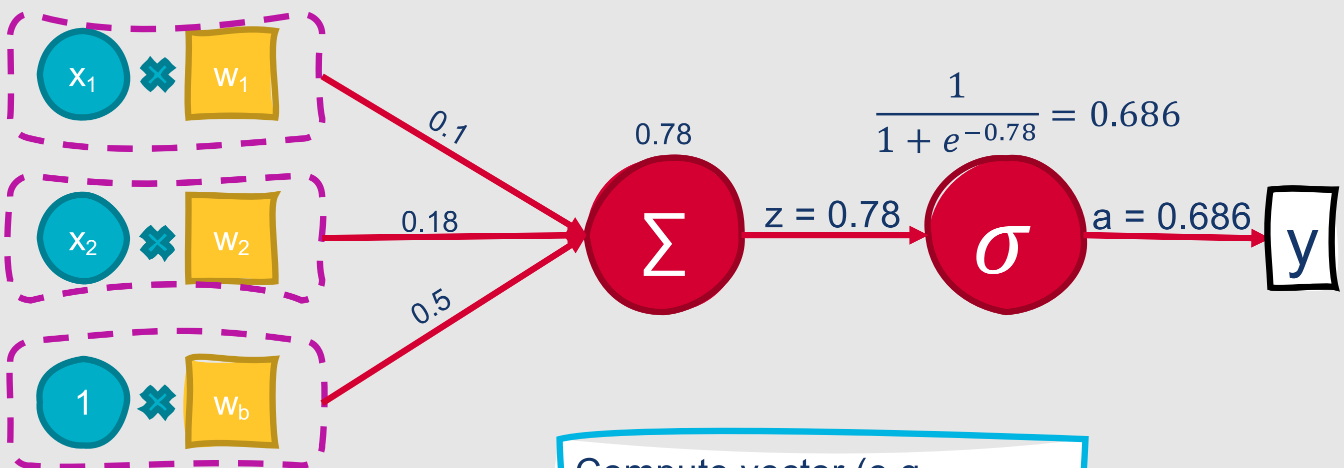
[0.5, 0.6]

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

Example: Computational Unit with Sigmoid Activation



Example: Computational Unit with Sigmoid Activation



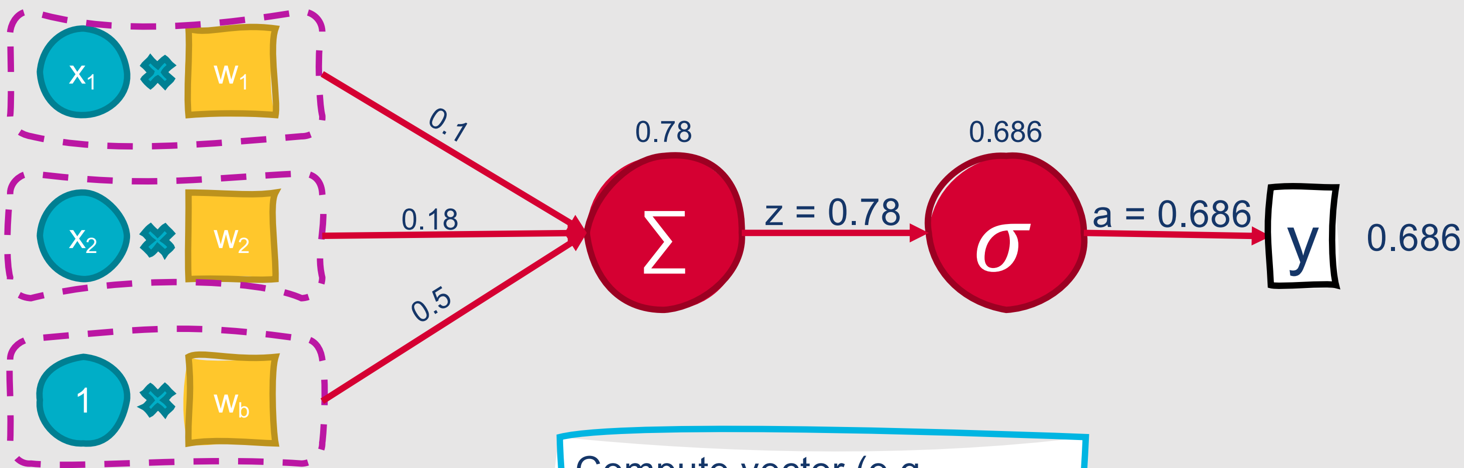
Input: “beautiful brutalist architecture”

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

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

Example: Computational Unit with Sigmoid Activation



Input: “beautiful brutalist architecture”

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

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

Other Popular Activation Functions

exponential linear unit (elu)

softmax

scaled exponential linear unit (selu)

softplus

softsign

rectified linear unit (relu)

hyperbolic tangent (tanh)

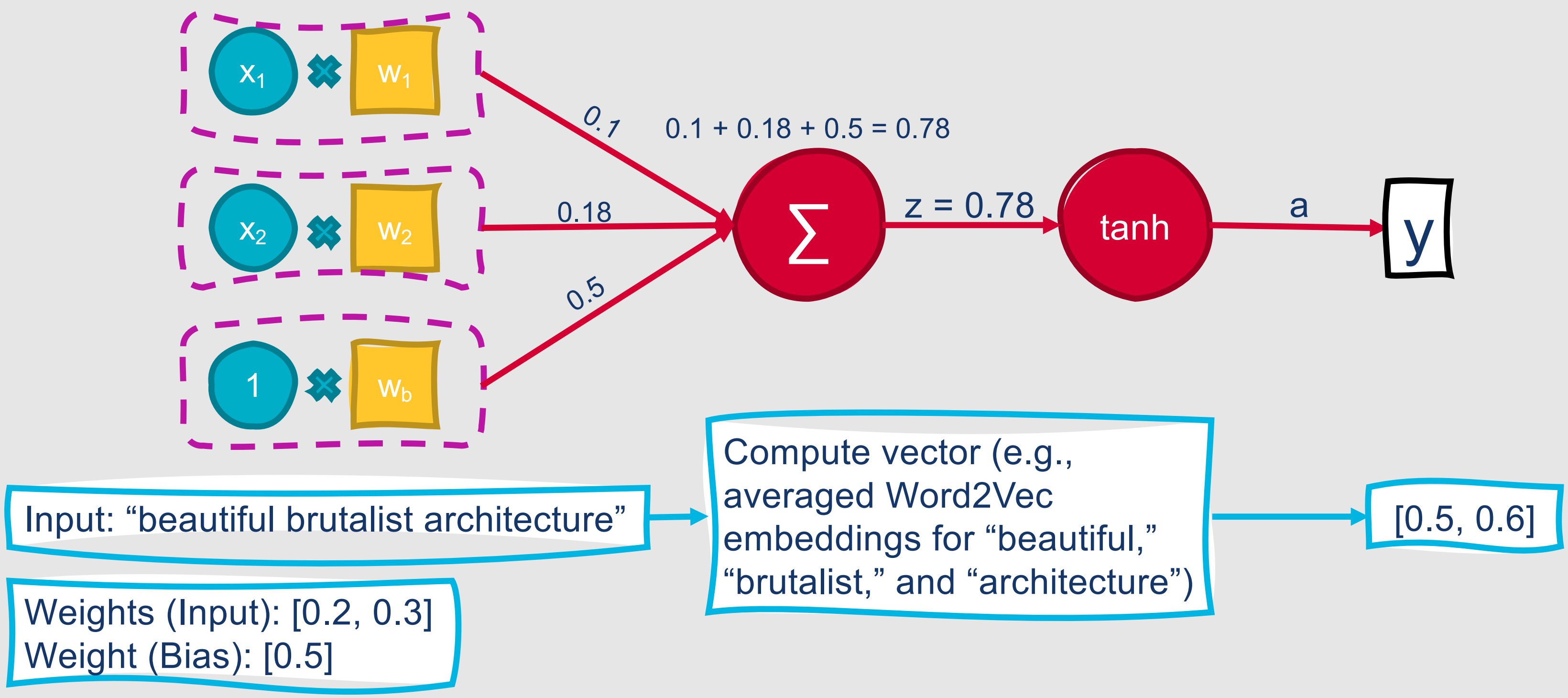
sigmoid

Particularly common activation functions

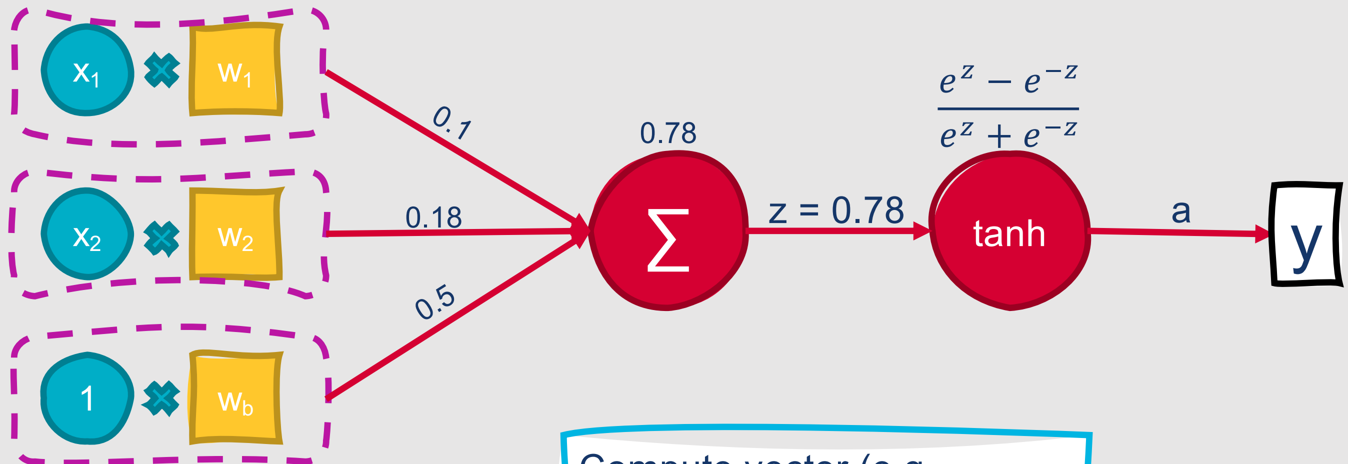
Activation: tanh

- Variant of sigmoid that ranges from -1 to +1
  - $y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- Larger derivatives → generally faster convergence

Example: Computational Unit with tanh Activation



Example: Computational Unit with tanh Activation



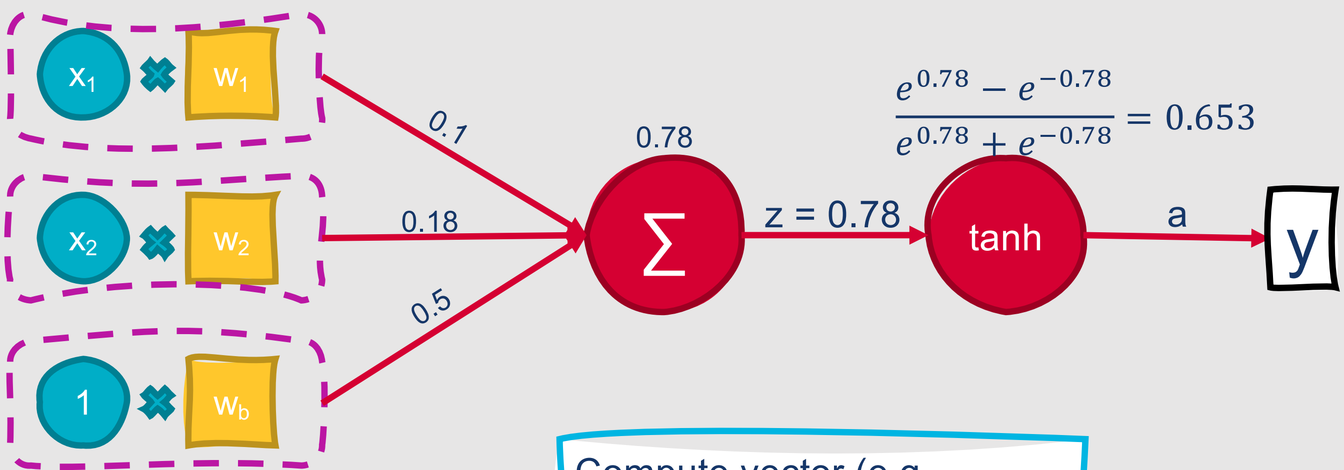
Input: "beautiful brutalist architecture"

Compute vector (e.g., averaged Word2Vec embeddings for "beautiful," "brutalist," and "architecture")

[0.5, 0.6]

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

Example: Computational Unit with tanh Activation



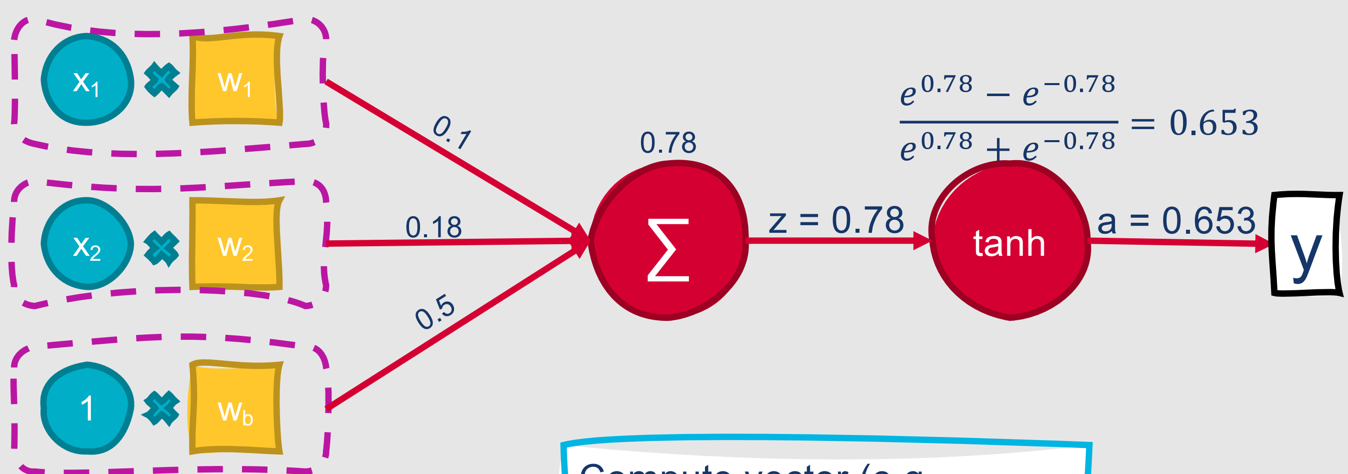
Input: “beautiful brutalist architecture”

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

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

Example: Computational Unit with tanh Activation



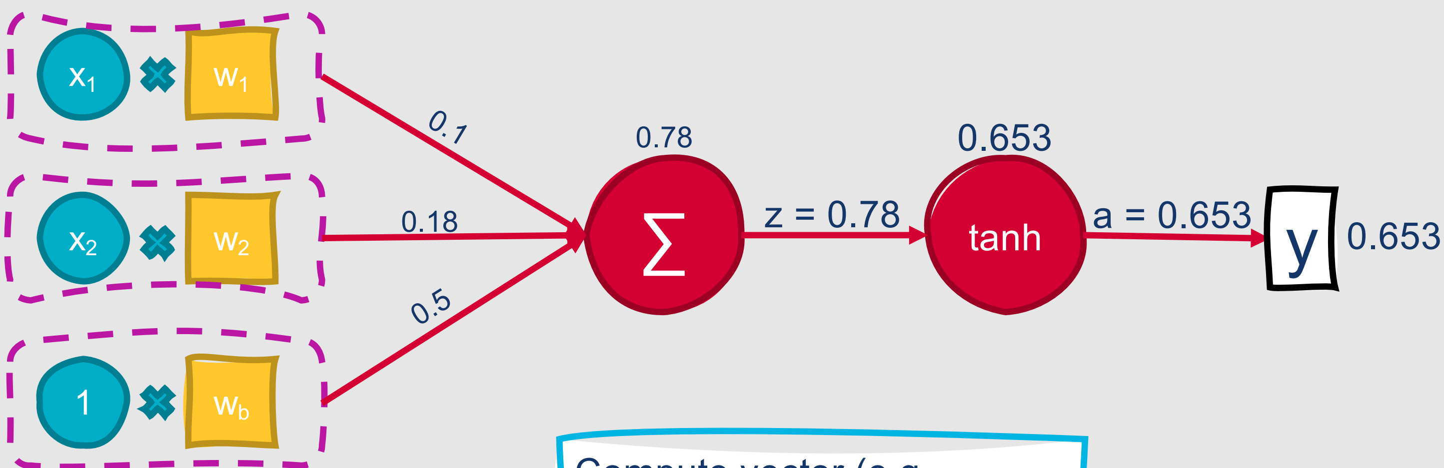
Input: “beautiful brutalist architecture”

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

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

Example: Computational Unit with tanh Activation



Input: “beautiful brutalist architecture”

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

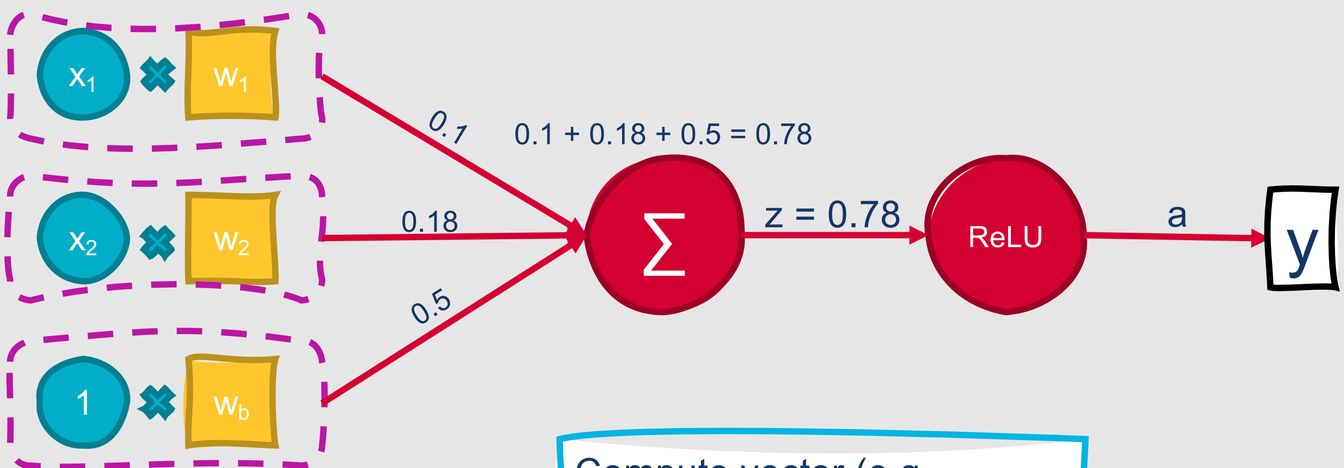
[0.5, 0.6]

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

Activation: ReLU

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

Example: Computational Unit with ReLU Activation



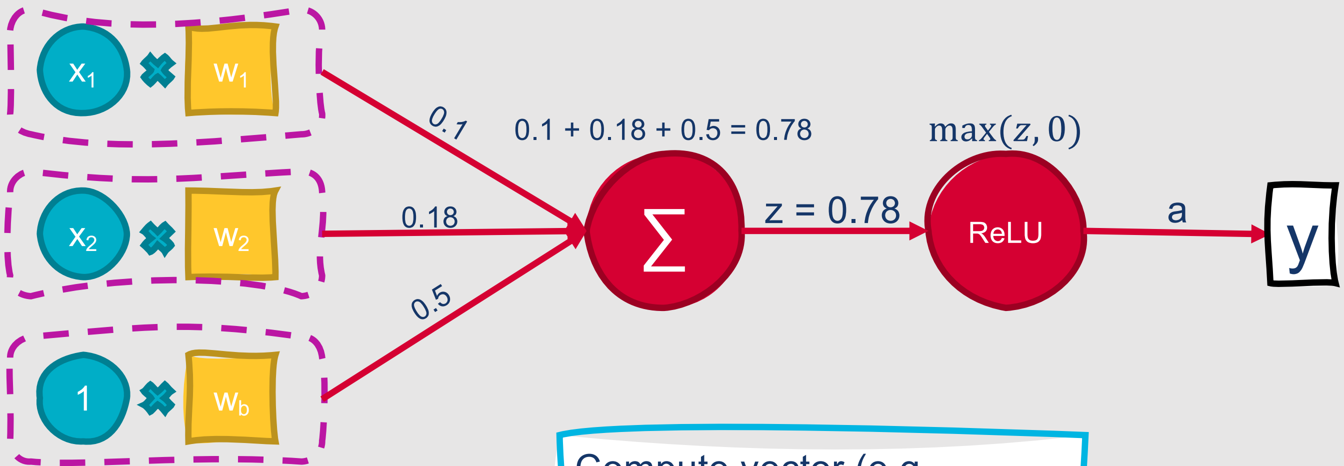
Input: “beautiful brutalist architecture”

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

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

Example: Computational Unit with ReLU Activation



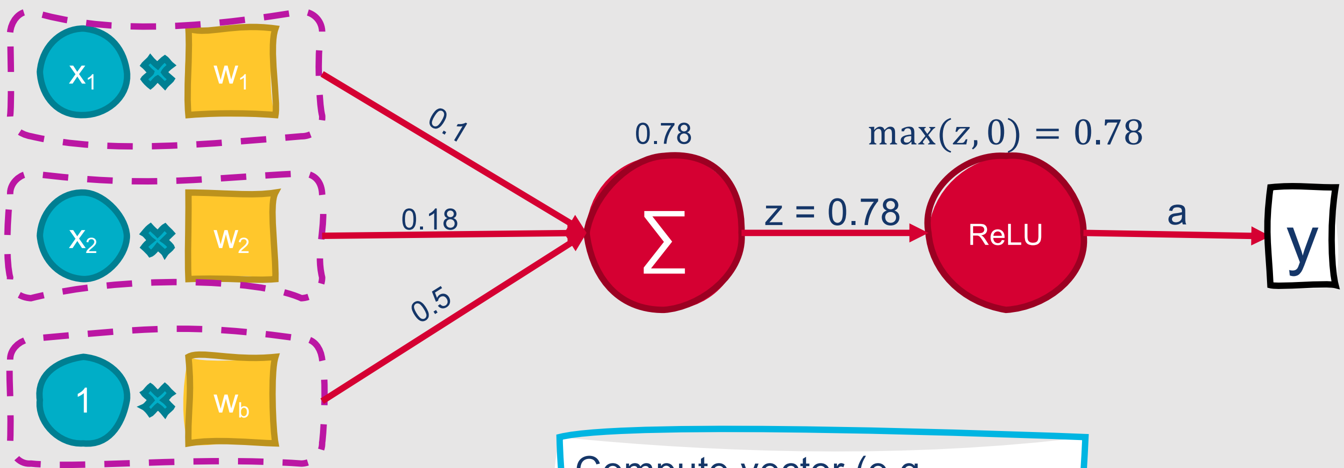
Input: “beautiful brutalist architecture”

Compute vector (e.g., averaged Word2Vec embeddings for “beautiful,” “brutalist,” and “architecture”)

[0.5, 0.6]

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

Example: Computational Unit with ReLU Activation



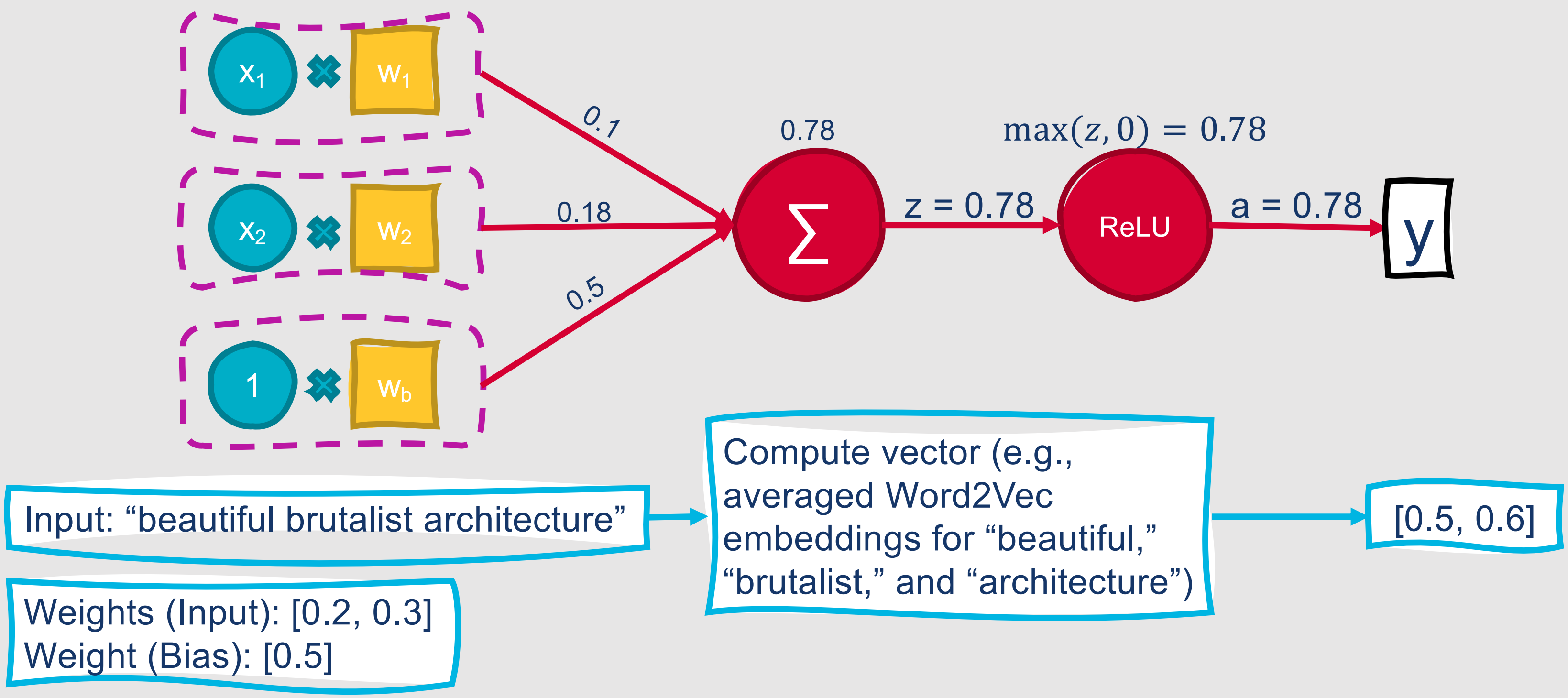
Input: "beautiful brutalist architecture"

Compute vector (e.g., averaged Word2Vec embeddings for "beautiful," "brutalist," and "architecture")

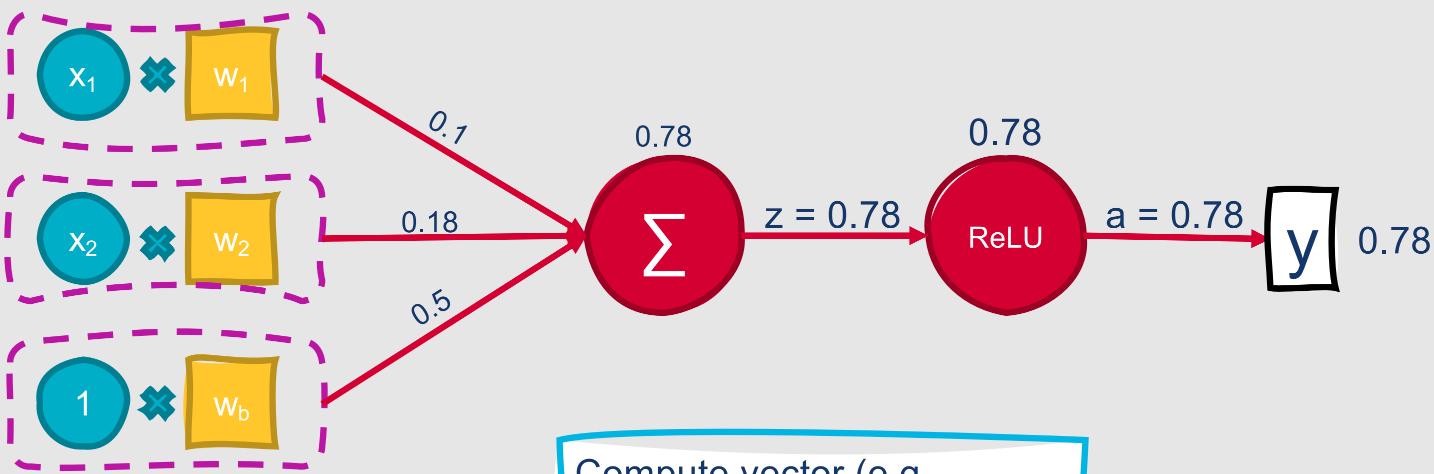
[0.5, 0.6]

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

Example: Computational Unit with ReLU Activation



Example: Computational Unit with ReLU Activation



Input: "beautiful brutalist architecture"

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

Compute vector (e.g., averaged Word2Vec embeddings for "beautiful," "brutalist," and "architecture")

[0.5, 0.6]

Comparing sigmoid, tanh, and ReLU

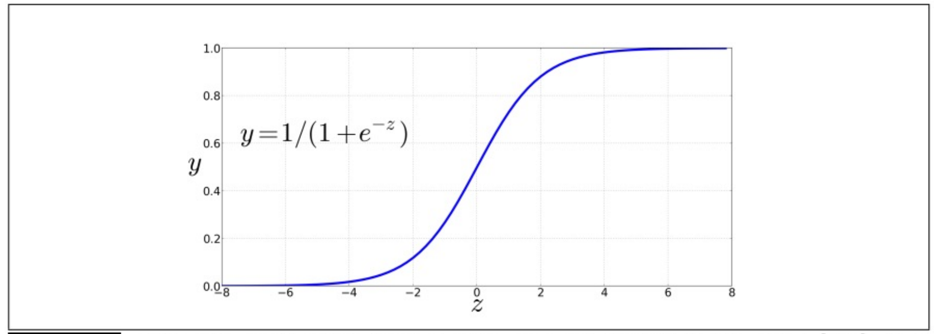


Figure 7.1 The sigmoid function takes a real value and maps it to the range [0,1]. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

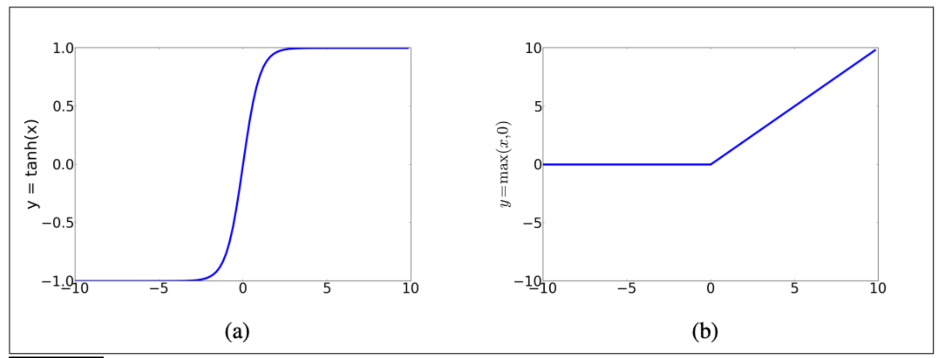


Figure 7.3 The tanh and ReLU activation functions.

This Week's Topics

Neural networks
Computational units
Combining layers of units

Thursday

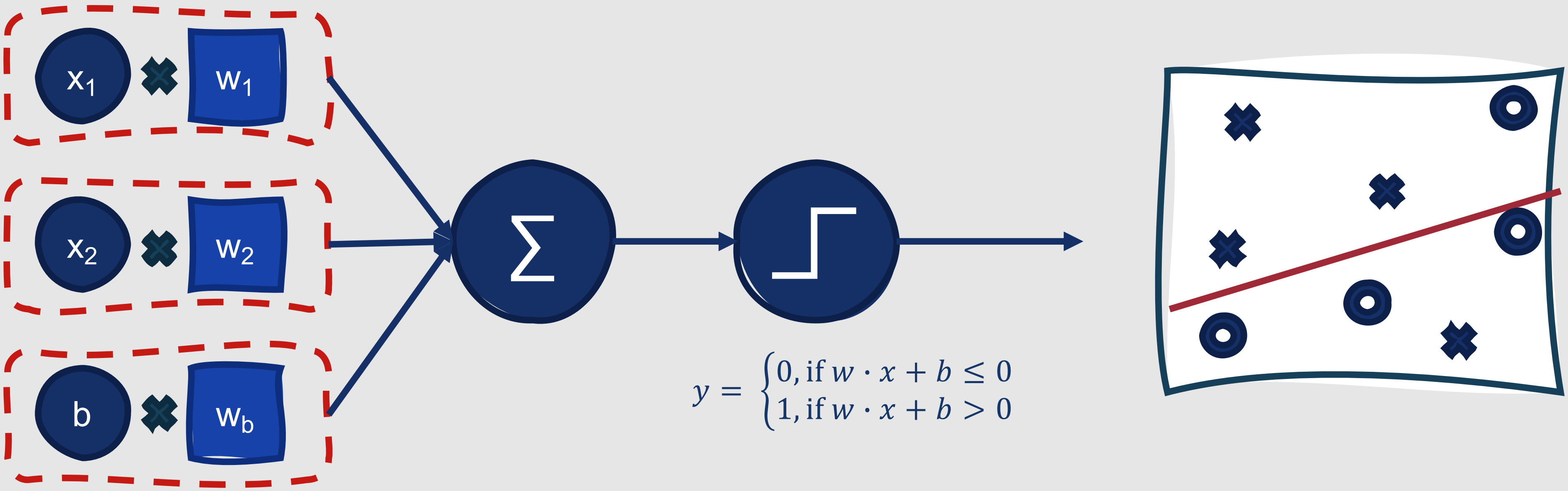
Tuesday

Backpropagation
Neural language models
Recurrent neural networks
Other popular deep learning architectures

Combining Computational Units

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

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

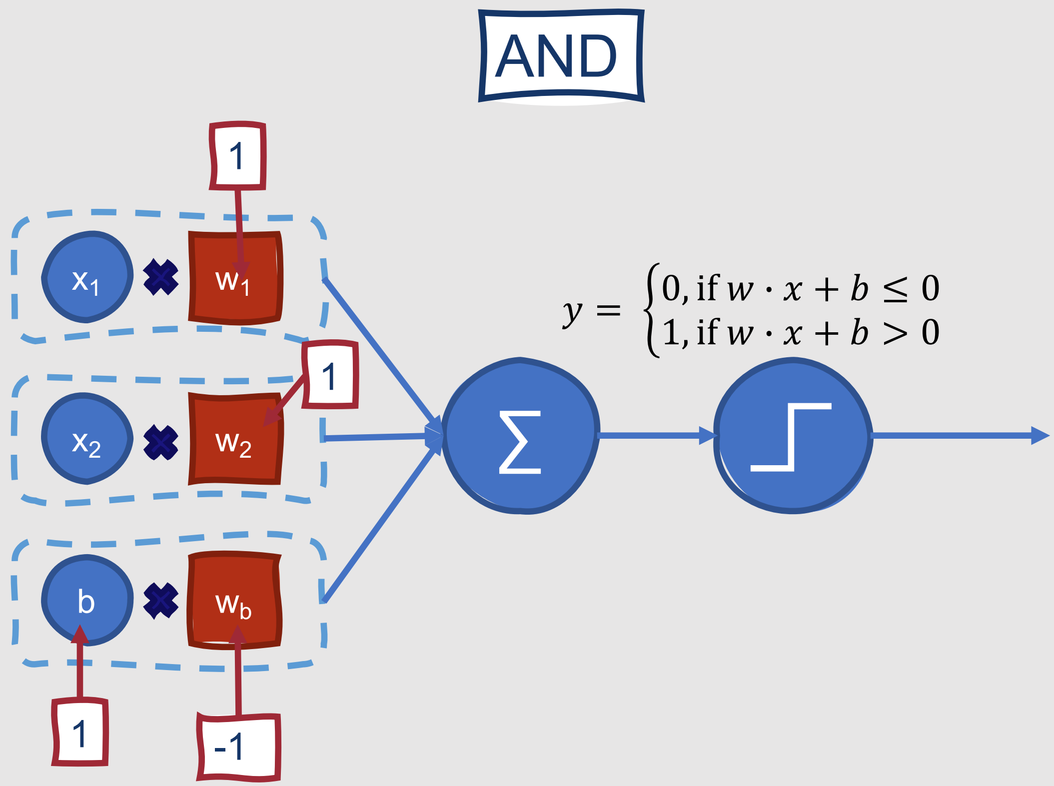


$$y = \begin{cases} 0, \text{if } w \cdot x + b \leq 0 \\ 1, \text{if } w \cdot x + b > 0 \end{cases}$$

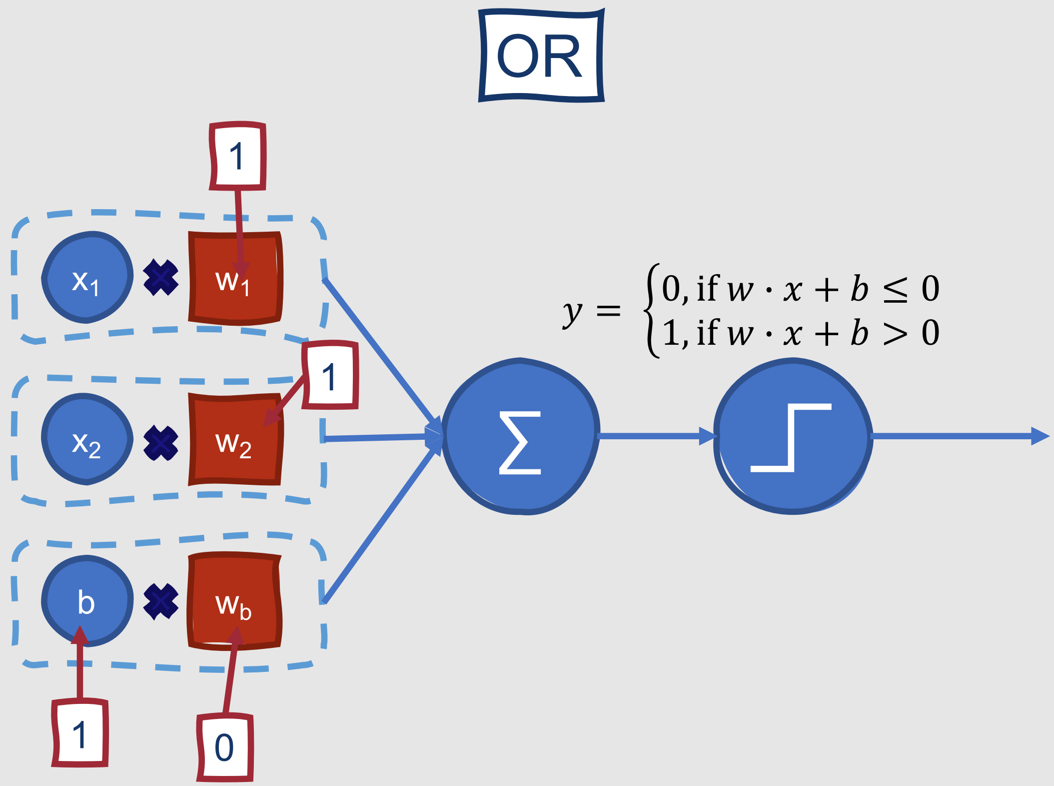
AND and OR can both be solved using a single perceptron.

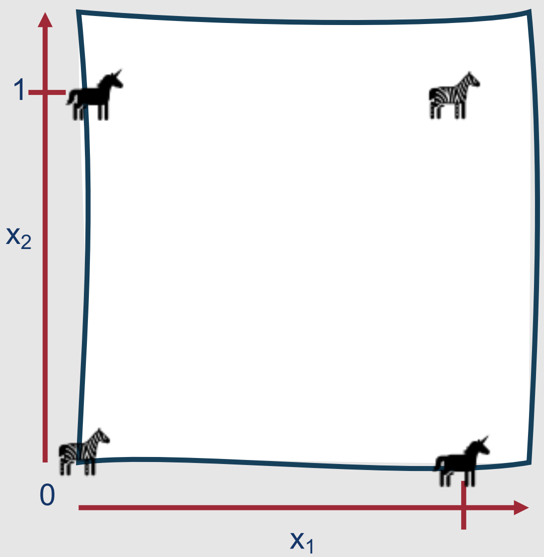
- **Perceptron:** A function that outputs a binary value based on whether the product of its inputs and associated weights surpasses a threshold

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



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



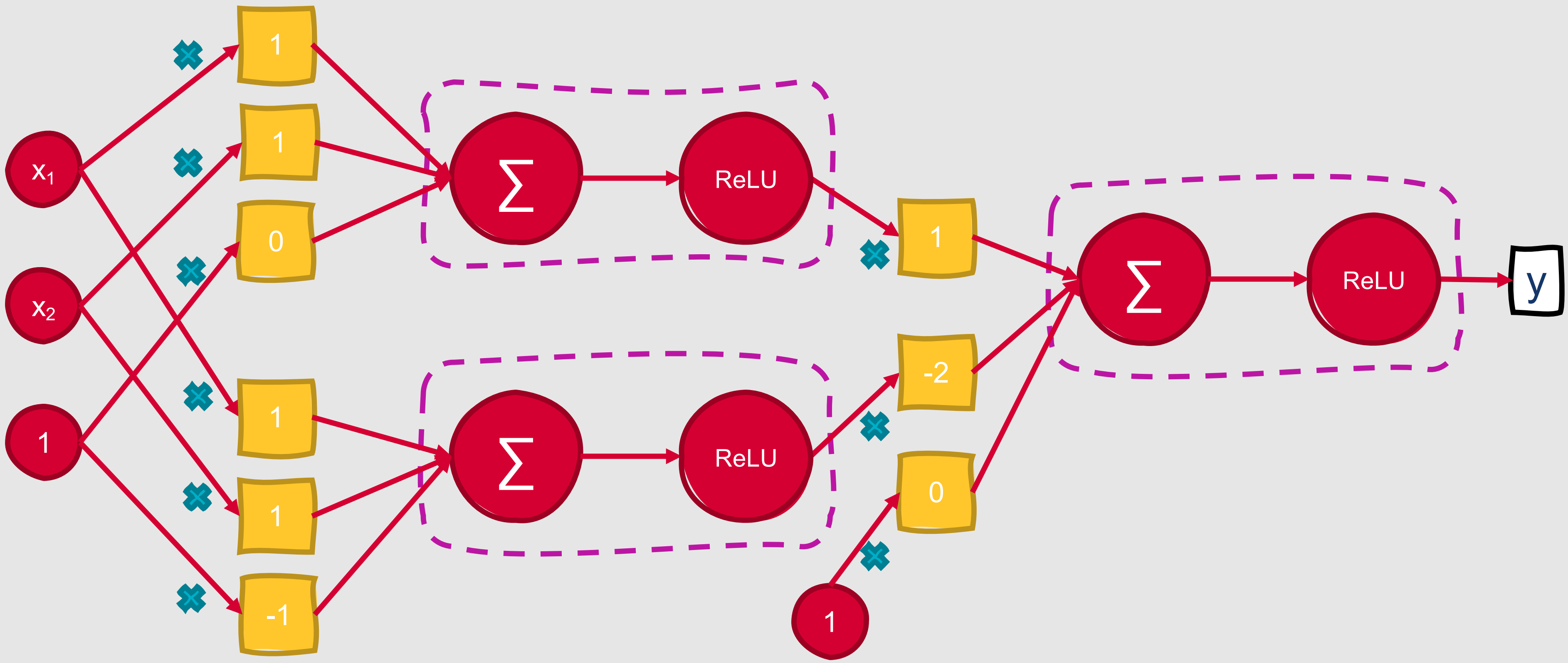


AND			OR			XOR		
x1	**x2**	**y**	**x1**	**x2**	**y**	**x1**	**x2**	**y**
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

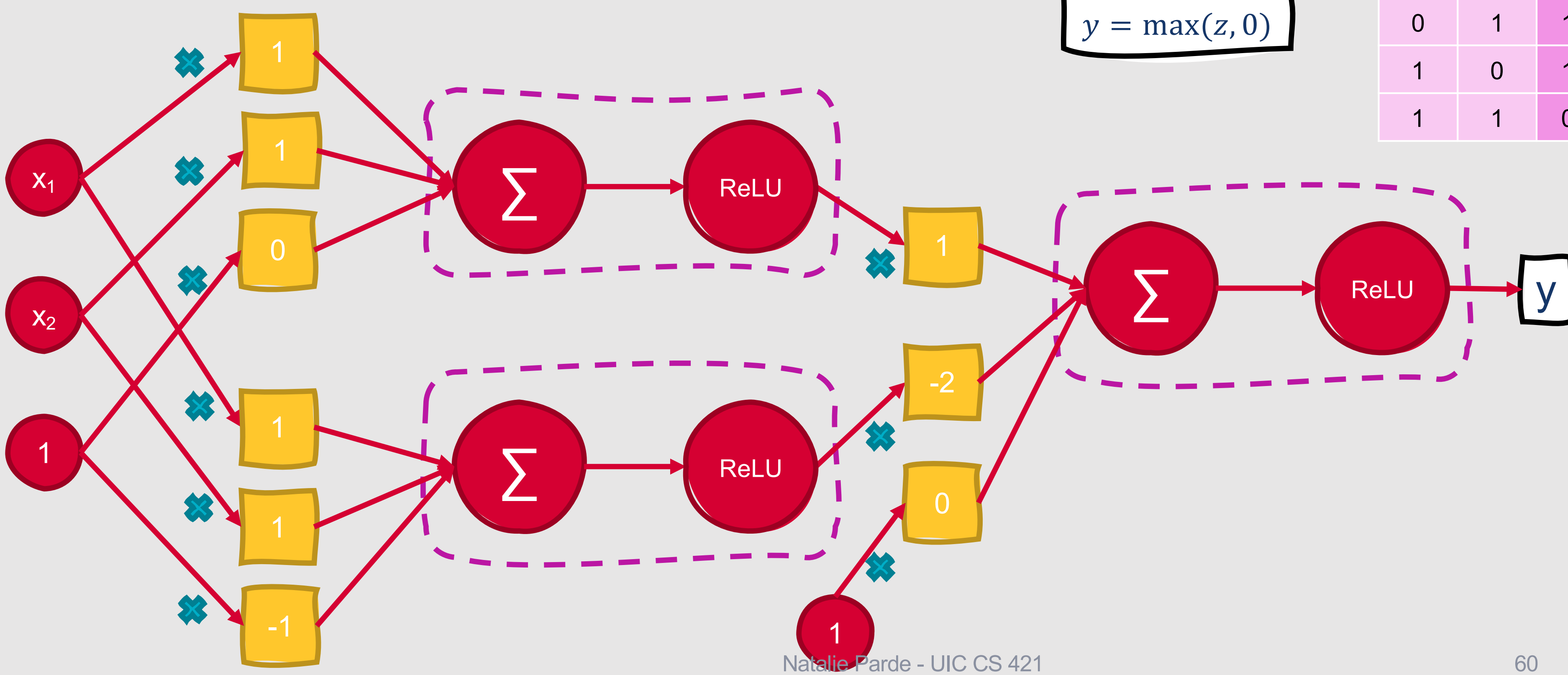
However, it's impossible to compute XOR using a single perceptron.

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

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



Truth Table Examples: XOR



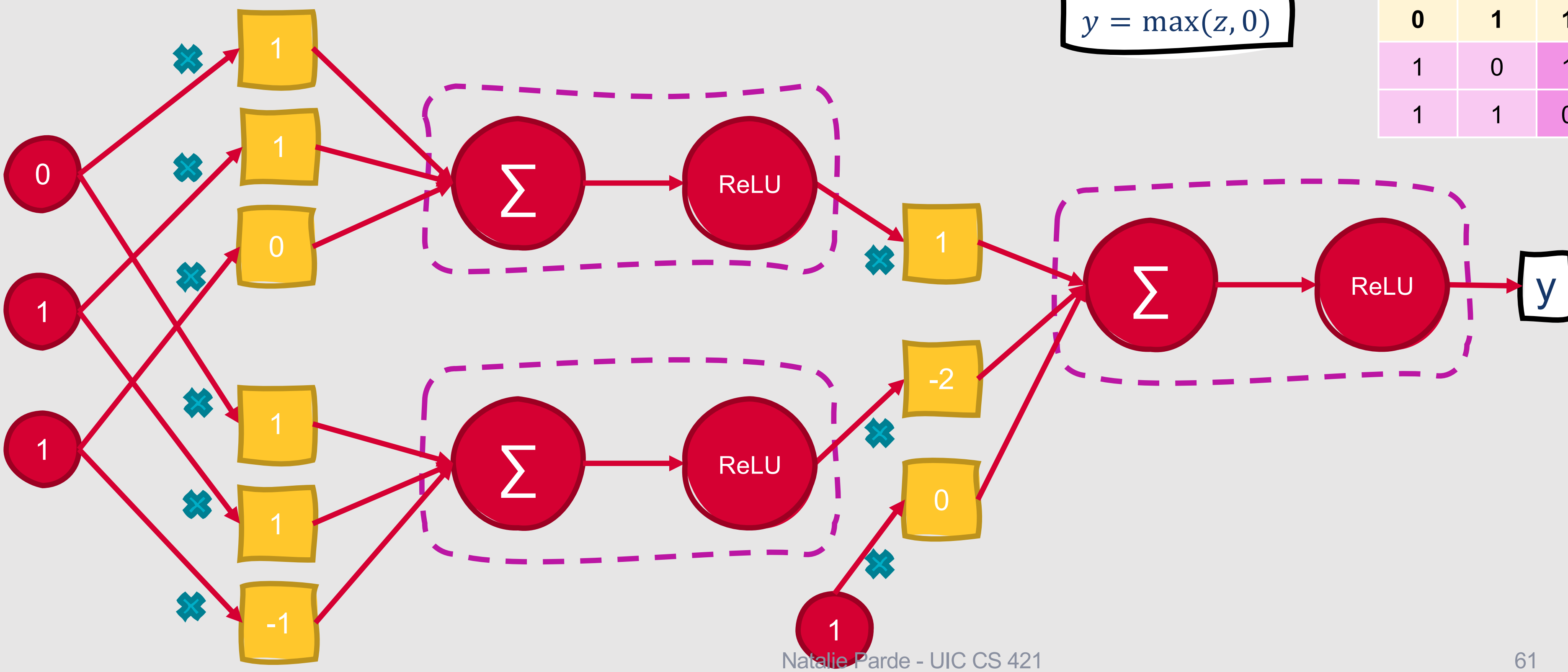
$y = \max(z, 0)$

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

Truth Table Examples: XOR

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

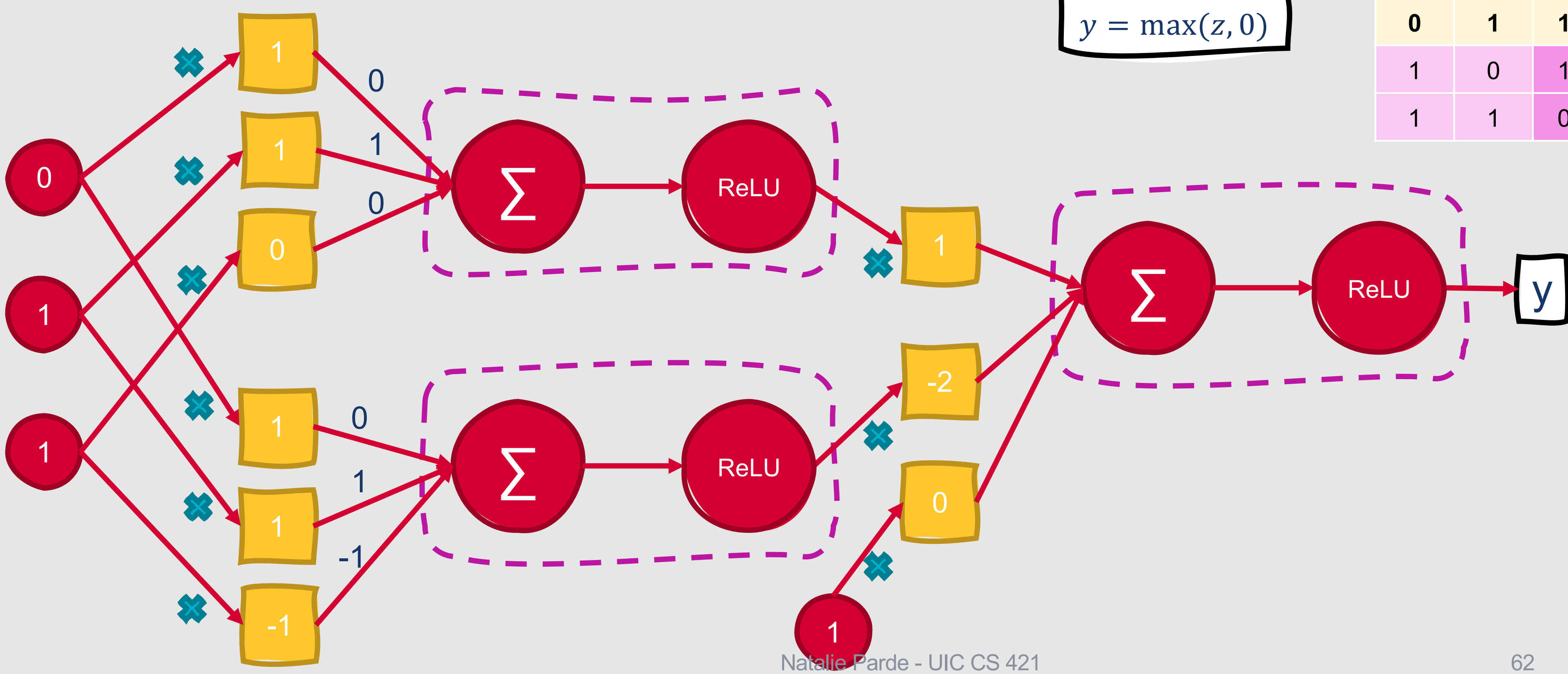
$$y = \max(z, 0)$$



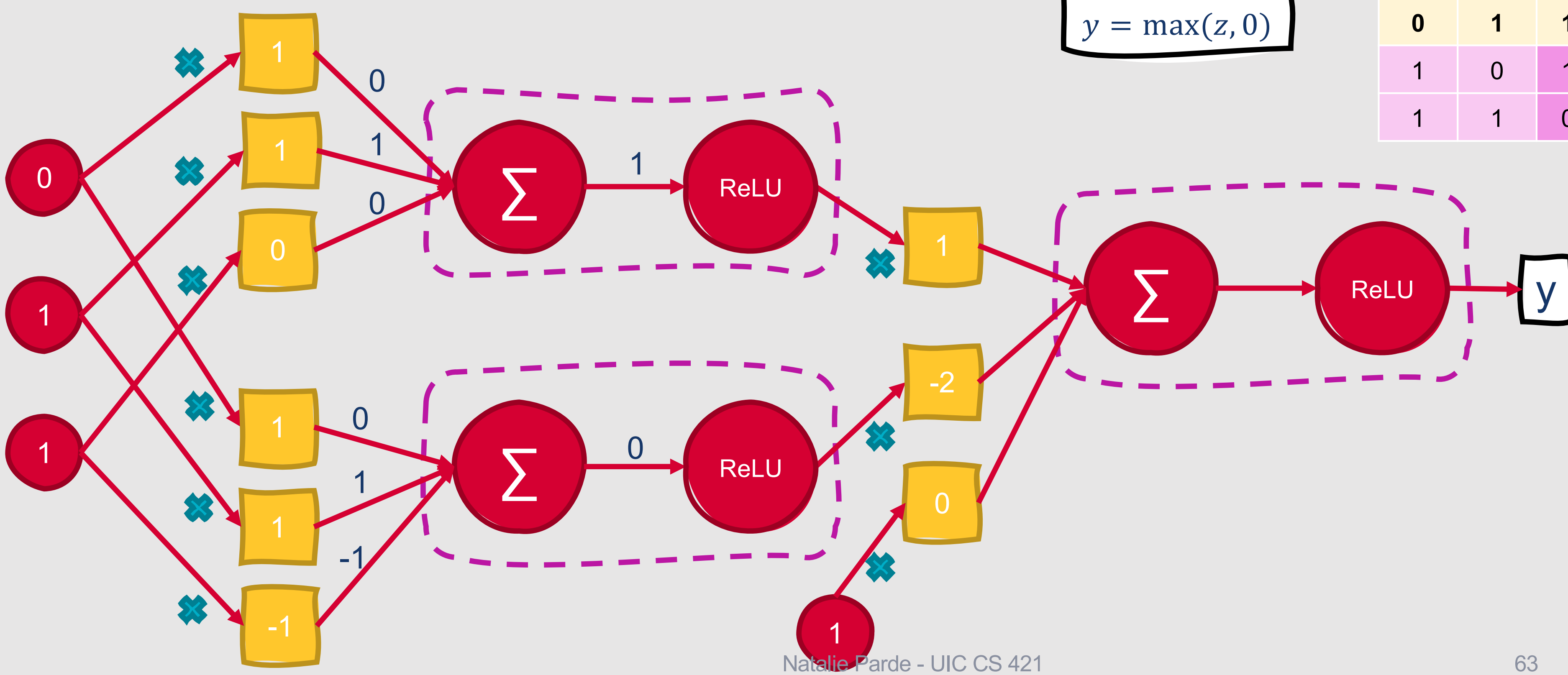
Truth Table Examples: XOR

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

$y = \max(z, 0)$



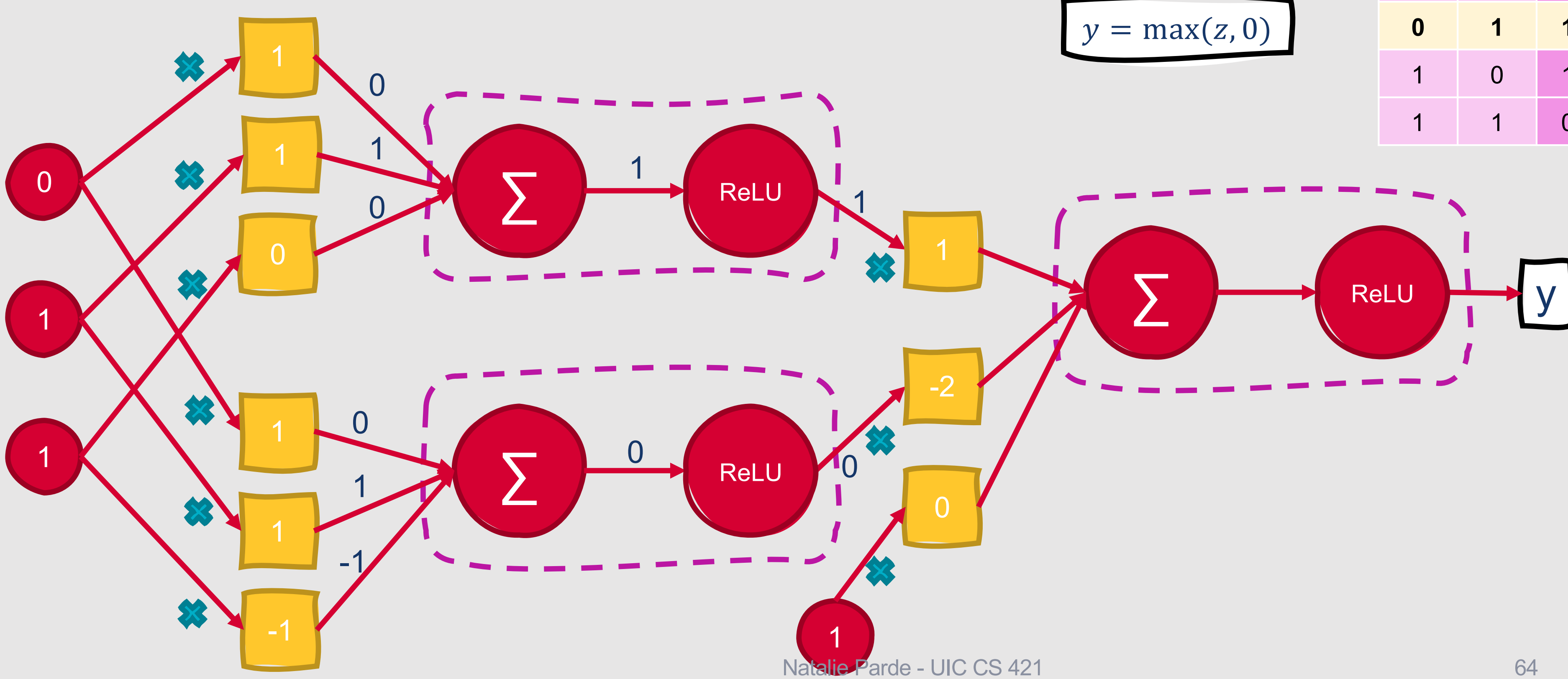
Truth Table Examples: XOR



$y = \max(z, 0)$

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

Truth Table Examples: XOR



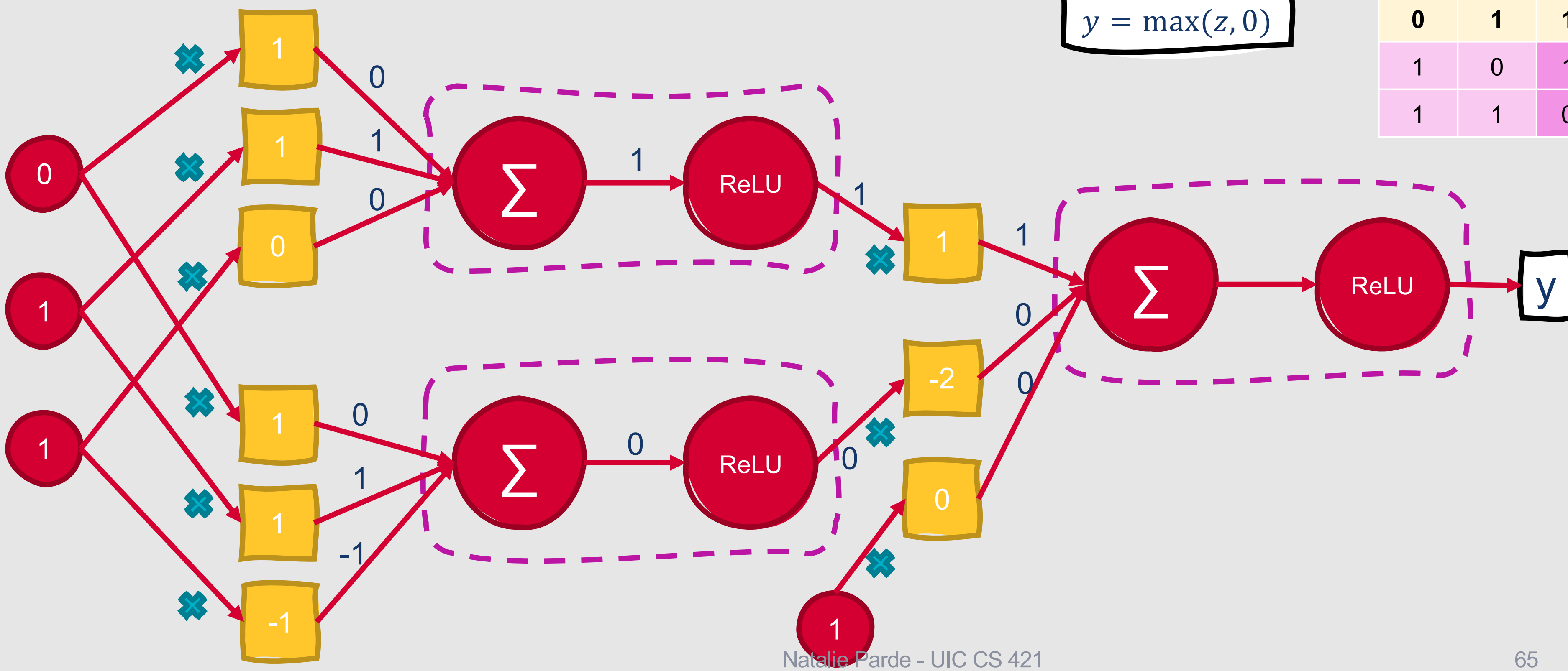
$y = \max(z, 0)$

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

Truth Table Examples: XOR

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

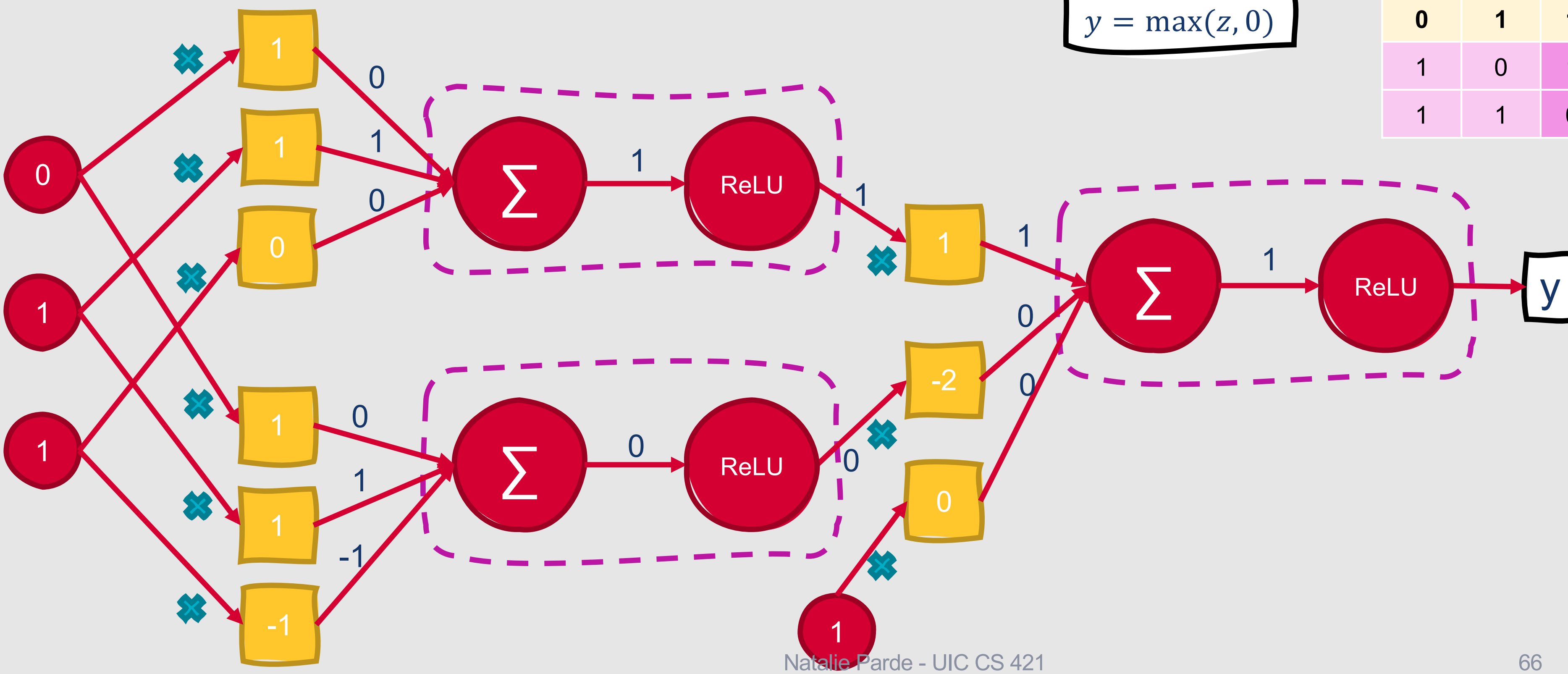
$y = \max(z, 0)$



Truth Table Examples: XOR

$y = \max(z, 0)$

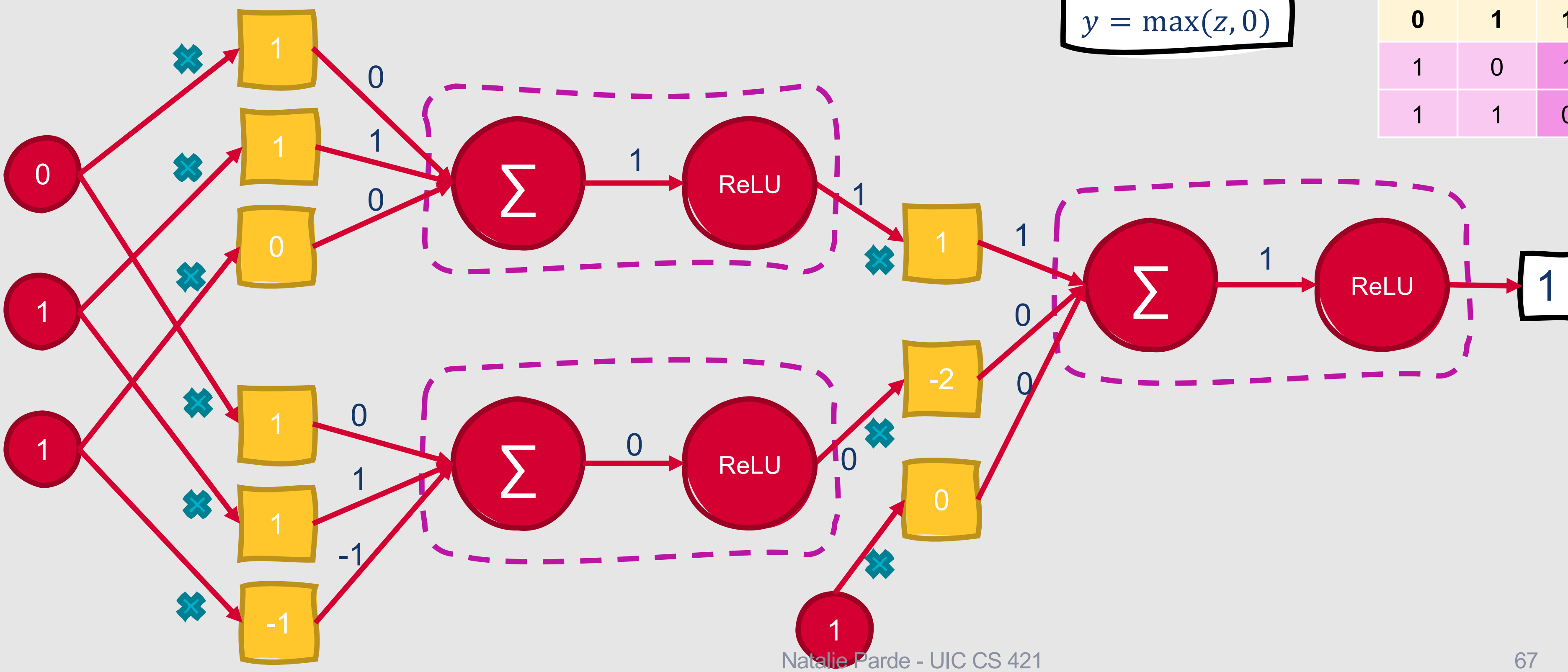
XOR		
x1	**x2**	**y**
0	0	0
0	**1**	**1**
1	0	1
1	1	0



Truth Table Examples: XOR

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

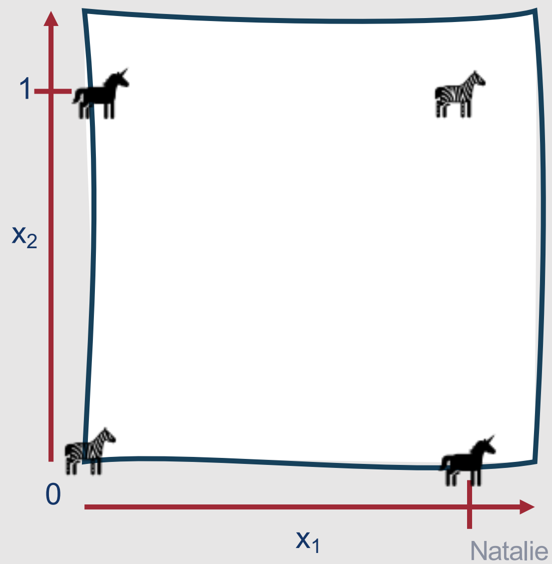
XOR		
x1	x2	y
0	0	0
0	**1**	**1**
1	0	1
1	1	0



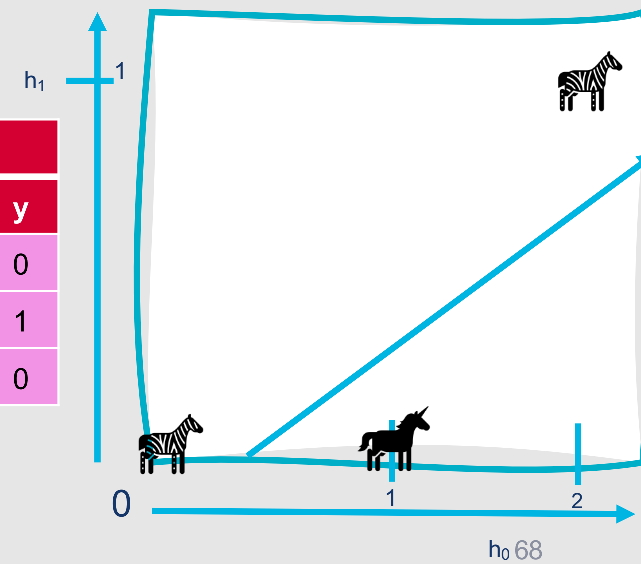
Why does this work?

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

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



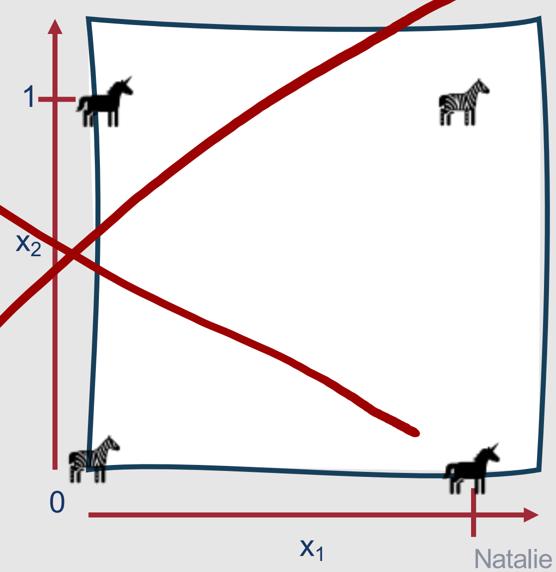
XOR		
h0	**h1**	**y**
0	0	0
1	0	1
2	1	0



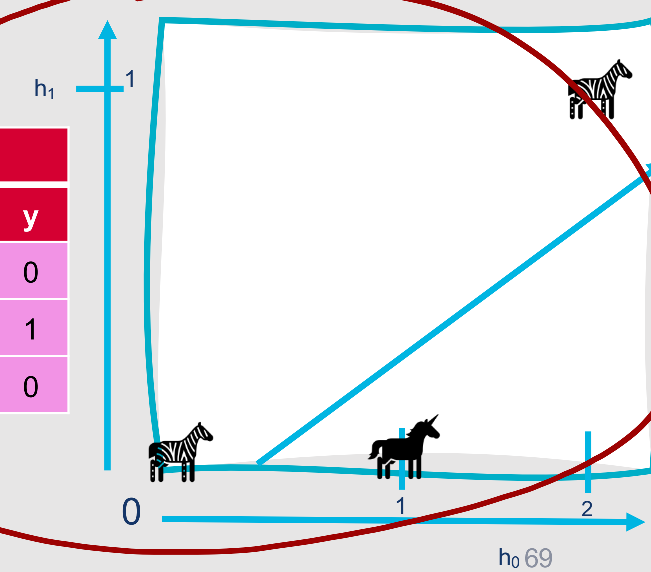
Why does this work?

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

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



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



Feedforward Network

- Final formulation for previous network:
  - $\mathbf{h} = \text{ReLU}(W\mathbf{x} + \mathbf{b})$
  - $y' = \text{ReLU}(U\mathbf{h} + \mathbf{b})$
- This represents a two-layer feedforward neural network
  - When numbering layers, count the hidden and output layers but not the inputs

We can generalize this for networks with > 2 layers.

- Let $W^{[n]}$ be the weight matrix for layer n, $\mathbf{b}^{[n]}$ be the bias vector for layer n, and so forth
- Let $g(\cdot)$ be any activation function
- Let $\mathbf{a}^{[n]}$ be the output from layer n, and $\mathbf{z}^{[n]}$ be the combination of weights and biases $W^{[n]}\,\mathbf{a}^{[n-1]} + \mathbf{b}^{[n]}$
- Let the input layer be $\mathbf{a}^{[0]}$

Neural Network: Formal Structure

- With this representation, a two-layer network becomes:
  - $z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$
  - $a^{[1]} = g^{[1]}\left(z^{[1]}\right)$
  - $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
  - $a^{[2]} = g^{[2]}(z^{[2]})$
  - $y' = a^{[2]}$
- We can easily generalize to networks with more layers:
  - For i in 1..*n*
    - $z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}$
    - $a^{[i]} = g^{[i]}(z^{[i]})$
  - $y' = a^{[n]}$

General Tips for Improving Neural Network Performance

- **Initialize weights** with small random numbers
- **Tune hyperparameters**
  - Learning rate
  - Number of layers
  - Number of units per layer
  - Type of activation function
  - Type of optimization function

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

TensorFlow

- https://www.tensorflow.org/

Keras

- https://keras.io/

PyTorch

- https://pytorch.org/

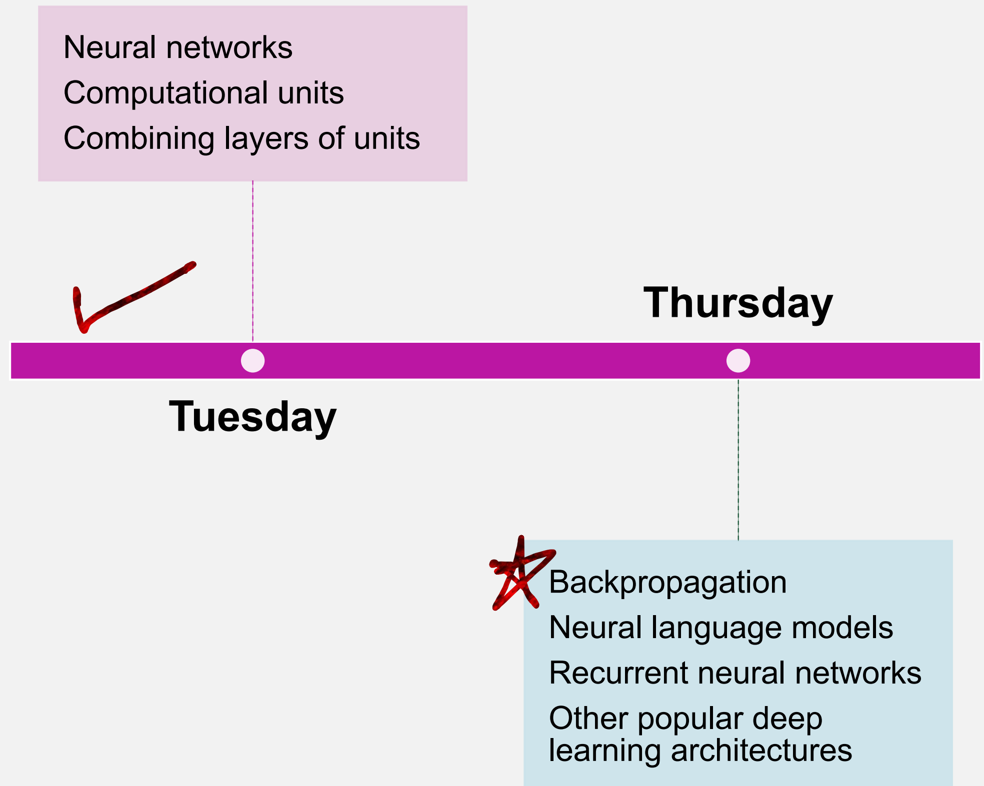
DL4J

- https://deeplearning4j.org/

Summary: Feedforward Neural Networks

- Neural networks are classification models that **implicitly learn** sophisticated feature representations
- **Feedforward neural networks** are comprised of interconnected layers of computing units through which information is passed forward from one layer to the next
- An **activation function** is a non-linear function applied to the weighted sum of inputs for a computing unit
- Computing units can be combined with another to solve complex tasks

This Week's Topics



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

???

There are two ways that we can pass information through a neural network.

- **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**
  - ???

Backpropagation

- Propagates loss values all the way back to the beginning of a neural network, even though it's only computed at the end of the network
- Why is this necessary?
  - Simply taking the derivative like we did for logistic regression only provides the gradient for the most recent (i.e., last) weight layer
  - What we need is a way to:
    - Compute the derivative with respect to weight parameters occurring earlier in the network as well
    - Even though we can only compute loss at a single point (the end of the network)

Backpropagation in a nutshell….

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

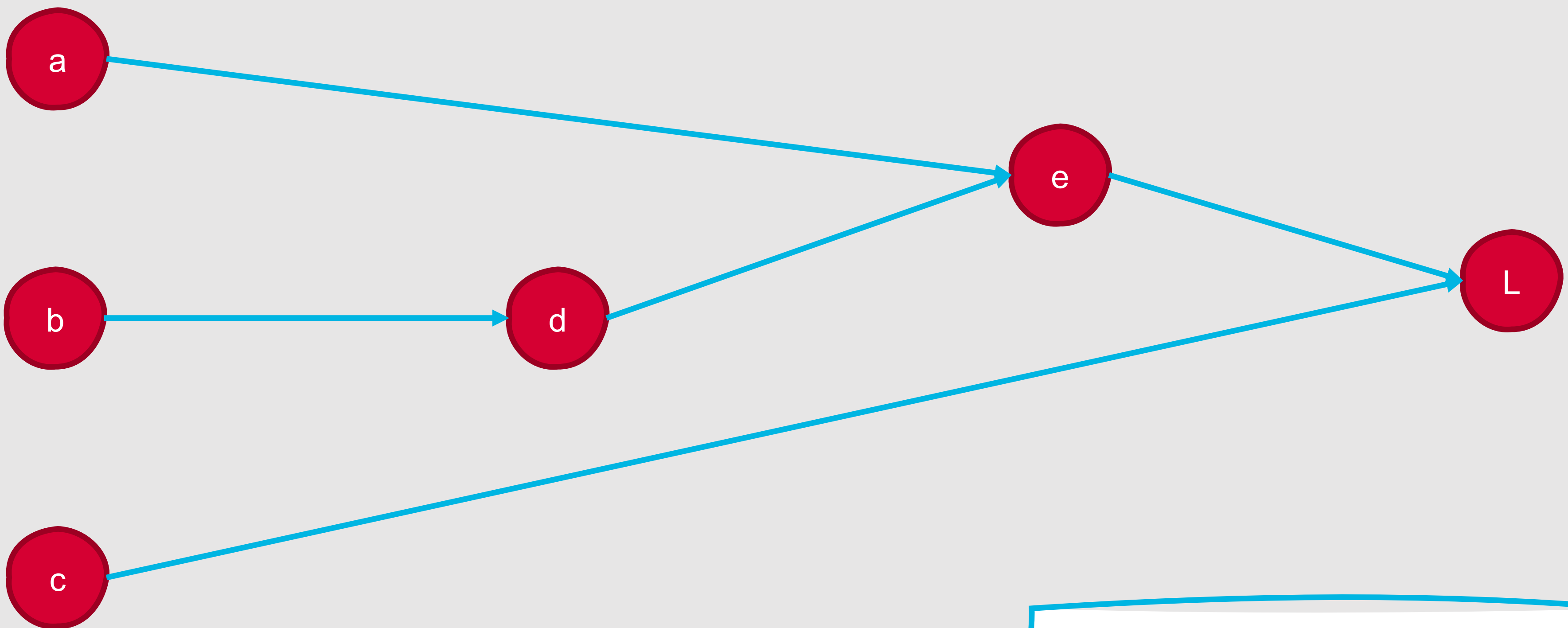
There are two ways that we can pass information through a neural network.

- **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

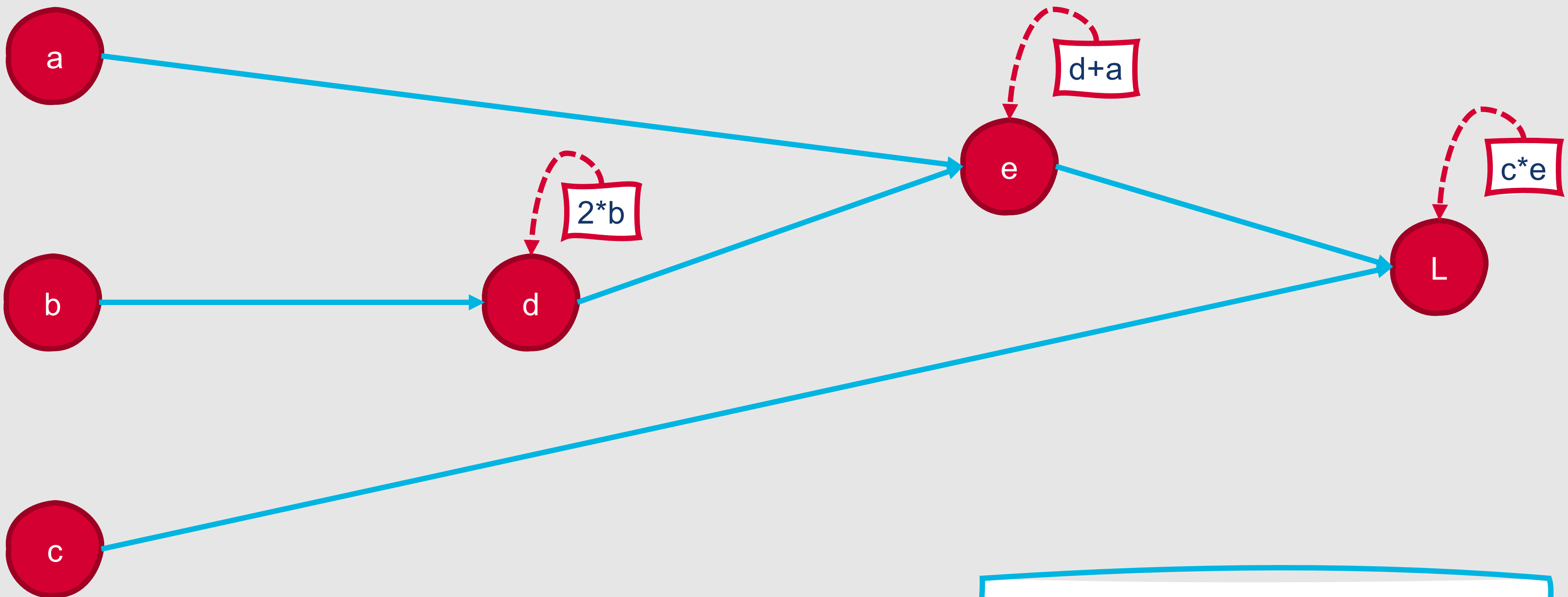
Goal: Represent $L(a, b, c) = c(a + 2b)$

Example: Forward Pass



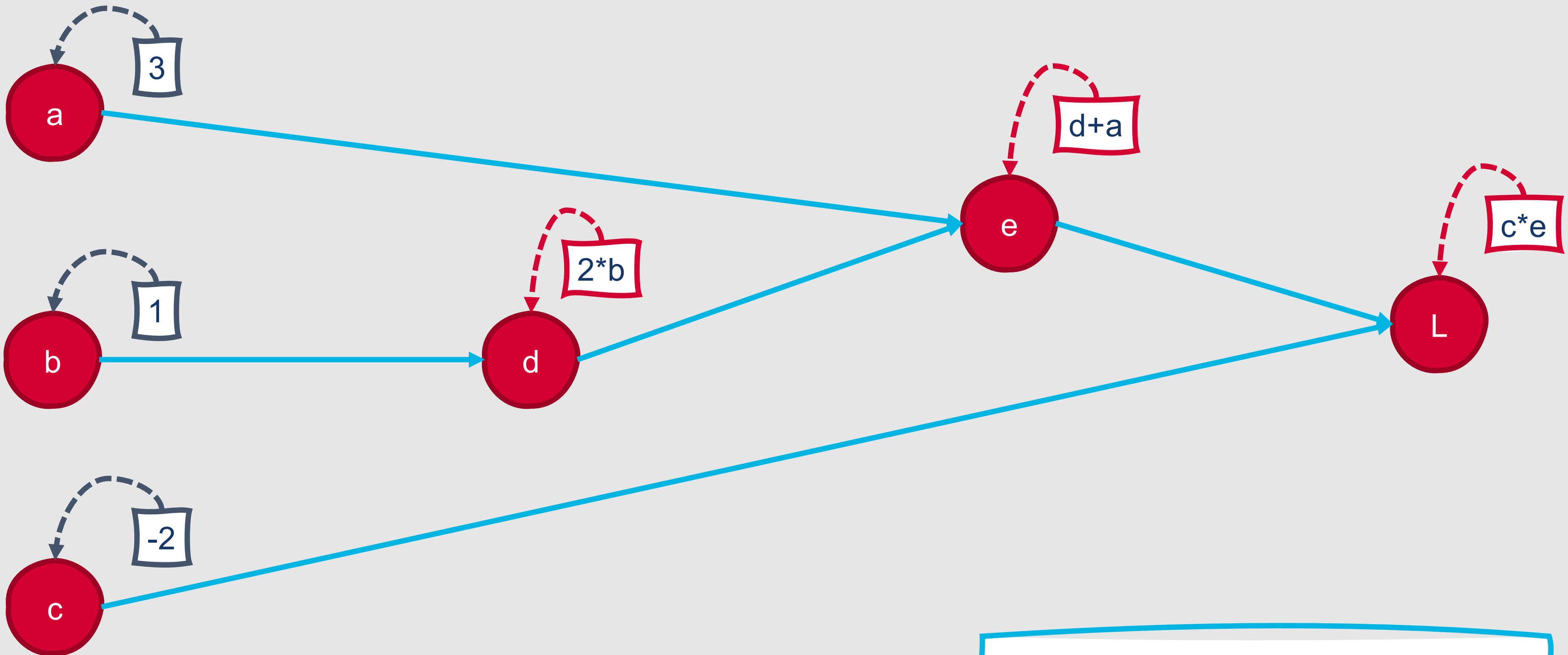
Goal: Represent L(a, b, c) = c(a + 2b)

Example: Forward Pass



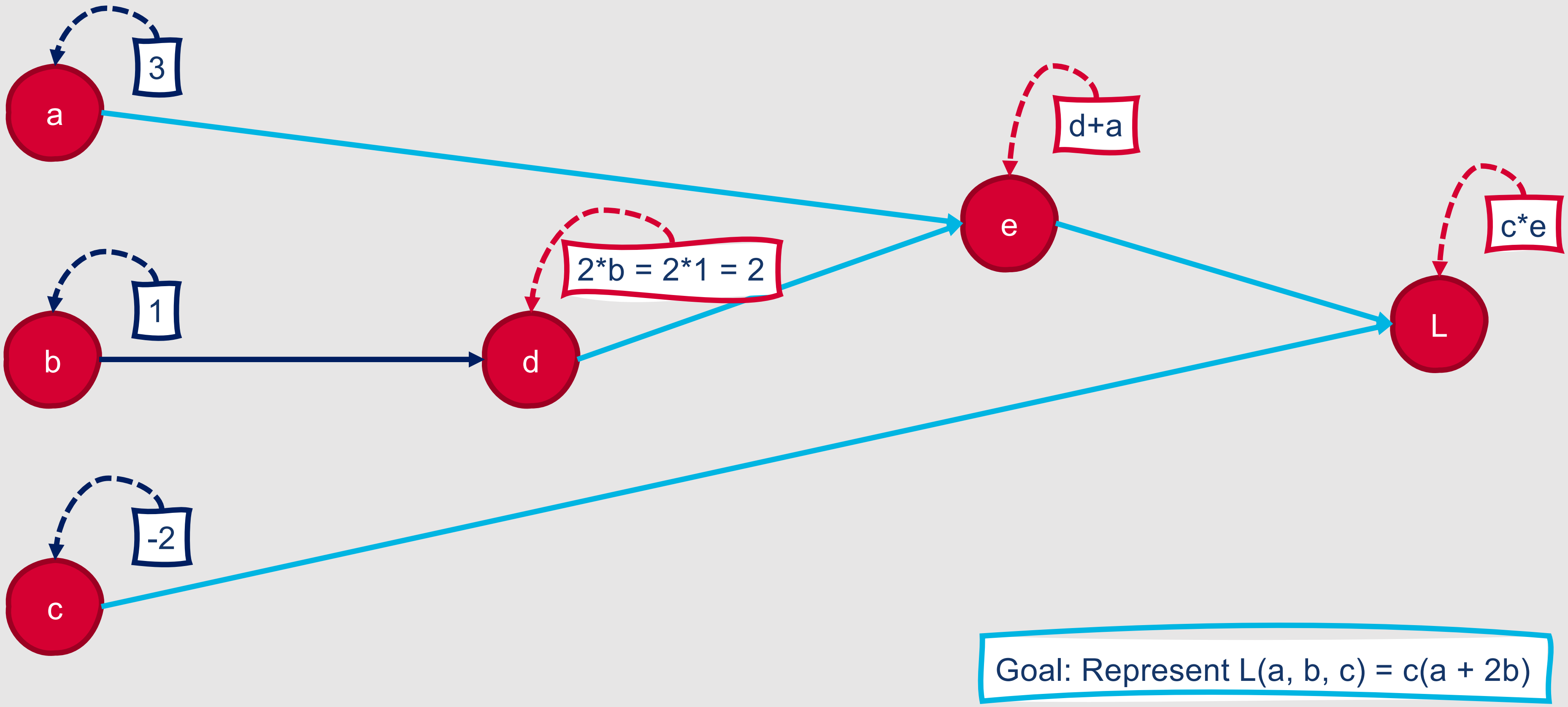
Goal: Represent L(a, b, c) = c(a + 2b)

Example: Forward Pass

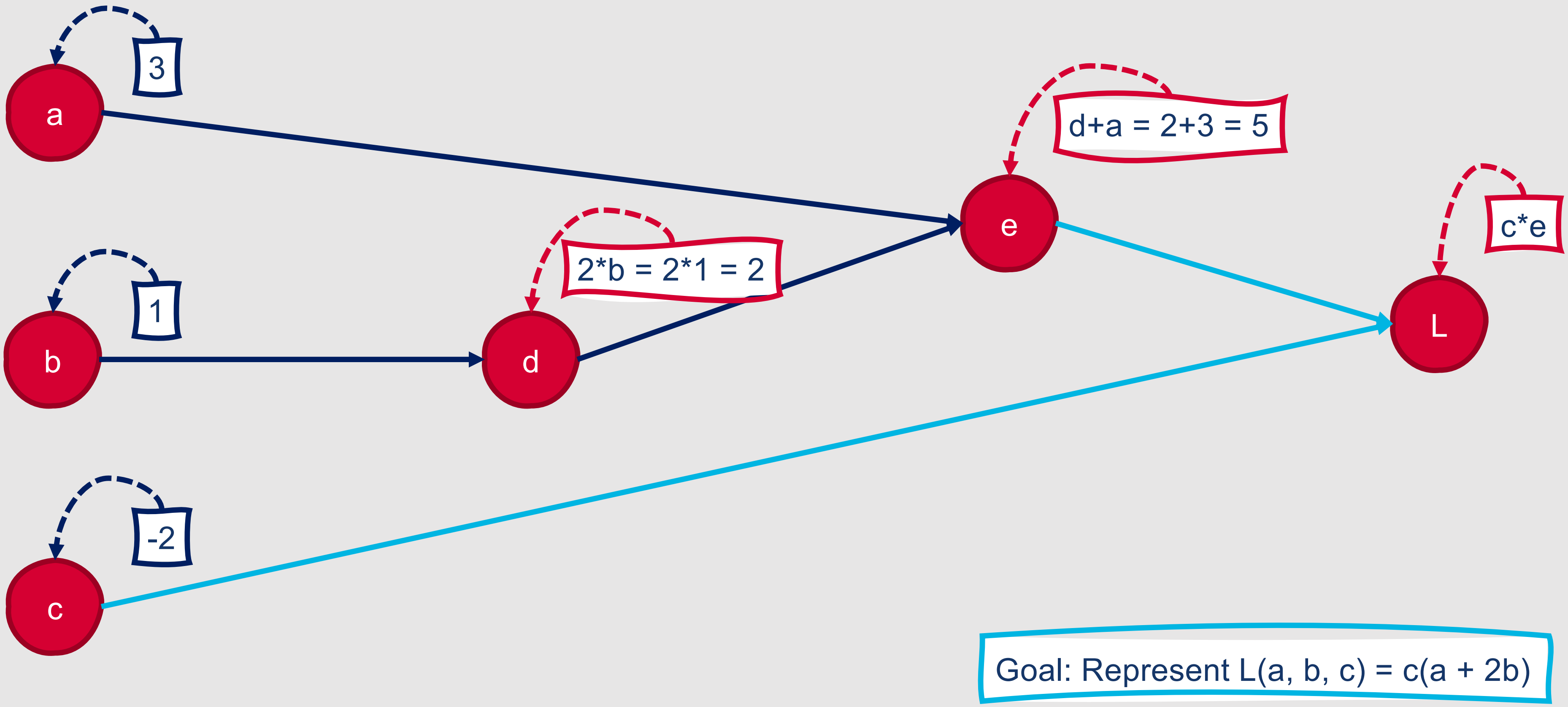


Goal: Represent L(a, b, c) = c(a + 2b)

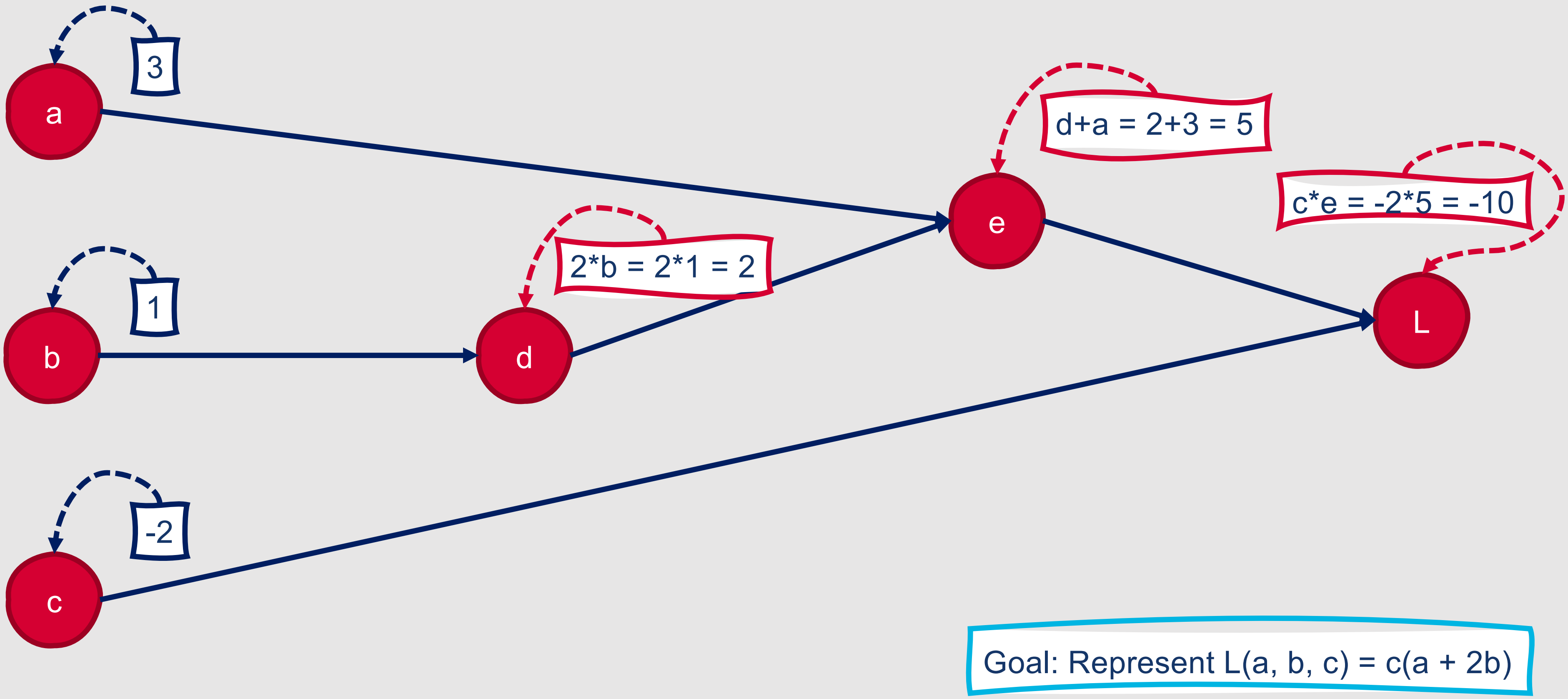
Example: Forward Pass



Example: Forward Pass



Example: Forward Pass



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

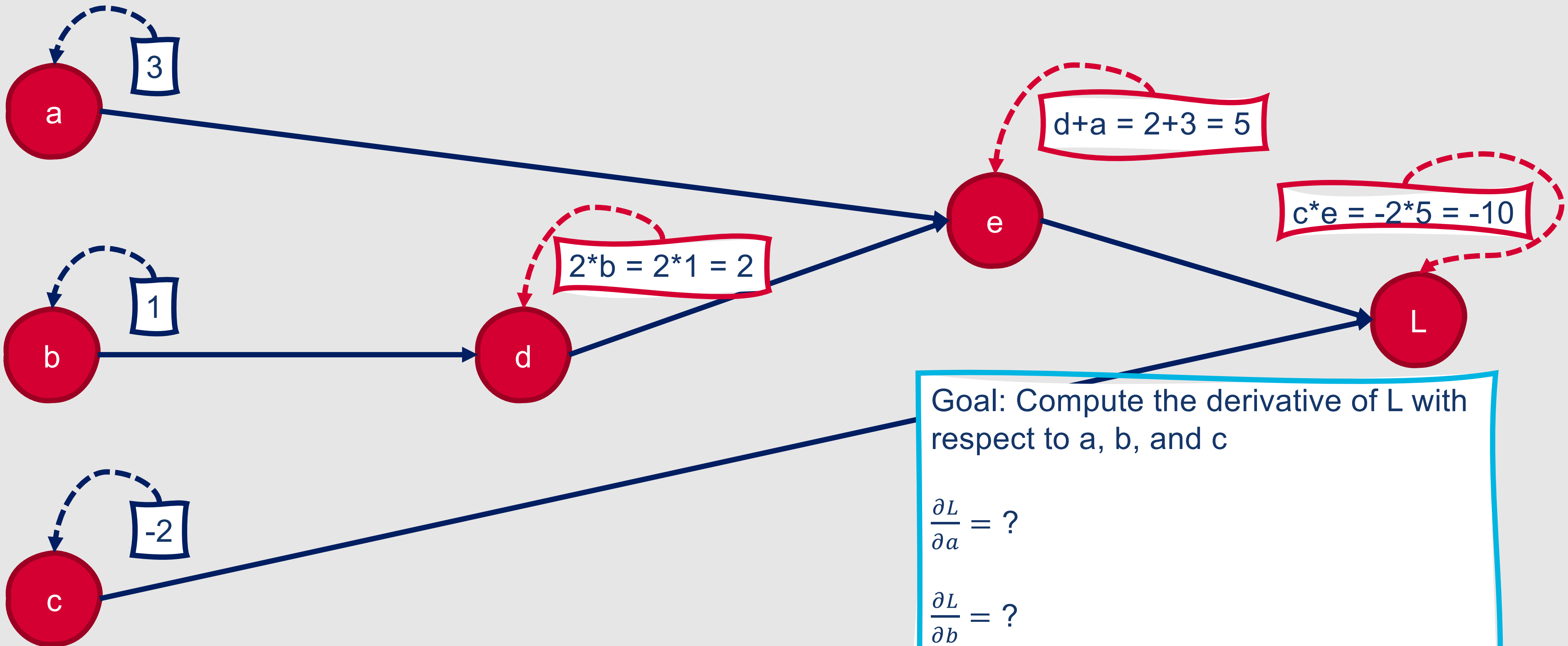
Derivatives of popular activation functions:

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

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

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

Example: Backward Pass



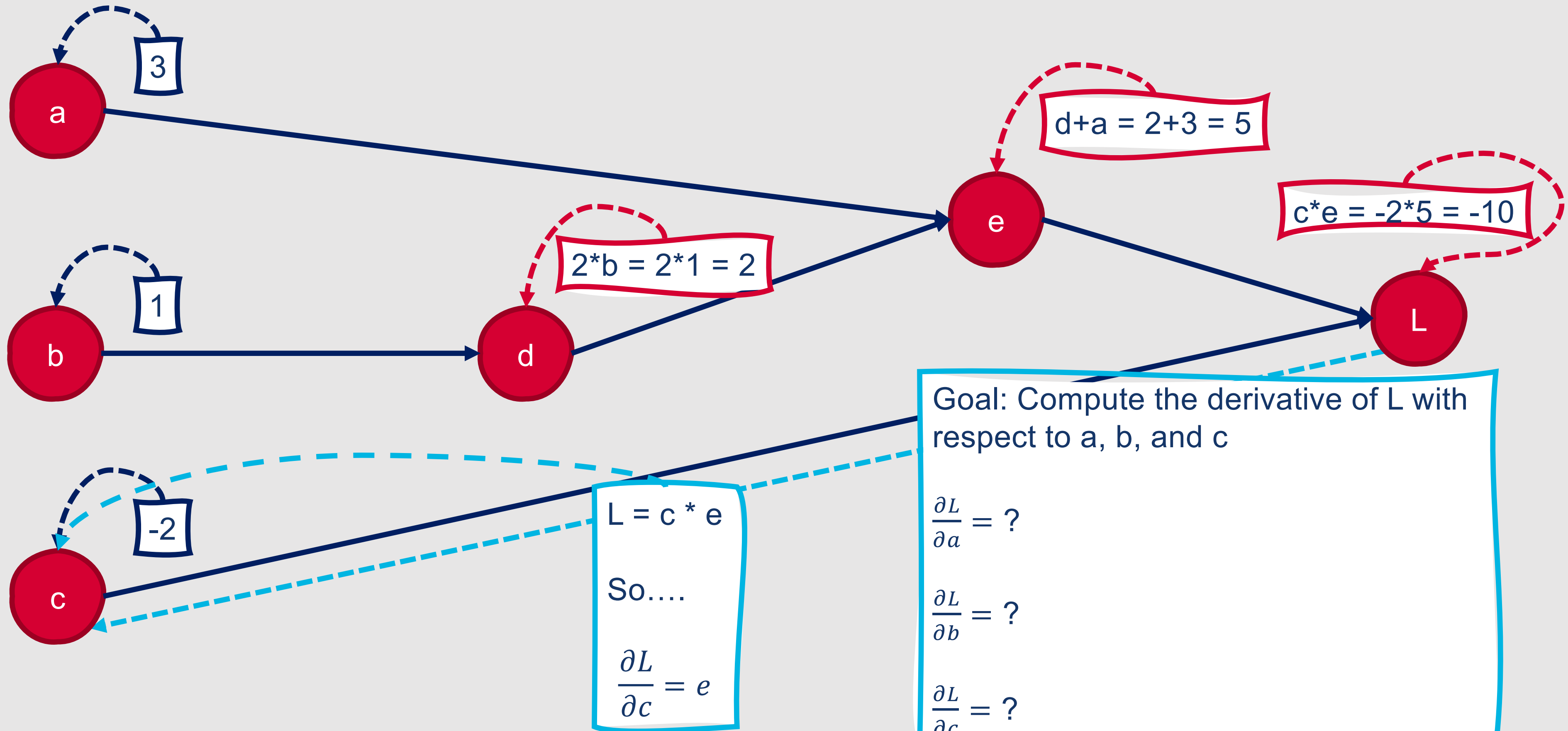
Goal: Compute the derivative of L with respect to a, b, and c

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

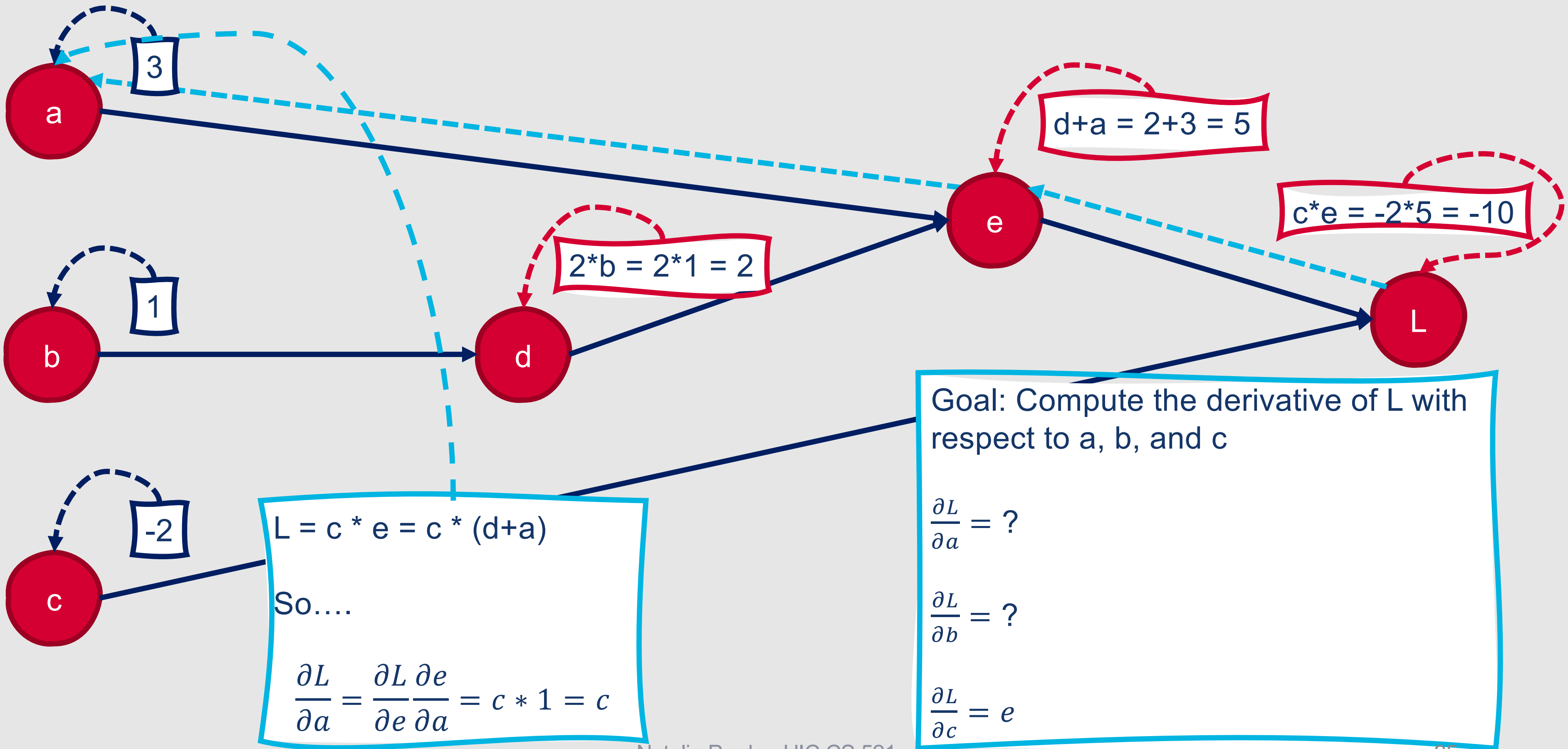
$\frac{\partial L}{\partial b} = ?$

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

Example: Backward Pass



Example: Backward Pass



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

So….

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

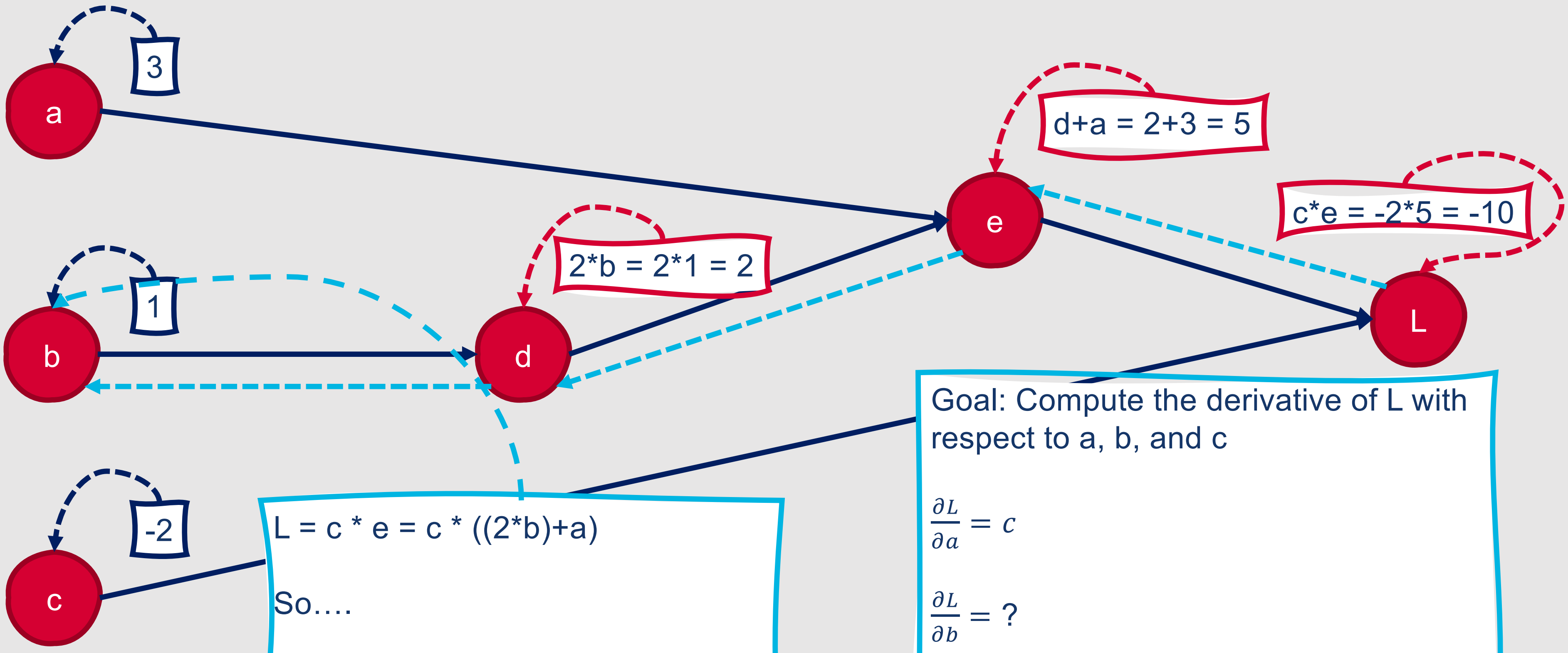
Goal: Compute the derivative of L with respect to a, b, and c

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

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

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

Example: Backward Pass



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

So….

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

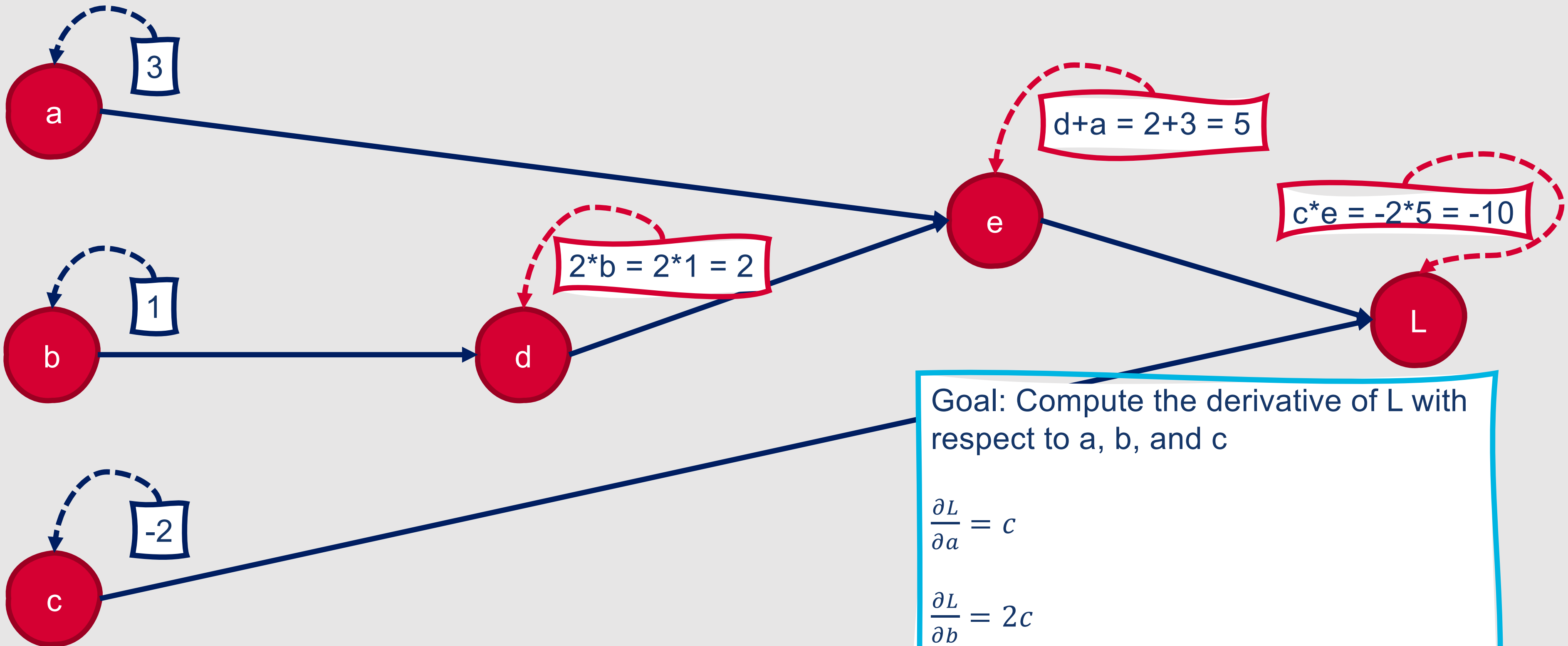
Goal: Compute the derivative of L with respect to a, b, and c

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

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

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

Example: Backward Pass



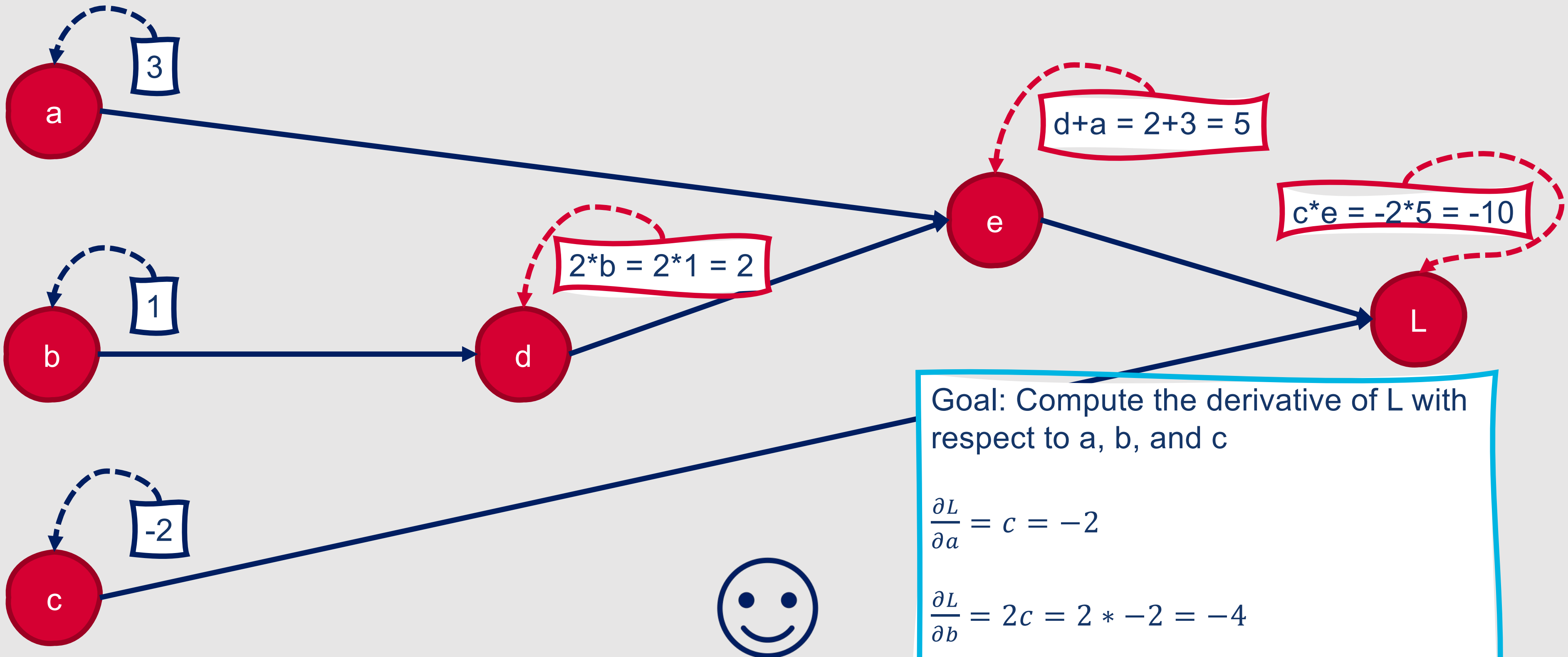
Goal: Compute the derivative of L with respect to a, b, and c

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

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

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

Example: Backward Pass



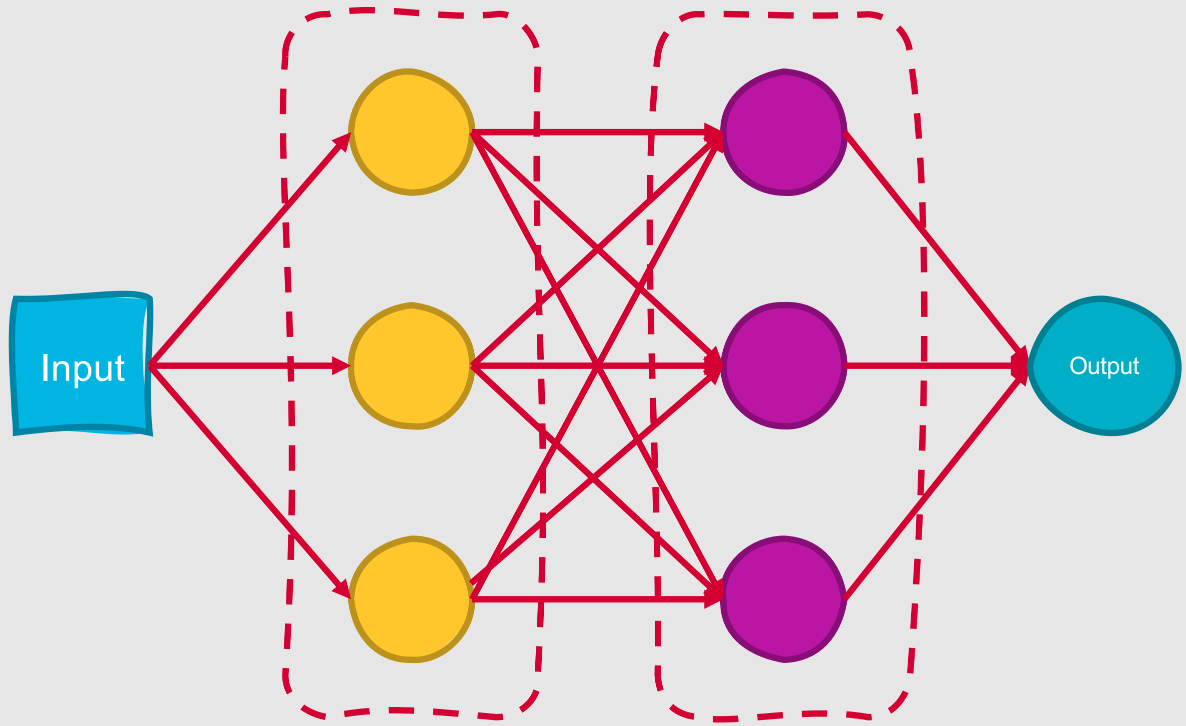
Goal: Compute the derivative of L with respect to a, b, and c

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

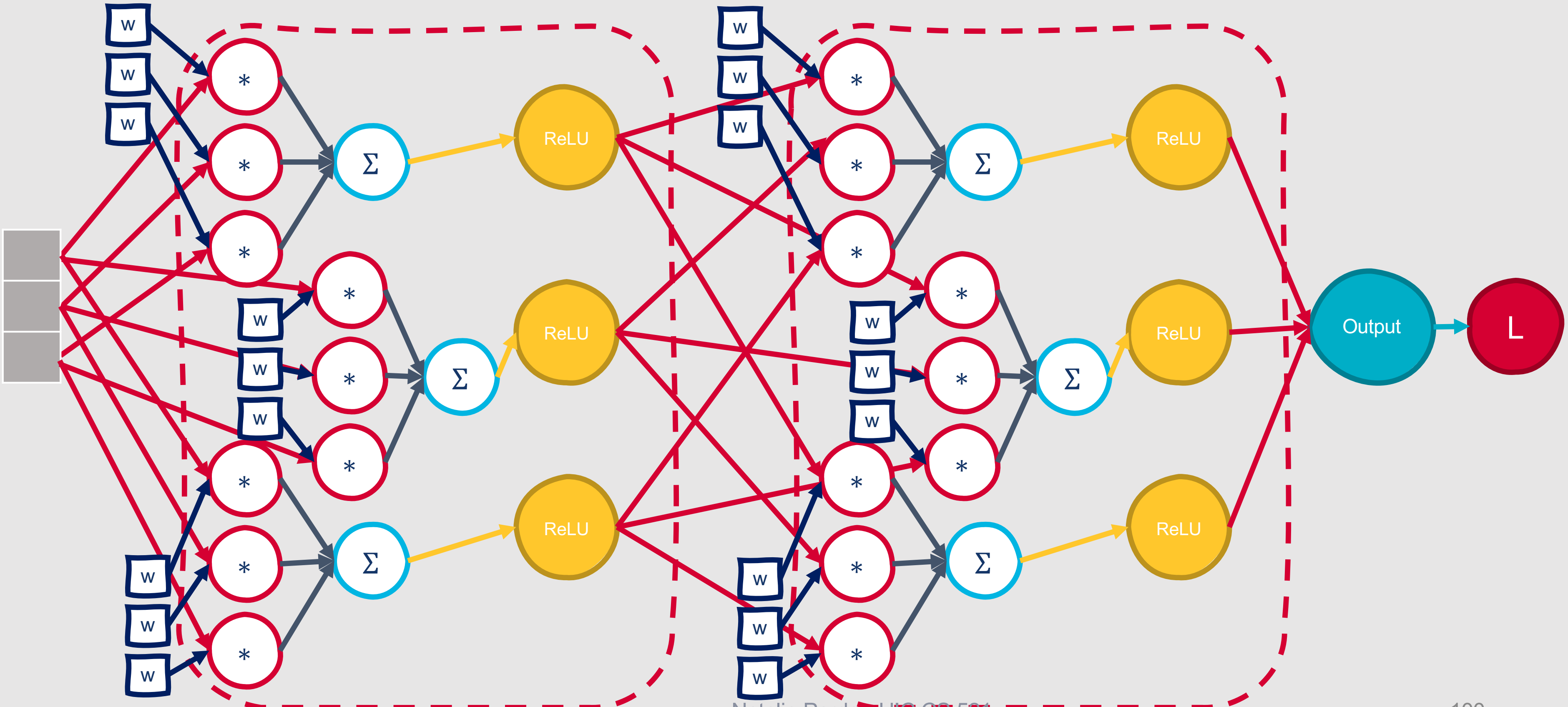
$$\frac{\partial L}{\partial b} = 2c = 2 * -2 = -4$$

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

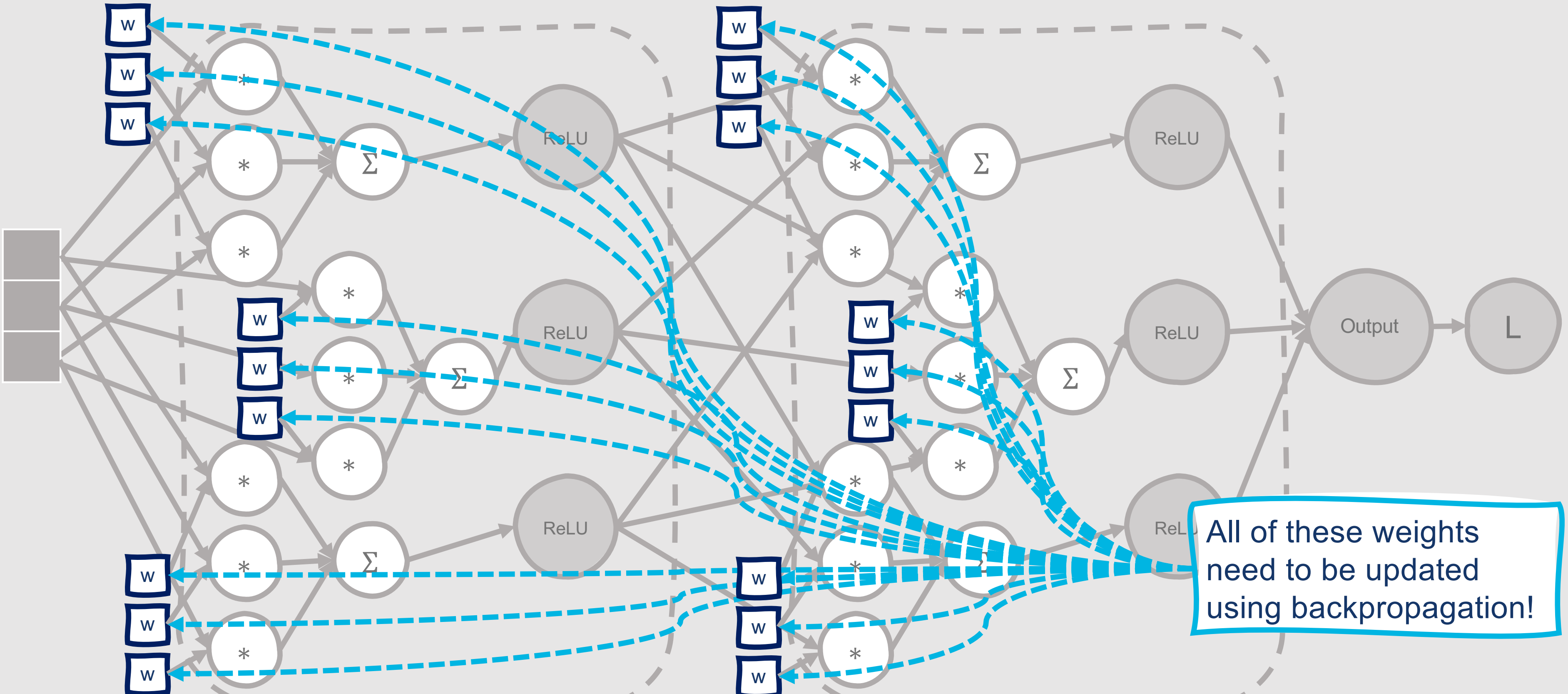
Computation graphs for neural networks involve numerous interconnected units.



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



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



This Week's Topics

Tuesday

- Neural networks
- Computational units
- Combining layers of units

Thursday

- Backpropagation
- Neural language models
- Recurrent neural networks
- Other popular deep learning architectures

Neural Language Models

- Popular application of neural networks
- Advantages over *n*-gram language models:
  - Can handle longer histories
  - Can generalize over contexts of similar words
- Disadvantage:
  - Slower to train
- Neural language models make more accurate predictions than n-gram language models trained on datasets of similar sizes

Feedforward Neural Language Model

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

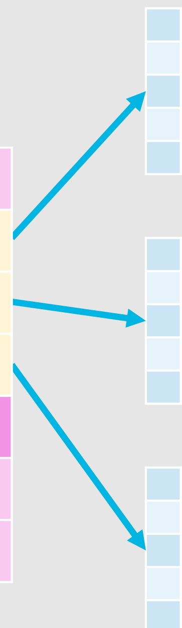
Neural Language Model

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

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

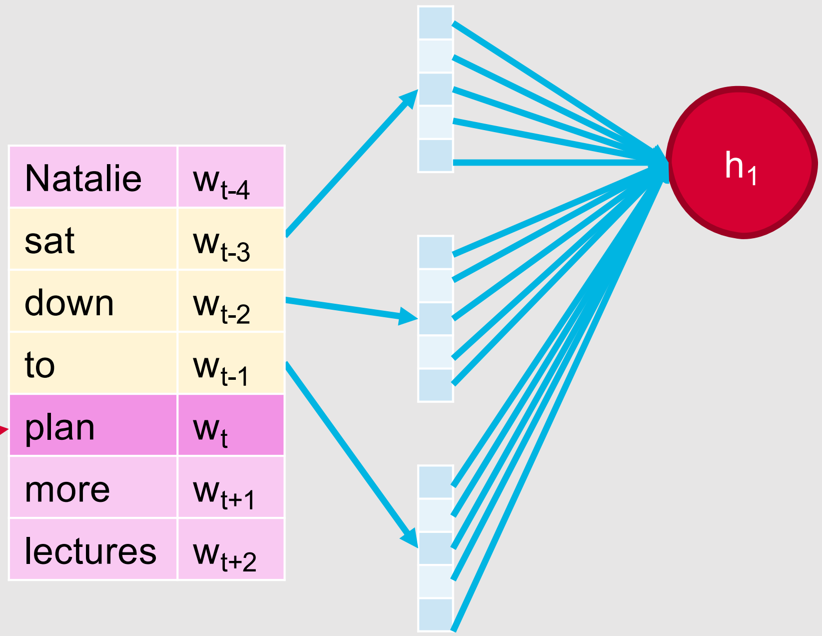
Neural Language Model

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



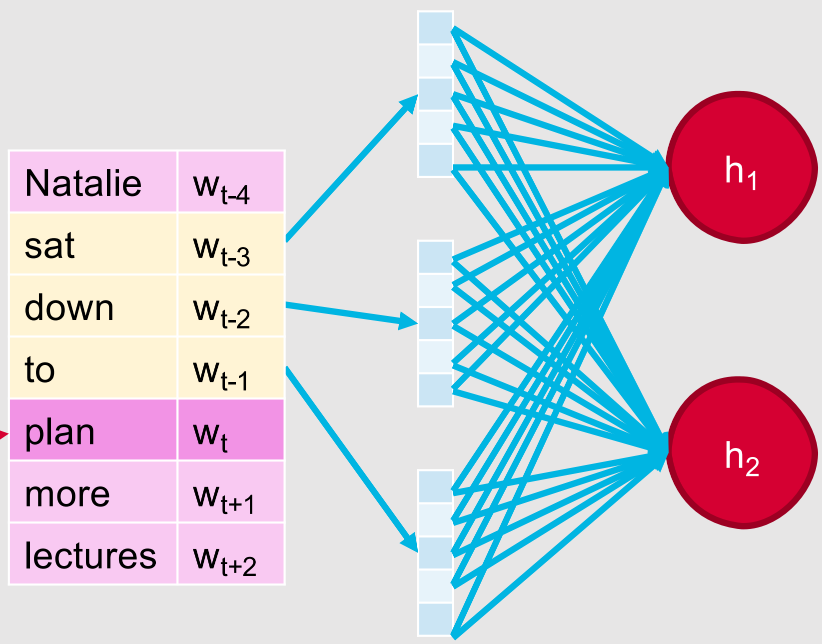
$$P(w_t = \text{“plan”} | w_{t-1} = \text{“to”}, w_{t-2} = \text{“down”}, w_{t-3} = \text{“sat”})$$

Neural Language Model



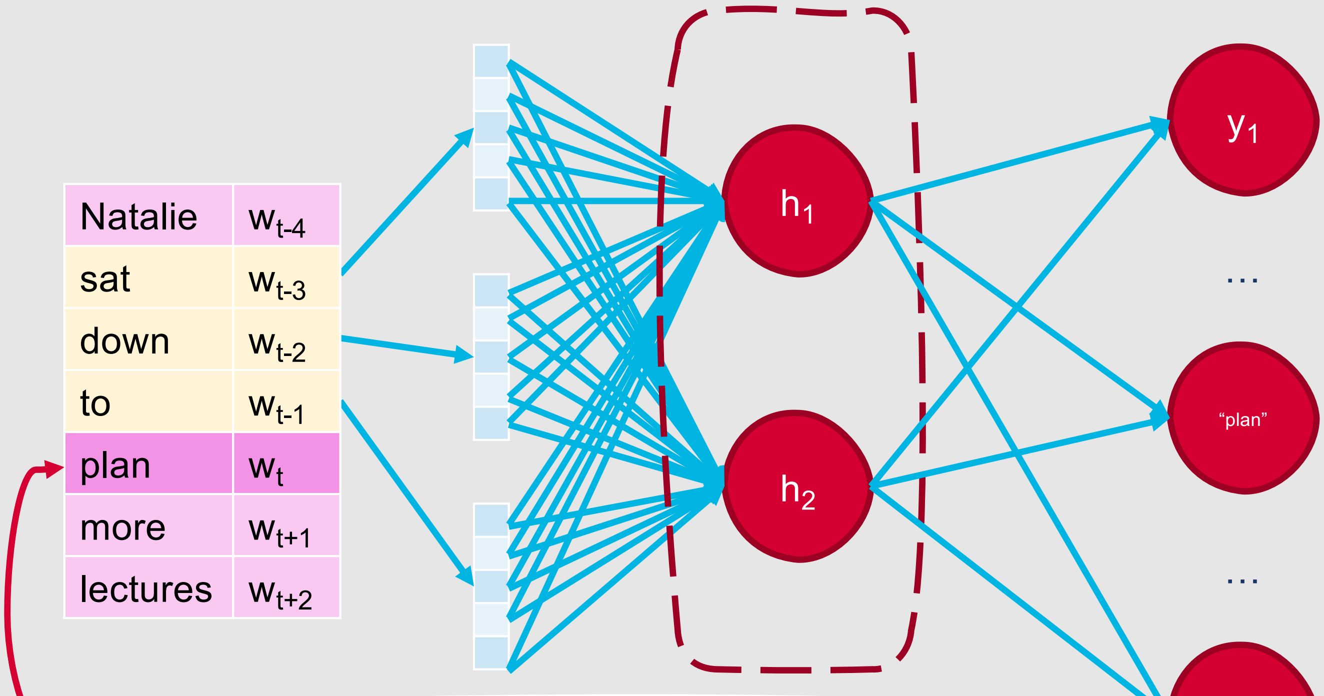
$$P(w_t = \text{“plan”} | w_{t-1} = \text{“to”}, w_{t-2} = \text{“down”}, w_{t-3} = \text{“sat”})$$

Neural Language Model



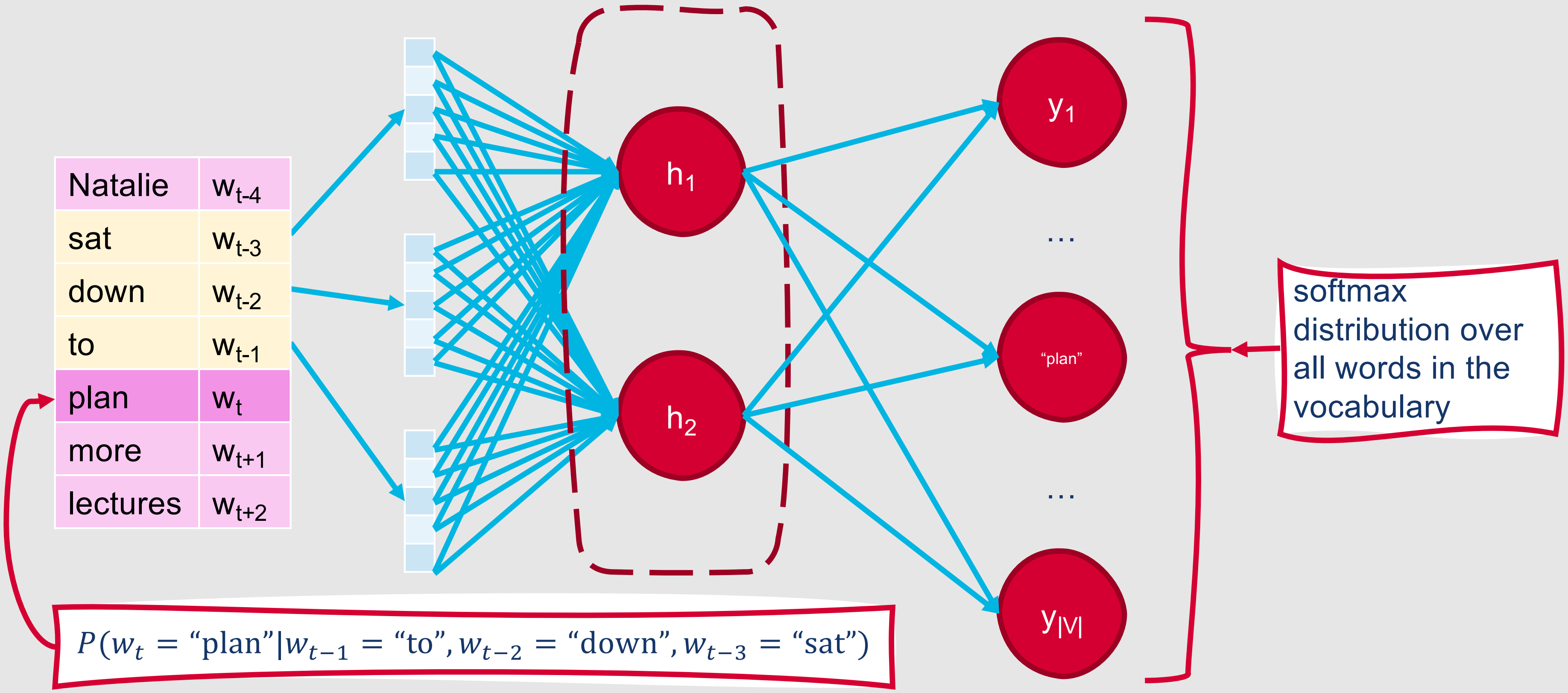
$$P(w_t = \text{“plan”} | w_{t-1} = \text{“to”}, w_{t-2} = \text{“down”}, w_{t-3} = \text{“sat”})$$

Neural Language Model



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

Neural Language Model



This Week's Topics

Tuesday

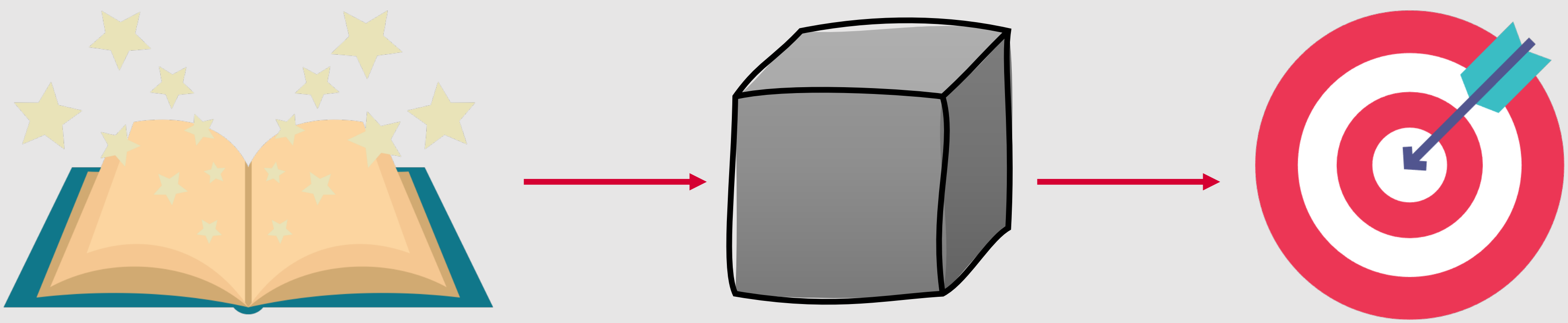
Neural networks
Computational units
Combining layers of units

Thursday

Backpropagation
Neural language models
Recurrent neural networks
Other popular deep learning architectures

Popular Deep Learning Architectures in Contemporary NLP

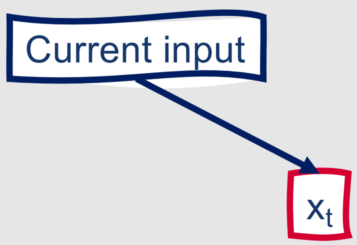
- **Recurrent Neural Networks**
- Convolutional Neural Networks
- Transformers



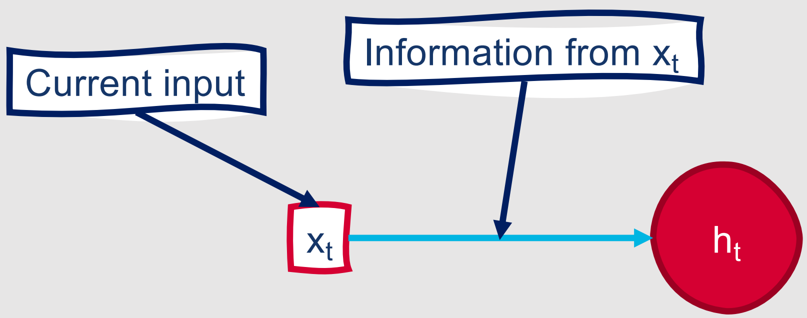
Recurrent Neural Networks (RNNs)

- General premise:
  - Deep learning models should be making decisions for sequential input based on decisions that have already been made at earlier points of the sequence
- Classic feedforward neural network:
  - Input to a layer is a vector of numbers representing the outputs of all units in the previous layer
- Modification for recurrent neural networks:
  - Input to a layer is a vector of numbers representing the outputs of all units in the previous layer **+ a vector of numbers representing the layer's output at the previous timestep**

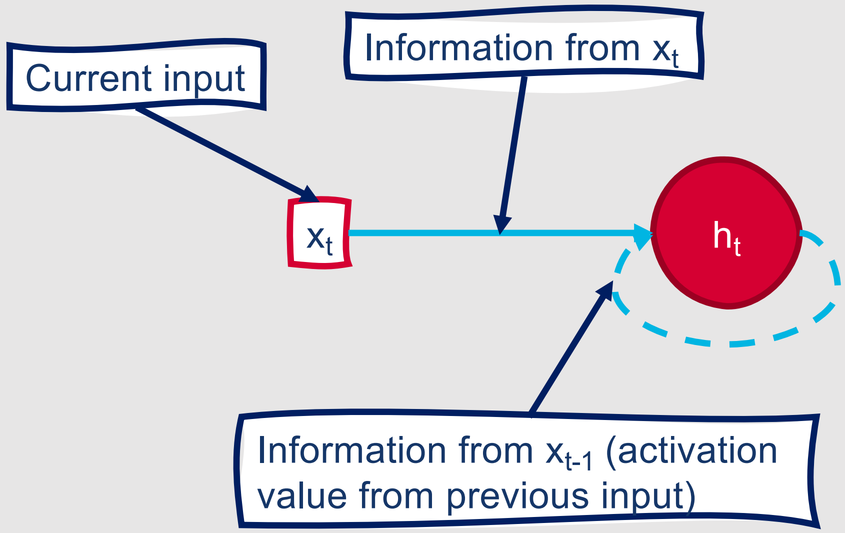
Structure of Single-Unit RNN Layer



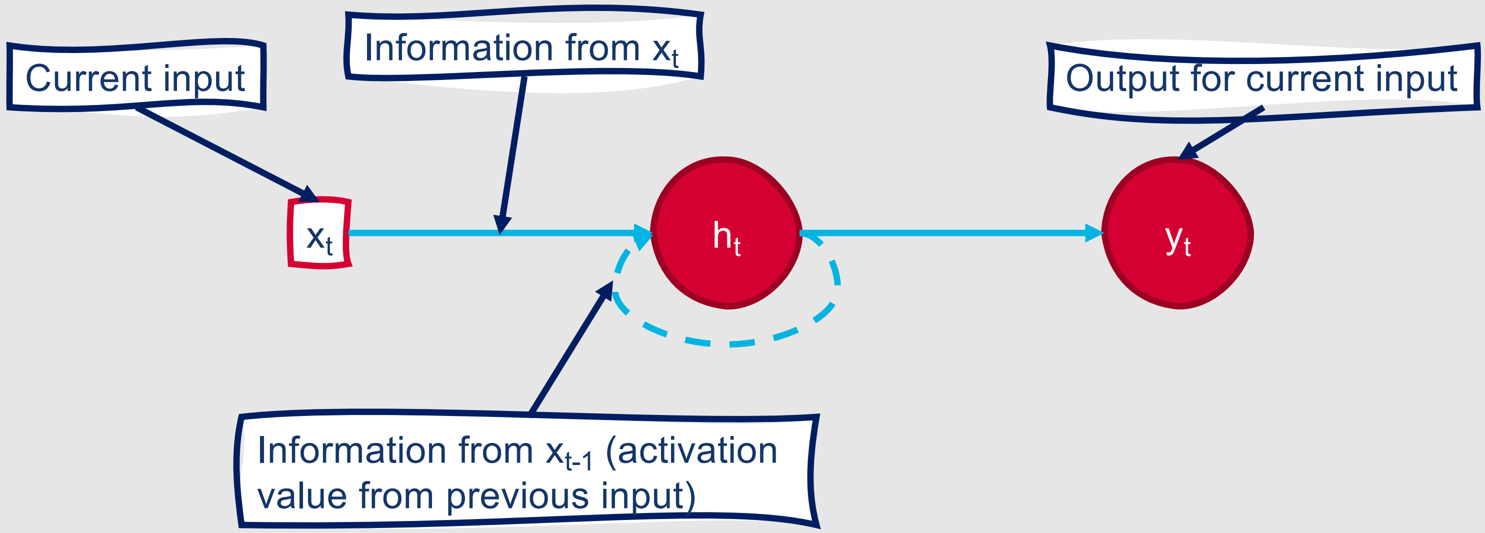
Structure of Single-Unit RNN Layer



Structure of Single-Unit RNN Layer



Structure of Single-Unit RNN Layer

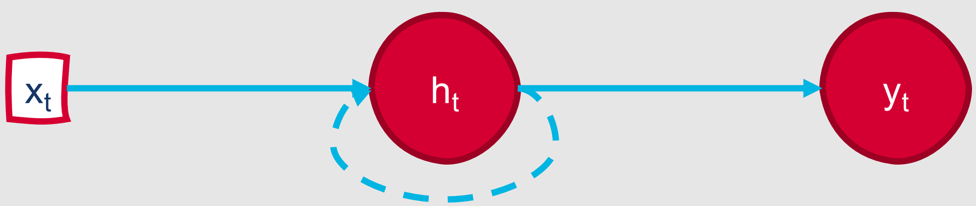


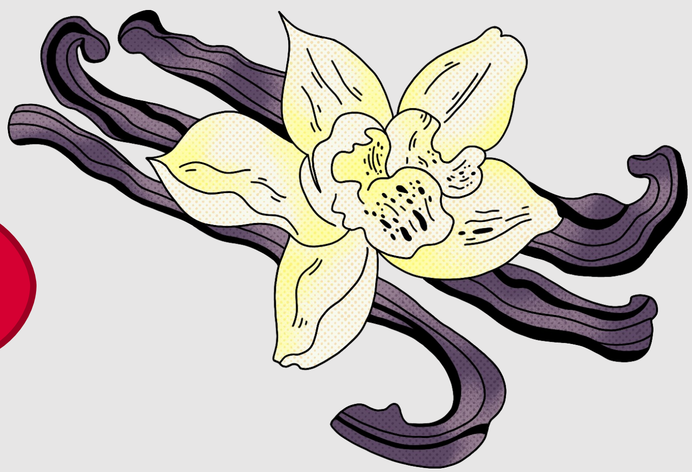
Why is this useful for NLP problems?

- Most data for NLP tasks is inherently sequential!
- Making use of sequences using feedforward neural networks requires:
  - Fixed-length context windows
  - Concatenated context vectors
- This limits the model's abilities, and prevents it from considering variable-length context

There are many popular variations of RNNs.

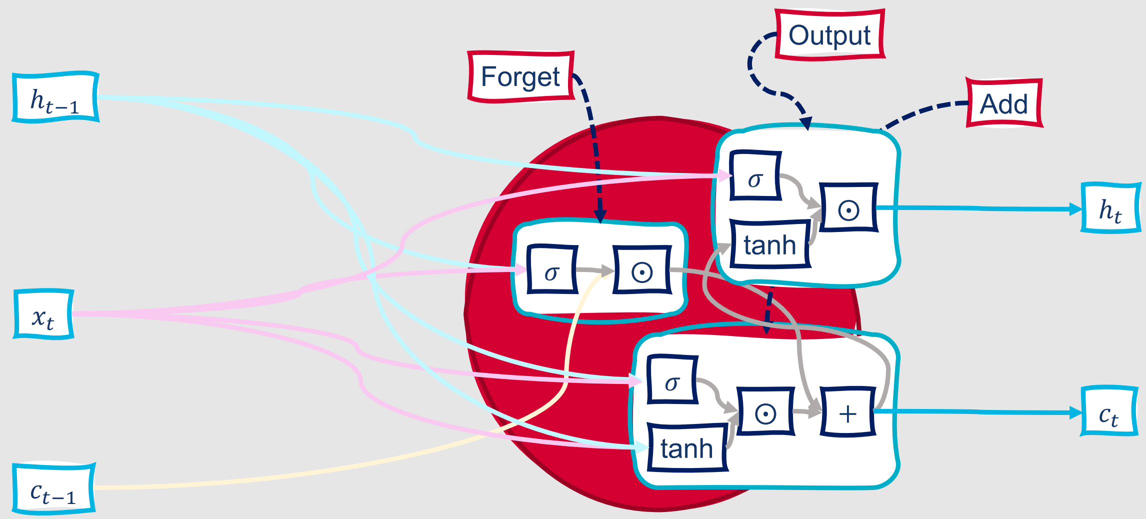
- "Standard" RNNs are often referred to informally as **vanilla RNNs**
- Some RNN architectures are modified to specifically improve the model's ability to consider long-term context
    - **Long short-term memory networks** (LSTMs)
    - **Gated recurrent units** (GRUs)





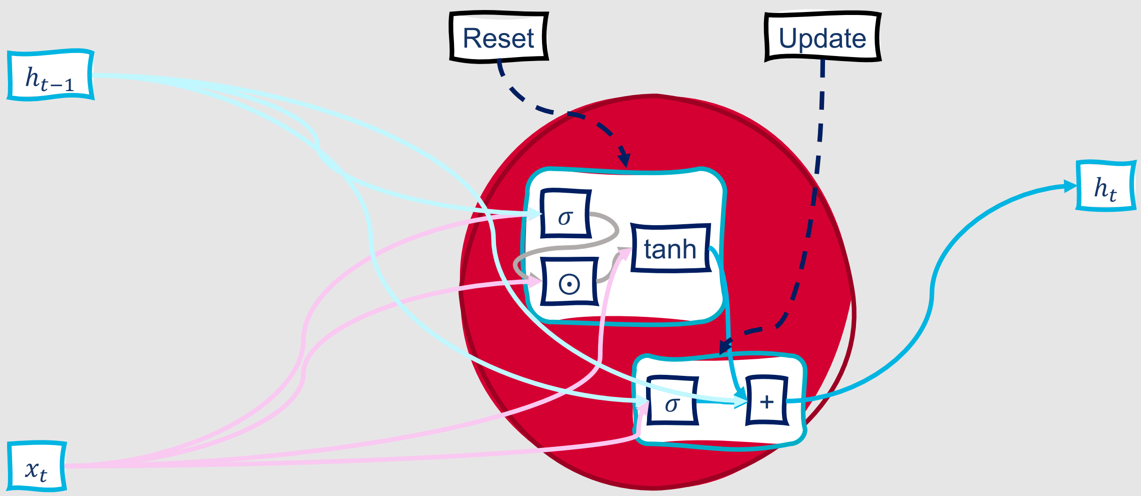
Long Short-Term Memory Networks (LSTMs)

- Specialized RNN units that incorporate gating mechanisms to remove information that is no longer needed from the context, and add information that is anticipated to be of use later
- Gating mechanisms include:
  - **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?

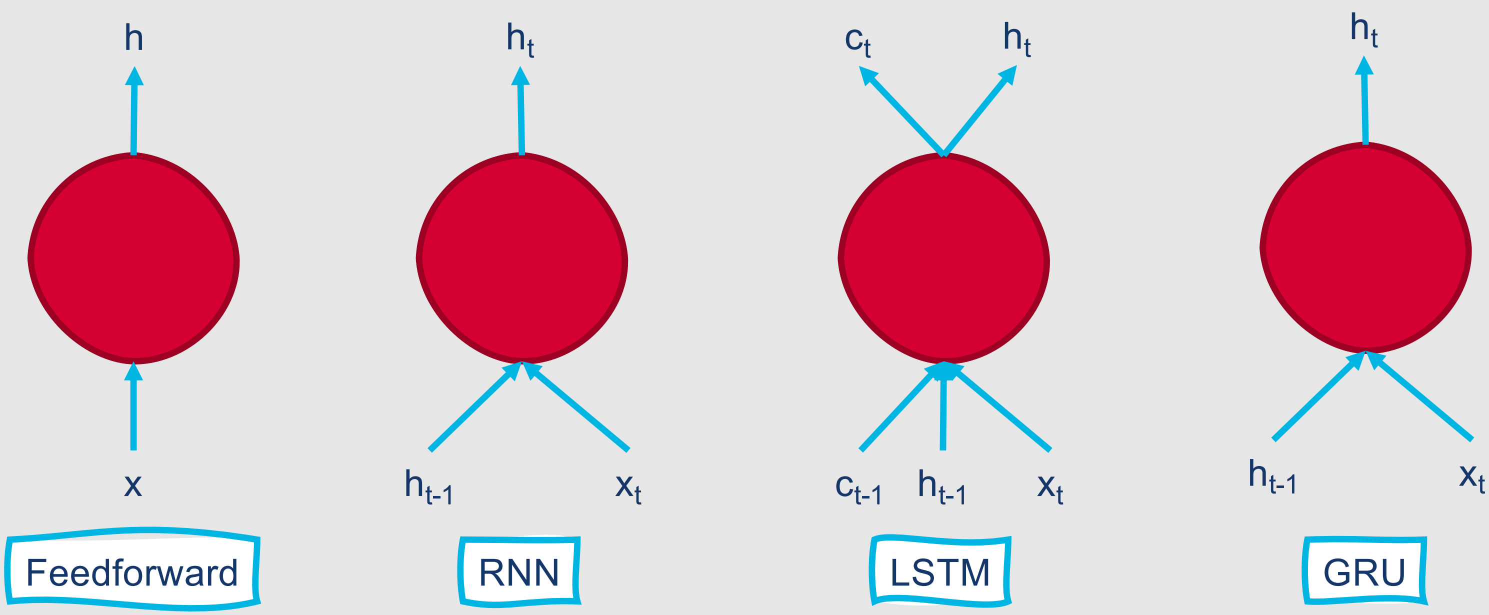


Gated Recurrent Units (GRUs)

- Also utilizes gating mechanisms to manage contexts, but uses a simpler architecture than LSTMs
- Only two gates:
  - **Reset gate:** Which elements of the previous hidden state are relevant to the current context?
  - **Update gate:** Which elements of the intermediate hidden state and of the previous hidden state need to be preserved for future use?



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

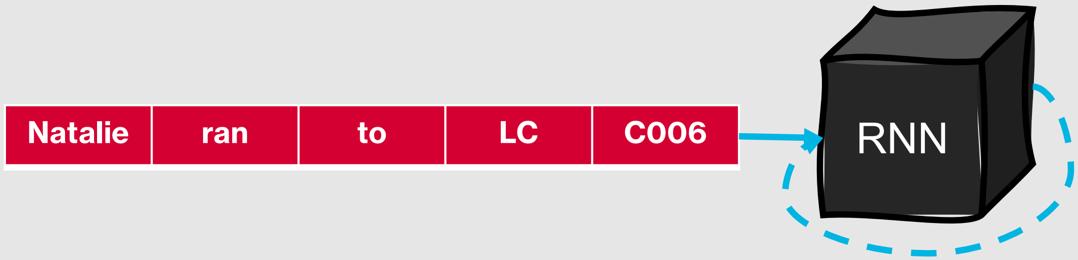
Why use LSTMs instead of GRUs?

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

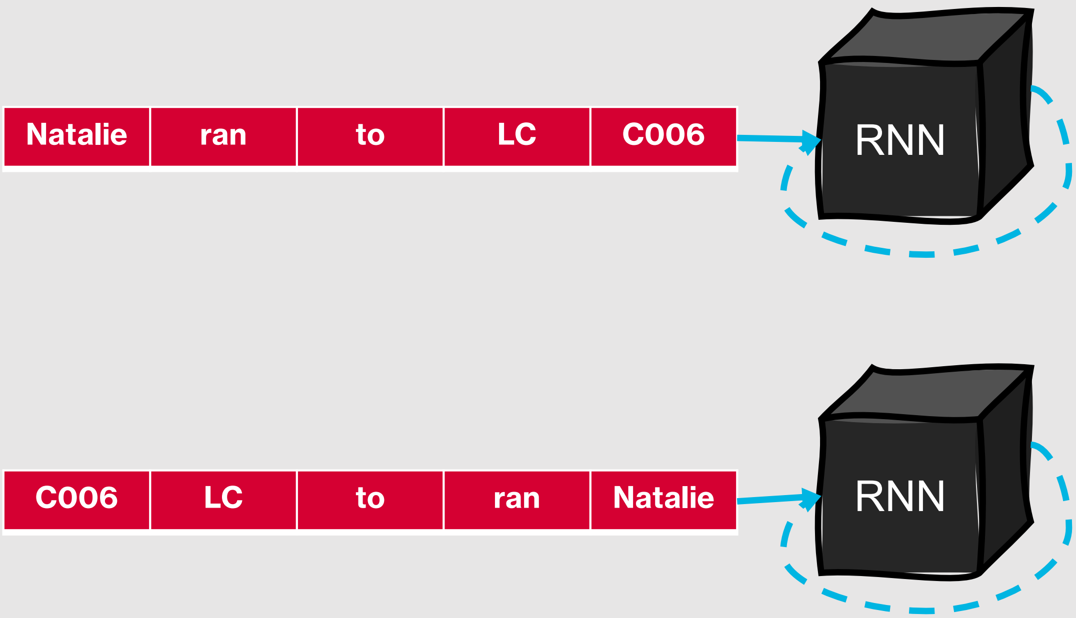
Bidirectional Models

- All RNN units can be combined with one another in the same way that feedforward units can be combined
  - Layers of vanilla RNN units
  - Layers of LSTM units
  - Layers of GRU units
- These layers can also be combined to implement **bidirectional** architectures that process input both from beginning to end and from end to beginning

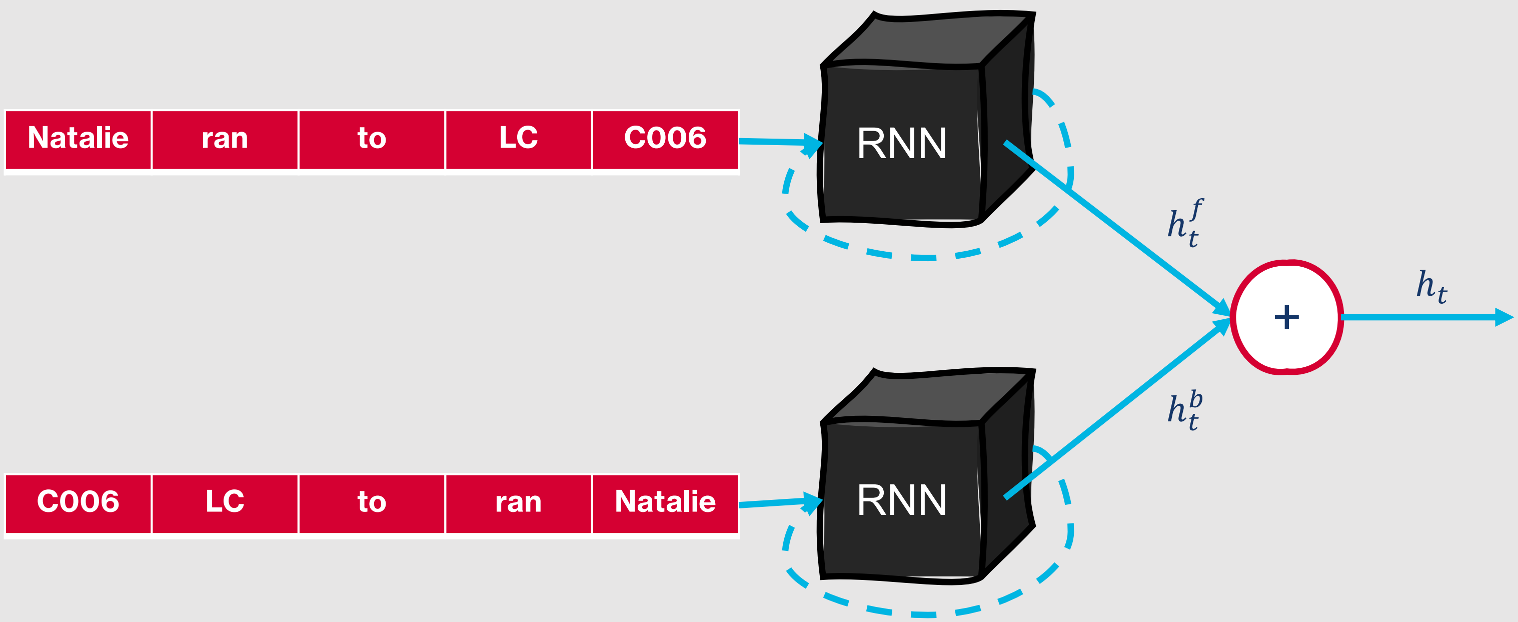
Bidirectional RNNs



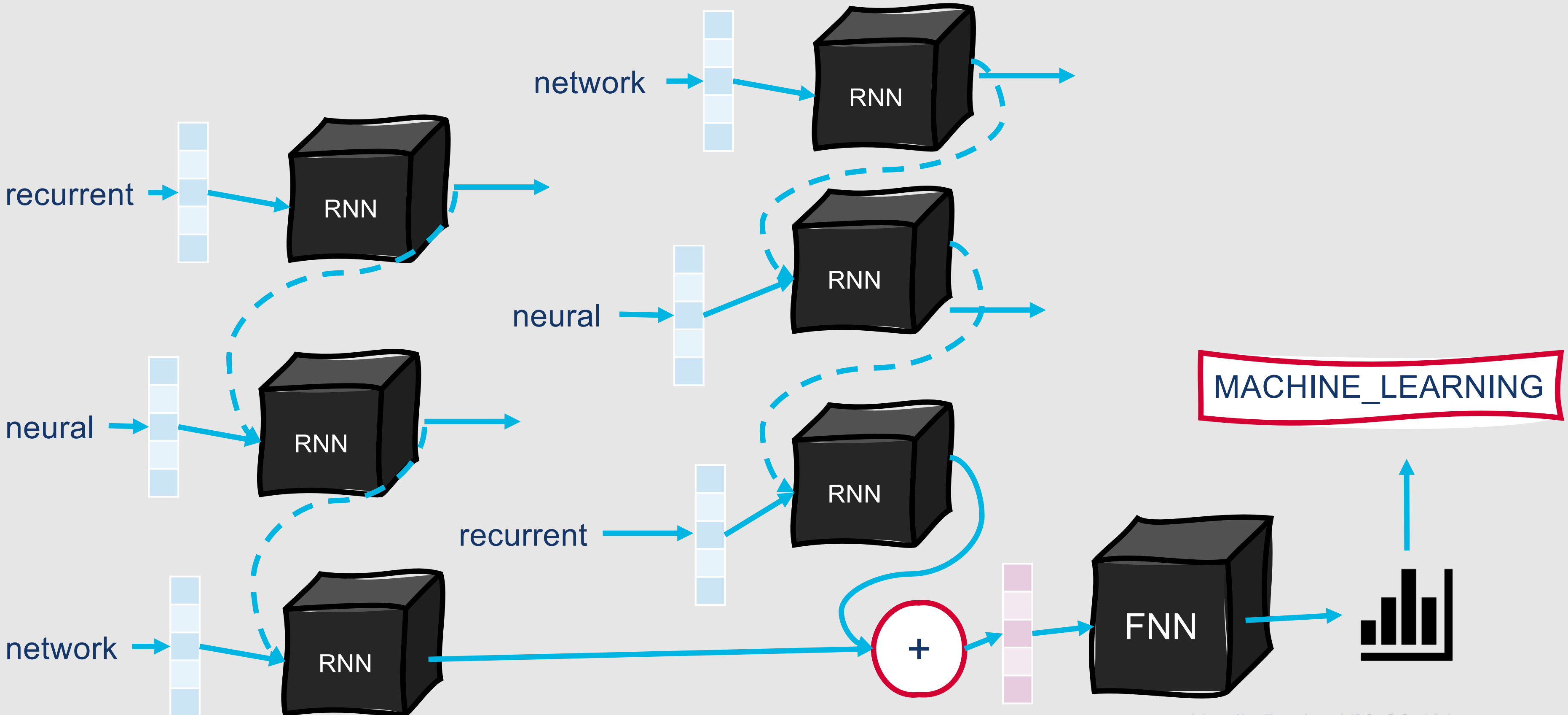
Bidirectional RNNs



Bidirectional RNNs



Sequence Classification with a Bidirectional RNN



This Week's Topics

Tuesday

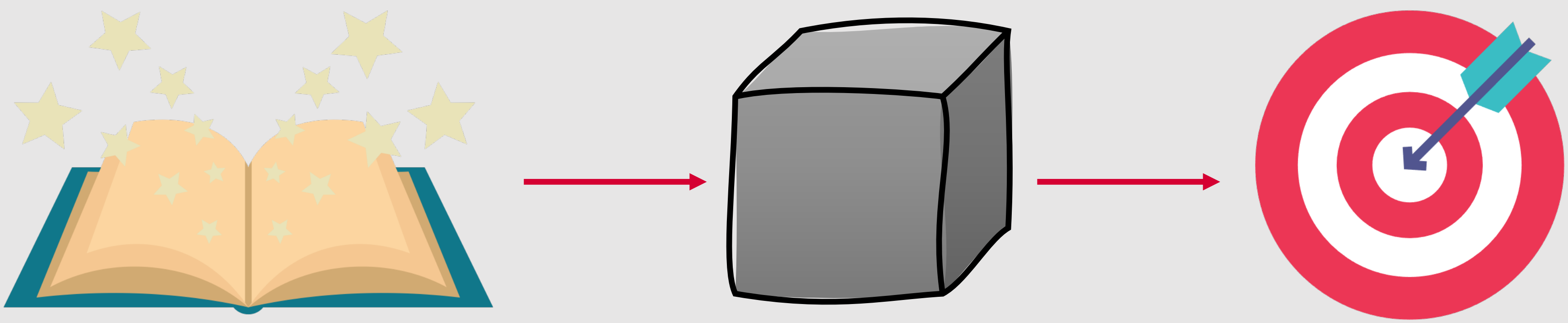
- Neural networks
- Computational units
- Combining layers of units

Thursday

- Backpropagation
- Neural language models
- Recurrent neural networks
- Other popular deep learning architectures

Popular Deep Learning Architectures in Contemporary NLP

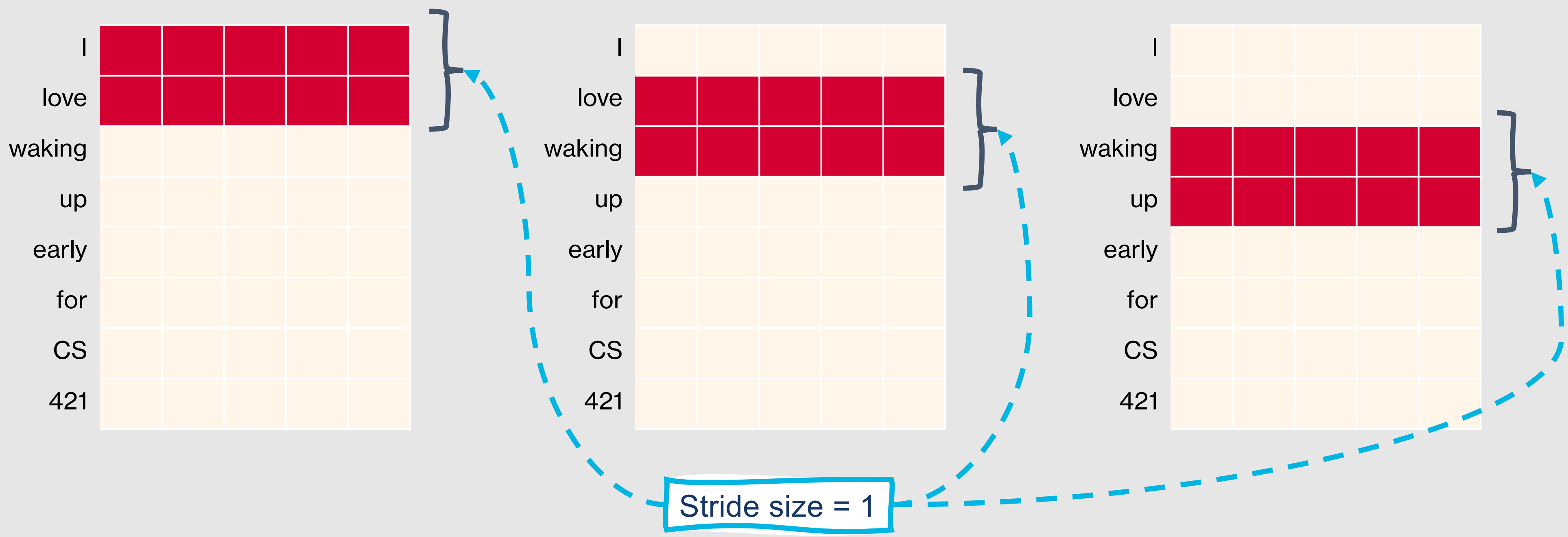
- Recurrent Neural Networks
- **Convolutional Neural Networks**
- Transformers



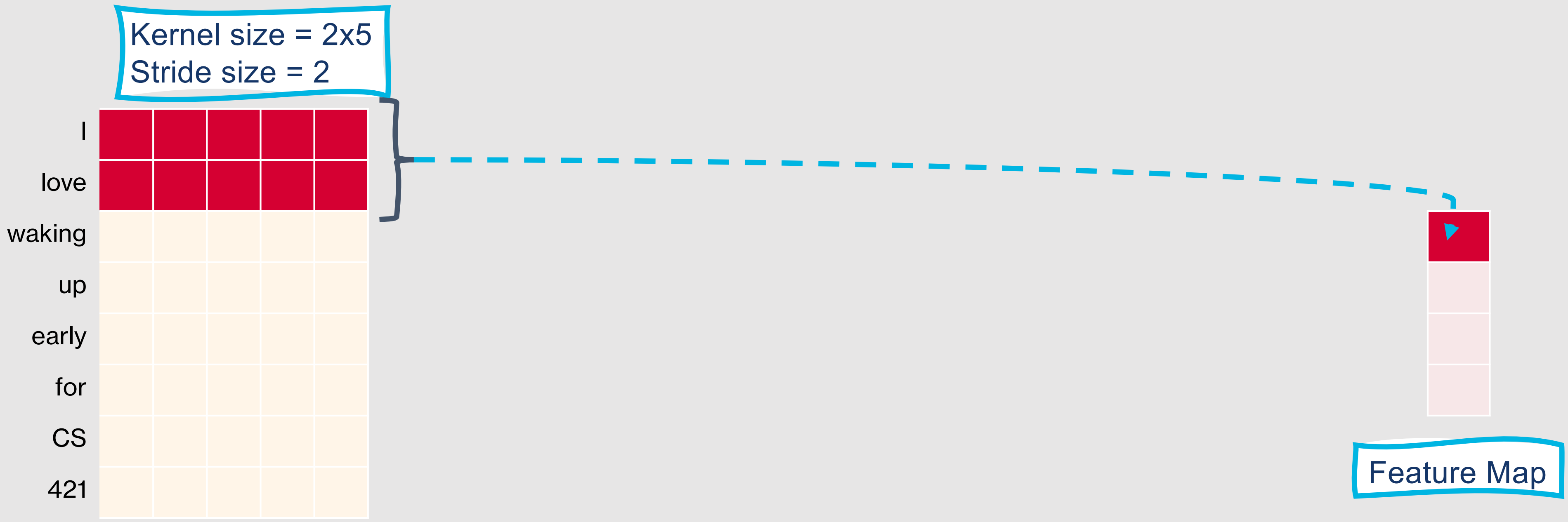
Convolutional Neural Networks (CNNs)

- General premise:
  - Deep learning models should be making decisions based on local regions of the context
- Classic feedforward neural network:
  - Input to a layer is a vector of numbers representing the outputs of all units in the previous layer
- Modification for convolutional neural networks:
  - Input to a layer is the output of **convolutional operations performed on subsets of the output** from the previous layer

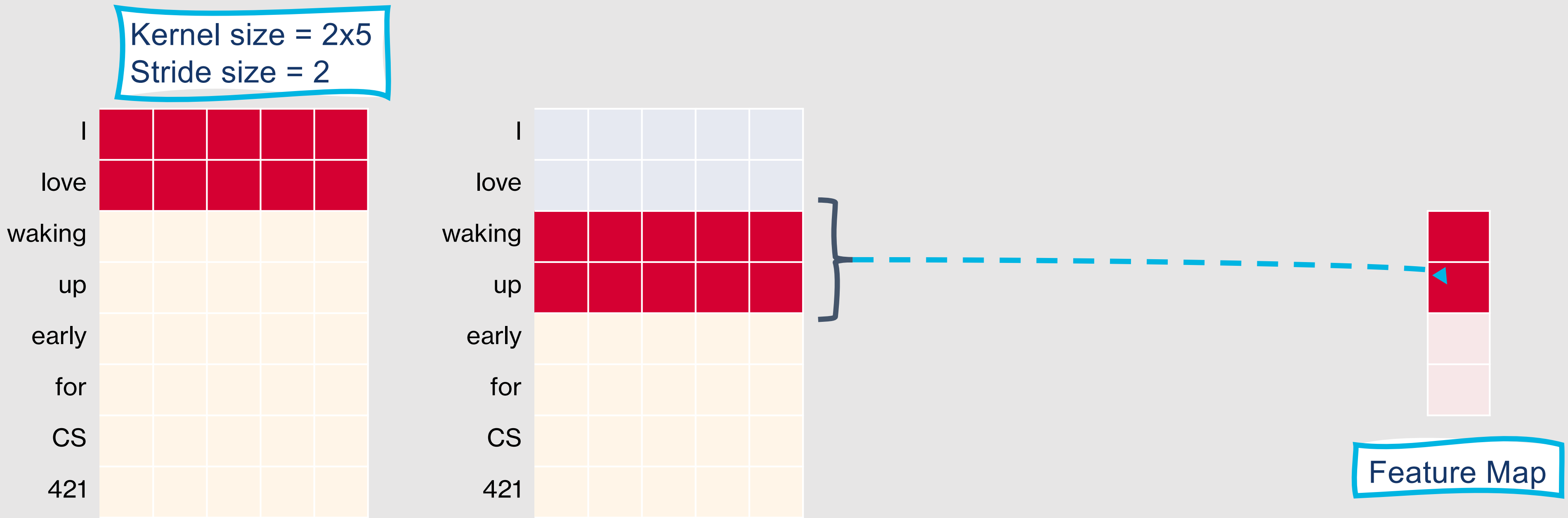
In NLP, convolutions are typically performed on entire rows of an input matrix, where each row corresponds to a word.



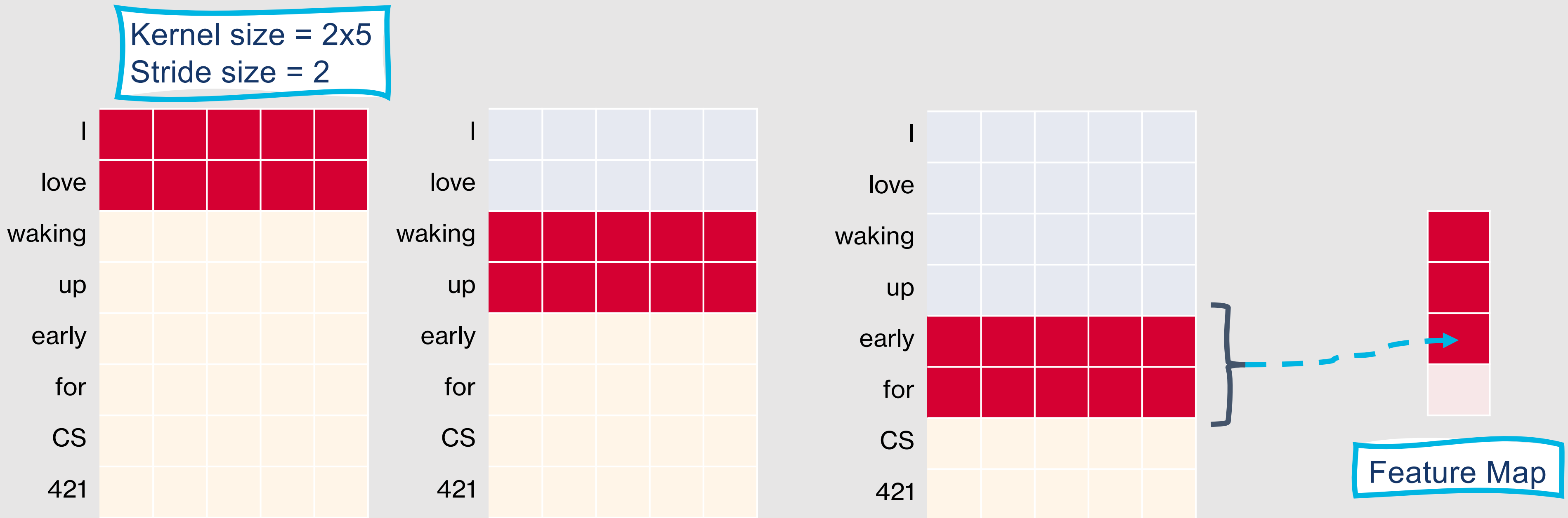
We apply convolutions with specific region (kernel) and stride sizes to an input matrix, and end up with a feature map.



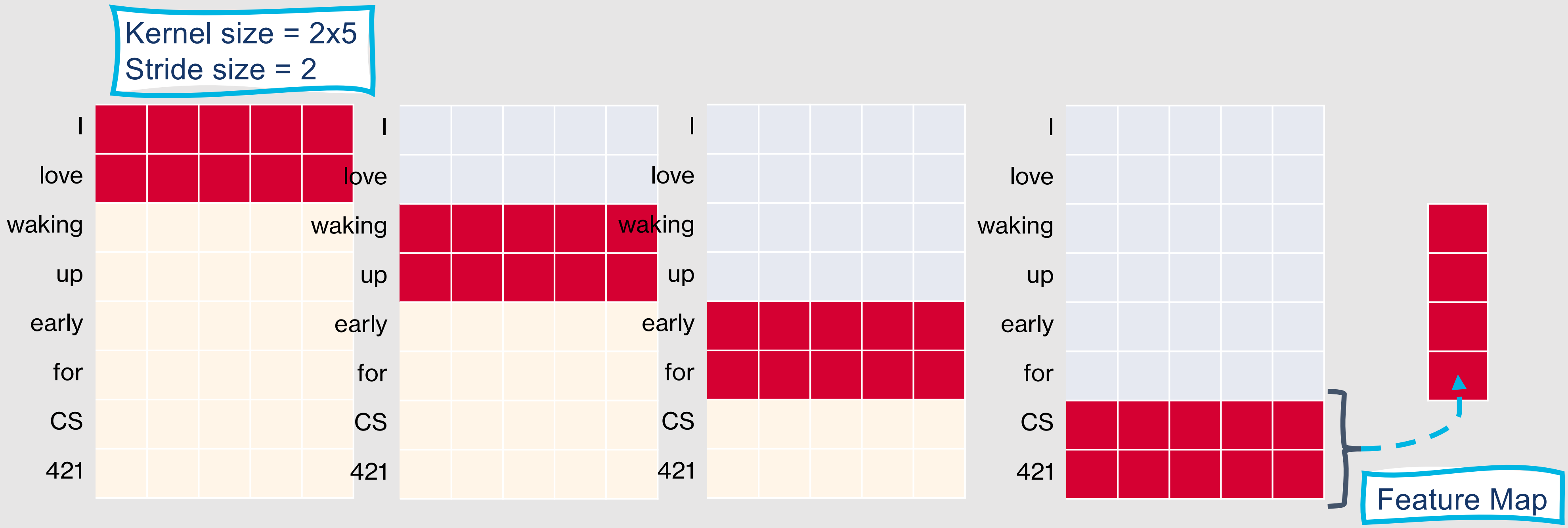
We apply convolutions with specific region (kernel) and stride sizes to an input matrix, and end up with a feature map.



We apply convolutions with specific region (kernel) and stride sizes to an input matrix, and end up with a feature map.

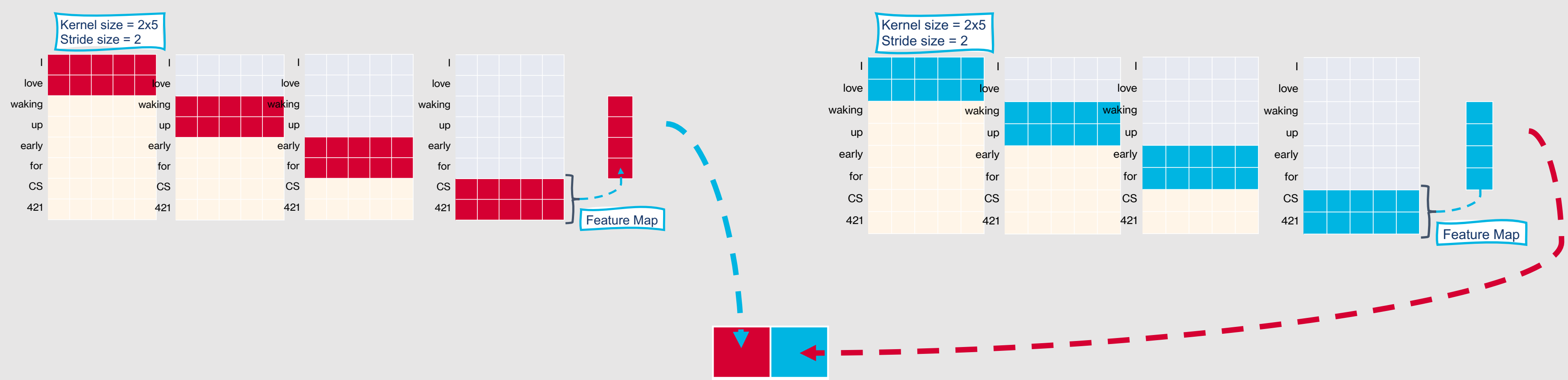


We apply convolutions with specific region (kernel) and stride sizes to an input matrix, and end up with a feature map.

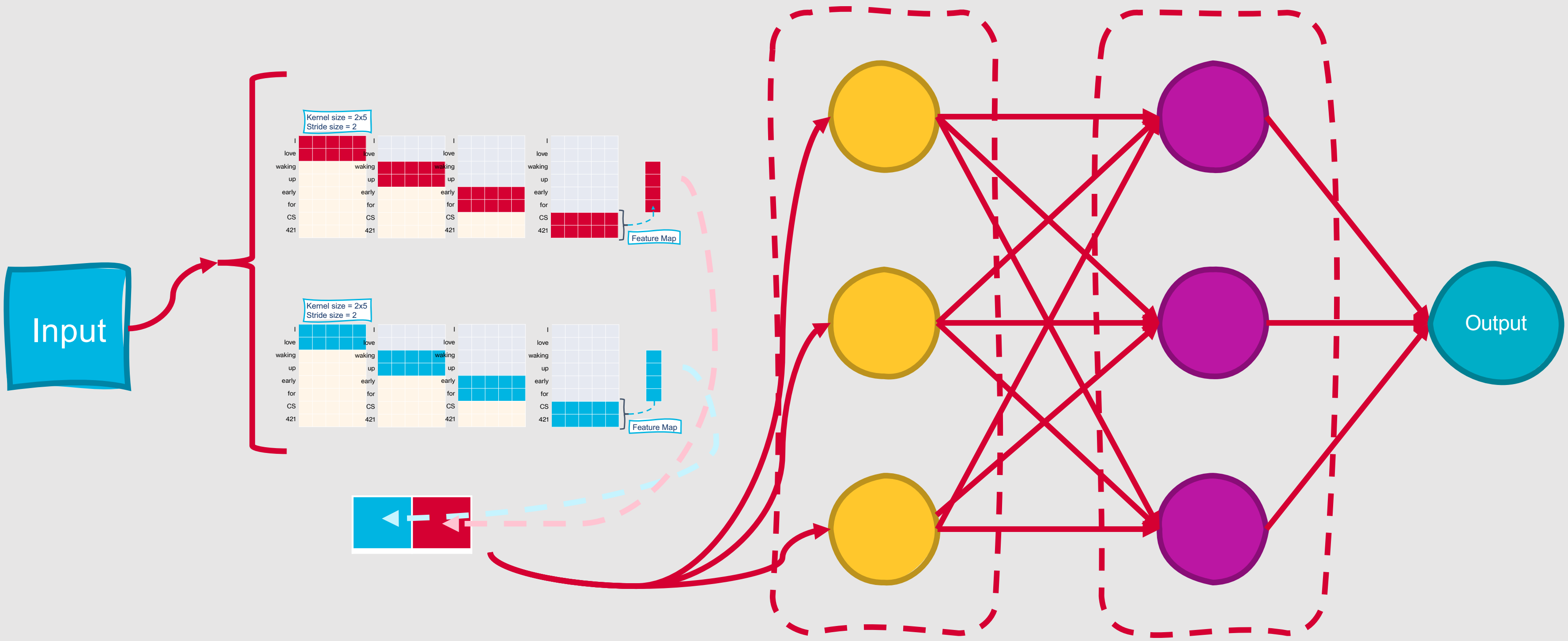


Typically, we learn multiple feature maps and then reduce the dimensionality of the learned feature maps by pooling (e.g., taking the average or maximum) subsets of their values.

- This is done to:
  - Further increase efficiency
  - Improve the model's invariance to small changes in the input



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

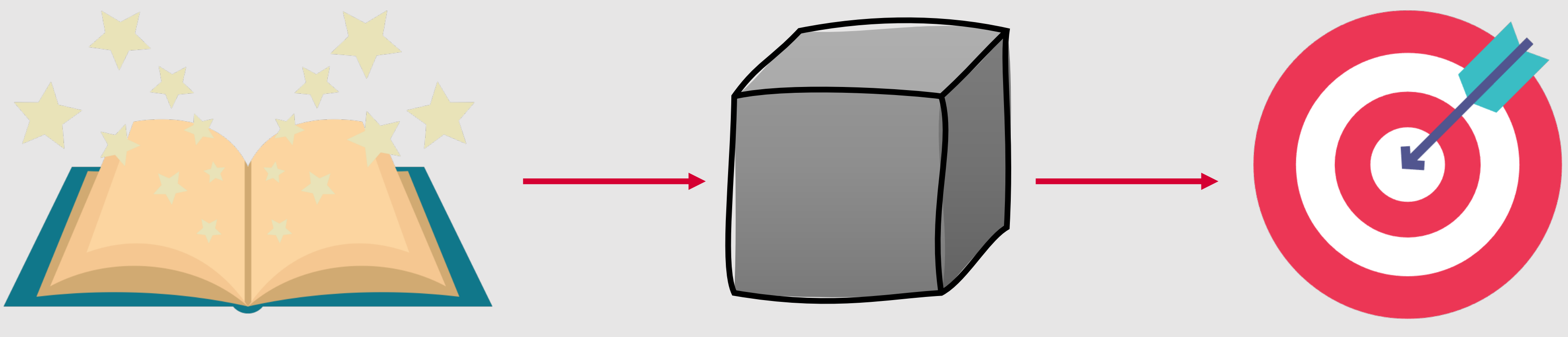


Why use CNNs for an NLP task?

- Originally designed for image classification!
- However, offers unique advantages for NLP tasks:
  - Extracts meaningful local structures from input
  - Increases efficiency of the training process relative to feedforward neural networks

Popular Deep Learning Architectures in Contemporary NLP

- Recurrent Neural Networks
- Convolutional Neural Networks
- **Transformers**



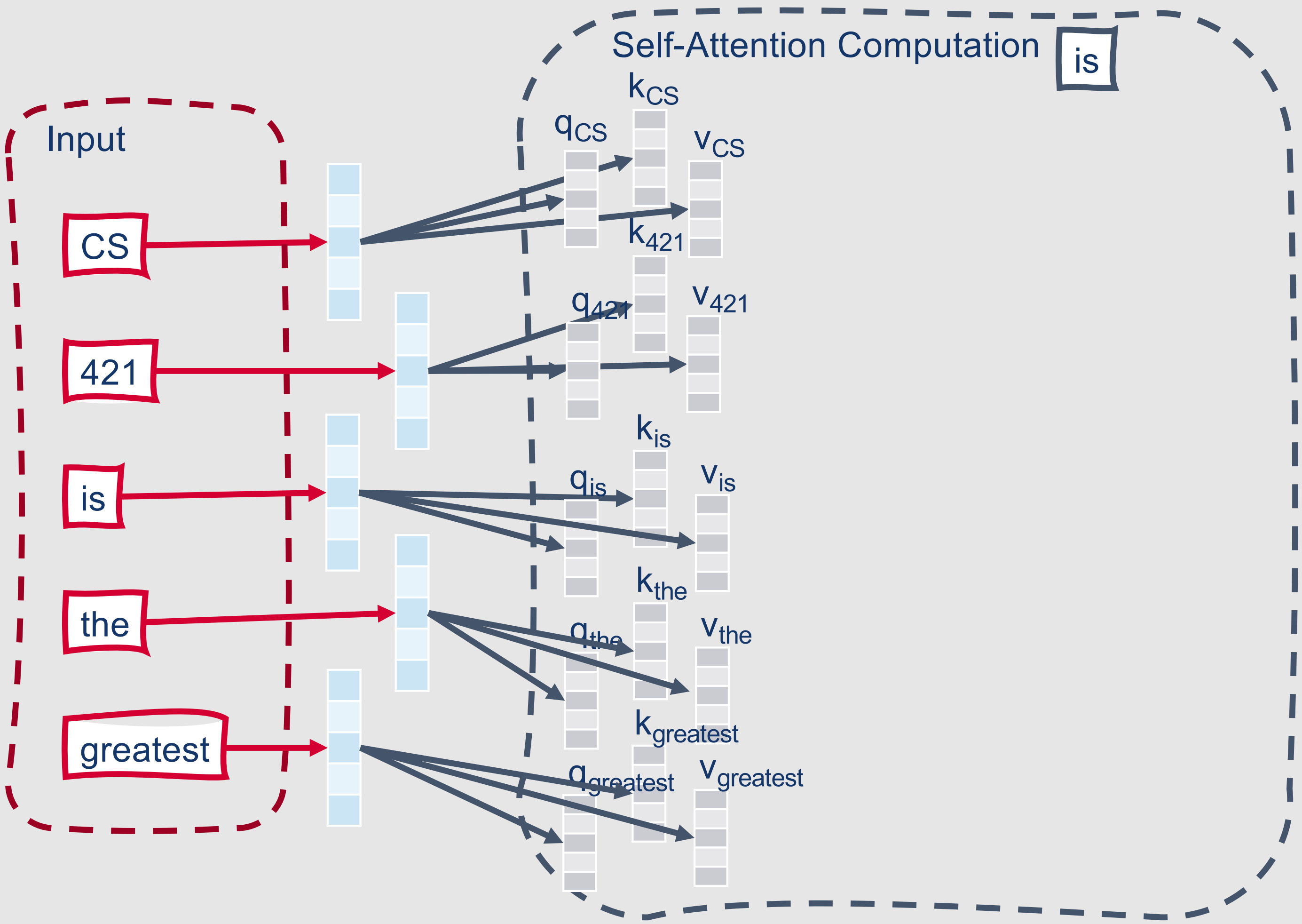
Transformers

- General premise:
  - Deep learning models don't need to wait to process items one after the other to incorporate sequential information
- Classic feedforward neural network:
  - Input to a layer is a vector of numbers representing the outputs of all units in the previous layer
- Modification for recurrent neural networks:
  - Input to a layer is a vector of numbers representing the outputs of all units in the previous layer + a vector of numbers representing the layer's output at the previous timestep
- Modification for Transformers:
  - Input to a feedforward layer is the output from a **self-attention layer** computed over the entire input sequence, indicating which words in the sequence are most important to one another

Self-Attention

1. Generate key, query, and value embeddings for each element of the input vector $\mathbf{x}$
   - $\mathbf{q}_i = \mathbf{W}^{\mathbf{Q}}\mathbf{x}_i$
   - $\mathbf{k}_i = \mathbf{W}^{\mathbf{K}}\mathbf{x}_i$
   - $\mathbf{v}_i = \mathbf{W}^{\mathbf{V}}\mathbf{x}_i$

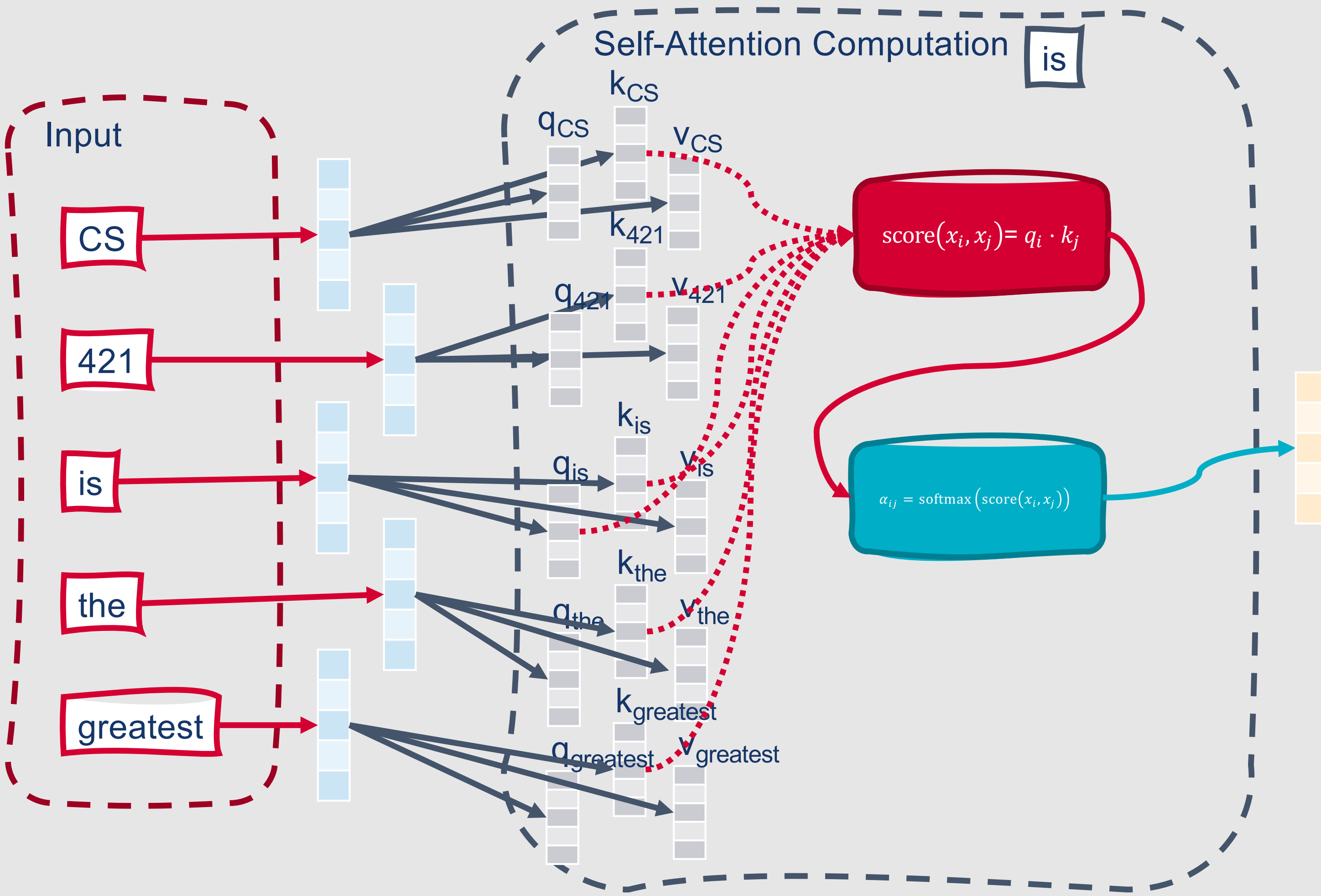
Bidirectional Self-Attention Layer



Self-Attention

1. Generate key, query, and value embeddings for each element of the input vector $\mathbf{x}$
   - $\mathbf{q}_i = \mathbf{W}^{\mathrm{Q}}\mathbf{x}_i$
   - $\mathbf{k}_i = \mathbf{W}^{\mathrm{K}}\mathbf{x}_i$
   - $\mathbf{v}_i = \mathbf{W}^{\mathrm{V}}\mathbf{x}_i$
2. Compute attention weights α by applying a softmax activation over the element-wise comparison scores between all possible query-key pairs in the full input sequence
   - $\mathrm{score}_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j$
   - $\alpha_{ij} = \frac{\exp(\mathrm{score}_{ij})}{\sum_{k=1}^{n} \exp(\mathrm{score}_{ik})}$

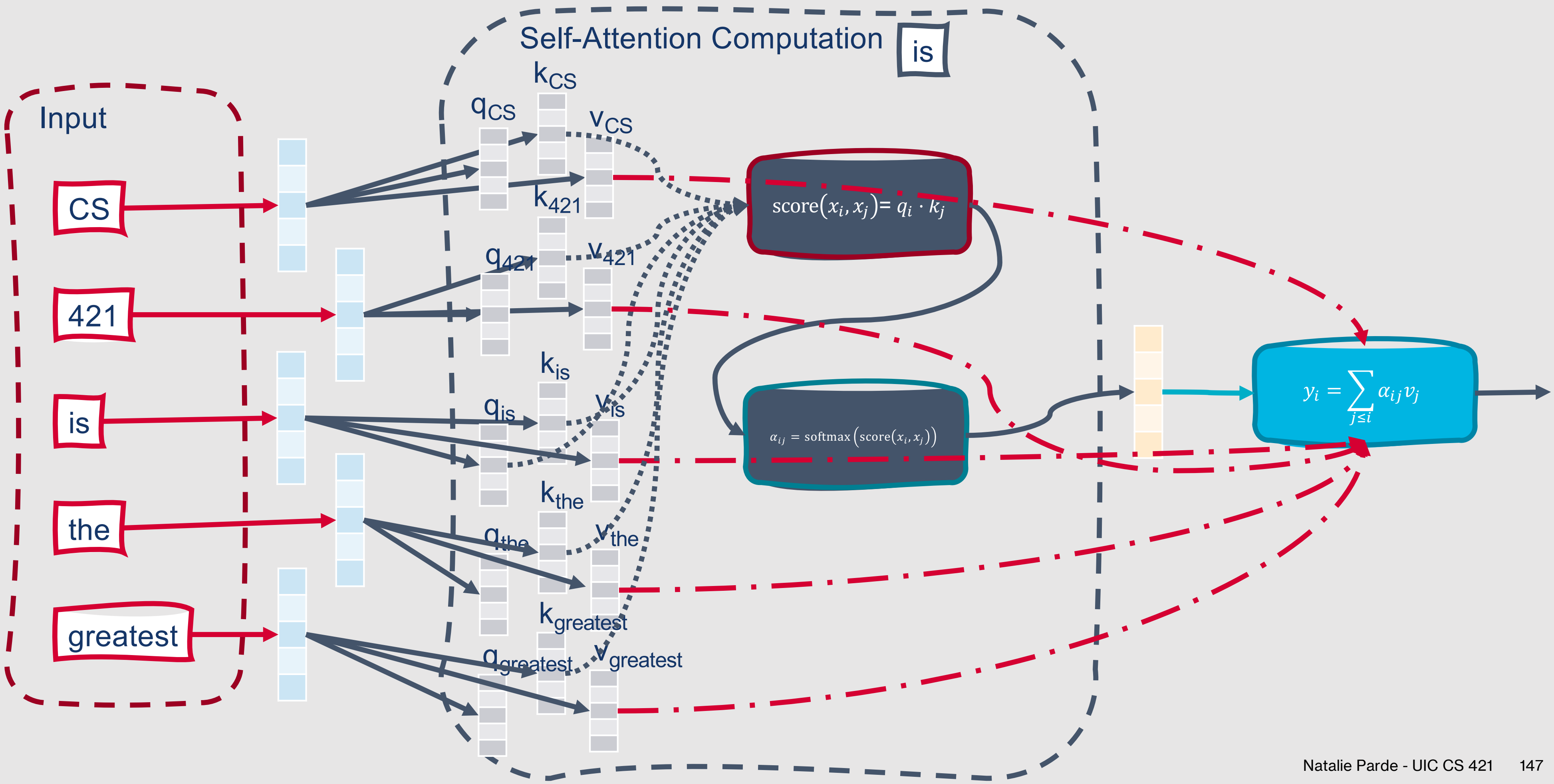
Bidirectional Self-Attention Layer



Self-Attention

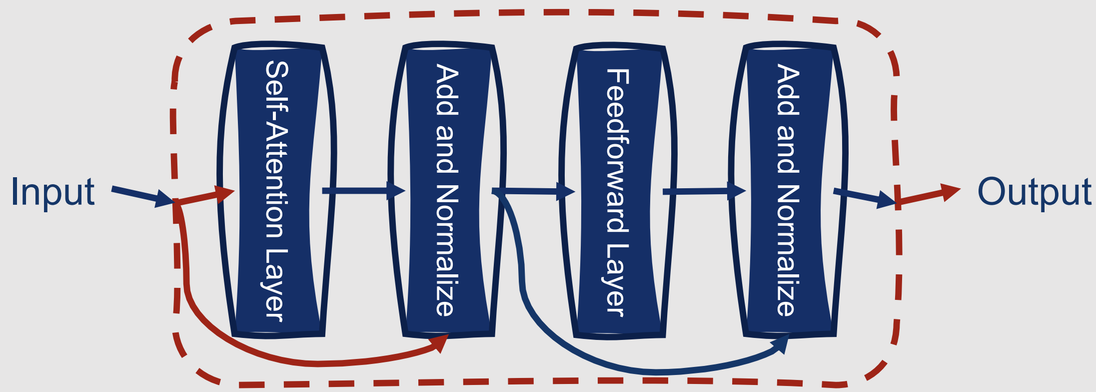
1. Generate key, query, and value embeddings for each element of the input vector $\mathbf{x}$
   - $\mathbf{q}_i = \mathbf{W}^{\mathrm{Q}}\mathbf{x}_i$
   - $\mathbf{k}_i = \mathbf{W}^{\mathrm{K}}\mathbf{x}_i$
   - $\mathbf{v}_i = \mathbf{W}^{\mathrm{V}}\mathbf{x}_i$
2. Compute attention weights α by applying a softmax activation over the element-wise comparison scores between all possible query-key pairs in the full input sequence
   - $\mathrm{score}_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j$
   - $\alpha_{ij} = \frac{\exp(\mathrm{score}_{ij})}{\sum_{k=1}^{n} \exp(\mathrm{score}_{ik})}$
3. Compute the output vector $\mathbf{y}_i$ as the attention-weighted sum of the input value vectors $\mathbf{v}$
   - $\boldsymbol{y}_{\boldsymbol{i}} = \sum_{j=1}^{n} \alpha_{ij}\mathbf{v}_j$

Bidirectional Self-Attention Layer



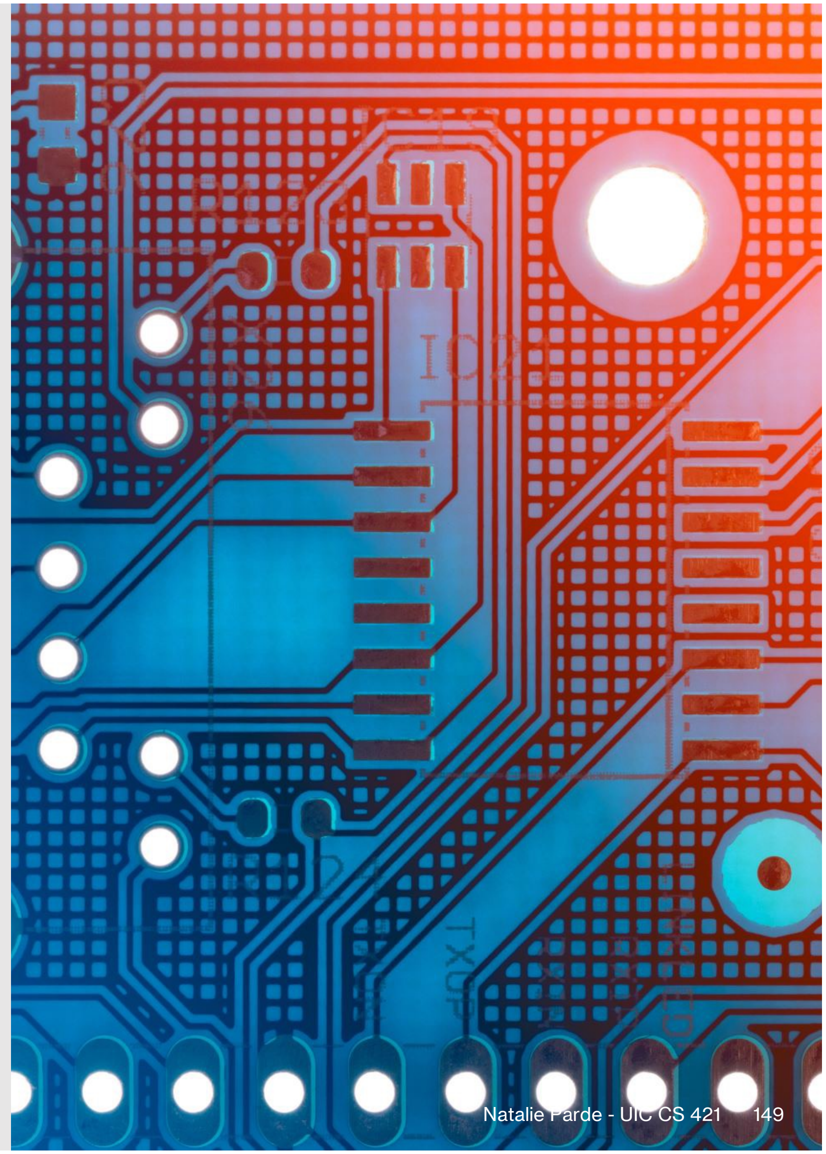
Transformer Blocks

- Transformers are implemented by stacking one or more blocks of the following layers:
  - Self-attention layer
  - Normalization layer
  - Feedforward layer
  - Another normalization layer
- Some of these layers have **residual** connections between them even though they do not immediately precede or proceed one another



Which of these architectures should you use?

- Depends on your:
  - Task
  - Dataset
  - Compute resources
- Current state-of-the-art models are usually Transformer-based; however, state-of-the-art Transformers require many compute resources
  - GPUs for performing lots of floating point operations
  - RAM for holding lots of data in memory
- Specialized tasks may also benefit from combined architectures (e.g., CNN-LSTM)!
- It’s good to experiment with numerous models to determine what works best for the problem you’re trying to solve, within the constraints of your compute environment



Summary: Deep Learning for NLP

- Loss can be propagated backward through the network from the output layer to earlier layers using **backpropagation**
- Network architectures can be optimized via a **fine-tuning** process
- Neural networks can be used to build **neural language models**
- **Recurrent neural networks** directly encode temporal context into the network's computational units
- **Convolutional neural networks** increase efficiency by performing operations over regions of input data
- **Transformers** calculate self-attention to encode temporal context for the full input in a single step