



Word Embeddings and Feedforward Neural Networks

Natalie Parde

UIC CS 421

What we know so far....

- **Word vectors:** Vectors of numbers used to encode language
 - Each vector represents a point in an n-dimensional semantic space
- Simple techniques to create word vectors:
 - Co-occurrence frequency (**bag of words**)
 - **TF-IDF**

1	0	0	1	0	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---

0.7	0	0	0	0	0.9	0.1	0	0	0.5
-----	---	---	---	---	-----	-----	---	---	-----

This Week's Topics

Cosine similarity
Word2Vec
Other dense embeddings
Using word embeddings

Thursday

Tuesday

Neural networks
Combining and optimizing
computational units
Neural language models

This Week's Topics



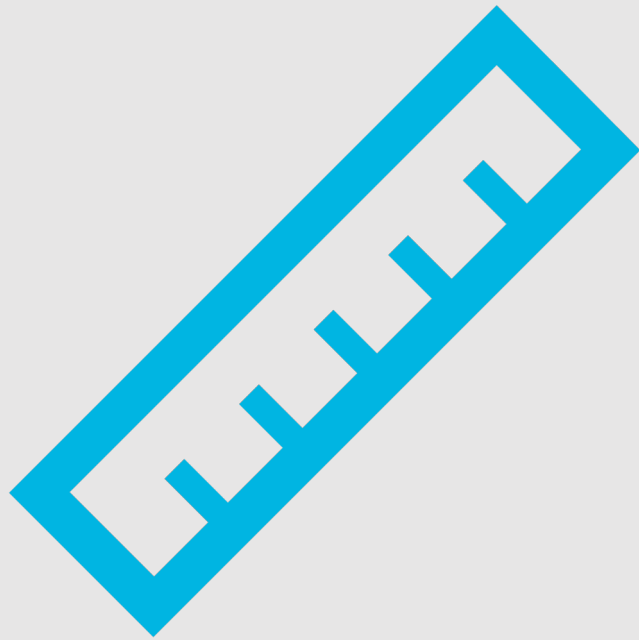
Cosine similarity
Word2Vec
Other dense embeddings
Using word embeddings

Thursday

Tuesday

Neural networks
Combining and optimizing
computational units
Neural language models

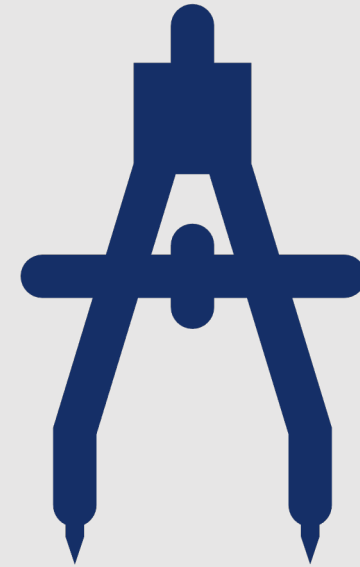
We can use these vectors to measure semantic similarity between words.



- Popular Approach: **Cosine similarity**
 - Based on the **dot product** from linear algebra
 - $\mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^N v_i w_i = v_1 w_1 + v_2 w_2 + \dots + v_N w_N$
- Intuition:
 - Similar vectors (e.g., large values in the same dimensions) will have high cosine similarity
 - Dissimilar vectors (e.g., zeros or low values in different dimensions) will have low cosine similarity
- We compute a **normalized dot product** to avoid issues related to word frequency
 - Non-normalized dot product will be higher for frequent words, regardless of how similar they are
 - We normalize by dividing the dot product by the lengths of the two vectors



Normalized Dot Product = Cosine Similarity



- The cosine similarity metrics between two vectors \mathbf{v} and \mathbf{w} can thus be computed as:

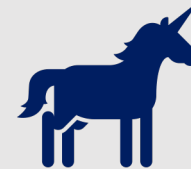
- $$\text{cosine}(\mathbf{v}, \mathbf{w}) = \frac{\mathbf{v} \cdot \mathbf{w}}{|\mathbf{v}| |\mathbf{w}|} = \frac{\sum_{i=1}^N v_i w_i}{\sqrt{\sum_{i=1}^N v_i^2} \sqrt{\sum_{i=1}^N w_i^2}}$$

- This value ranges between:
 - 0 (dissimilar) and 1 (similar) for frequency or TF-IDF vectors
 - -1 (dissimilar) and 1 (similar) for embedding vectors that may have negative values

Example: Computing Cosine Similarity

	glitter	data	computer
unicorn	442	8	2
digital	5	1683	1670
information	5	3982	3325

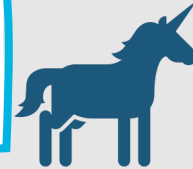
$$\cos(\text{unicorn}, \text{information}) = ?$$



Example: Computing Cosine Similarity

	glitter	data	computer
unicorn	442	8	2
digital	5	1683	1670
information	5	3982	3325

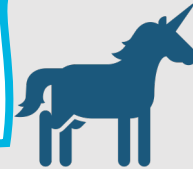
$$\cos(\text{unicorn}, \text{information}) = \frac{[442, 8, 2] \cdot [5, 3982, 3325]}{\sqrt{442^2 + 8^2 + 2^2} \sqrt{5^2 + 3982^2 + 3325^2}}$$



Example: Computing Cosine Similarity

	glitter	data	computer
unicorn	442	8	2
digital	5	1683	1670
information	5	3982	3325

$$\cos(\text{unicorn}, \text{information}) = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}}$$



Example: Computing Cosine Similarity

	glitter	data	computer
unicorn	442	8	2
digital	5	1683	1670
information	5	3982	3325

$$\cos(\text{unicorn}, \text{information}) = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = 0.017$$

Example: Computing Cosine Similarity

	glitter	data	computer
unicorn	442	8	2
digital	5	1683	1670
information	5	3982	3325

$$\cos(\text{unicorn}, \text{information}) = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = 0.017$$

$$\cos(\text{digital}, \text{information}) = \frac{5*5+1683*3982+1670*3325}{\sqrt{5^2+1683^2+1670^2}\sqrt{5^2+3982^2+3325^2}} = 0.996$$

Example: Computing Cosine Similarity

	glitter	data	computer
unicorn	442	8	2
digital	5	1683	1670
information	5	3982	3325

$$\cos(\text{unicorn}, \text{information}) = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = 0.017$$

$$\cos(\text{digital}, \text{information}) = \frac{5*5+1683*3982+1670*3325}{\sqrt{5^2+1683^2+1670^2}\sqrt{5^2+3982^2+3325^2}} = 0.996$$



Result: *information* is way closer to *digital* than it is to *unicorn*!

This Week's Topics



Cosine similarity
Word2Vec
Other dense embeddings
Using word embeddings

Thursday

Tuesday

Neural networks
Combining and optimizing
computational units
Neural language models

Limitations of Bag-of- Words Style Vectors

- Very **high-dimensional**
- Lots of **empty** (zero-valued) cells
- Struggle with inferring deeper semantic content:
 - Synonyms
 - Antonyms
 - Positive/negative connotations
 - Related contexts

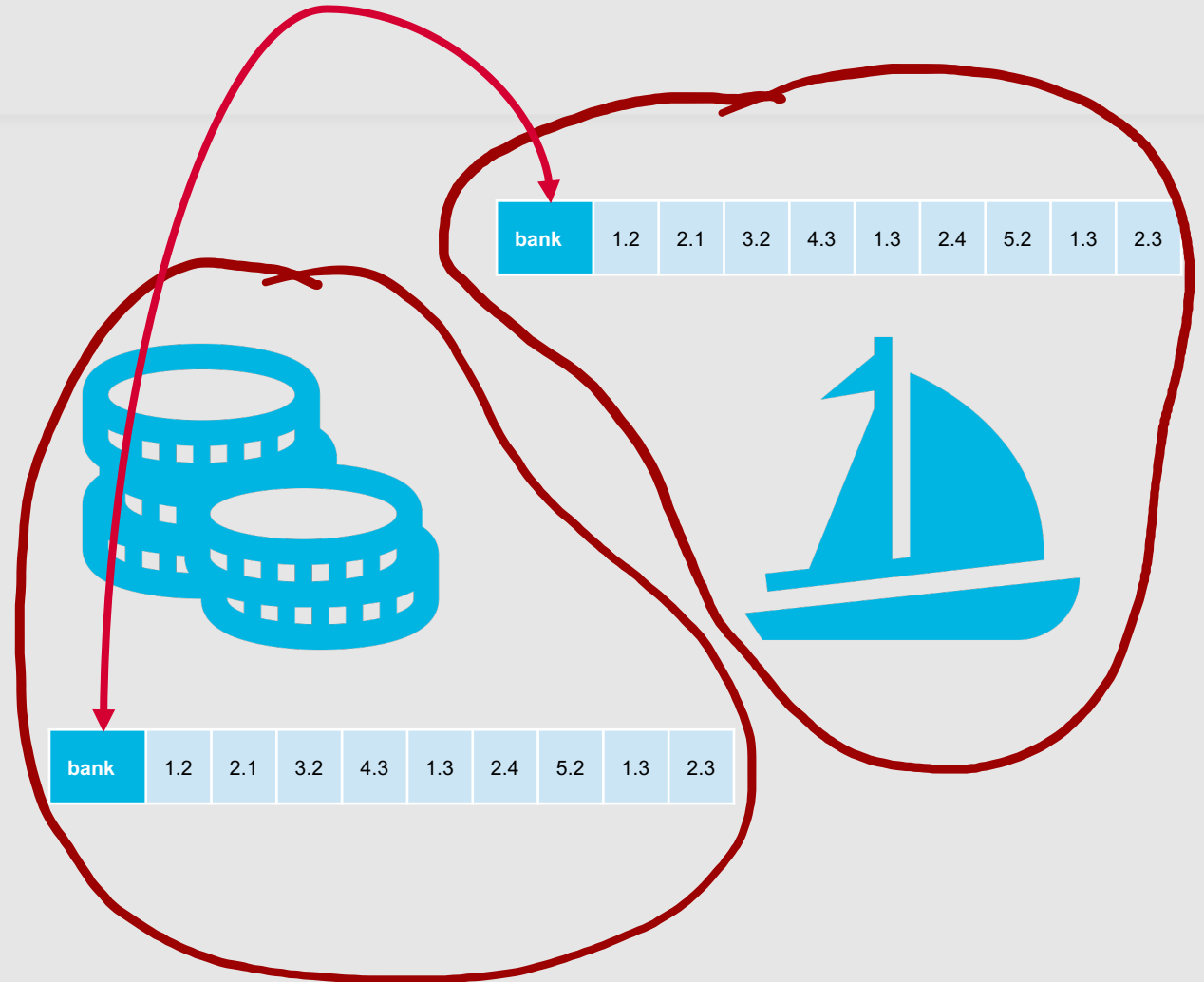
+
•
○

What would our “dream vector” look like?

- **Lower-dimensional** (~ 50-1000 cells)
 - Easier to include as **features**
 - Classifiers have to learn ~100 weights instead of ~50,000
 - Fewer **parameters** → lower chance of overfitting
 - May generalize better to new data
- Most dimensions with **non-zero** values
- We'd also prefer to be able to encode other semantic dimensions of meaning
 - *Good* should be:
 - Far from *bad*
 - Close to *great*
 - For this, we need vector dimensions to correspond to meaning directly, rather than specific words

Enter Word2Vec....

- **Word2Vec:** A method for automatically learning dense word representations from large text corpora
 - Fast
 - Efficient to train
 - Non-contextual (homonyms have the same representations)





Word2Vec

- Technically a tool for implementing word vectors:
 - <https://code.google.com/archive/p/word2vec>
- The algorithm that people usually refer to as *Word2Vec* is the **skip-gram** model with **negative sampling**

Word2Vec Intuition

- Instead of counting how often each word occurs near each context word, train a classifier on a **binary prediction task**
 - Is word w likely to occur near context word c ?
- The twist: **We don't actually care about the classifier!**
- We use the **learned classifier weights** from this prediction task as our word embeddings

+

•

○

**None of this
requires
manual
supervision.**

- Text (without any other labels) is framed as **implicitly supervised** training data
 - Given the question: Is word w likely to occur near context word c ?
 - If w occurs near c in the training corpus, the gold standard answer is “yes”
- Similar setup to **neural language modeling** (neural networks that predict the next word based on prior words), but simpler:
 - Fewer layers
 - Makes **binary yes/no predictions** rather than predicting words

What does the classification task look like?

- **Goal:** Train a classifier that, given a tuple (t, c) of a target word t paired with a context word c (e.g., (super, bowl) or (super, laminator)), will return the probability that c is a real context word
 - $P(+ | t, c)$
- Context is defined by our context window (in this case, ± 2 words)

this	sunday,	watch	the	super	bowl	at	5:30	p.m.
		c1	c2	t	c3	c4		

High-Level Overview: How Word2Vec Works

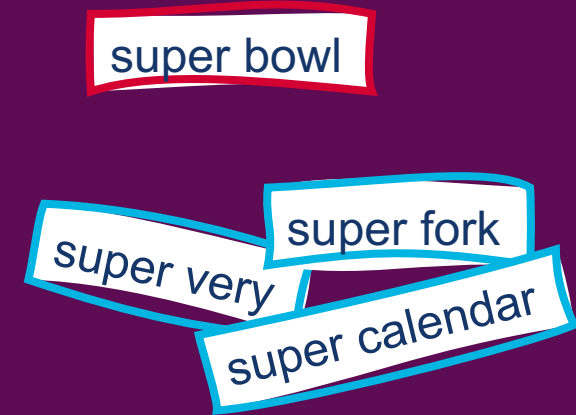
- Treat the target word w and a neighboring context word c as positive samples



super bowl

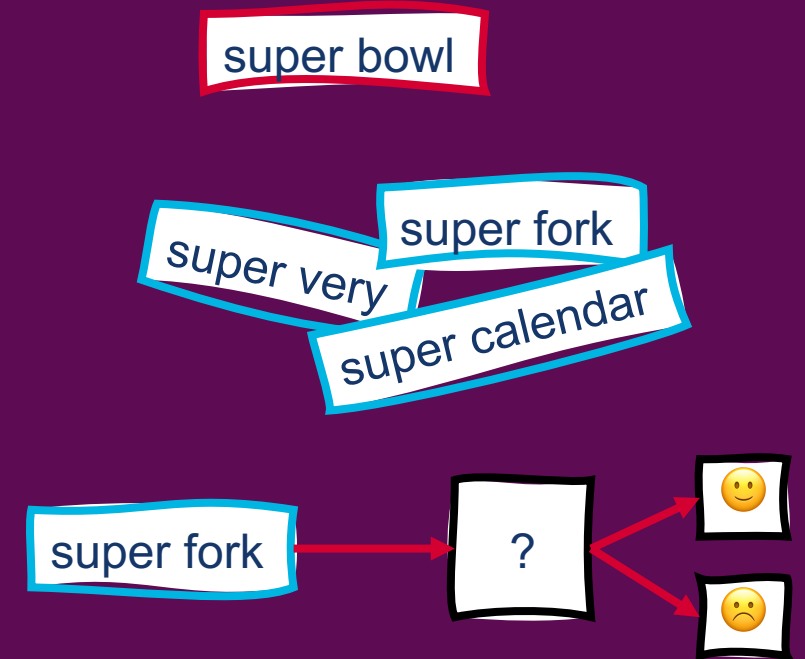
High-Level Overview: How Word2Vec Works

- Treat the target word w and a neighboring context word c as positive samples
- Randomly sample other words in the lexicon to get negative samples



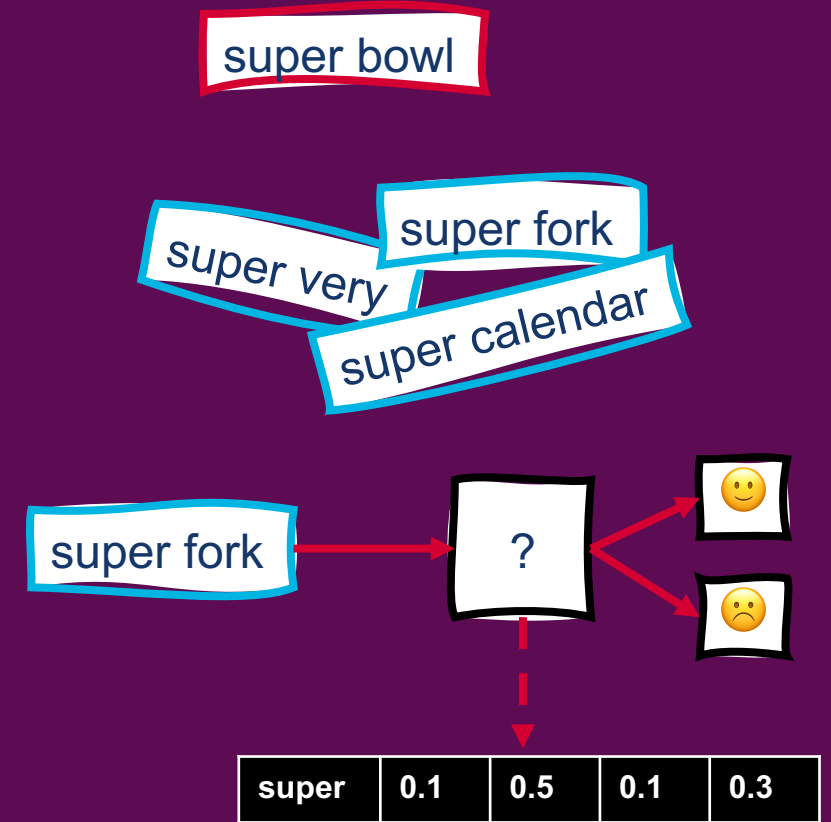
High-Level Overview: How Word2Vec Works

- Treat the target word w and a neighboring context word c as positive samples
- Randomly sample other words in the lexicon to get negative samples
- Train a classifier to distinguish between those two cases



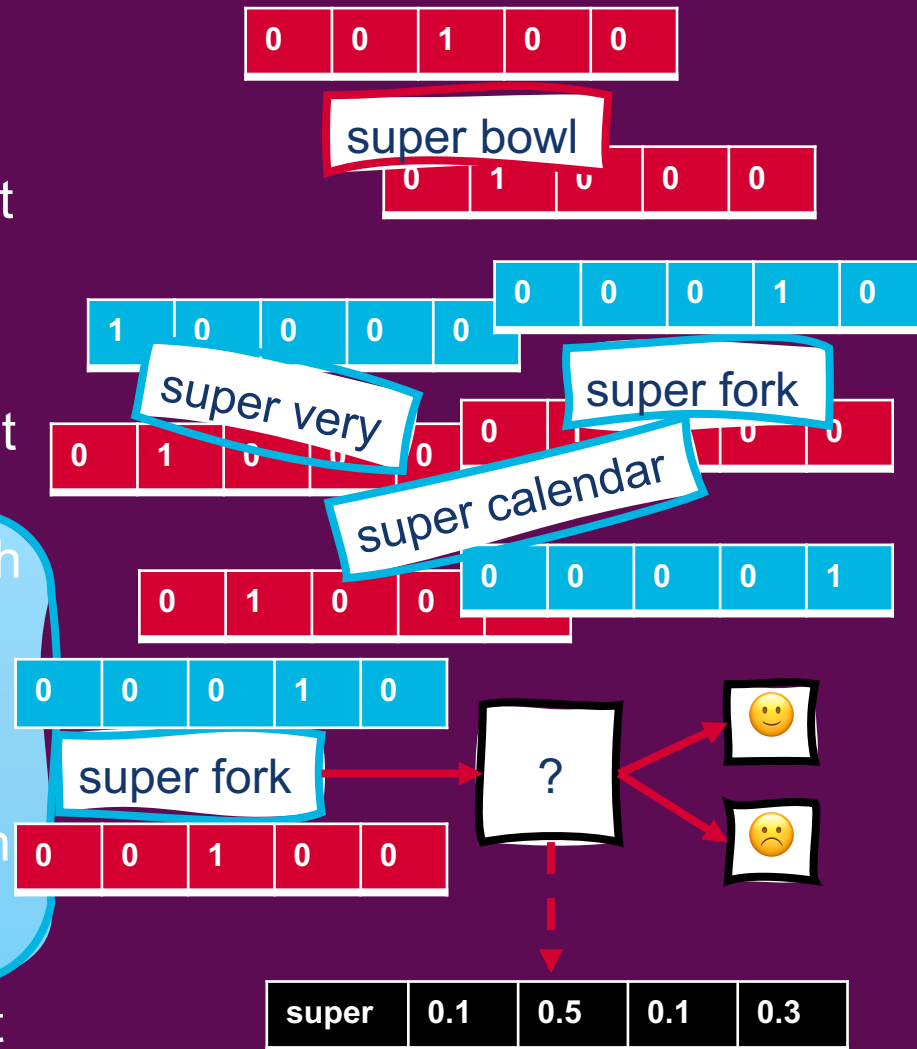
High-Level Overview: How Word2Vec Works

- Treat the target word w and a neighboring context word c as positive samples
- Randomly sample other words in the lexicon to get negative samples
- Train a classifier to distinguish between those two cases
- Use the weights from that classifier as the word embeddings



High-Level Overview: How Word2Vec Works

- Represent all words in a vocabulary as a vector
- Treat the target word w and a neighboring context word c as positive samples
- Randomly sample other words in the lexicon to get negative samples
- Find the similarity for each (t,c) pair and use this to calculate $P(+|(t,c))$
- Train a classifier to maximize these probabilities to distinguish between positive and negative cases
- Use the weights from that classifier as the word embeddings





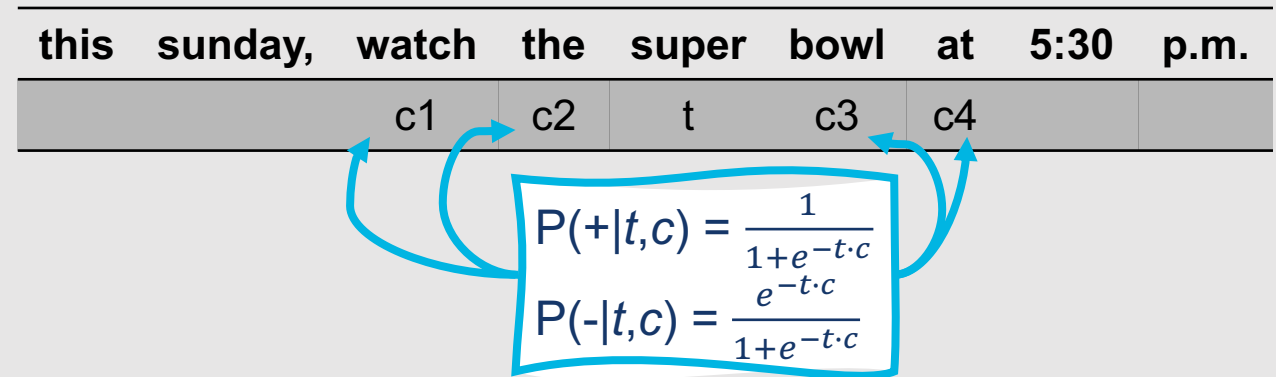
How do we *compute* $P(+ | t, c)$?

- This is based on vector similarity
- We can assume that vector similarity is proportional to the dot product between two vectors
 - $\text{Similarity}(t, c) \propto t \cdot c$
- More similar vectors \rightarrow more likely that c occurs near t

A dot product gives us a number, not a probability.

- How do we turn it into a probability?
 - **Sigmoid function** (just like we did with logistic regression!)
 - We can set:
 - $P(+|t,c) = \frac{1}{1+e^{-t \cdot c}}$
- Then:
 - $P(+ | t,c) = \frac{1}{1+e^{-t \cdot c}}$
 - $P(- | t,c) = 1 - P(+ | t,c) = \frac{e^{-t \cdot c}}{1+e^{-t \cdot c}}$

What if we want to know the probability that a span of text occurs in the context of the target word?



- Simplifying assumption: **All context words are independent**
- So, we can just multiply their probabilities:
 - $P(+|t, c_{1:k}) = \prod_{i=1}^k \frac{1}{1+e^{-t \cdot c_i}}$, or
 - $\log P(+|t, c_{1:k}) = \sum_{i=1}^k \log \frac{1}{1+e^{-t \cdot c_i}}$



With this in mind....

this	sunday,	watch	the	super	bowl	at	5:30	p.m.
	c1	c2	t	c3	c4			
	$P(+ super, watch) = .7$	$P(+ super, the) = .5$		$P(+ super, bowl) = .9$	$P(+ super at) = .5$			

$$P(+|t, c_{1:k}) = .7 * .5 * .9 * .5 = .1575$$

- Given t and a context window of k words $c_{1:k}$, we can assign a probability based on how similar the context window is to the target word
- However, we still have some unanswered questions....
 - **How do we determine our input vectors?**
 - **How do we learn word embeddings** throughout this process (this is the real goal of training our classifier in the first place)?

Input Vectors

- Typically represented as **one-hot vectors**
 - **Binary bag-of-words approach:** Place a “1” in the position corresponding to a given word, and a “0” in every other position

super

0	0	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---

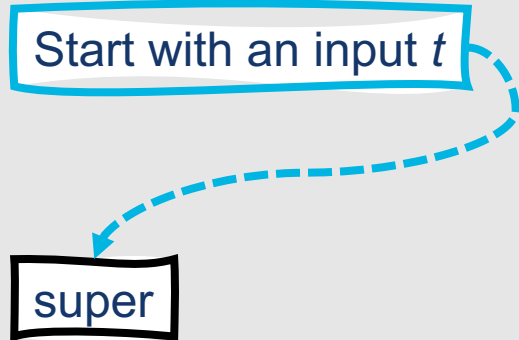
bowl

0	0	1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---

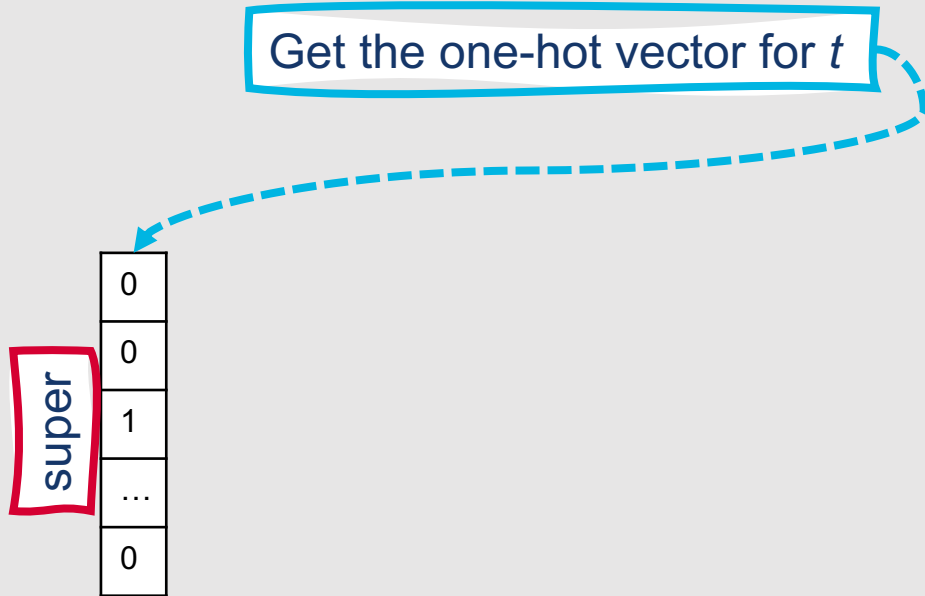
Learned Embeddings....

- Embeddings are the weights learned for a two-layer classifier that predicts $P(+ | t, c)$
- Recall from our discussion of logistic regression:
 - $y = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-w \cdot x + b}}$
- This is quite similar to the probability we're trying to optimize:
 - $P(+ | t, c) = \frac{1}{1+e^{-t \cdot c}}$

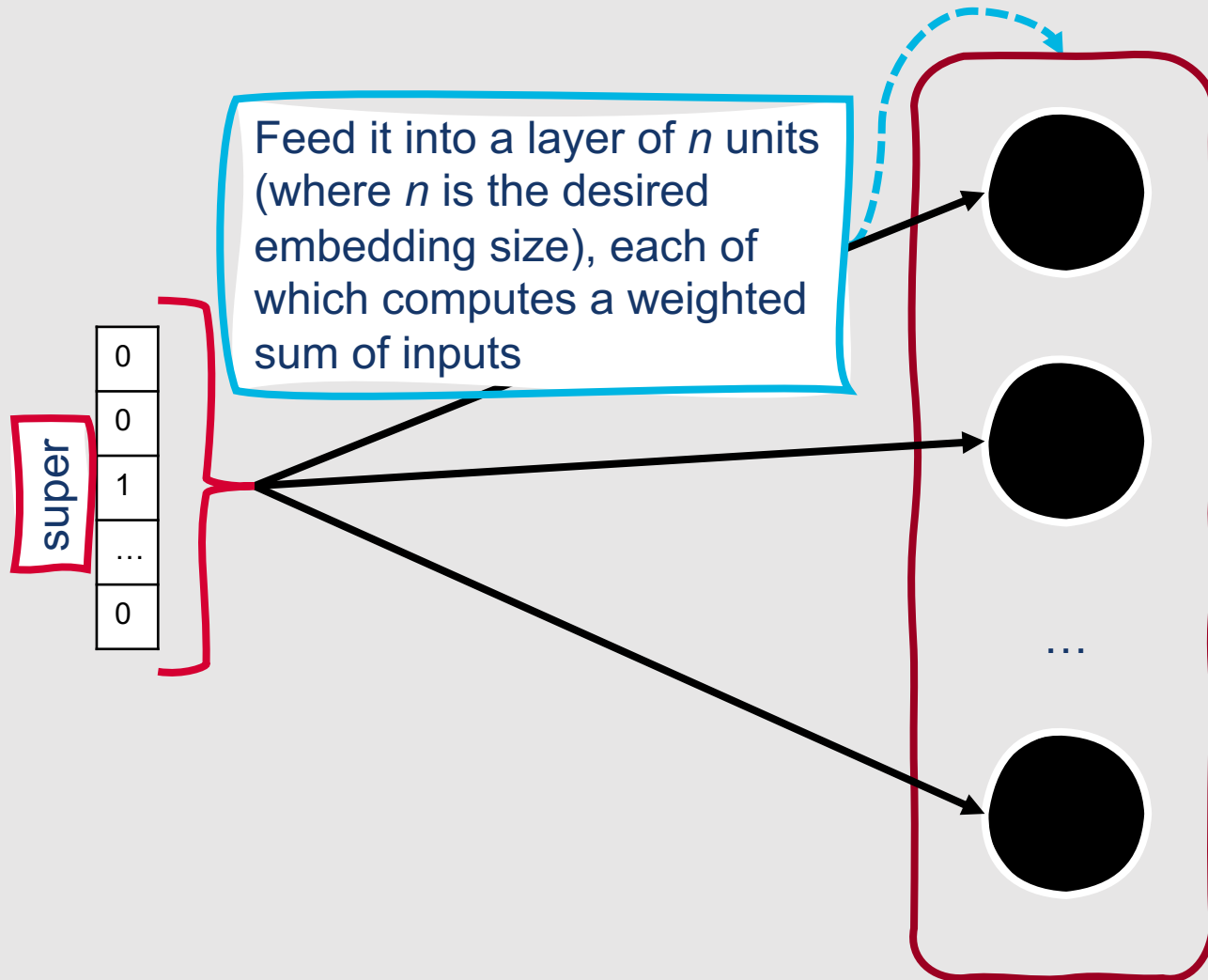
What does this look like?



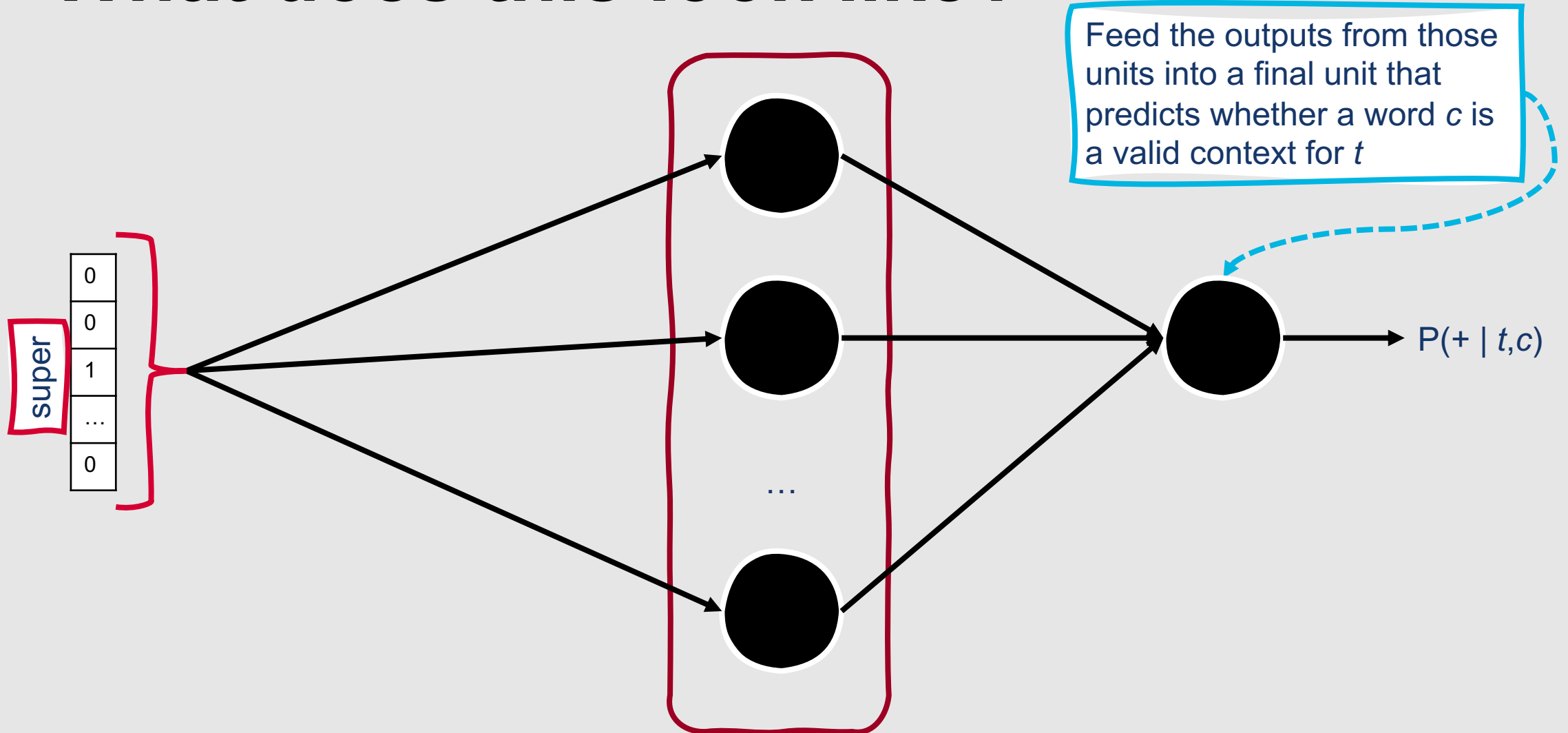
What does this look like?



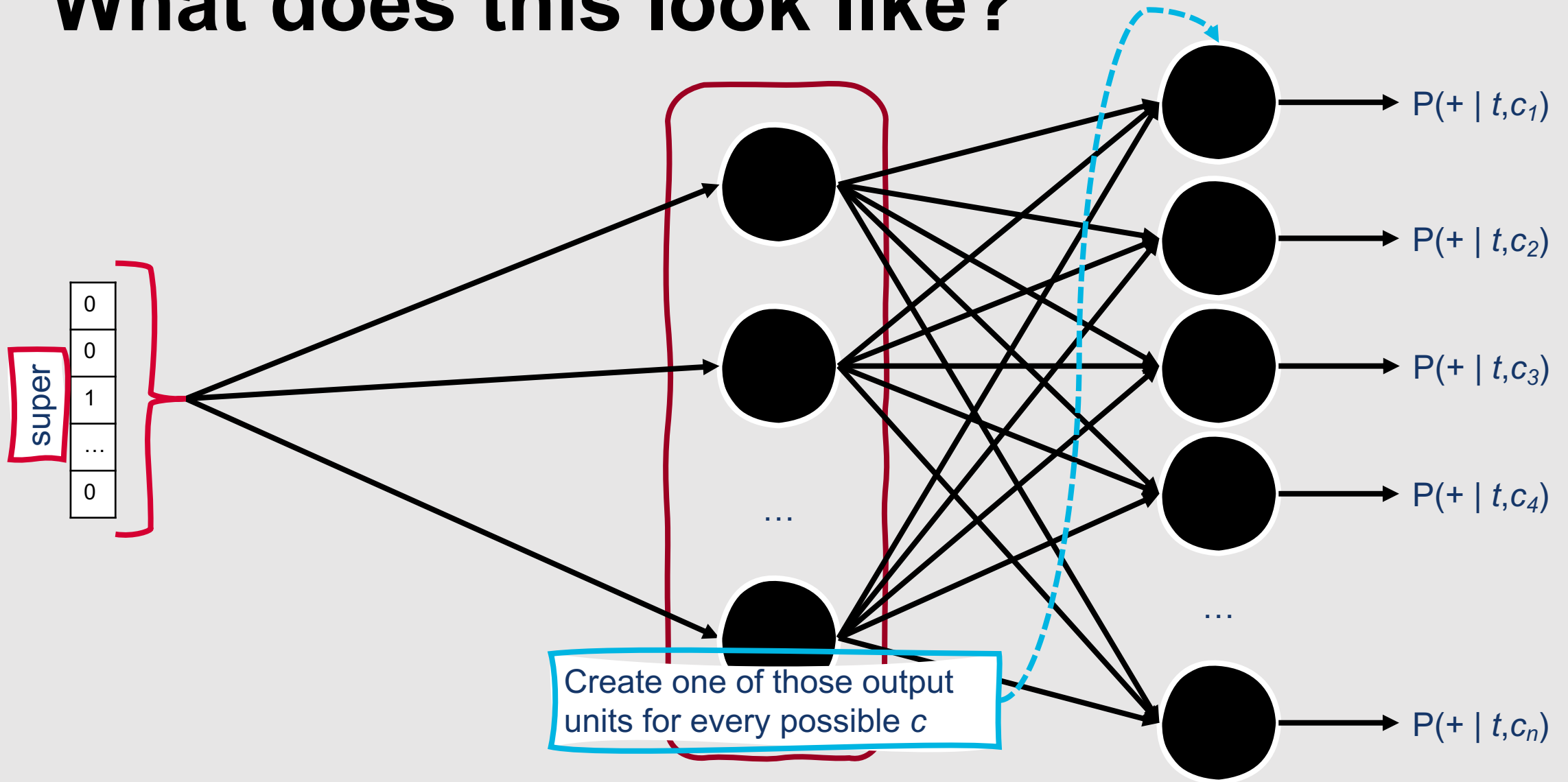
What does this look like?



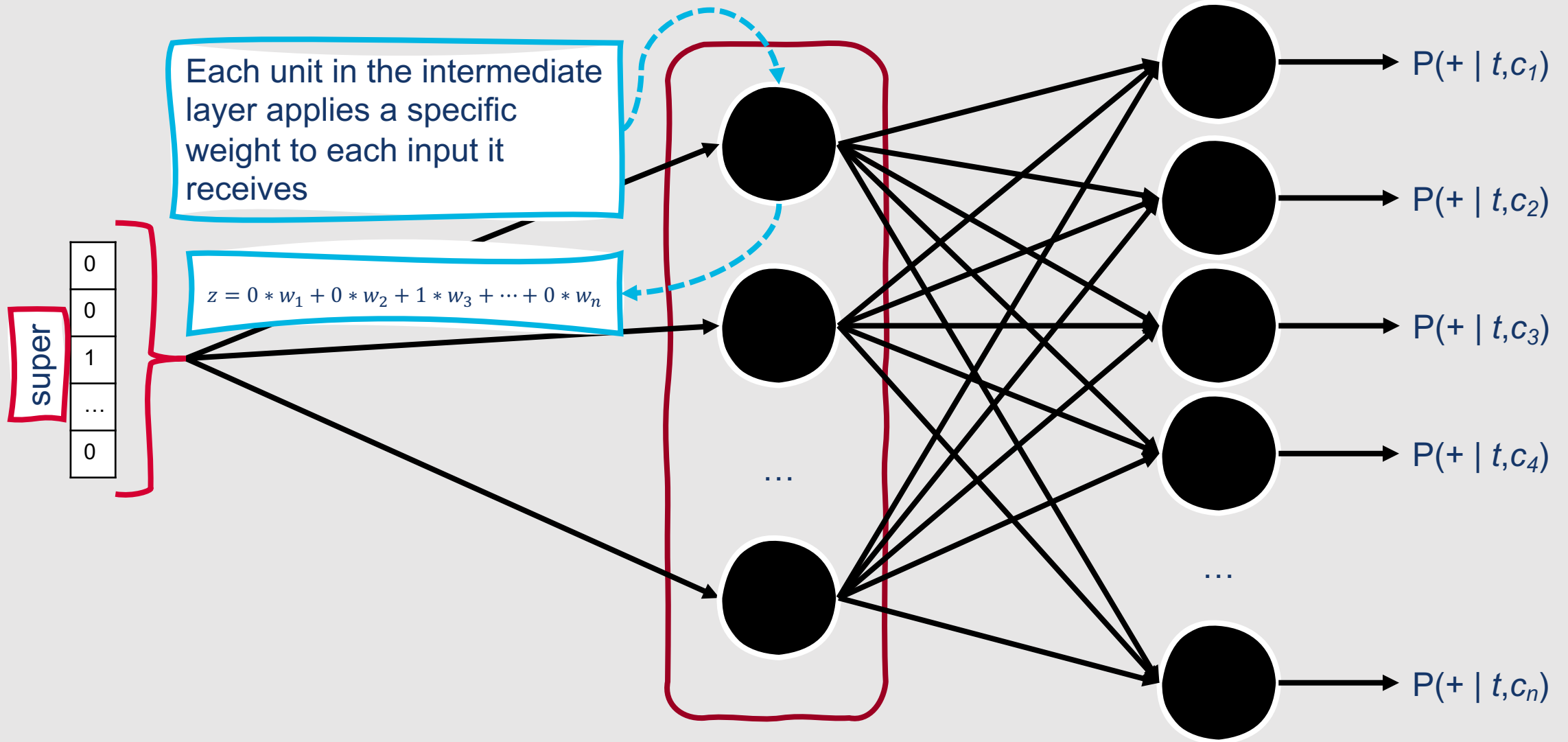
What does this look like?



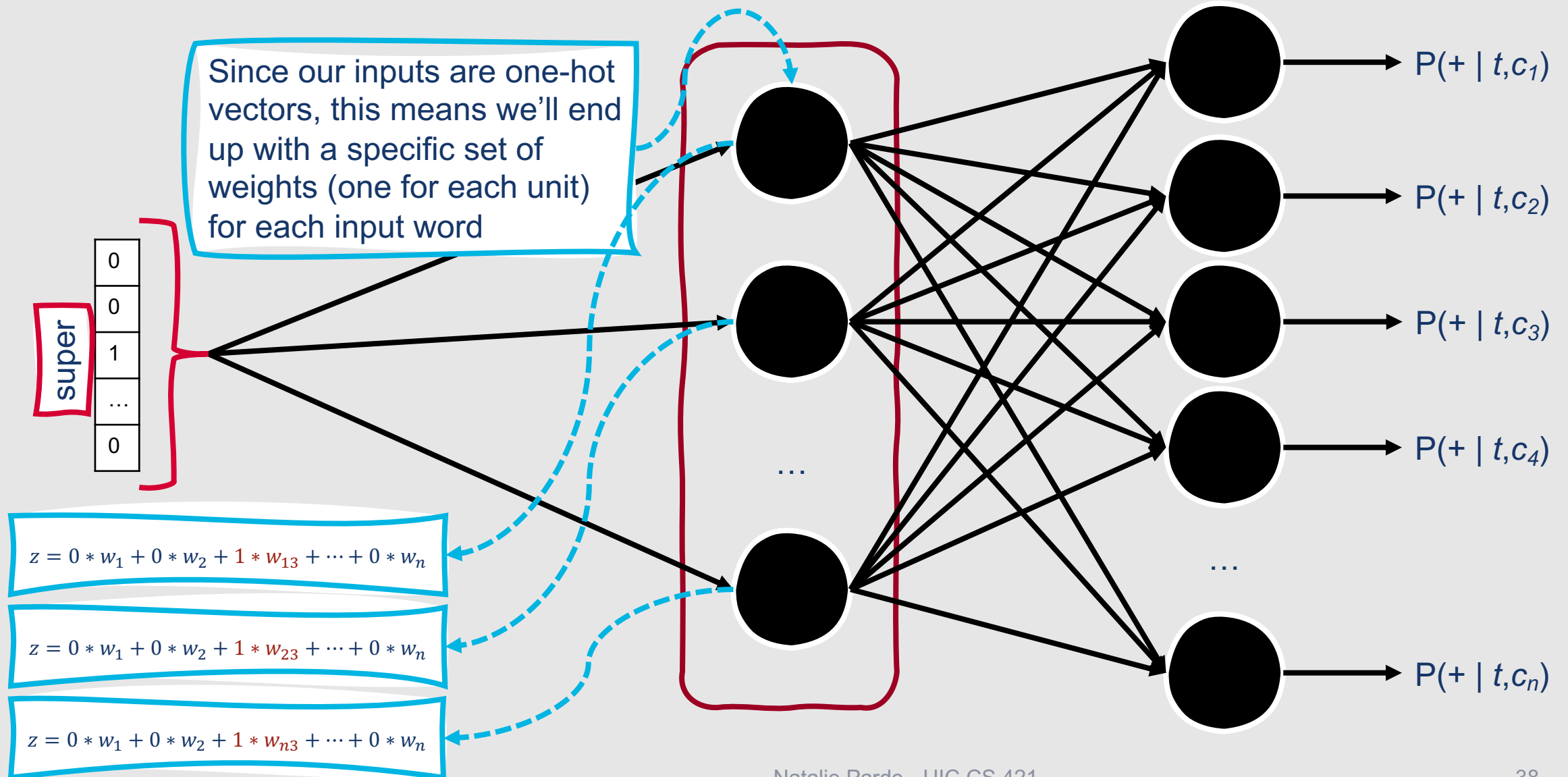
What does this look like?



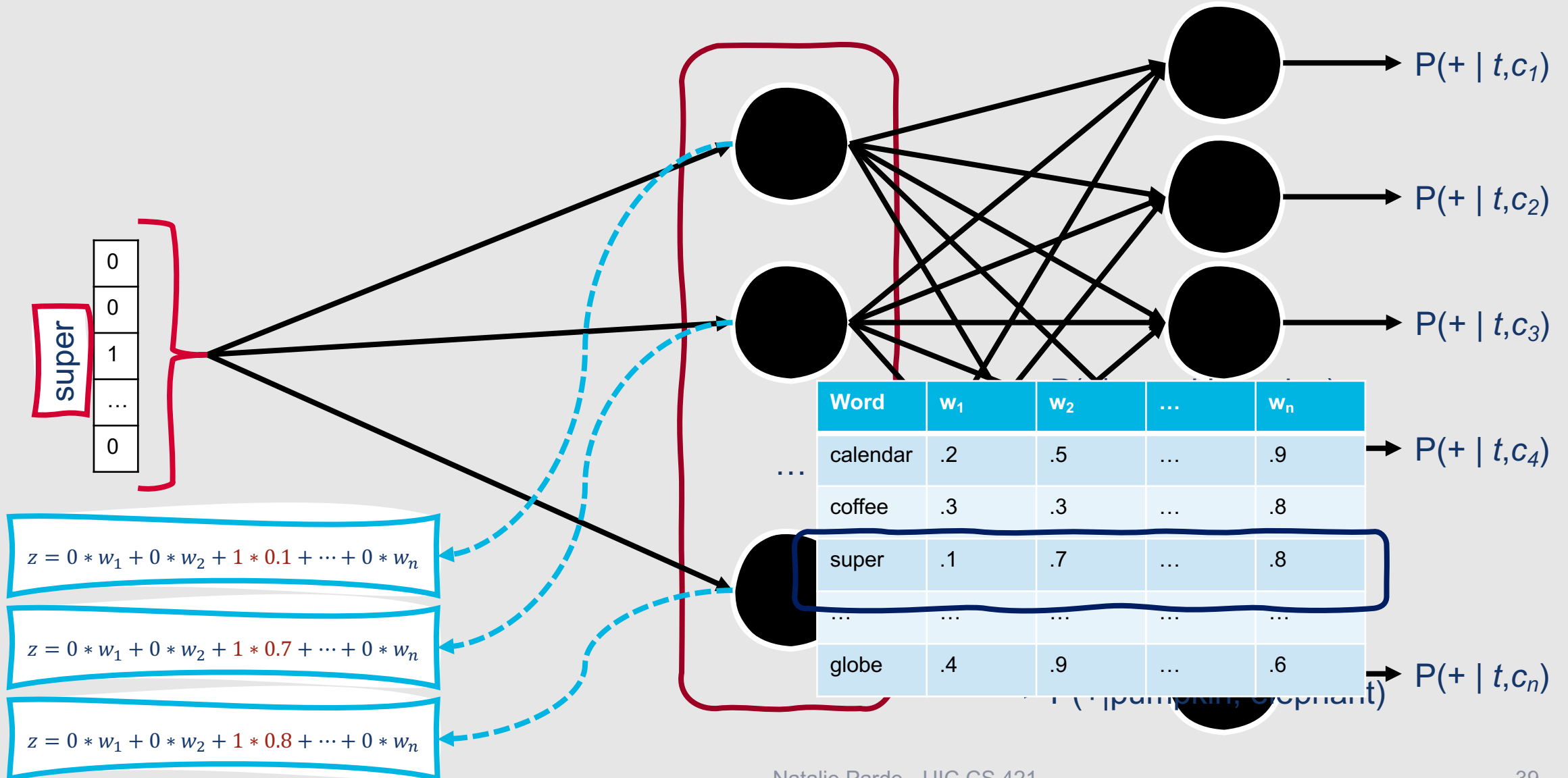
Behind the scenes....



Behind the scenes....



These are the weights we're interested in! ✓

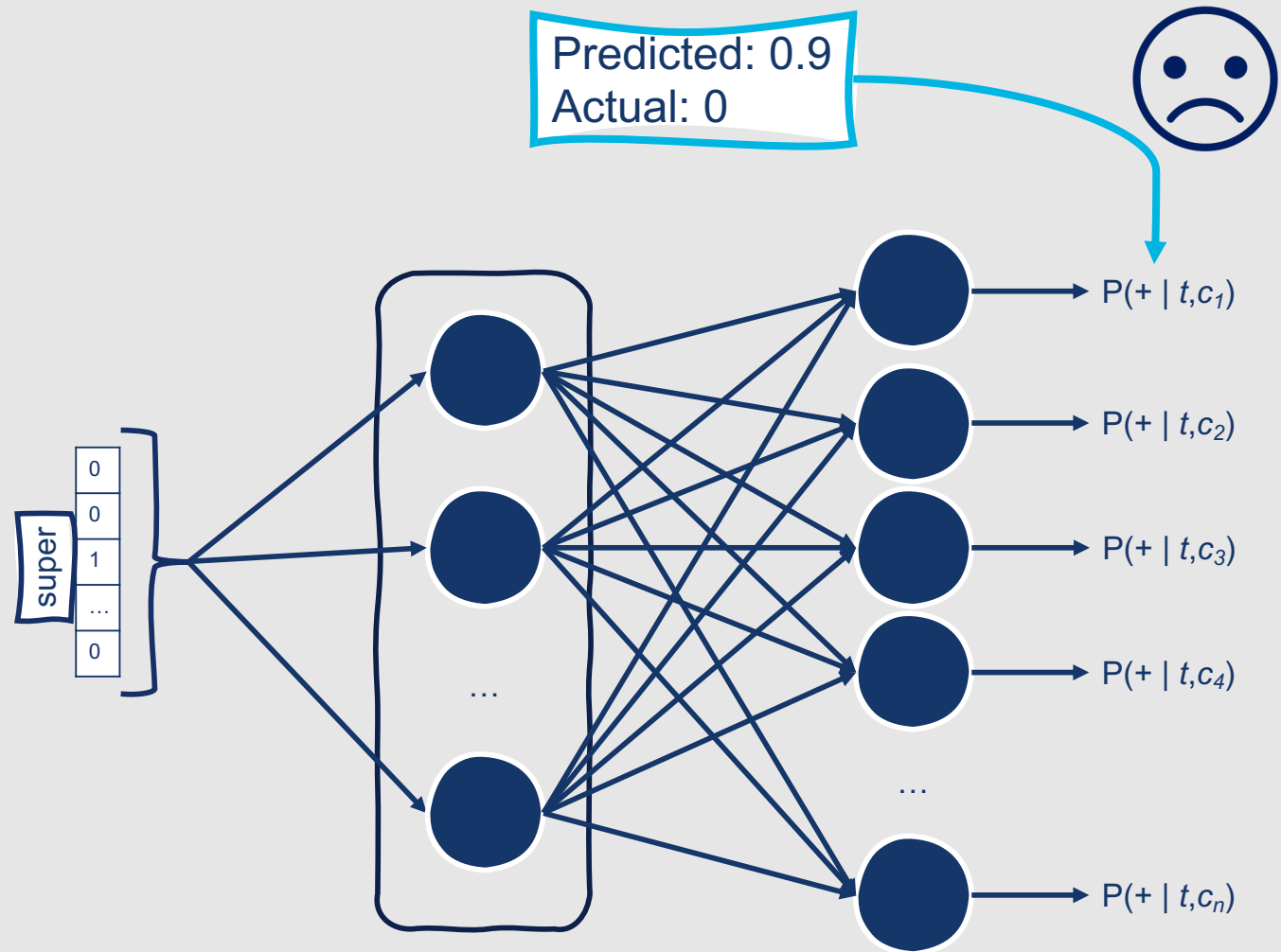




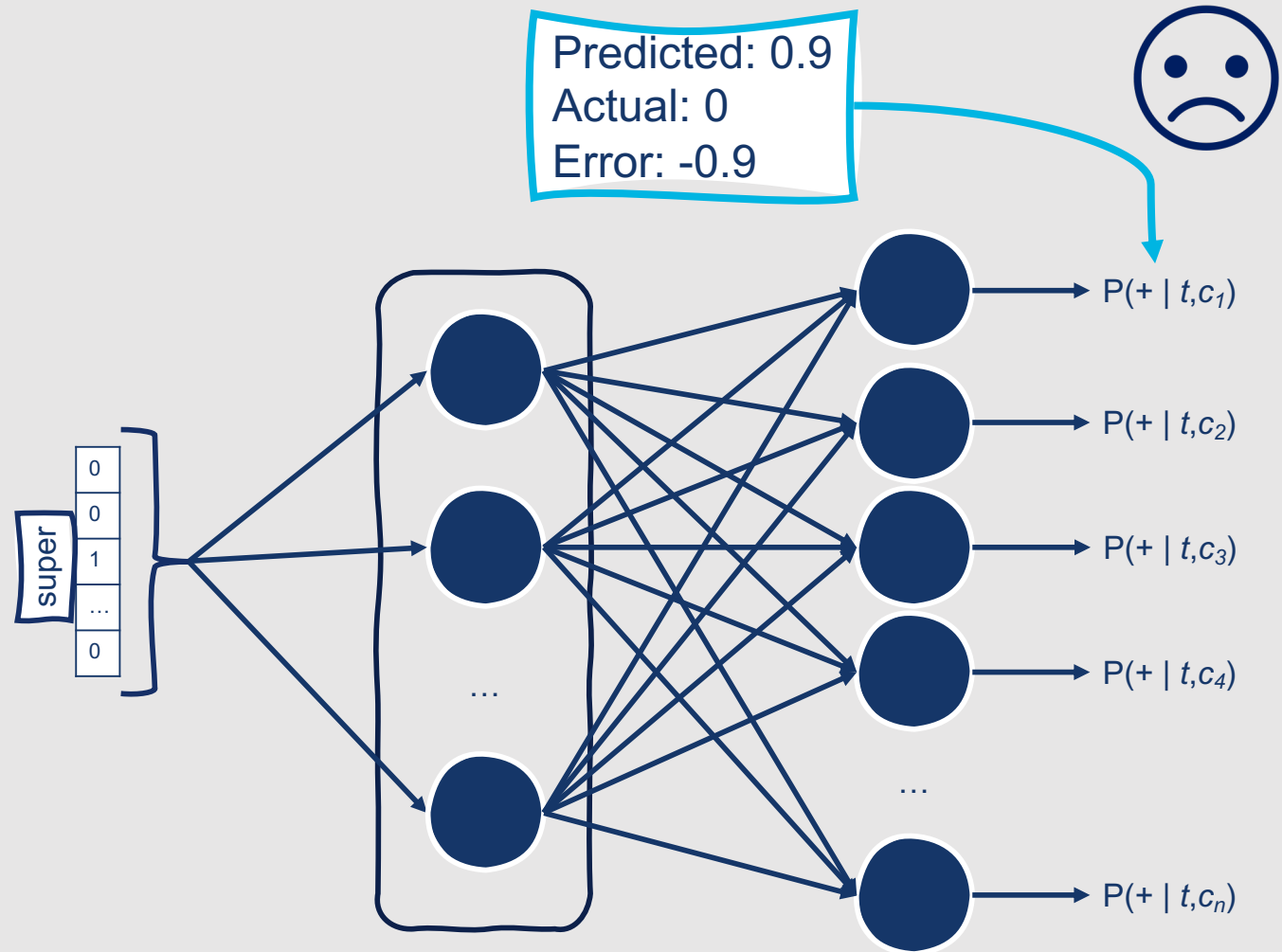
How do we optimize these weights over time?

- Weights are **initialized to some random value** for each word
- They are then iteratively updated to be more similar for words that occur in similar contexts in the training set, and less similar for words that do not
 - Specifically, we want to find weights that maximize $P(+|t,c)$ for words that occur in similar contexts and minimize $P(-|t,c)$ for words that do not, given the information we have at the time

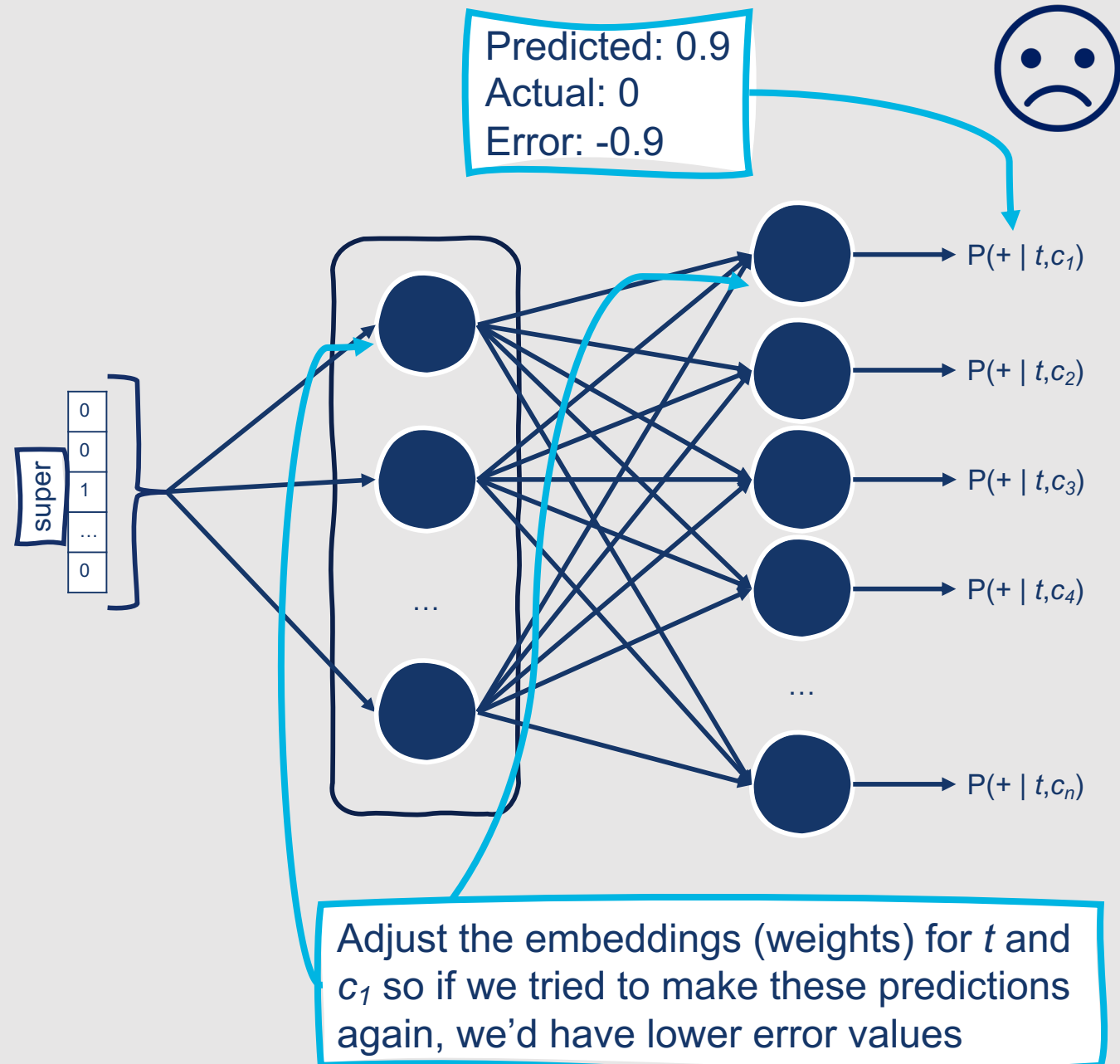
Since we initialize our weights randomly, the classifier's first prediction will almost certainly be wrong.



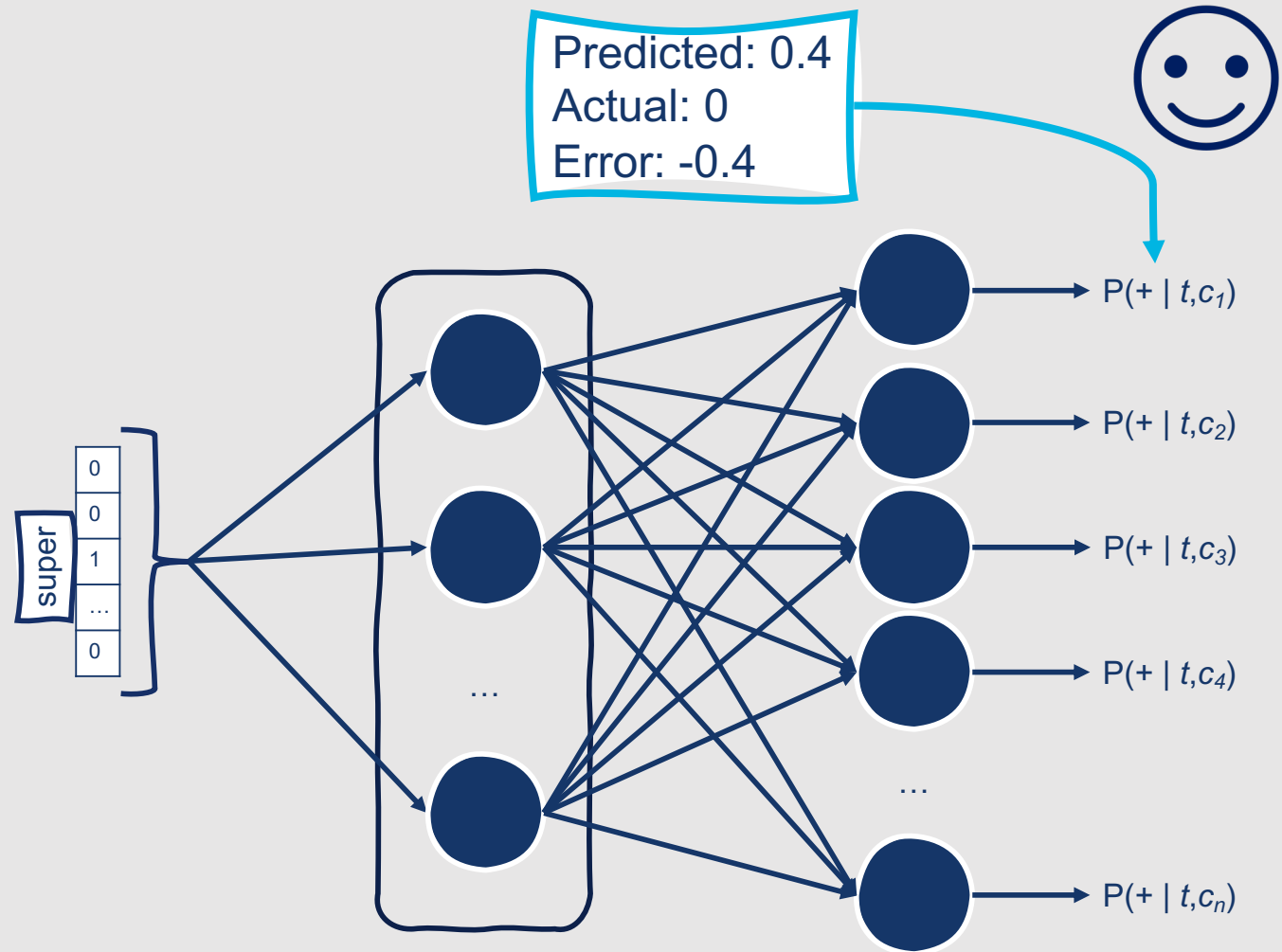
However, the error values from our incorrect guesses are what allow us to improve our embeddings over time.



However, the error values from our incorrect guesses are what allow us to improve our embeddings over time.



However, the error values from our incorrect guesses are what allow us to improve our embeddings over time.





What is our training data?

this	sunday,	watch	the	super	bowl	at	5:30
		c1	c2	t	c3	c4	

Positive Examples

t	c
super	watch
super	the
super	bowl
super	at

- We assume that all occurrences of words in similar contexts in our training corpus are **positive samples**



What is our training data?

this	sunday,	watch	the	super	bowl	at	5:30
		c1	c2	t	c3	c4	

Positive Examples

t	c
super	watch
super	the
super	bowl
super	at

- However, we also need negative samples!
- In fact, Word2Vec uses more negative than positive samples (the exact ratio can vary)
- We need to create our own negative examples



What is our training data?

this	sunday,	watch	the	super	bowl	at	5:30
		c1	c2	t	c3	c4	

Positive Examples

t	c
super	watch
super	the
super	bowl
super	at

Negative Examples

t	c
super	calendar
super	exam
super	loud
super	bread
super	cellphone
super	enemy
super	penguin
super	drive

- How to create negative examples?
 - Target word + “noise” word that is sampled from the training set
 - Noise words are chosen according to their weighted unigram frequency $p_\alpha(w)$, where α is a weight:
 - $p_\alpha(w) = \frac{\text{count}(w)^\alpha}{\sum_{w'} \text{count}(w')^\alpha}$
 - Often, $\alpha = 0.75$ to give rarer noise words slightly higher probability of being randomly sampled
- Randomly select noise words according to weighted unigram frequency

Learning Skip-Gram Embeddings

- The model uses these positive and negative samples to:
 - Maximize the vector similarity of the (target, context) pairs drawn from positive examples
 - Minimize the vector similarity of the (target, context) pairs drawn from negative examples
- Parameters (target and context weight vectors) are fine-tuned by:
 - Applying stochastic gradient descent
 - Optimizing a cross-entropy loss function



What if we want to predict a target word from a set of context words instead?

- **Continuous Bag of Words (CBOW)**
 - Another variation of Word2Vec
- Very similar to skip-gram model!

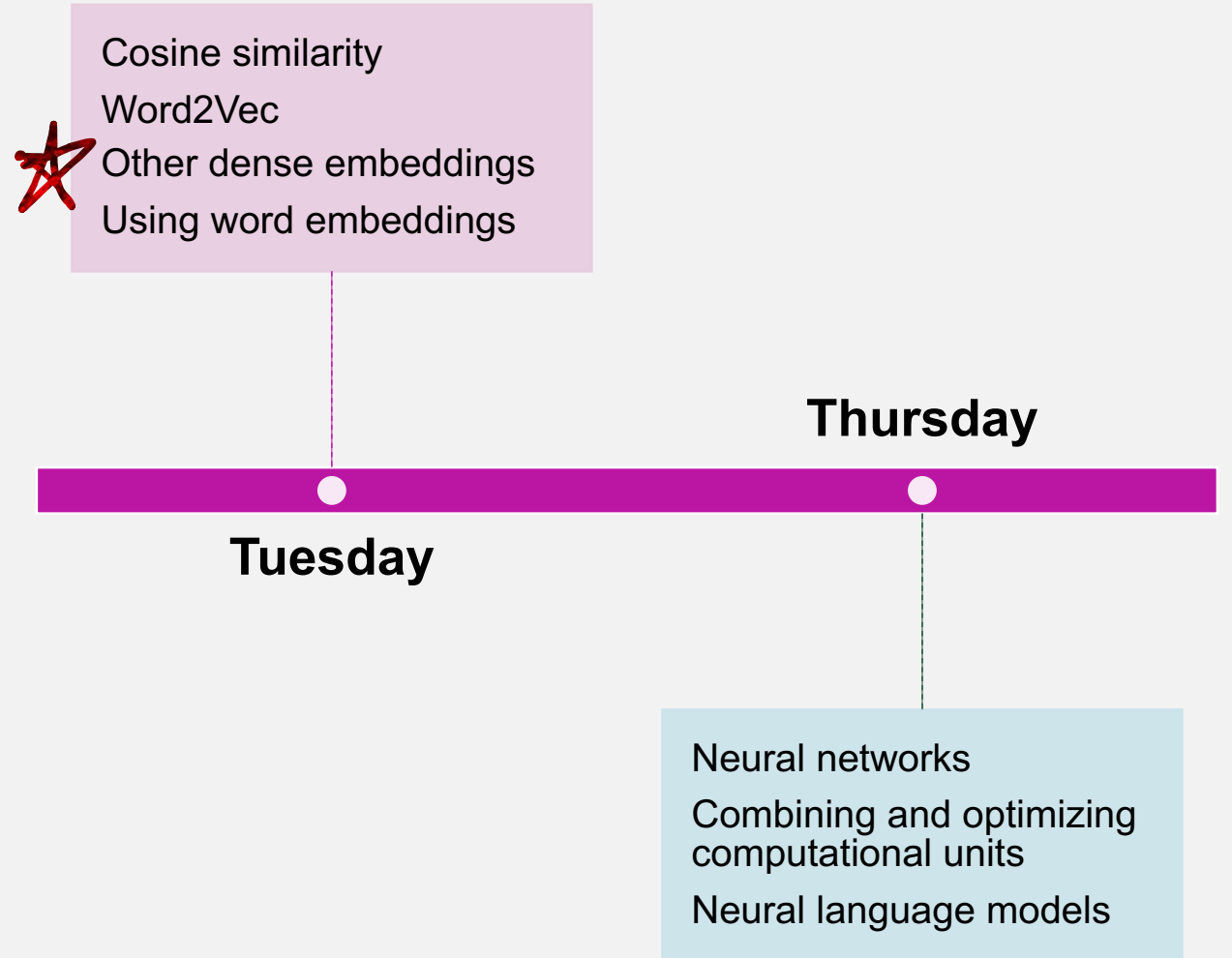
In general, skip-gram embeddings are good with:

- Small datasets
- Rare words and phrases

CBOW embeddings are good with:

- Larger datasets (faster to train)
- Frequent words

This Week's Topics



Are there any other variations of Word2Vec?

- **fastText**
 - An extension of Word2Vec that also incorporates **subwords**
 - Designed to better handle unknown words and sparsity in language

fastText

- Each word is represented as:
 - Itself
 - A bag of constituent n-grams





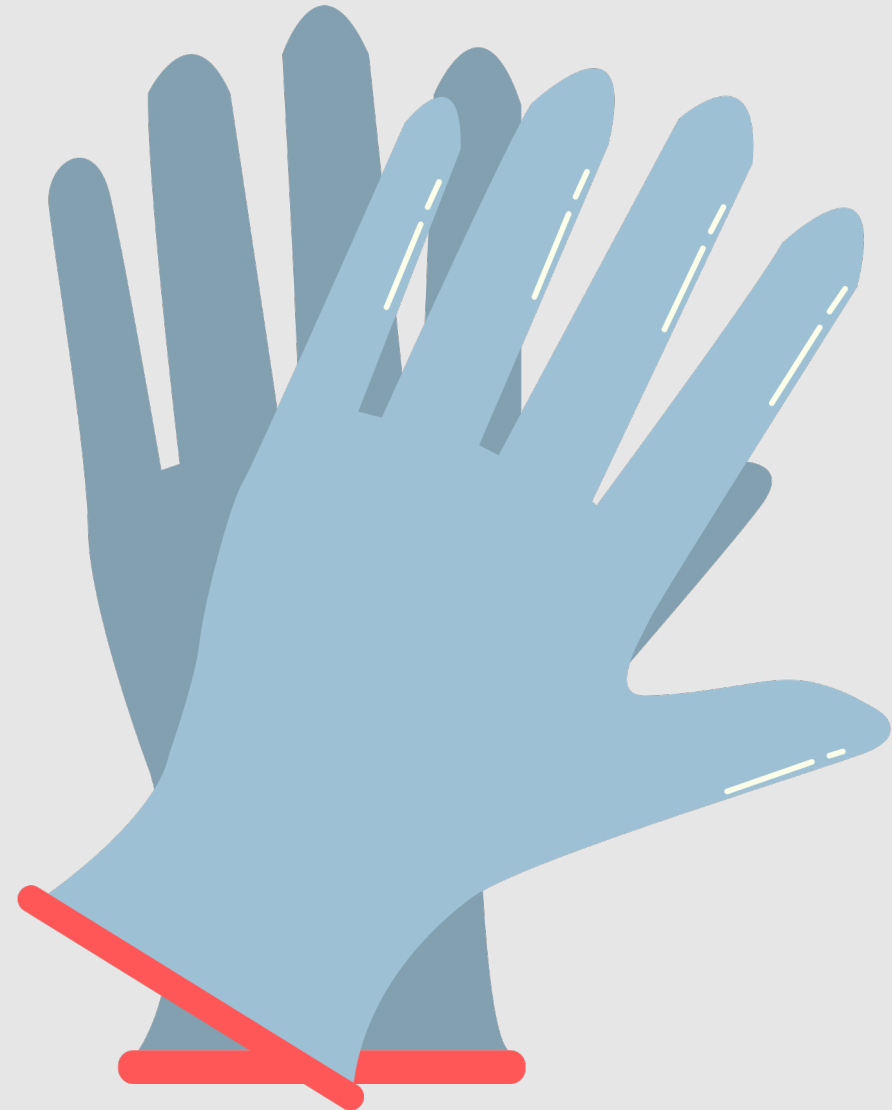
fastText

- Skip-gram embedding is learned for each constituent n-gram
- Word is represented by the sum of all embeddings of its constituent n-grams
- Key advantage of this extension?
 - Allows embeddings to be predicted for unknown words based on subword constituents alone

Source code available online:
<https://fasttext.cc/>

Other Types of Dense Word Embeddings

- Word2Vec is an example of a **predictive** word embedding model
 - Learns to predict whether words belong in a target word's context
- Other models are **count-based**
 - Remember co-occurrence matrices?
- GloVE combines aspects of both predictive and count-based models



Global Vectors for Word Representation (GloVe)

- Co-occurrence matrices quickly grow extremely large
- GloVe learns to predict weights in a lower-dimensional space that correspond to the co-occurrence probabilities between words
- Why is this useful?
 - Predictive models → black box
 - They work, but why?
 - GloVe models are easier to interpret

How does GloVe work?

	c_1	...	c_n
t_1	123	...	456
...
t_n	0	...	789

Build a huge word-context co-occurrence matrix

How does GloVe work?

	c_1	...	c_n
t_1	123	...	456
...
t_n	0	...	789

Build a huge word-context co-occurrence matrix

Define soft constraints for each word pair

Scaler biases for t_i and c_j

$$w_i^T w_j + b_i + b_j = \log X_{ij}$$

Vector for t_i

Vector for c_j

Co-occurrence count for $t_i c_j$

How does GloVe work?

	c_1	...	c_n
t_1	123	...	456
...
t_n	0	...	789

Build a huge word-context co-occurrence matrix

Define soft constraints for each word pair

$$w_i^T w_j + b_i + b_j = \log X_{ij}$$

Define a cost function

$$J = \sum_{i=1}^V \sum_{j=1}^V f(X_{ij}) (w_i^T w_j + b_i + b_j - \log X_{ij})^2$$

Weighting function:

$$f(X_{ij}) = \begin{cases} \left(\frac{X_{ij}}{x_{max}}\right)^\alpha, & X_{ij} < XMAX \\ 1, & \text{otherwise} \end{cases}$$

How does GloVe work?

	c_1	...	c_n
t_1	123	...	456
...
t_n	0	...	789

Build a huge word-context co-occurrence matrix

Define soft constraints for each word pair

$$w_i^T w_j + b_i + b_j = \log X_{ij}$$

Define a cost function

$$J = \sum_{i=1}^V \sum_{j=1}^V f(X_{ij}) (w_i^T w_j + b_i + b_j - \log X_{ij})^2$$

Minimize the cost function to learn ideal embedding values for w_i and w_j

How does GloVe work?

	c_1	...	c_n
t_1	123	...	456
...
t_n	0	...	789

Build a huge word-context co-occurrence matrix

Define soft constraints for each word pair

$$w_i^T w_j + b_i + b_j = \log X_{ij}$$

Define a cost function

$$J = \sum_{i=1}^V \sum_{j=1}^V f(X_{ij}) (w_i^T w_j + b_i + b_j - \log X_{ij})^2$$

Minimize the cost function to learn ideal embedding values for w_i and w_j

0.4 0.7 1.2 4.3 0.9 6.7 1.3 0.5 0.7 5.3

Why does GloVe work?

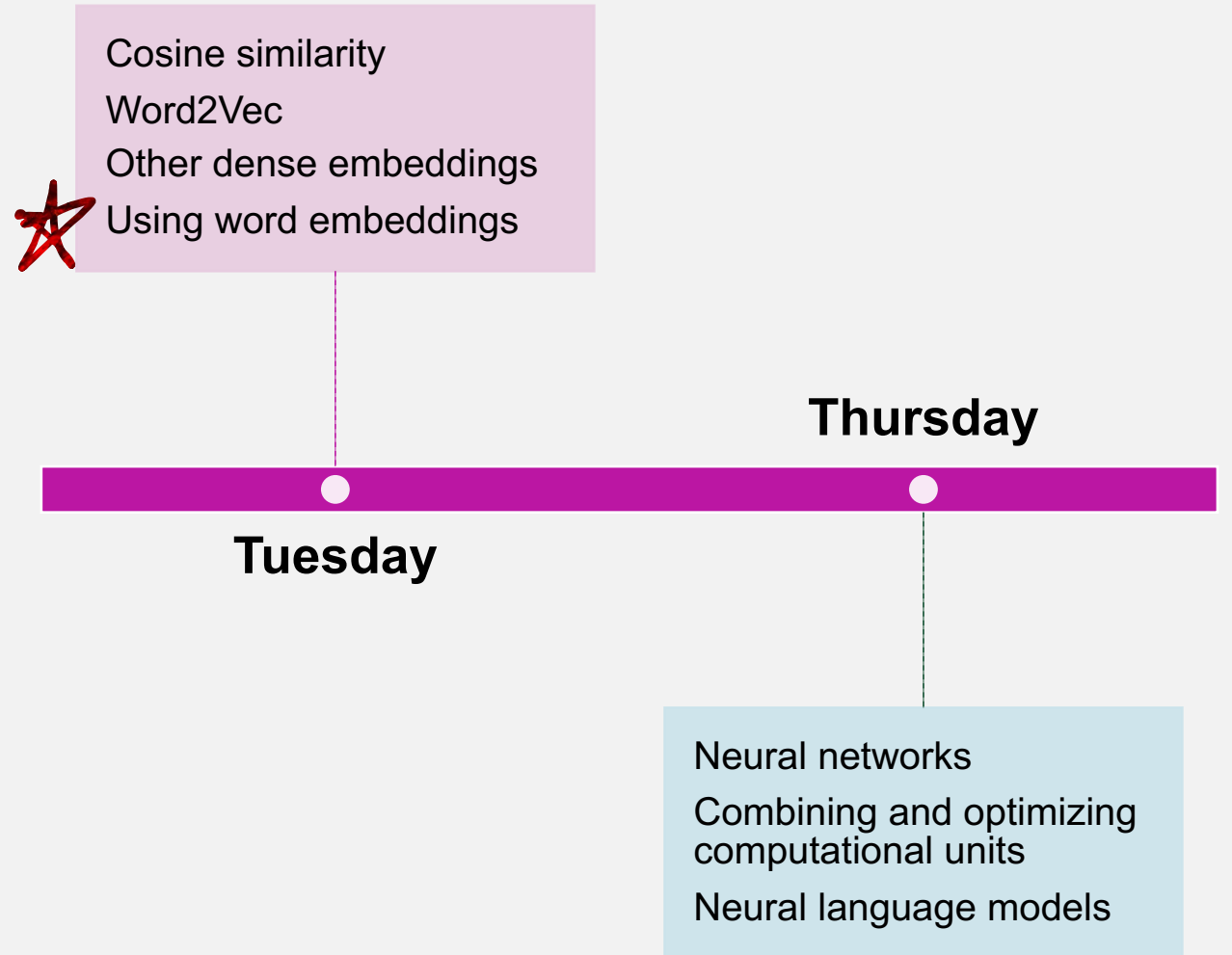
- Ratios of co-occurrence probabilities have the potential to encode word similarities and differences
- These similarities and differences are useful components of meaning
 - GloVe embeddings perform particularly well on analogy tasks



Which is better ... Word2Vec or GloVe?

- In general, Word2Vec and GloVe produce similar embeddings
- Word2Vec → slower to train but less memory intensive
- GloVe → faster to train but more memory intensive
- Word2Vec and GloVe both produce context-independent embeddings
- Contextual embeddings:
 - ELMo (Peters et al., 2018; <https://www.aclweb.org/anthology/N18-1202/>)
 - BERT (Devlin et al., 2019; <https://www.aclweb.org/anthology/N19-1423/>)

This Week's Topics



Evaluating Vector Models

- Extrinsic Evaluation
 - Add the vectors as features in a downstream NLP task, and see whether and how this changes performance relative to a baseline model
 - Most important evaluation metric for word embeddings!
 - Word embeddings are rarely needed in isolation
 - They are almost solely used to boost performance in downstream tasks
- Intrinsic Evaluation
 - Performance at predicting:
 - Word similarity
 - Text similarity
 - Analogy



Evaluating Performance at Predicting Word Similarity

- Compute the **cosine similarity** between vectors for pairs of words
- Compute the **correlation** between those similarity scores and word similarity ratings for the same pairs of words manually assigned by humans
- Corpora for doing this:
 - WordSim-353
 - SimLex-999
 - TOEFL Dataset
 - *Levied* is closest in meaning to: (a) imposed, (b) believed, (c) requested, (d) correlated

Analogy

- We can capture **relational meanings** in word embeddings by computing the offsets between values in the same columns for different vectors
- Famous examples (Mikolov et al., 2013; Levy and Goldberg, 2014):
 - king - man + woman = queen
 - Paris - France + Italy = Rome





Context window size influences what you learn!

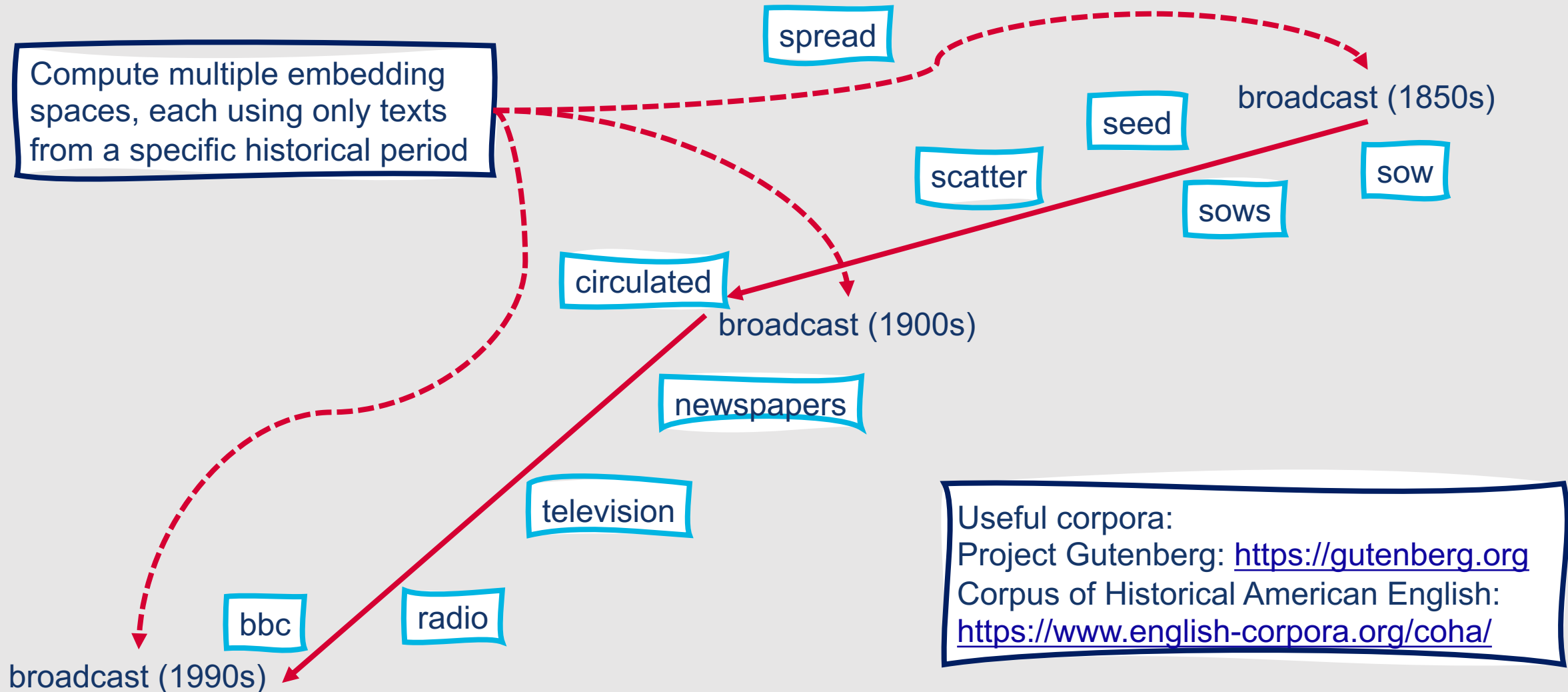
- Shorter context window → more **syntactic representations**
 - Information is from immediately nearby words
 - Most similar words tend to be semantically similar words with the same parts of speech
- Longer context window → more **topical representations**
 - Information can come from longer-distance dependencies
 - Most similar words tend to be topically related, but not necessarily similar (e.g., *diner* and *eats*, rather than *spoon* and *fork*)

Word embeddings have many practical applications.



- Features for text classification tasks
- Representations for computational social science studies
 - Studying word meaning over time
 - Studying implicit associations between words

Embeddings and Historical Semantics



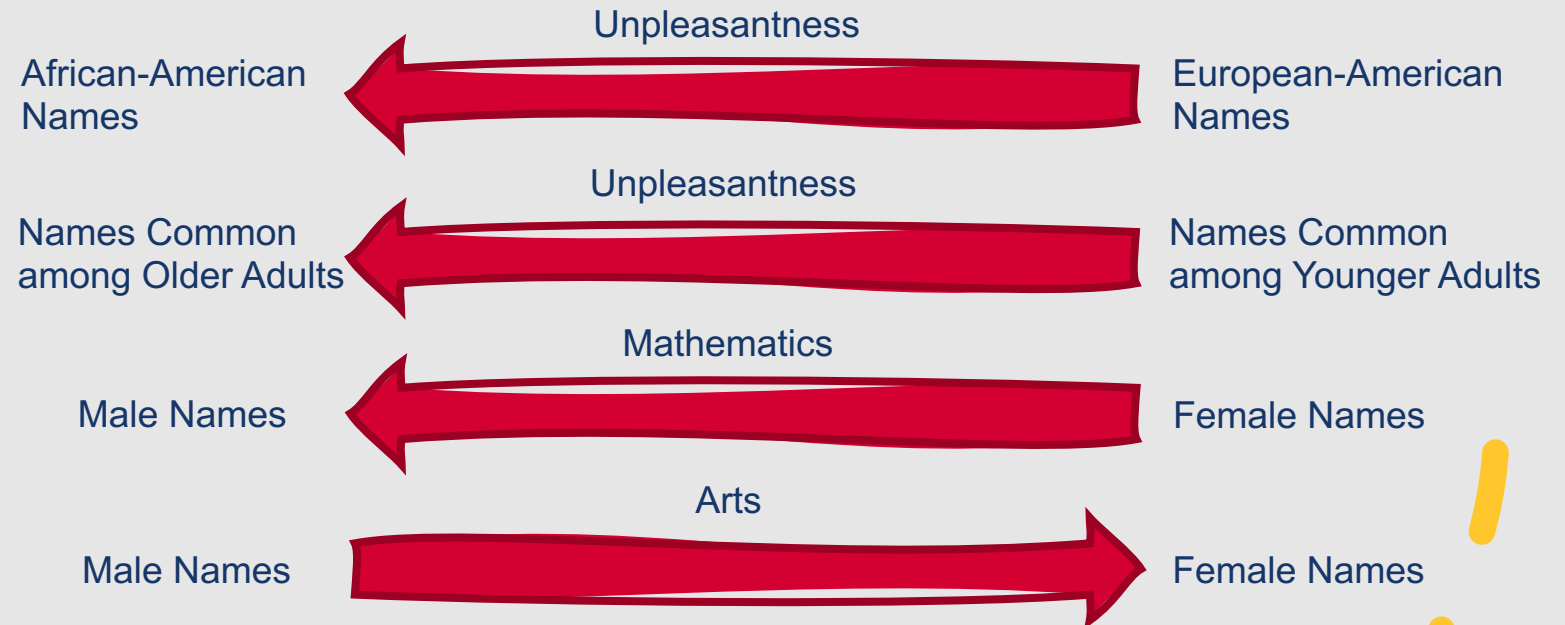
Unfortunately, word embeddings can also end up reproducing implicit biases and stereotypes latent in text.

- Recall: king - man + woman = queen
- Word embeddings trained on news corpora also produce:
 - man - computer programmer + woman = homemaker
 - doctor - father + mother = nurse
- Very problematic for real-world applications (e.g., CV/resume scoring models)



Bias and Embeddings

- Caliskan et al. (2017) identified known, harmful implicit associations in GloVe embeddings



How do we keep the useful associations present in word embeddings, but get rid of the harmful ones?

- Recent research has begun examining ways to **debias** word embeddings by:
 - Transforming embedding spaces to remove gender stereotypes but preserve definitional gender
 - Changing training procedures to eliminate these issues before they arise
- Increasingly active area of study:
 - <https://facctconference.org>



Summary: Word Embeddings

- **Cosine similarity**, commonly used to calculate word vector similarity, measures the distance between vectors by computing the normalized dot product between them
- **Word2Vec** is a **predictive** word embedding approach that learns word representations by training a classifier to predict whether a **context word** should be associated with a given **target word**
- **GloVe** is a **count-based** word embedding approach that learns an optimized, lower-dimensional version of a co-occurrence matrix
- **Word embeddings** can be evaluated through their incorporation in other language tasks, and they can be used to model syntactic and semantic properties of language over time
- Word embeddings may reflect the same **biases** found in the data used to train them

This Week's Topics

Cosine similarity
Word2Vec
Other dense embeddings
Using word embeddings



Tuesday

Thursday



Neural networks
Combining and optimizing computational units
Neural language models

+

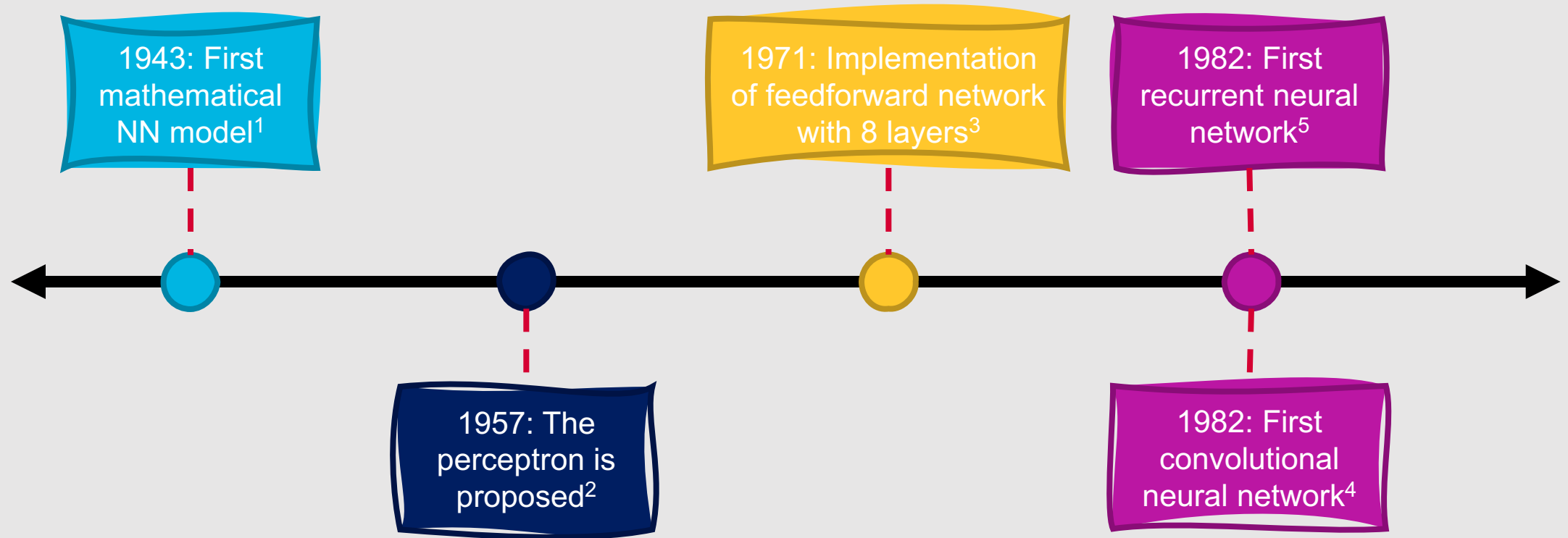
•

○

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

Are neural networks new?



¹McCulloch, 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.

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

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

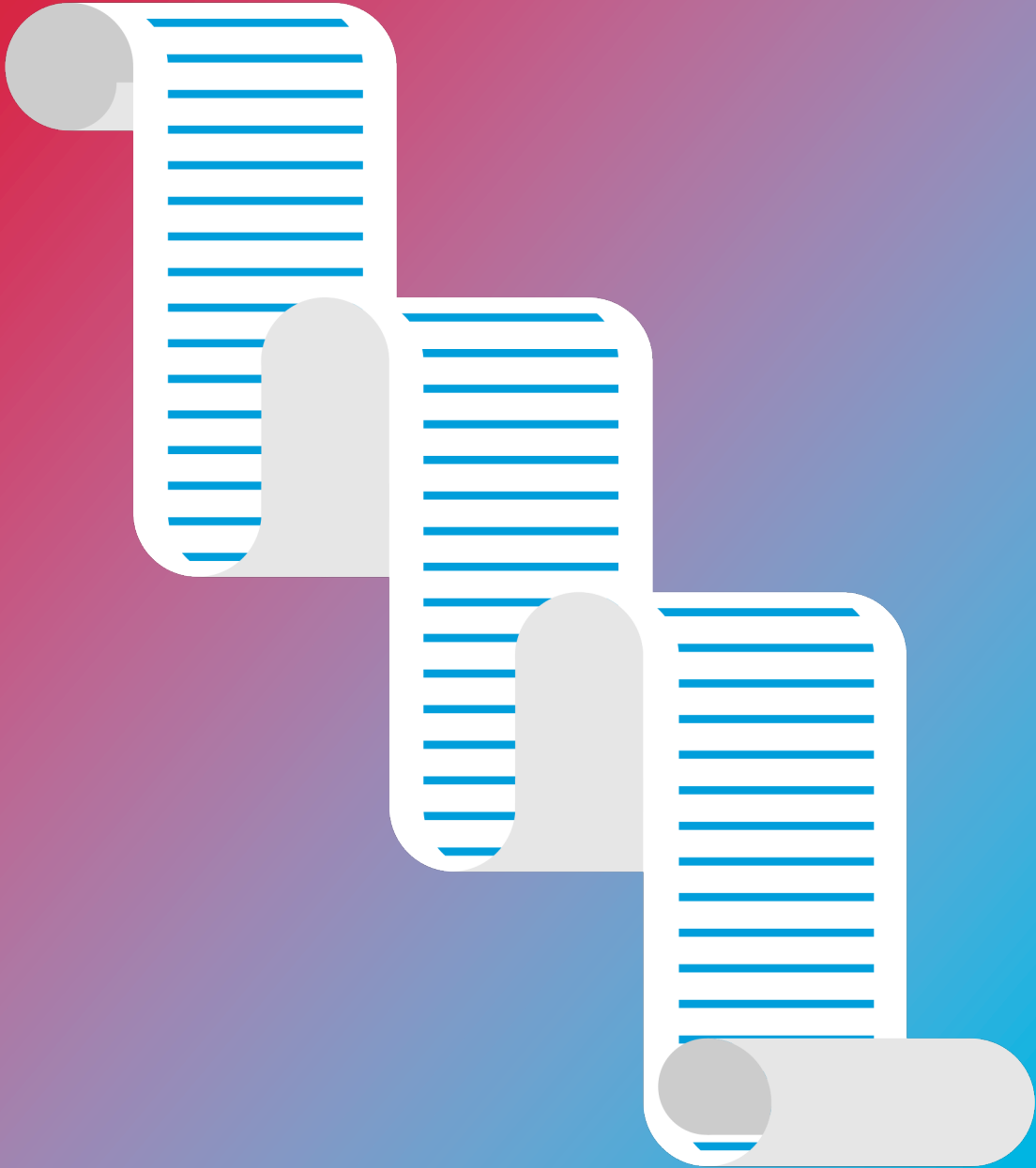
⁴Fukushima, K., & Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets* (pp. 267-285). Springer, Berlin, Heidelberg.

⁵Hopfield, 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





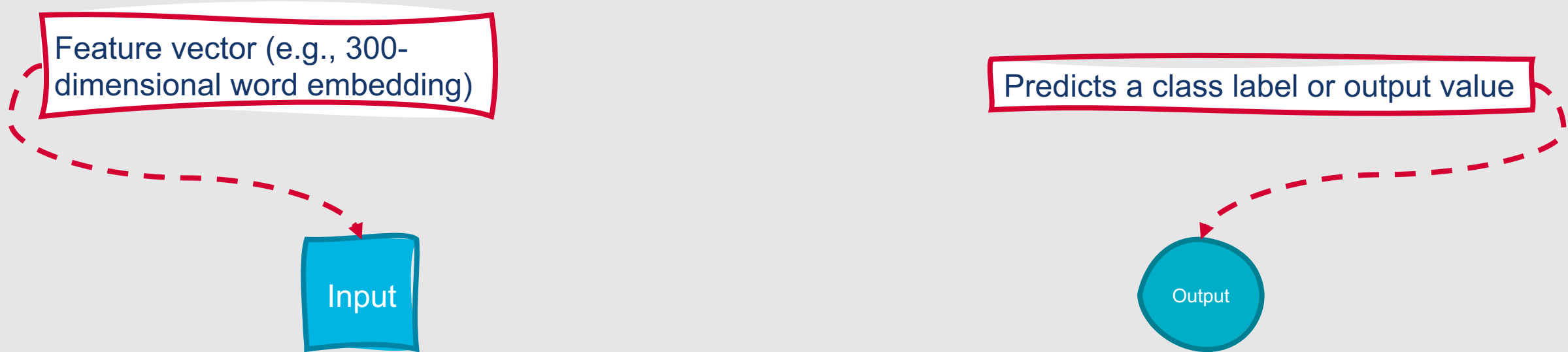
There are many types of neural networks!

- Feedforward neural networks
- Recurrent neural networks
- Convolutional neural networks
- Transformers
-

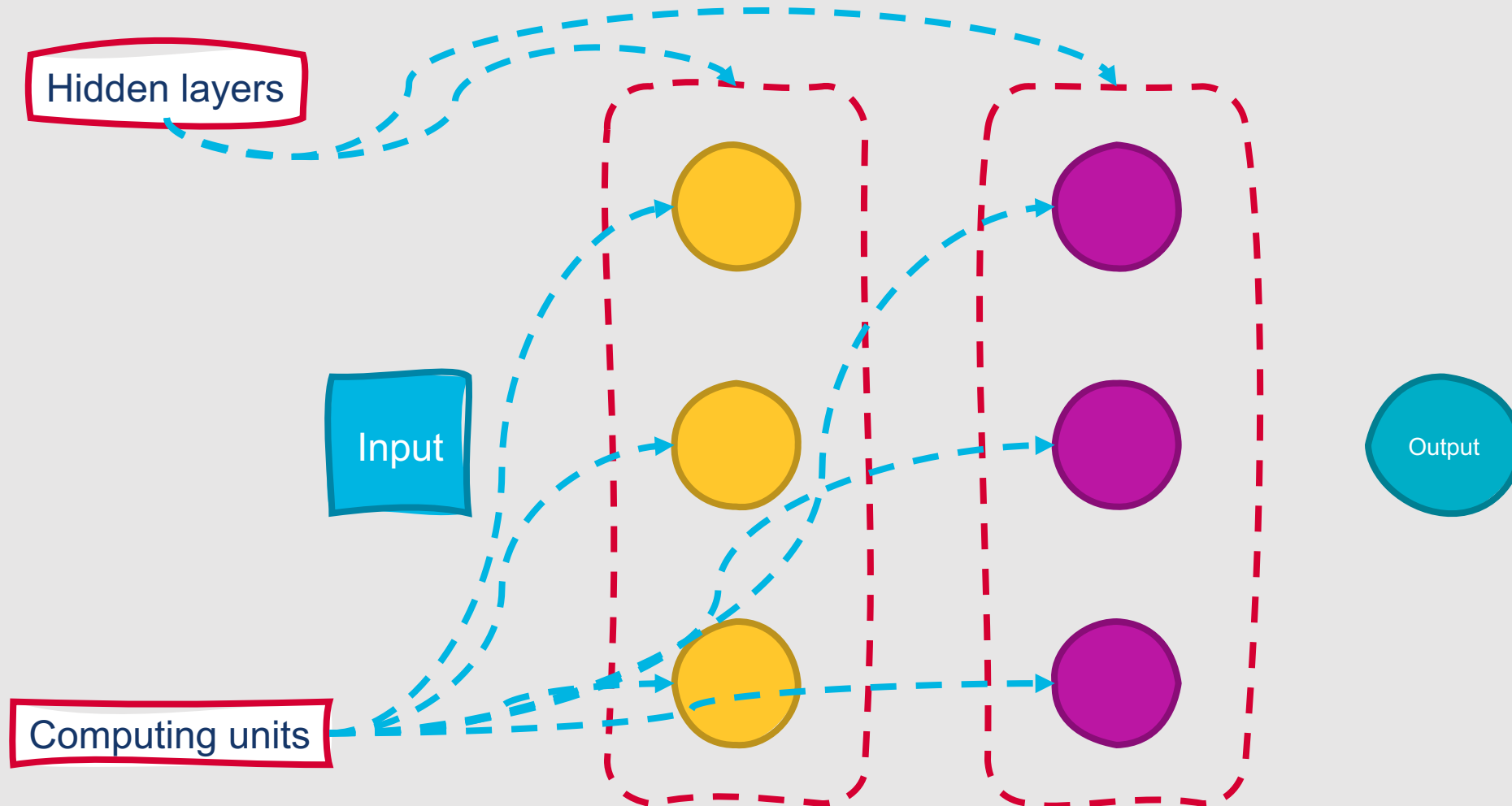
Feedforward Neural Networks

- 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

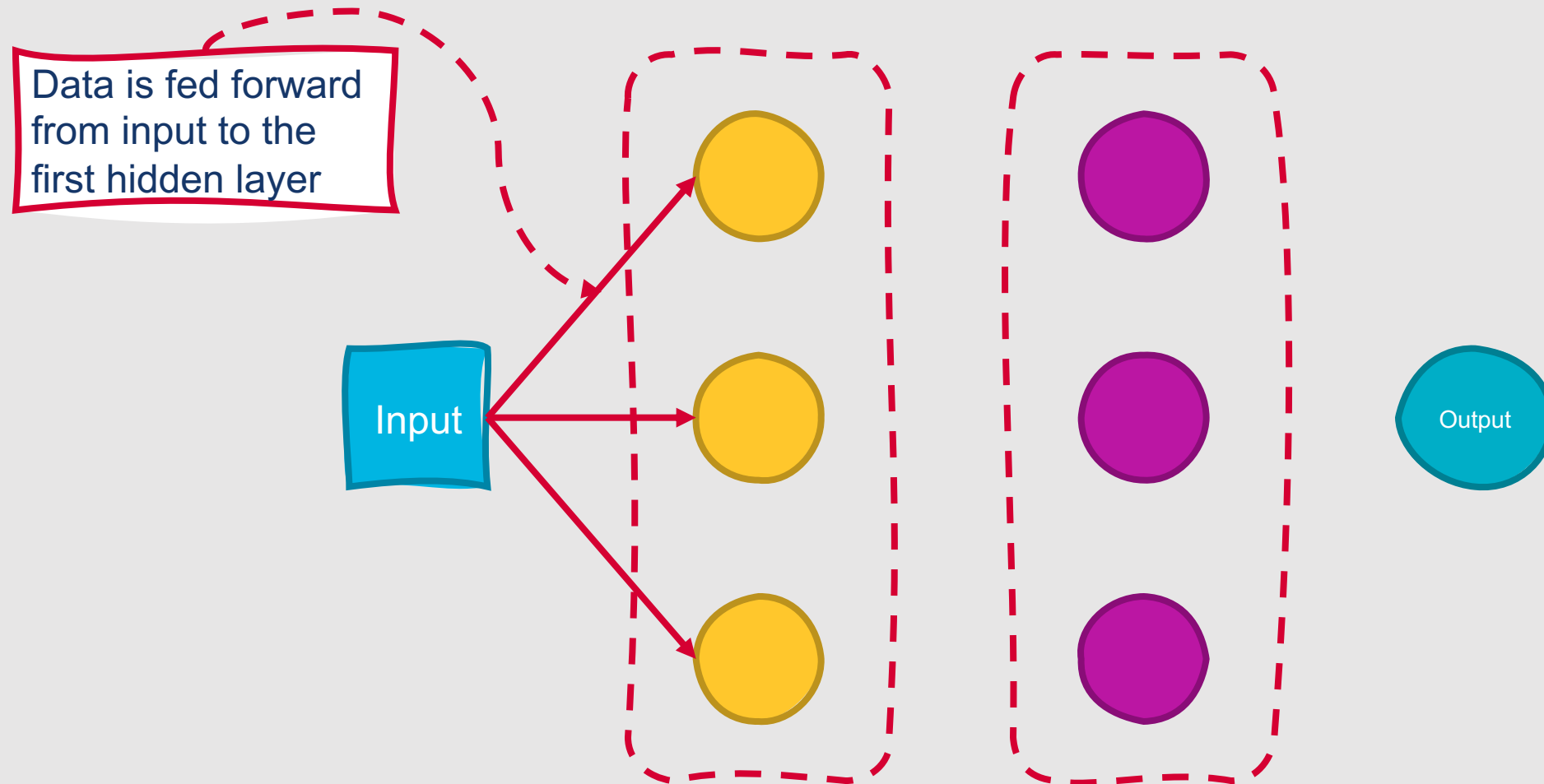
Feedforward Neural Networks



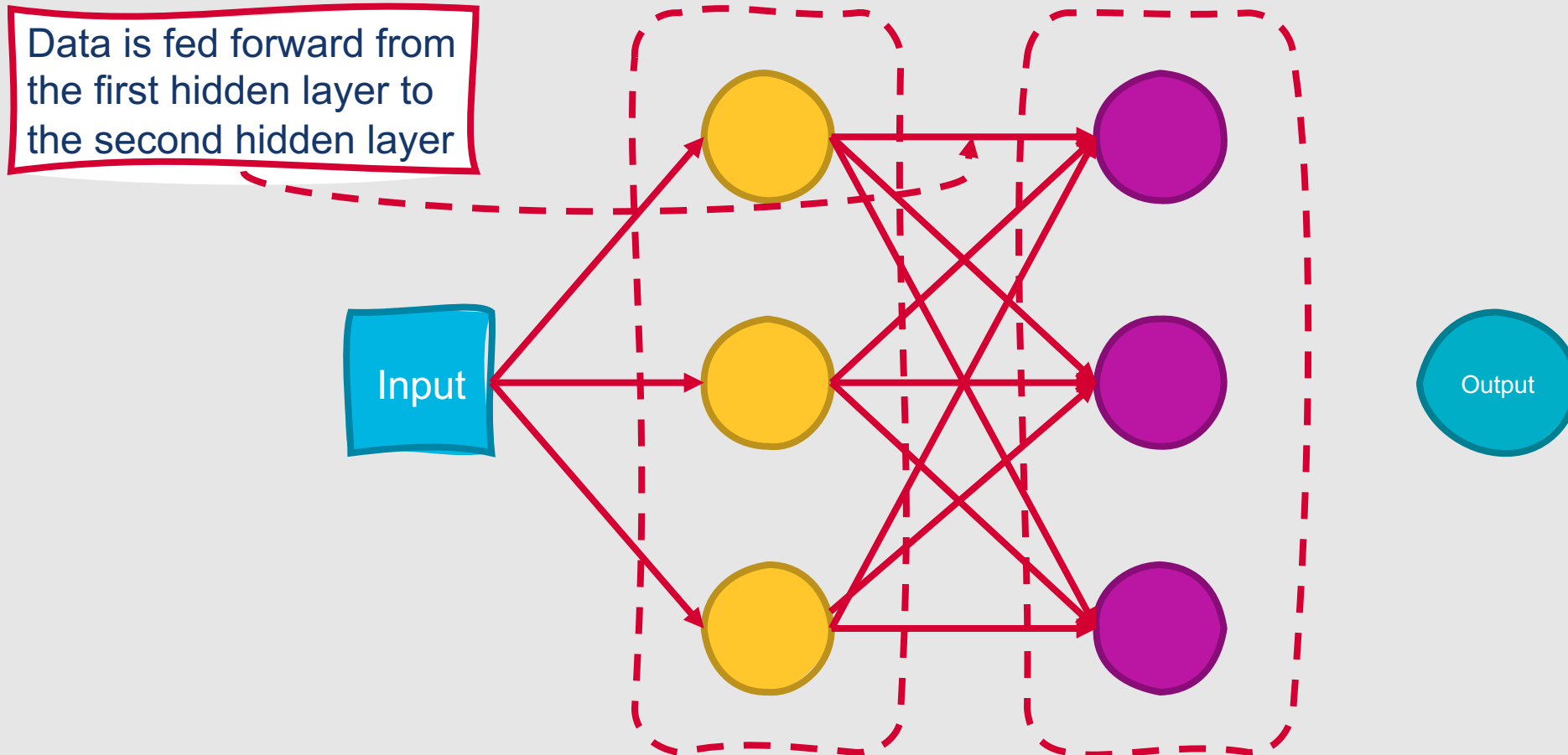
Feedforward Neural Networks



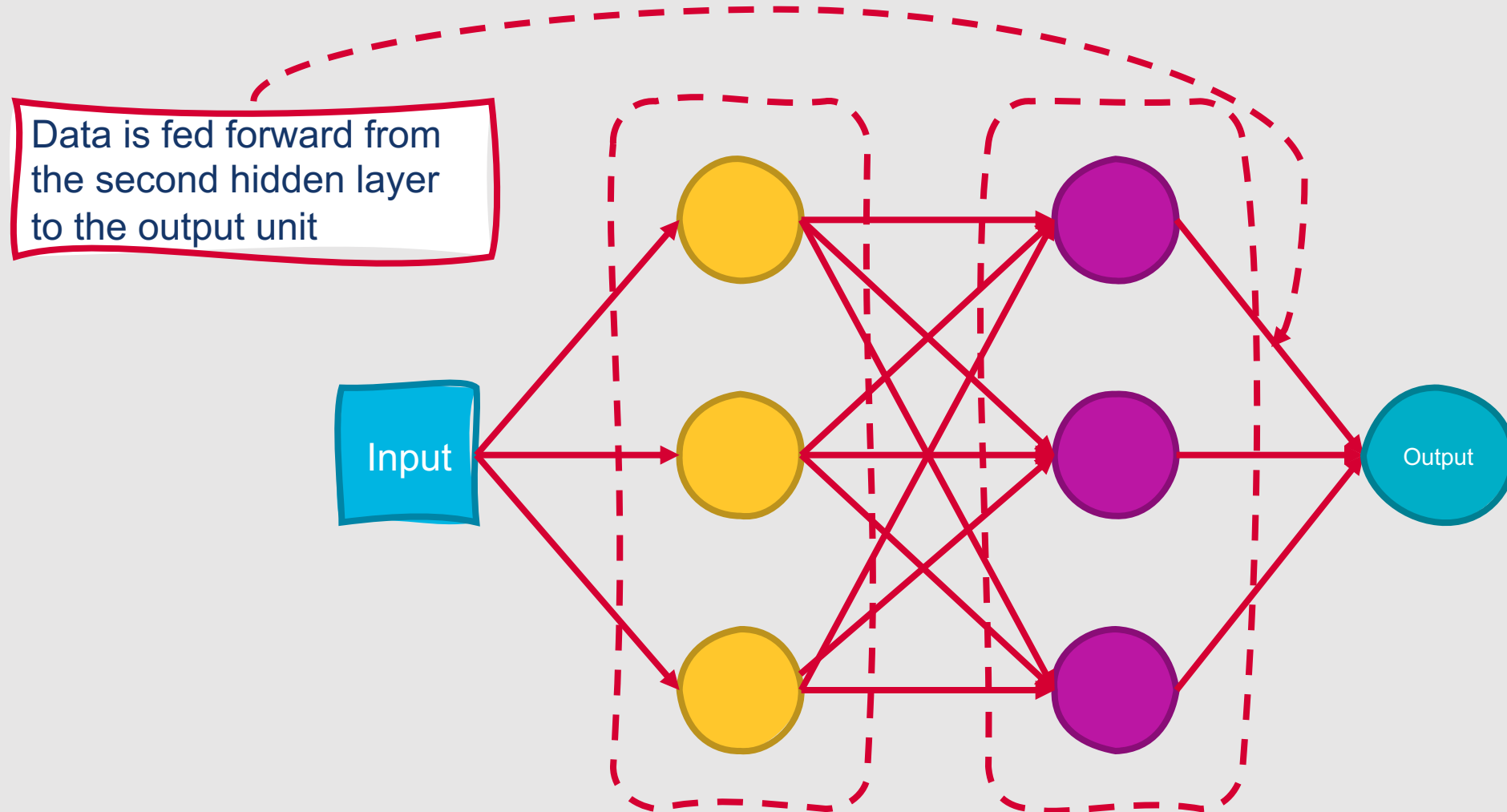
Feedforward Neural Networks



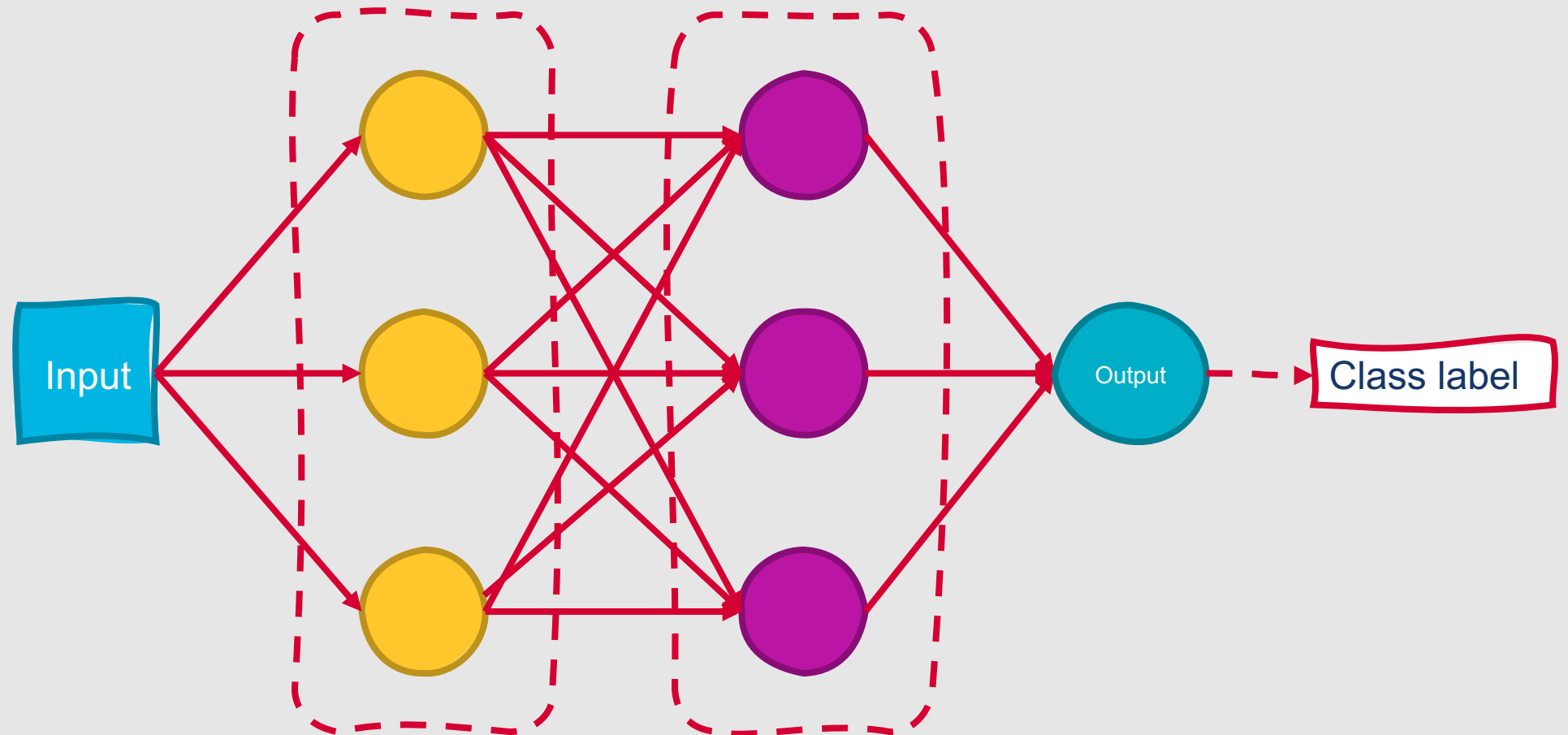
Feedforward Neural Networks



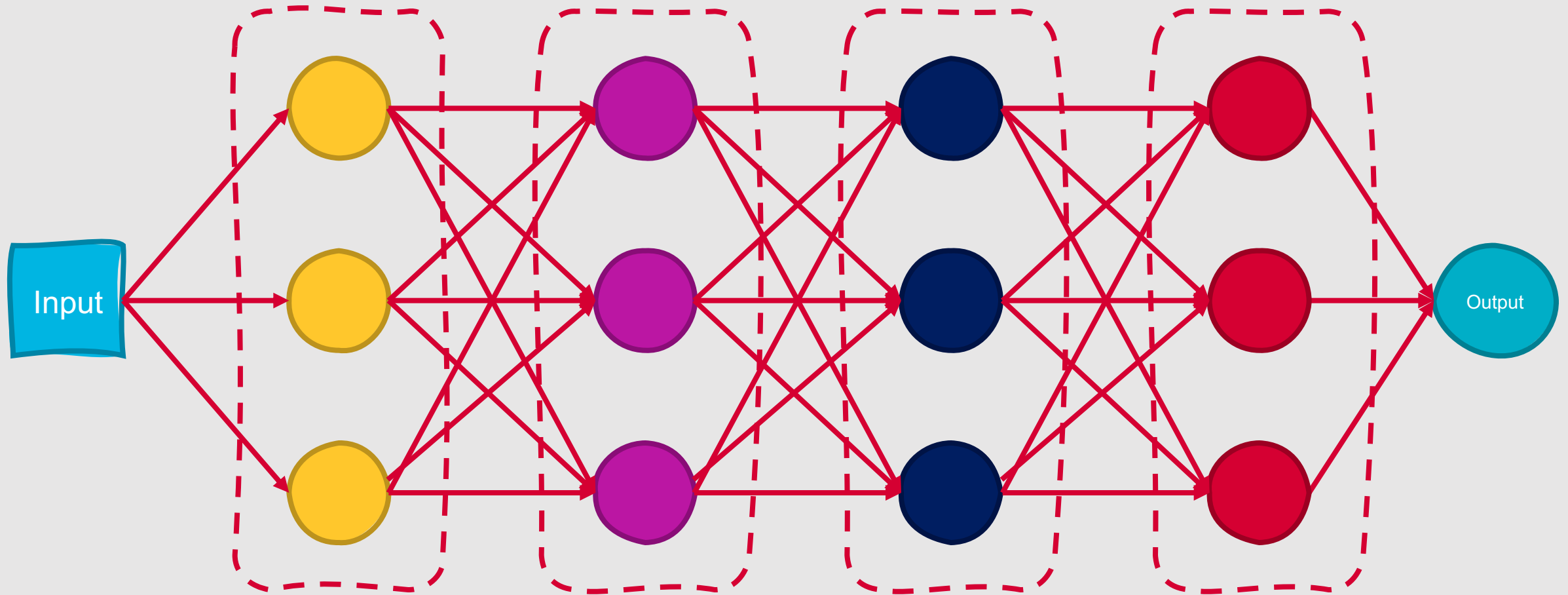
Feedforward Neural Networks



Feedforward Neural Networks

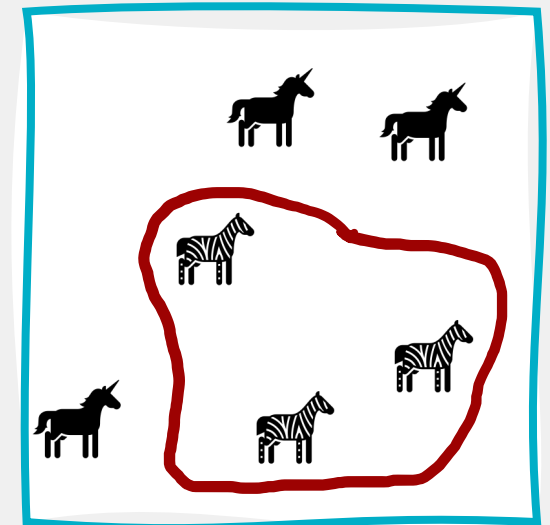
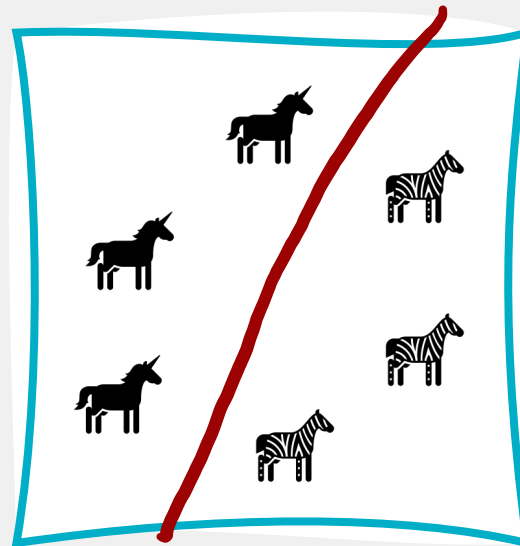


People often refer to multi-layer neural networks as “deep learning.”



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

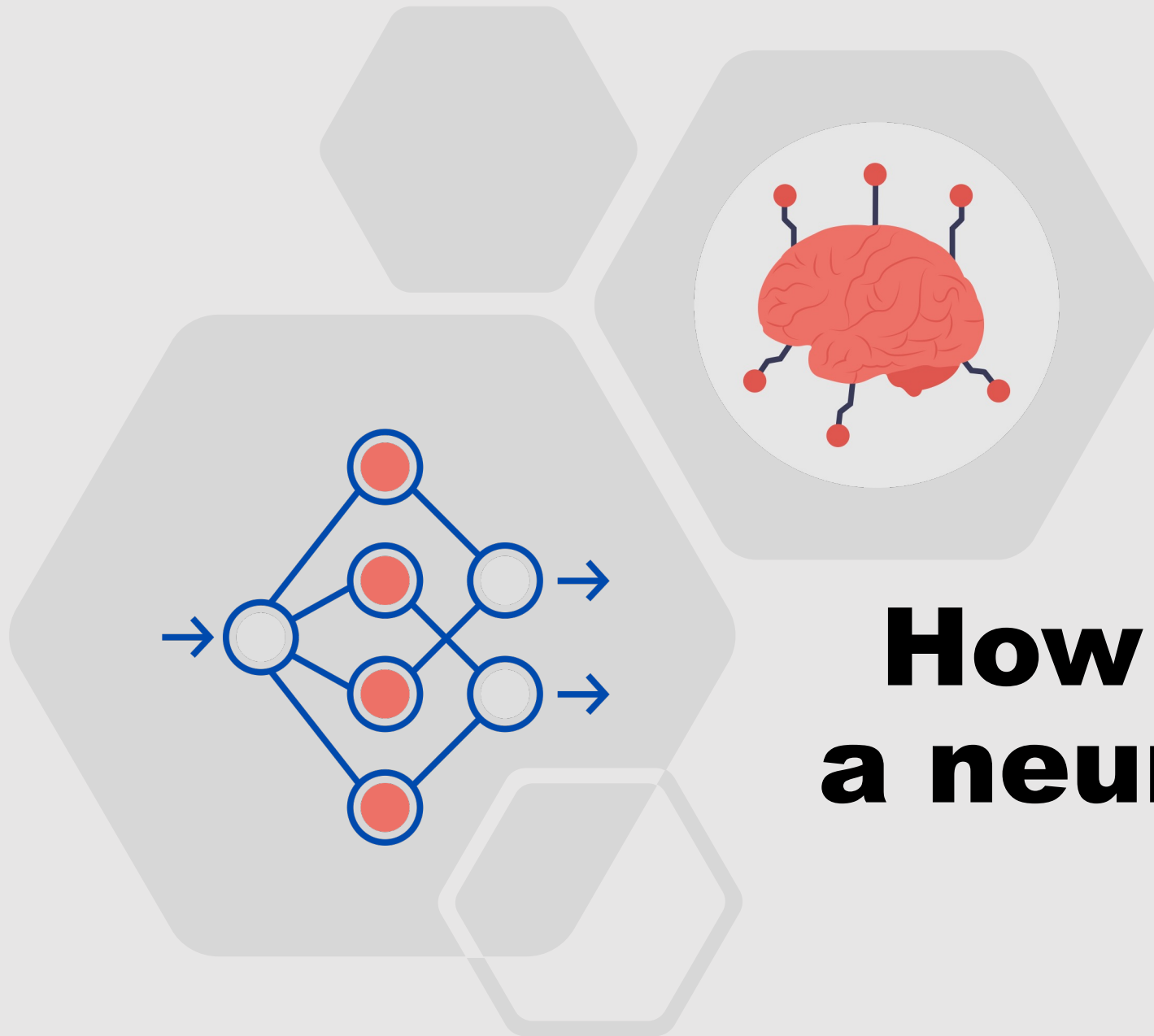


**Neural
networks
aren't
necessarily
the best
classifier
for all
tasks!**

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

In general, deeper network →
more data needed

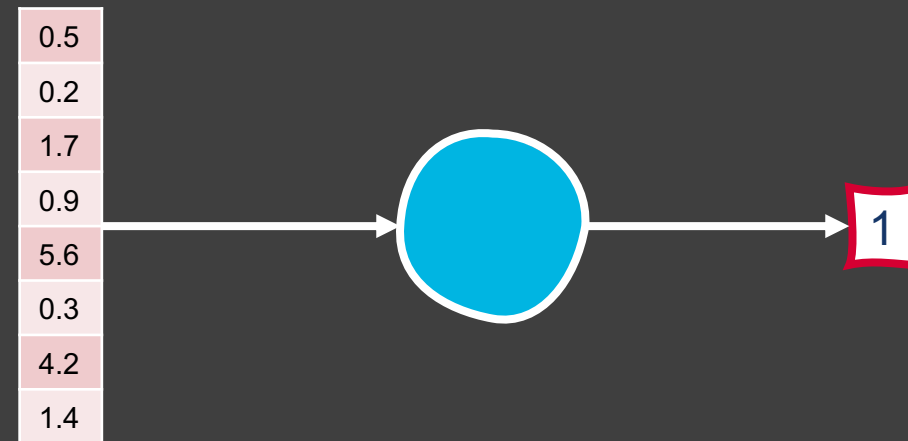
Neural nets tend to work very well
for large-scale problems, but not
as well for small-scale problems



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:
 - $z = \mathbf{w} \cdot \mathbf{x} + b$

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

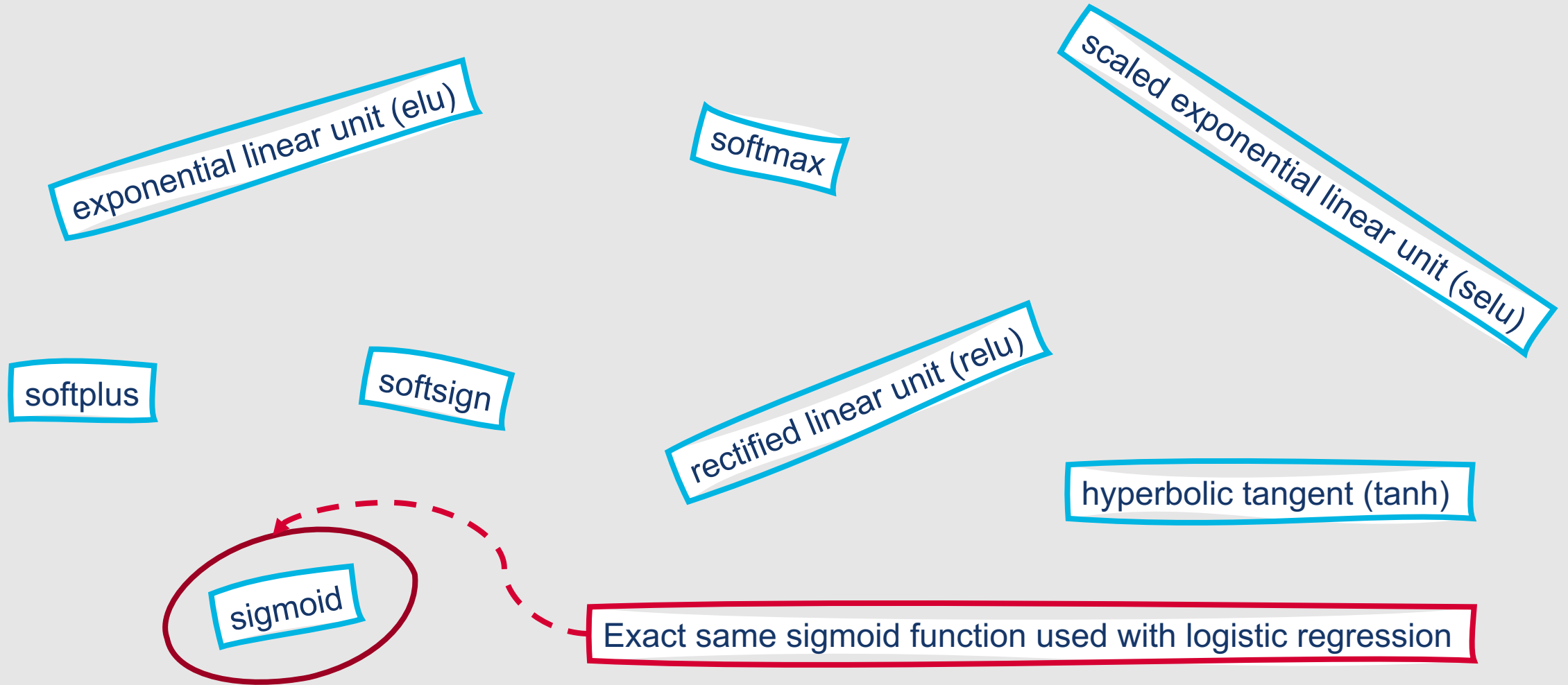
softsign

rectified linear unit (relu)

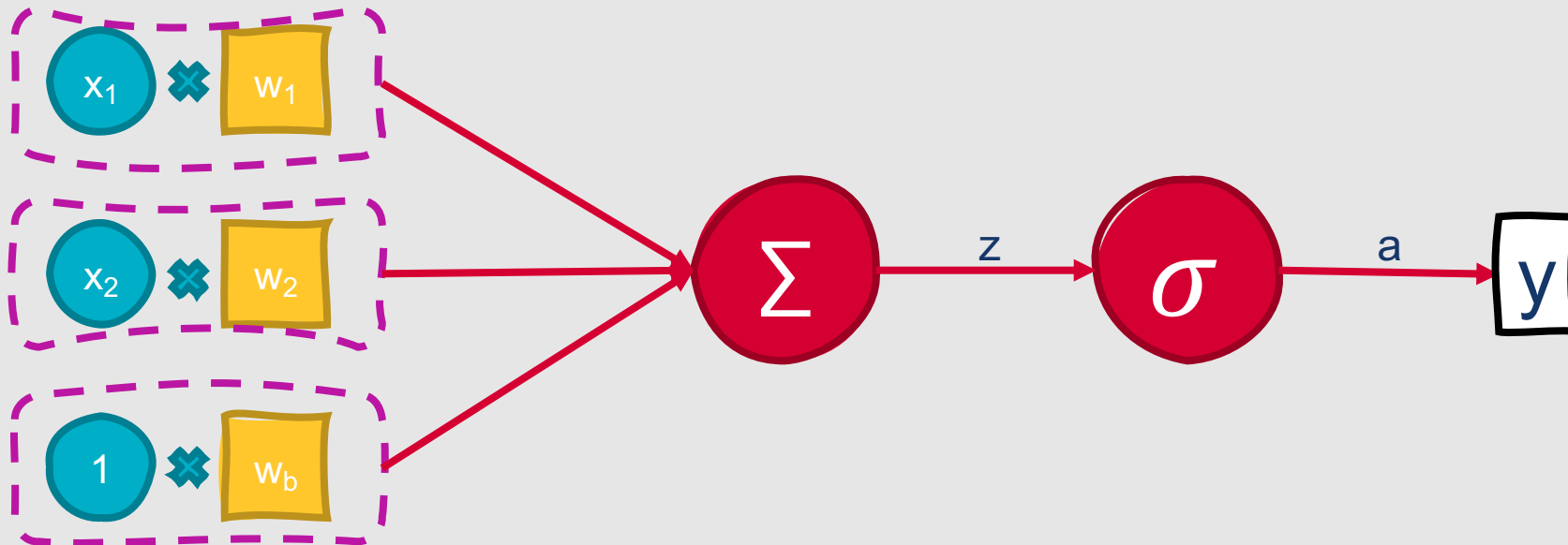
hyperbolic tangent (tanh)

sigmoid

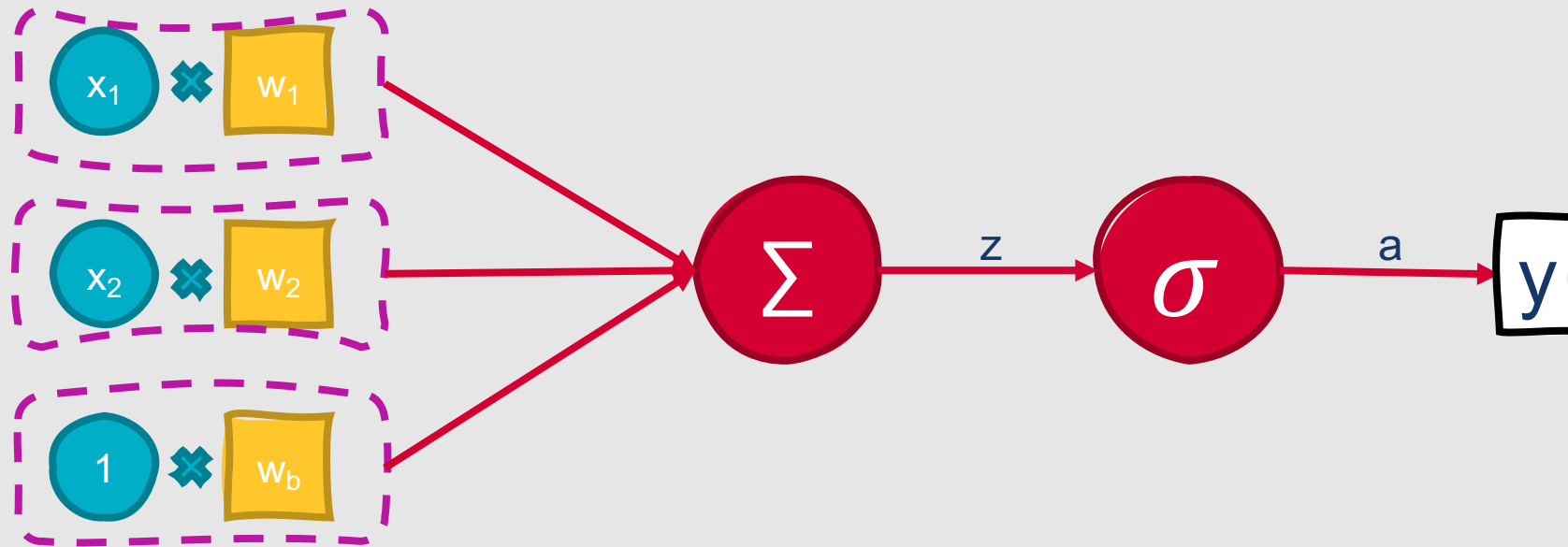
There are many different activation functions!



Computational Unit with Sigmoid Activation



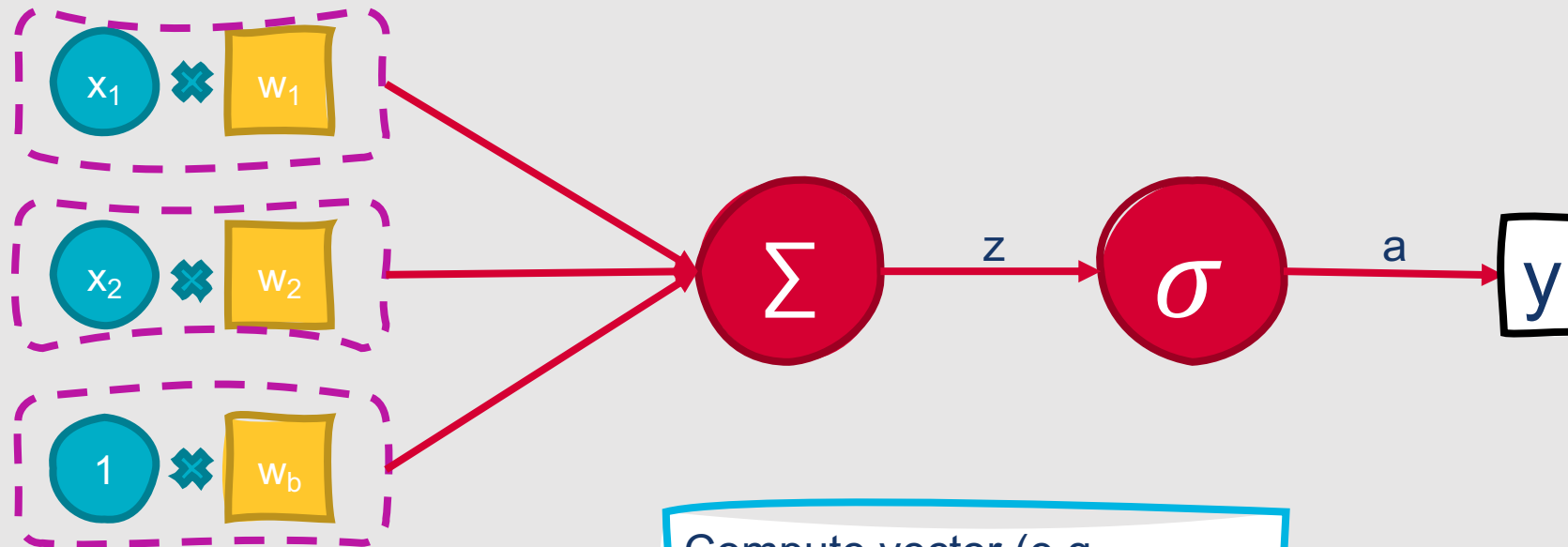
Example: Computational Unit with Sigmoid Activation



Input: “beautiful brutalist architecture”

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

Example: Computational Unit with Sigmoid Activation



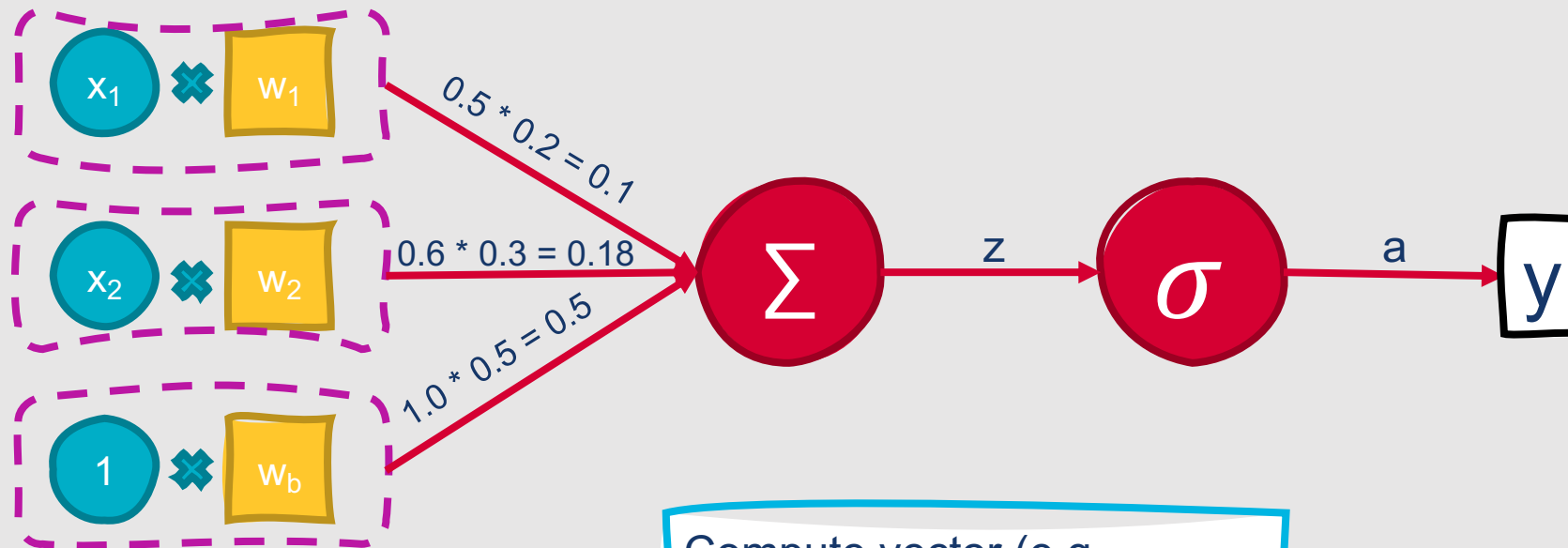
Input: "beautiful brutalist architecture"

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 Sigmoid Activation



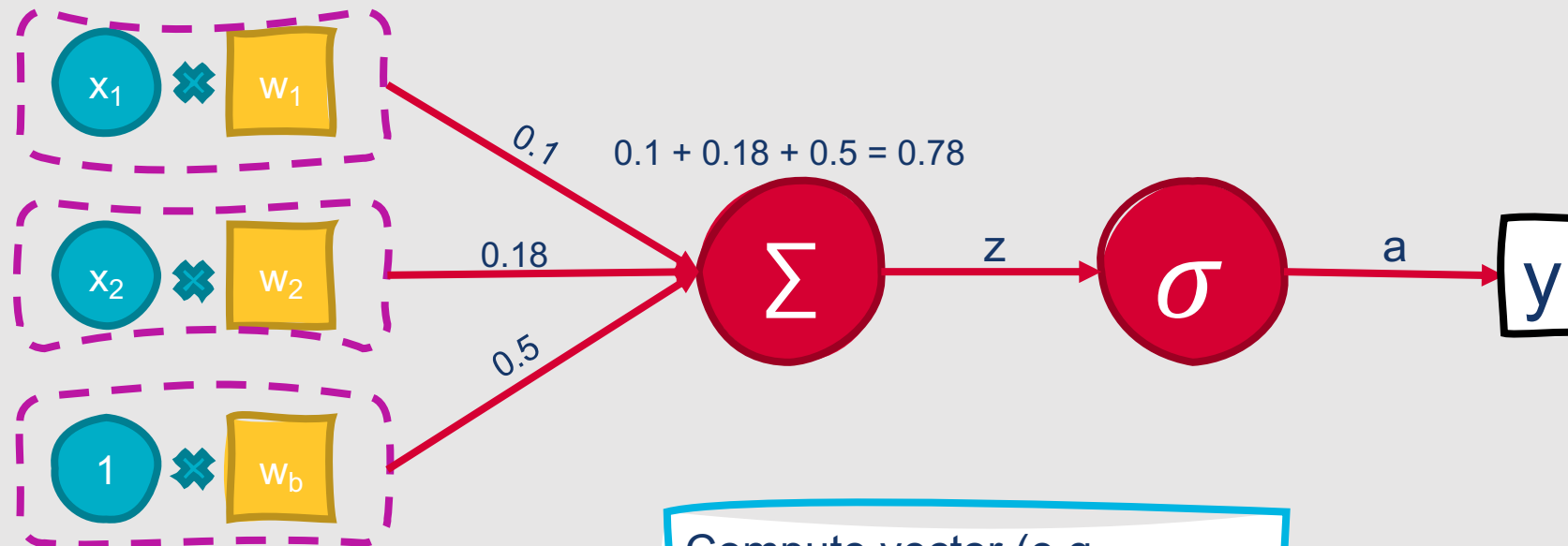
Input: "beautiful brutalist architecture"

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 Sigmoid Activation



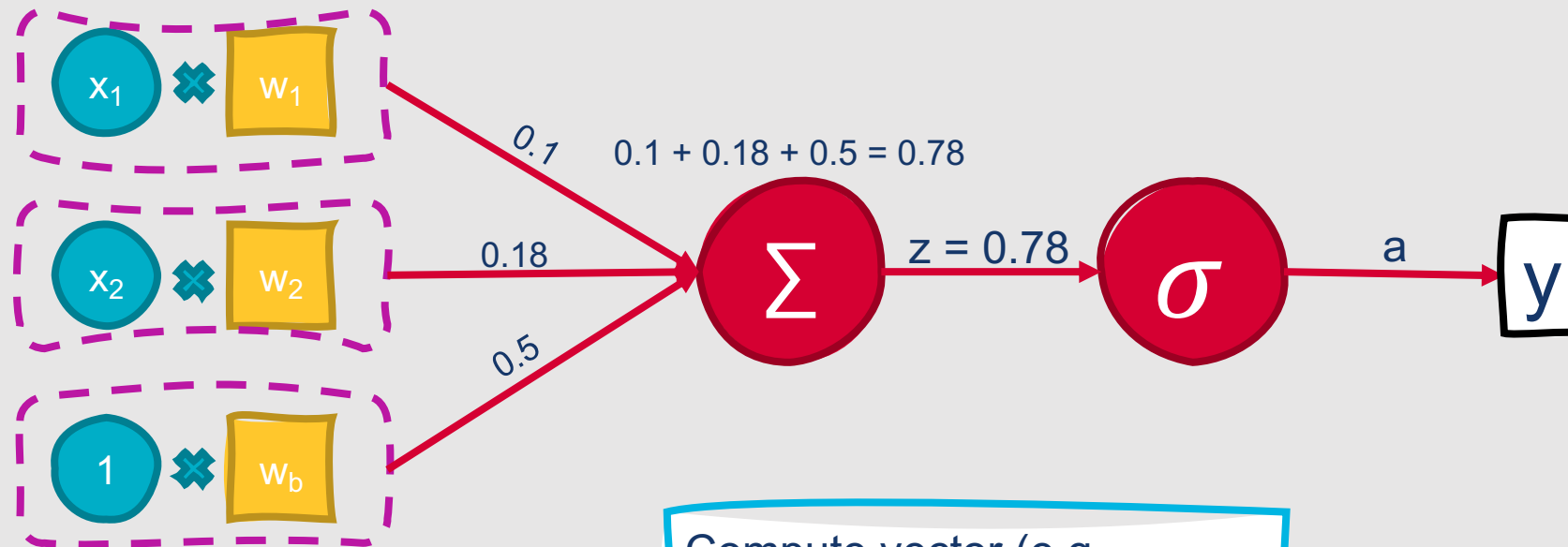
Input: "beautiful brutalist architecture"

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 Sigmoid Activation



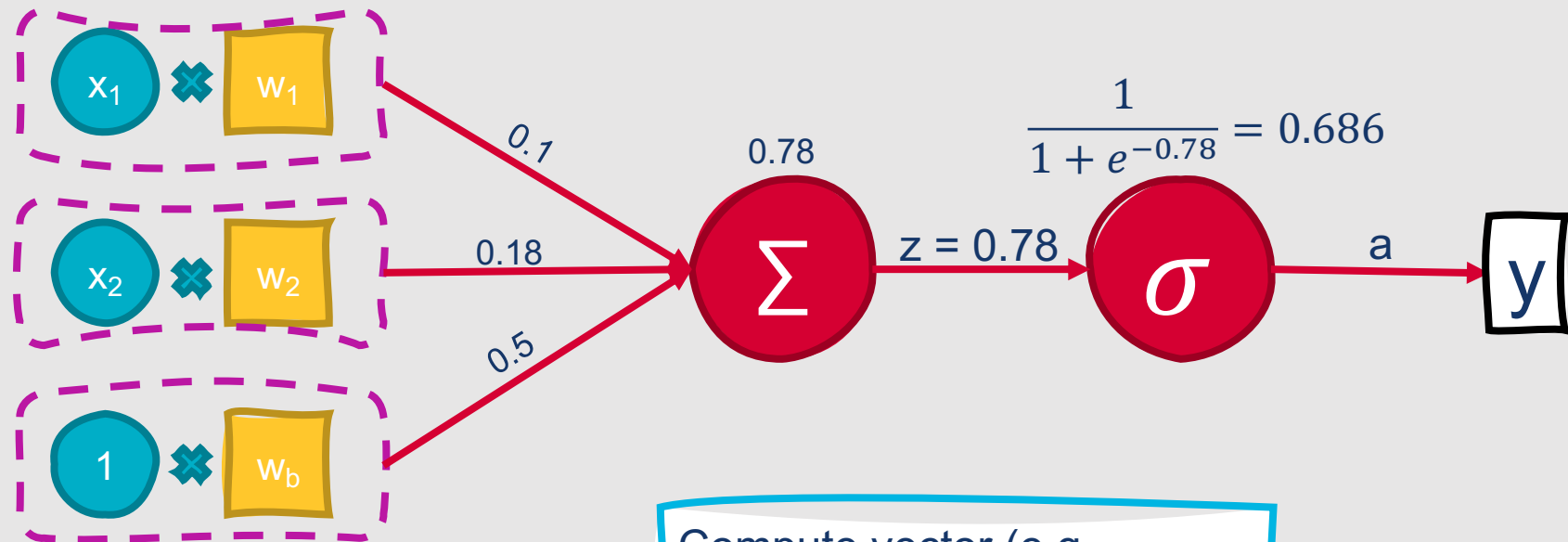
Input: "beautiful brutalist architecture"

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 Sigmoid Activation



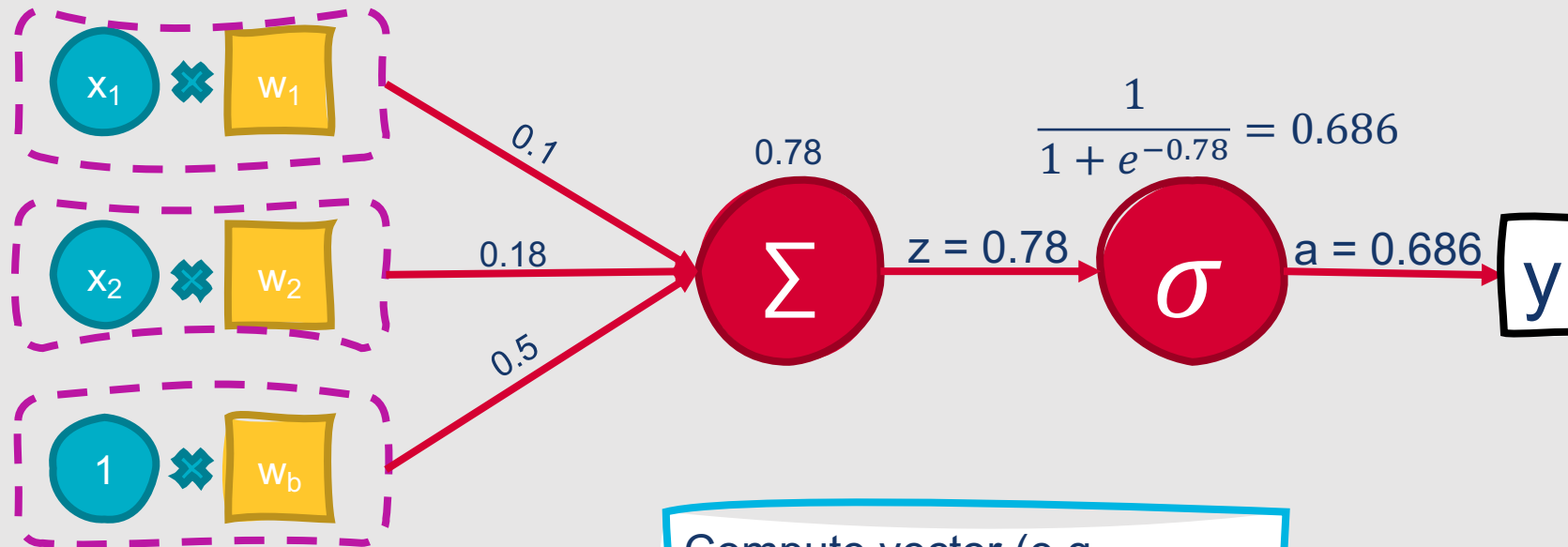
Input: "beautiful brutalist architecture"

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 Sigmoid Activation



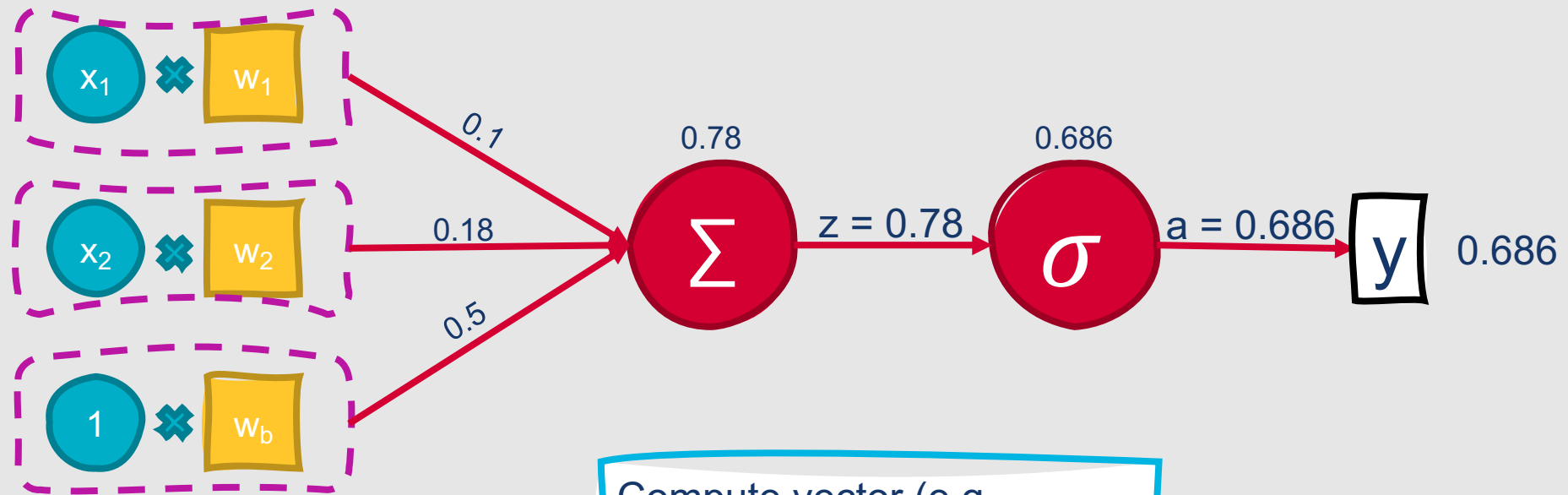
Input: "beautiful brutalist architecture"

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 Sigmoid Activation



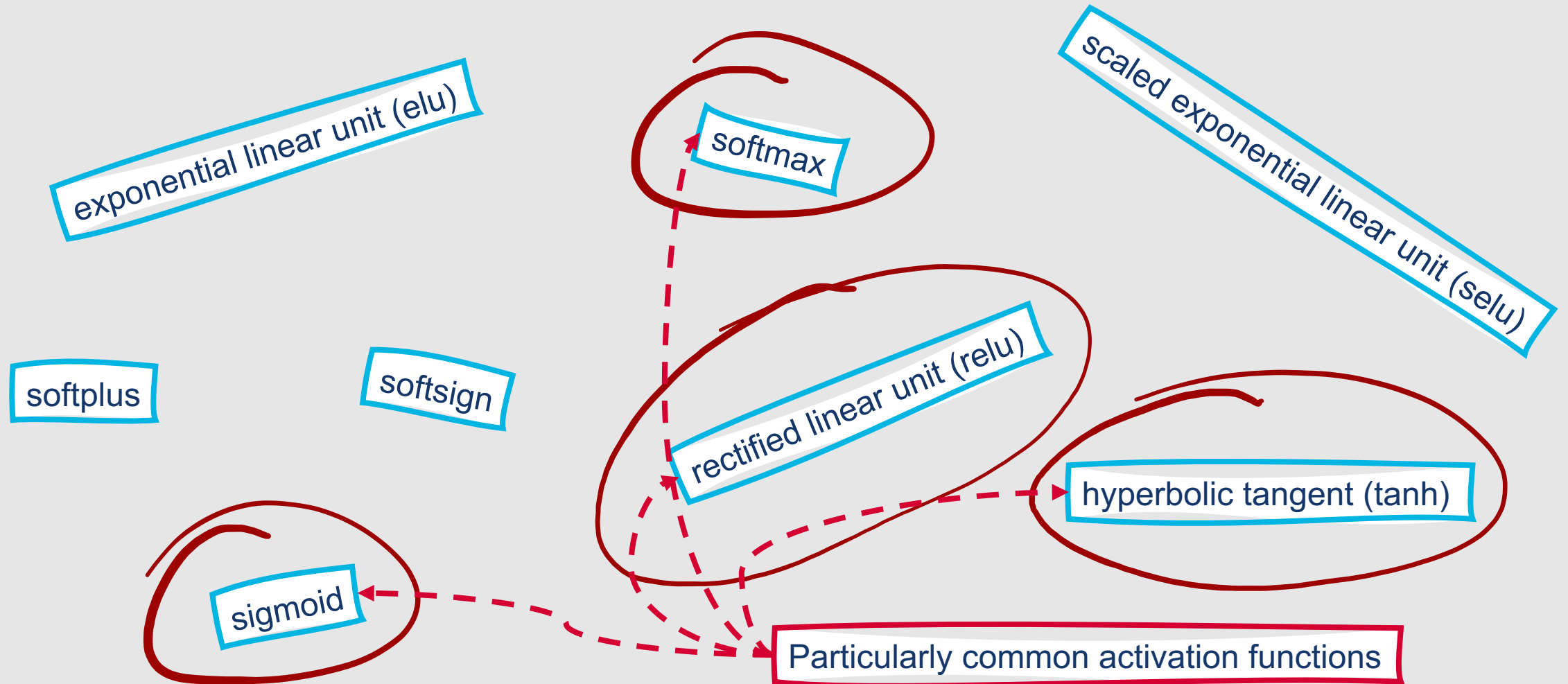
Input: "beautiful brutalist architecture"

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]

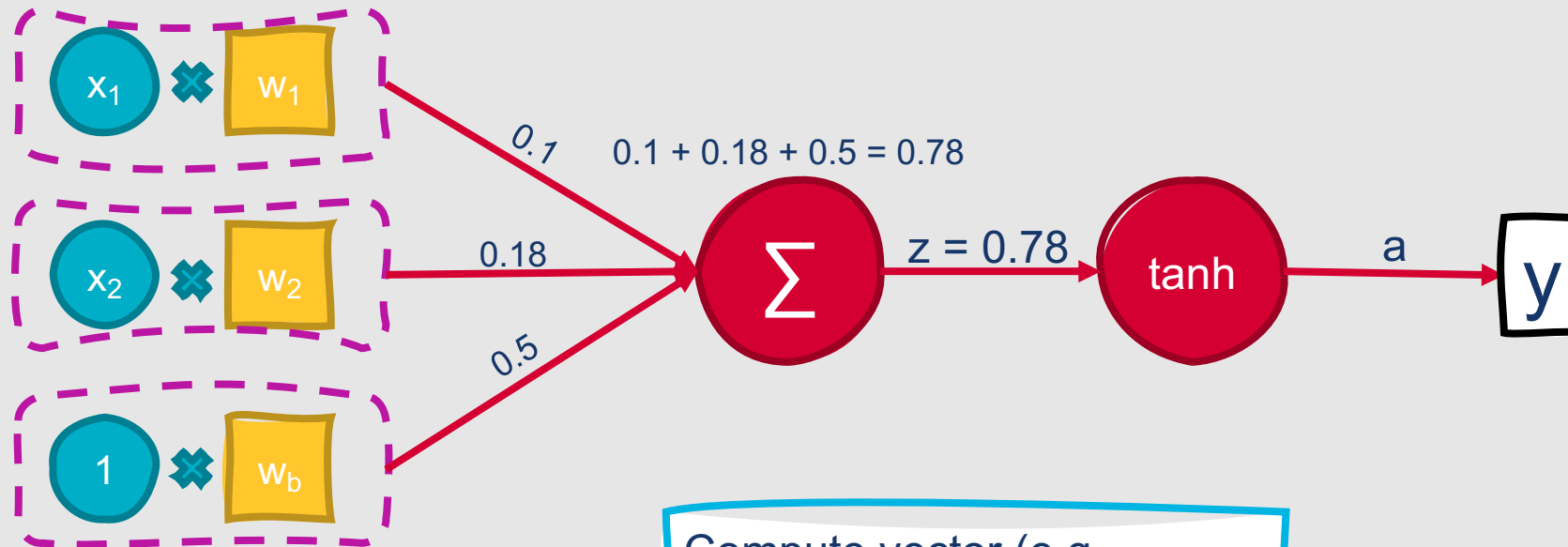
Other Popular Activation Functions



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



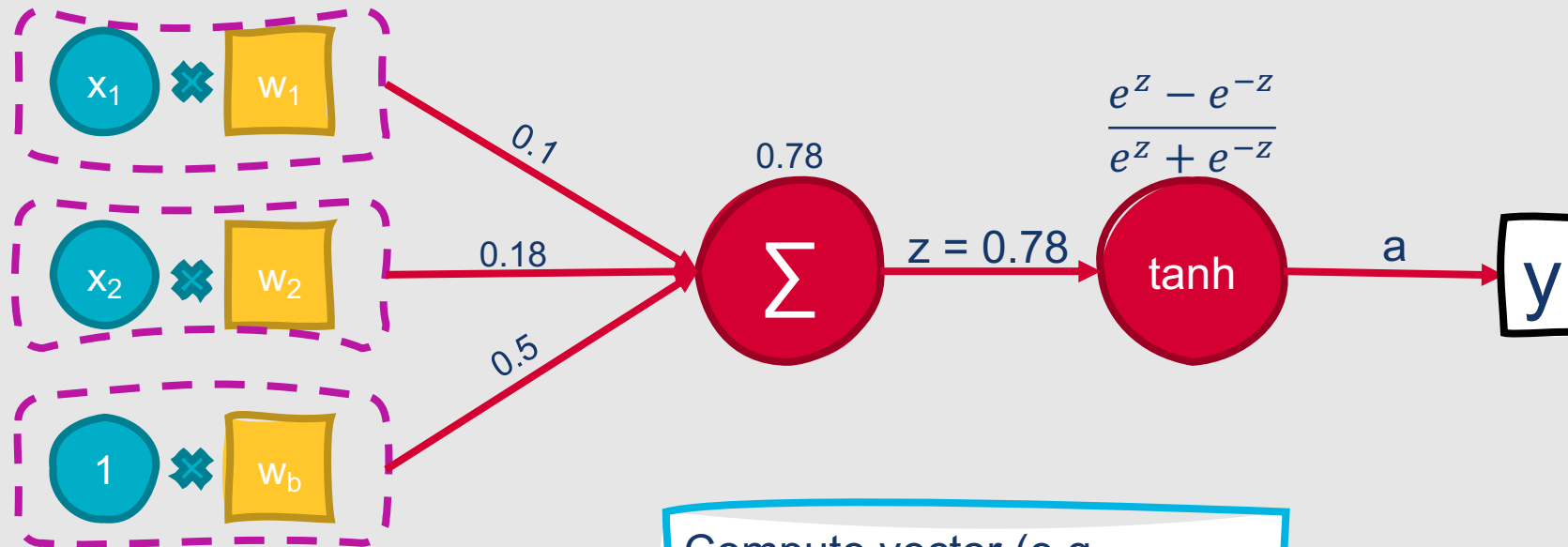
Input: "beautiful brutalist architecture"

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 tanh Activation



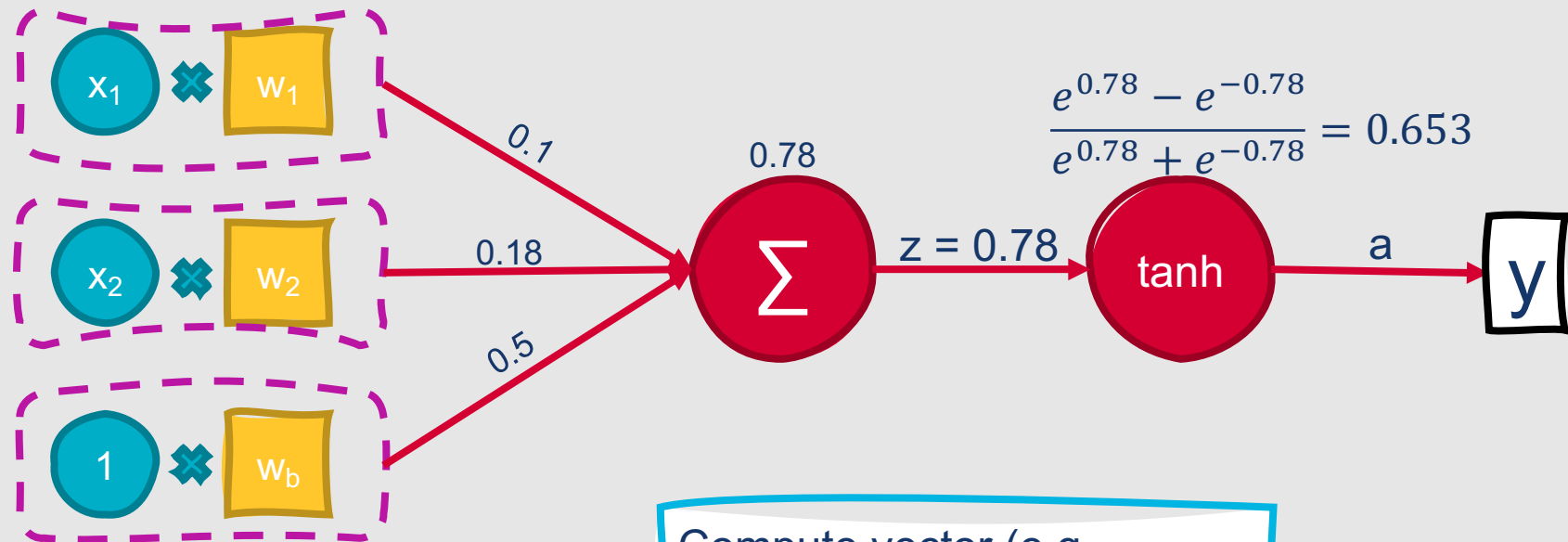
Input: "beautiful brutalist architecture"

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 tanh Activation



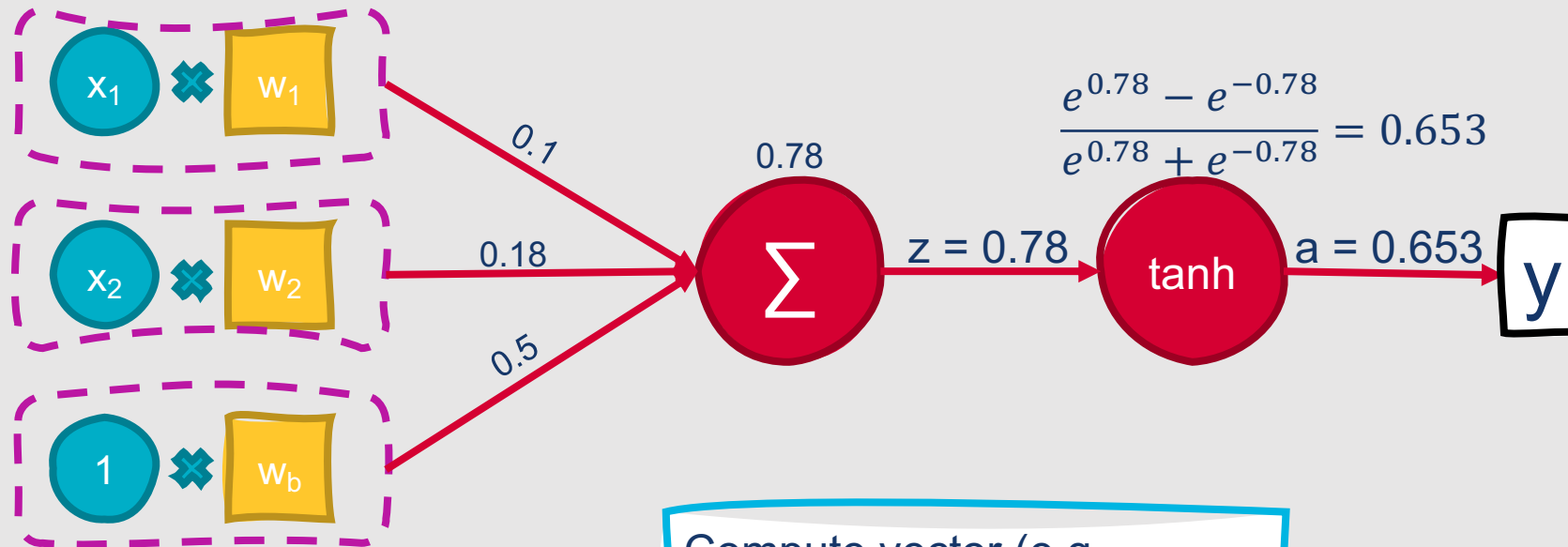
Input: "beautiful brutalist architecture"

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 tanh Activation



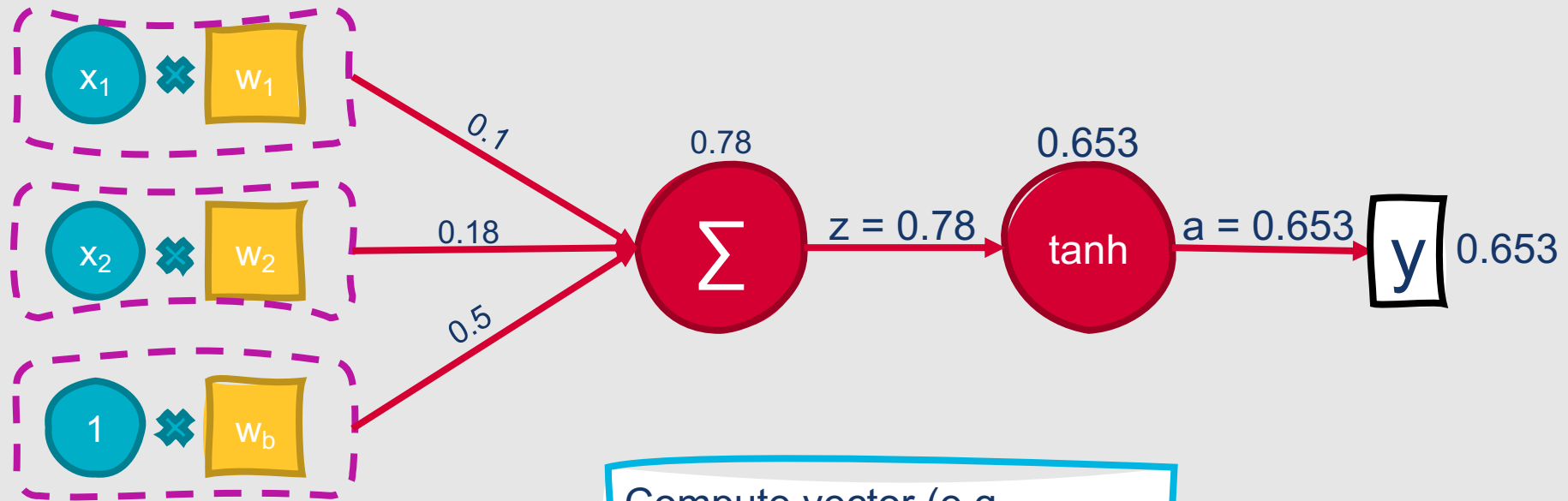
Input: "beautiful brutalist architecture"

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 tanh Activation



Input: "beautiful brutalist architecture"

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

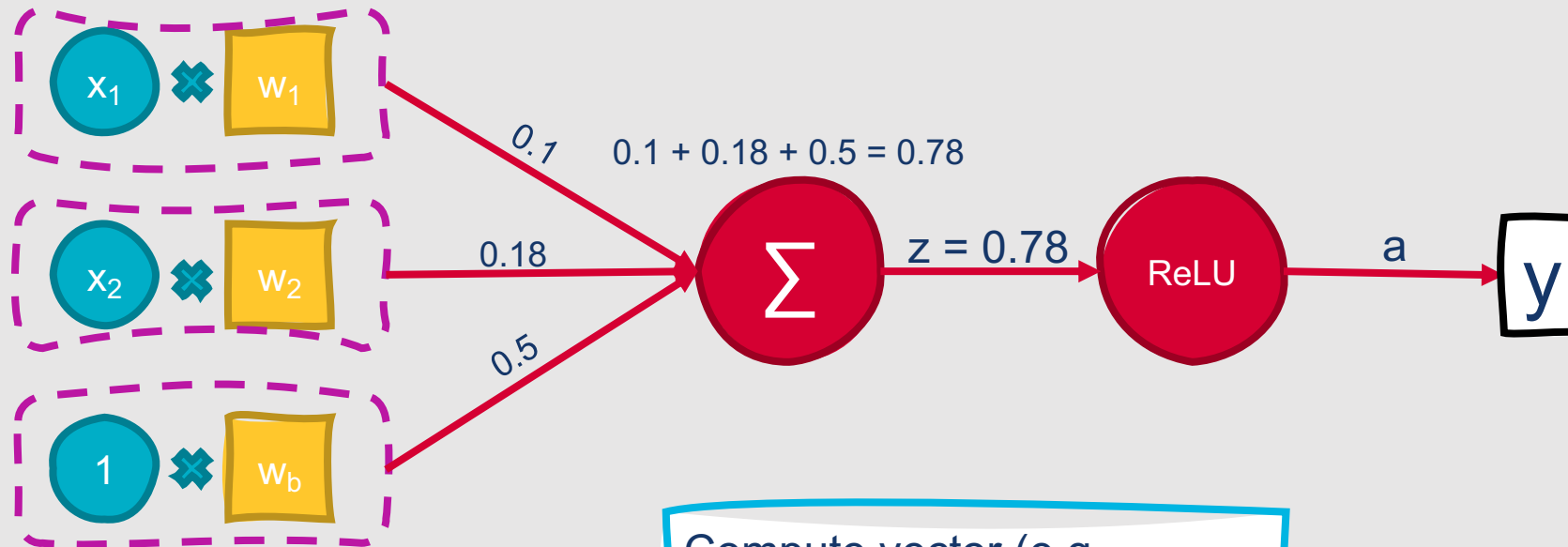
Compute vector (e.g., averaged Word2Vec embeddings for "beautiful," "brutalist," and "architecture")

[0.5, 0.6]

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



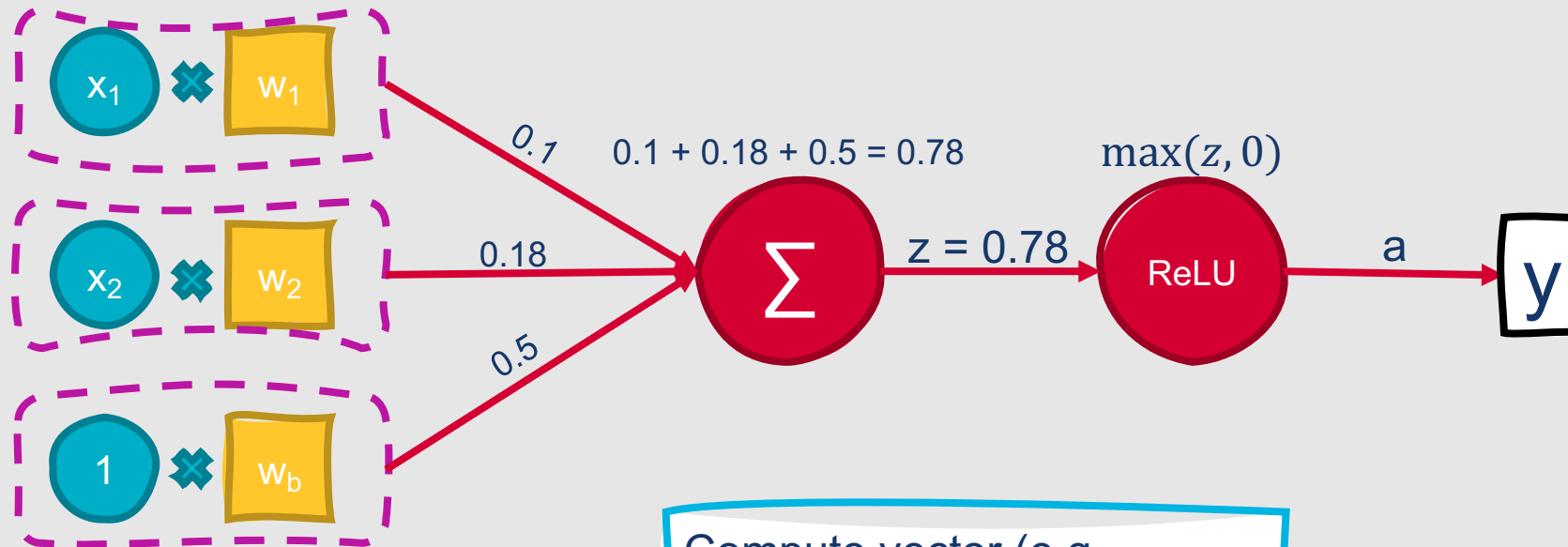
Input: "beautiful brutalist architecture"

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



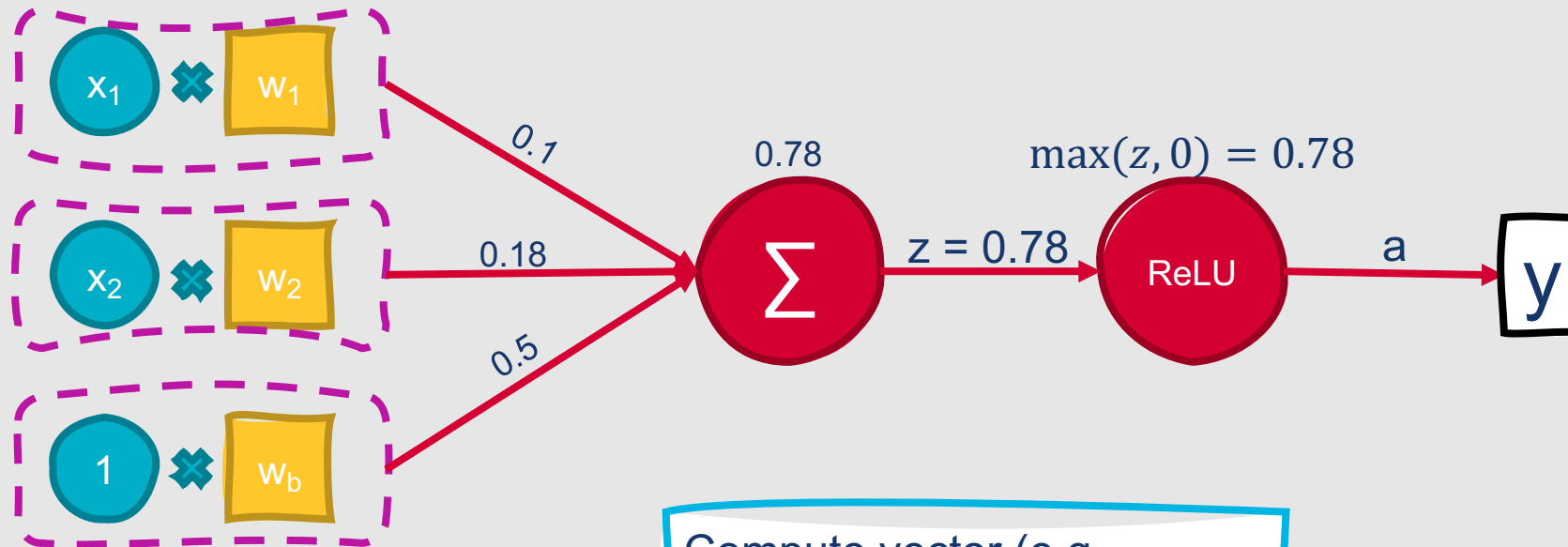
Input: "beautiful brutalist architecture"

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



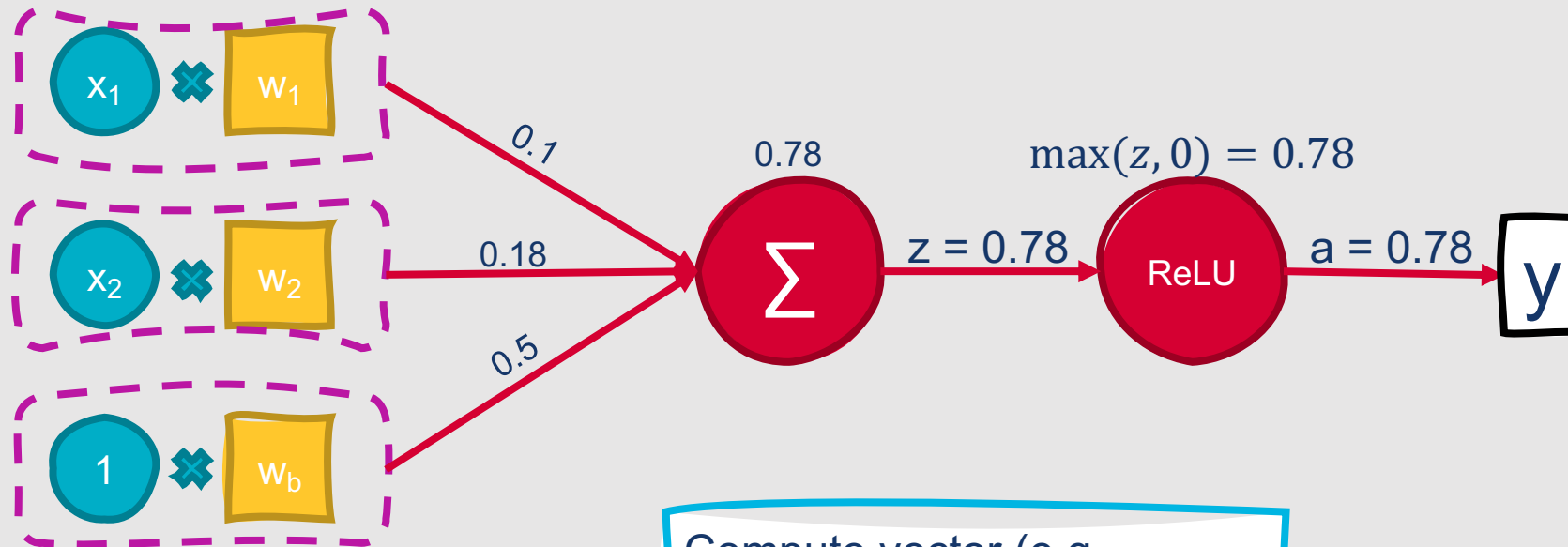
Input: "beautiful brutalist architecture"

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



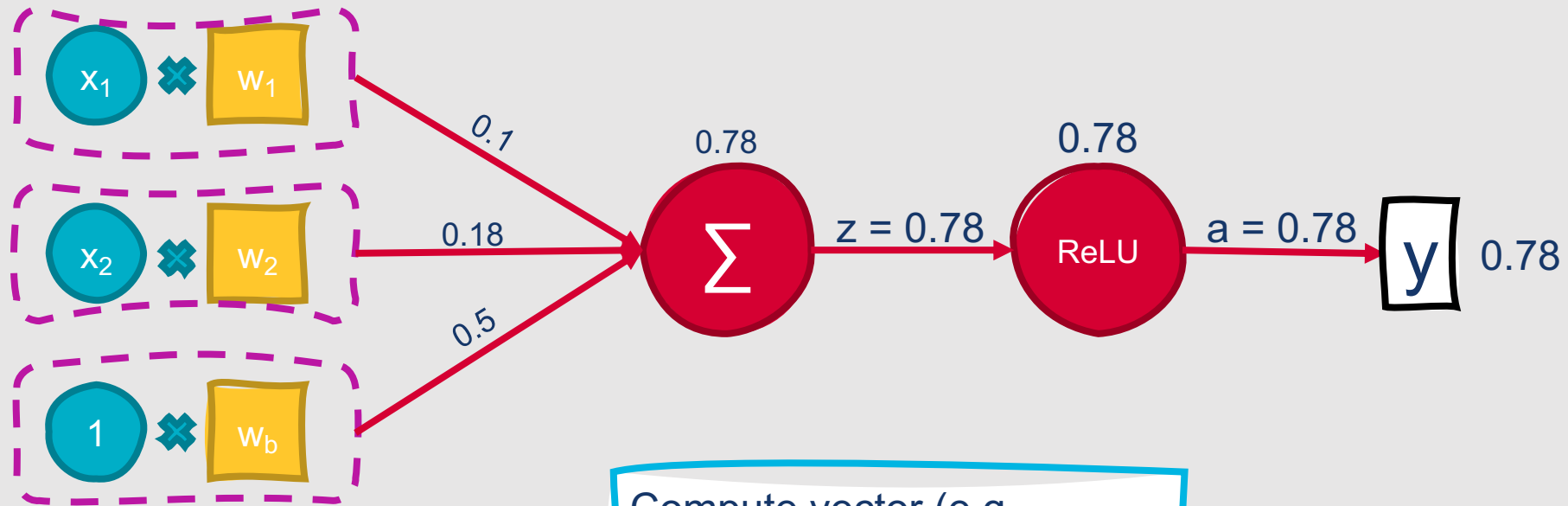
Input: "beautiful brutalist architecture"

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"

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

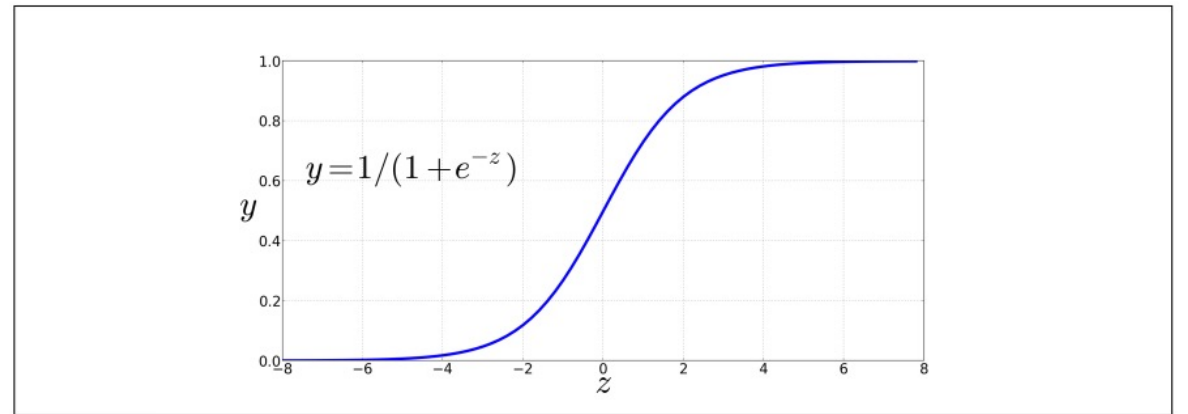


Figure 7.1 The sigmoid function takes a real value and maps it to the range $[0,1]$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

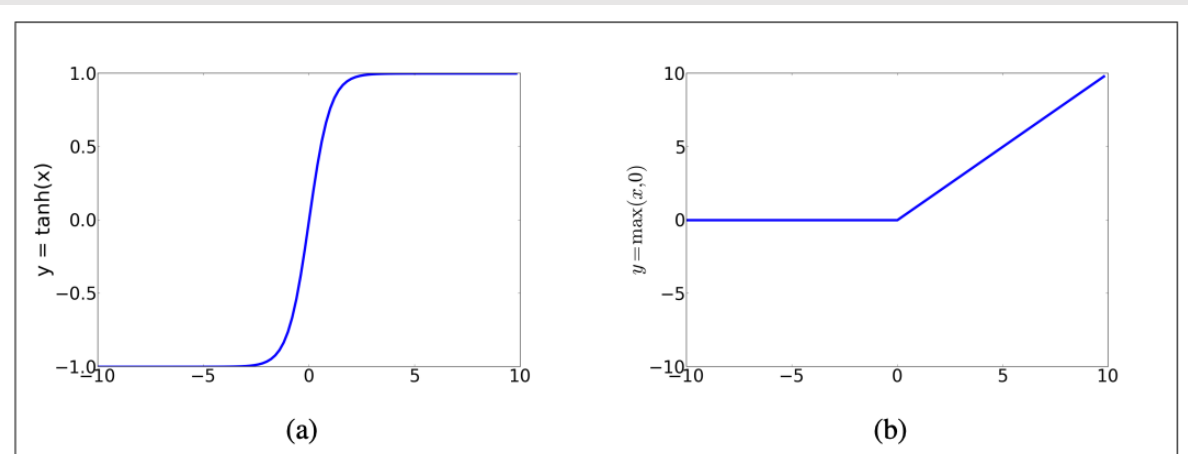


Figure 7.3 The tanh and ReLU activation functions.

This Week's Topics

Cosine similarity
Word2Vec
Other dense embeddings
Using word embeddings

Thursday

Tuesday

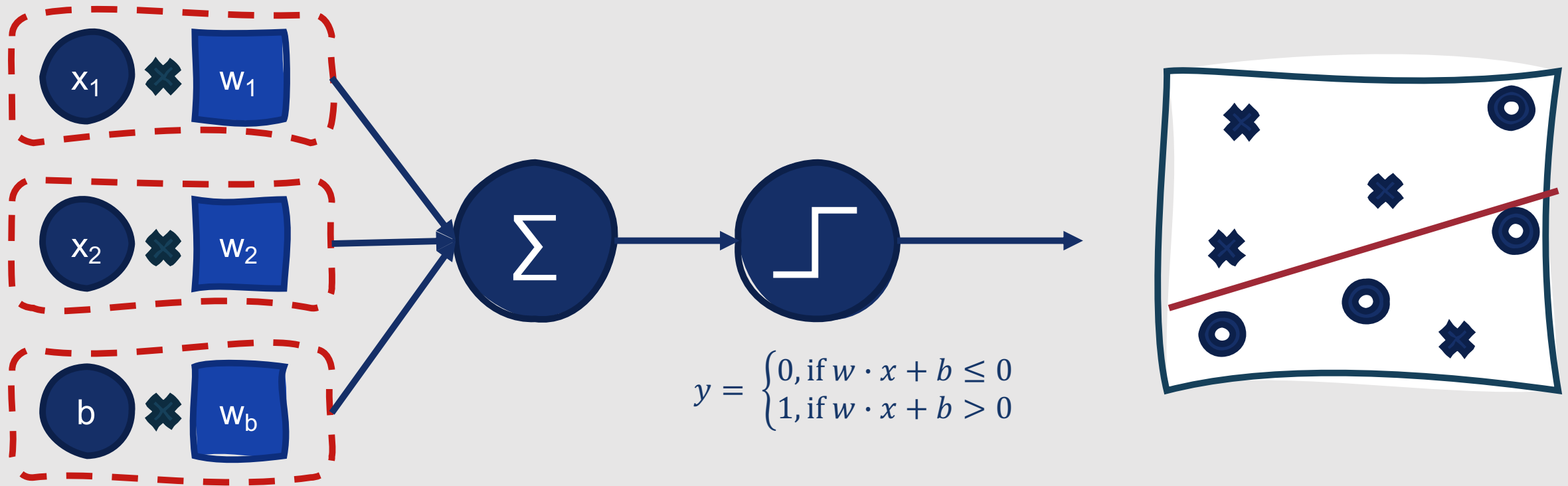


Neural networks
Combining and optimizing computational units
Neural language models

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

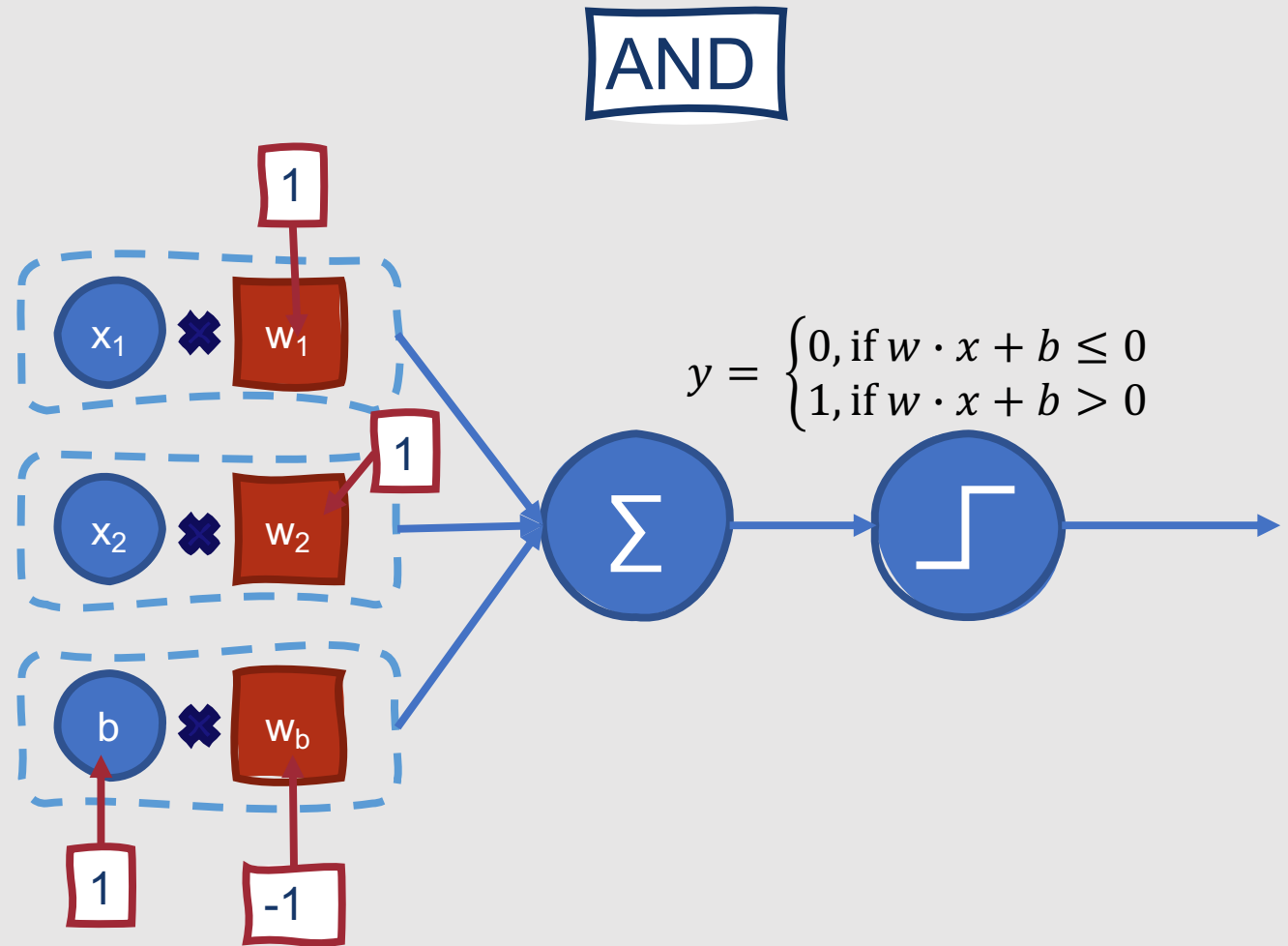
AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0



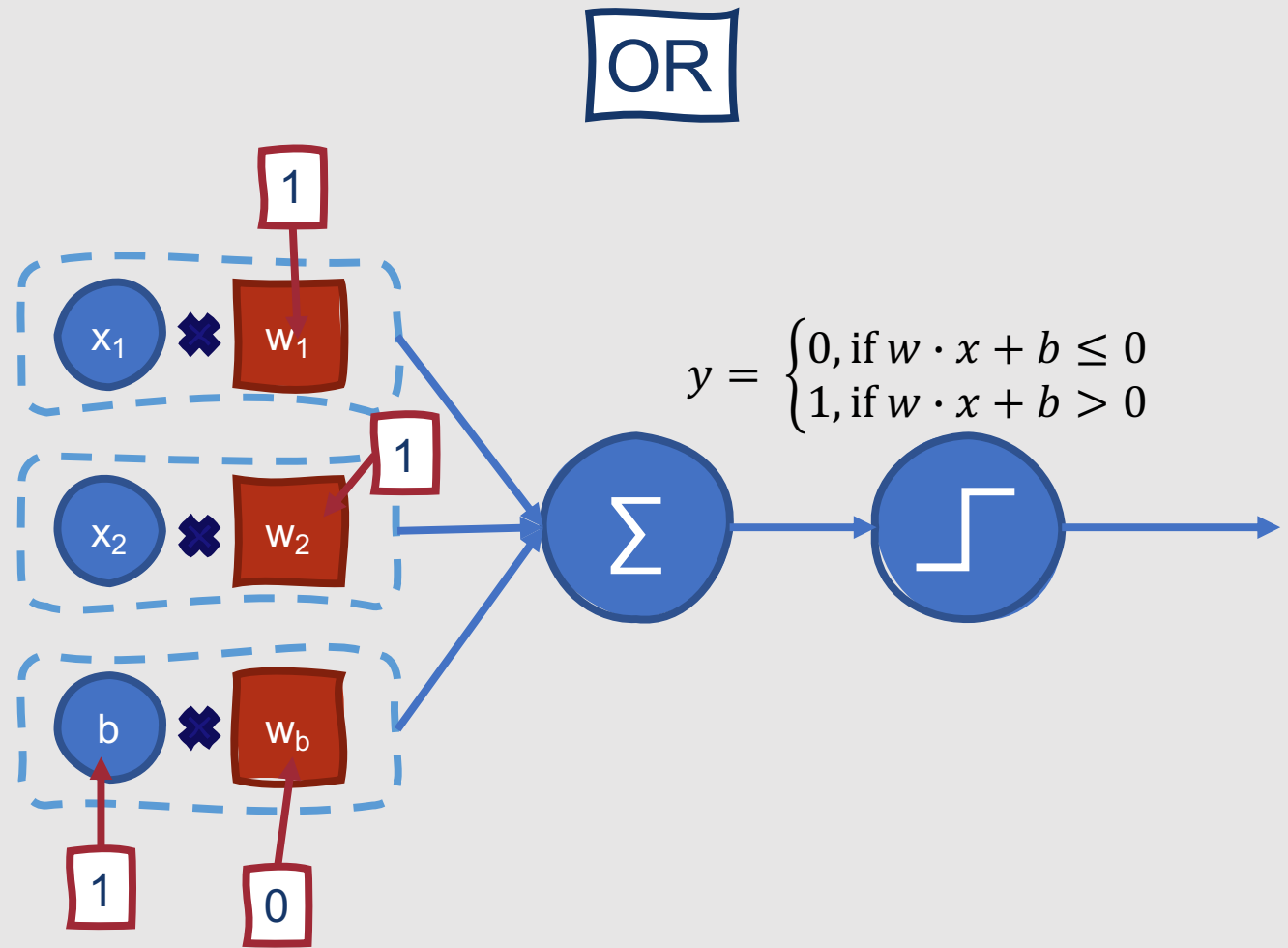
AND and OR can both be solved using a single perceptron.

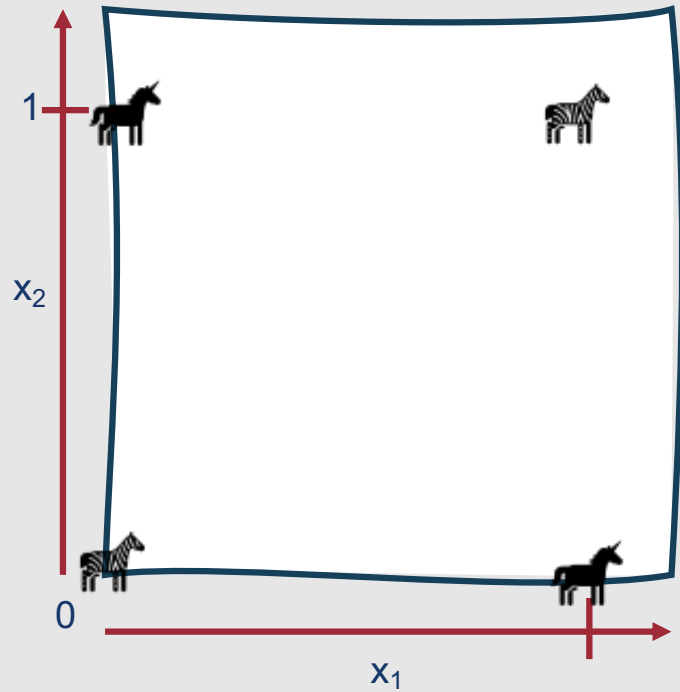
- **Perceptron:** A function that outputs a binary value based on whether the product of its inputs and associated weights surpasses a threshold

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



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



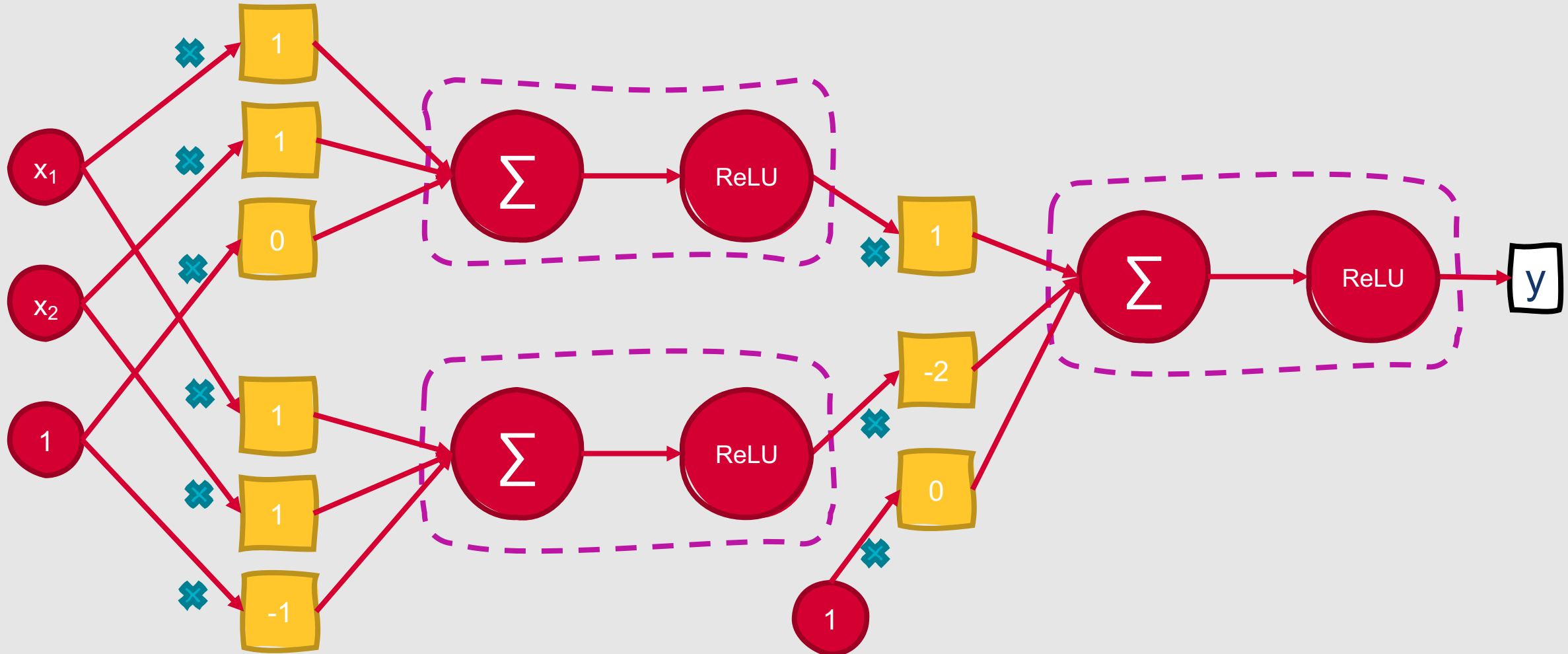


AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

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

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

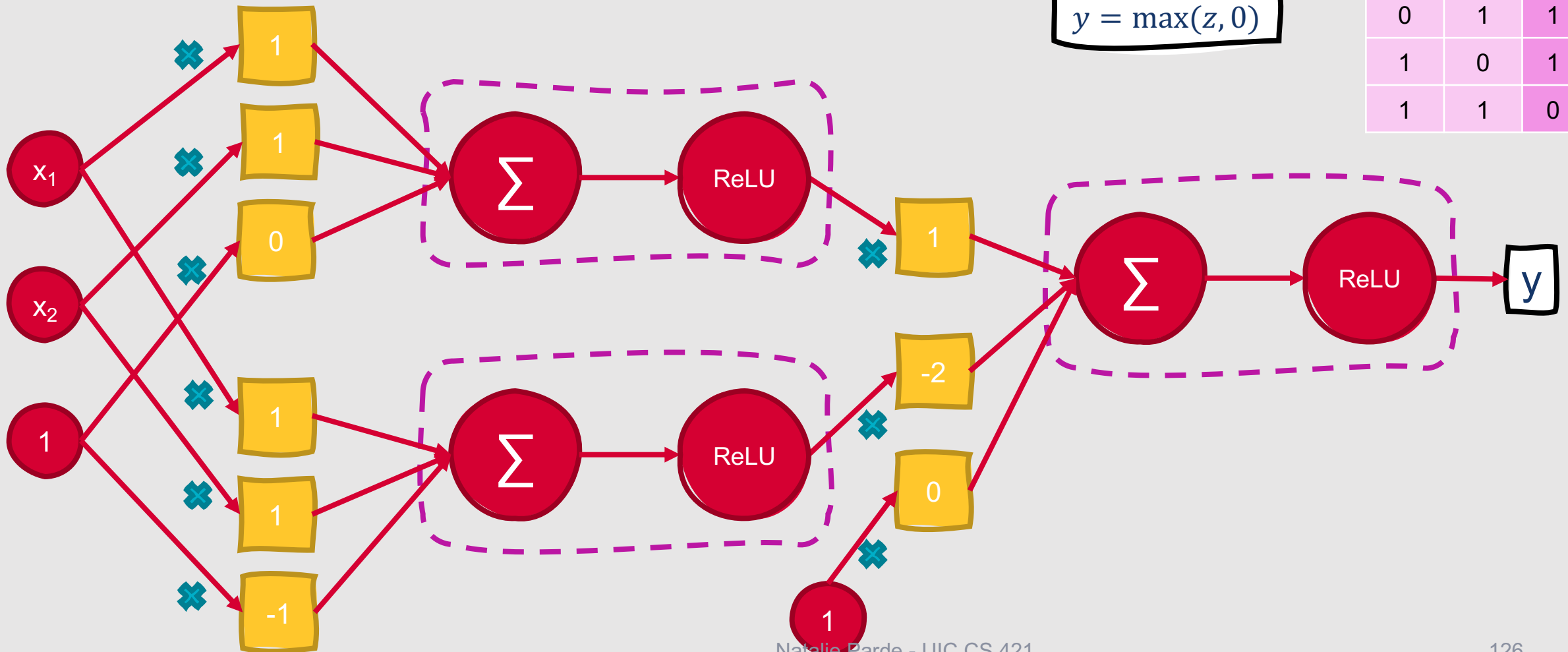
The only successful way to compute XOR is by combining these smaller units into a larger network.



Truth Table Examples: XOR

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

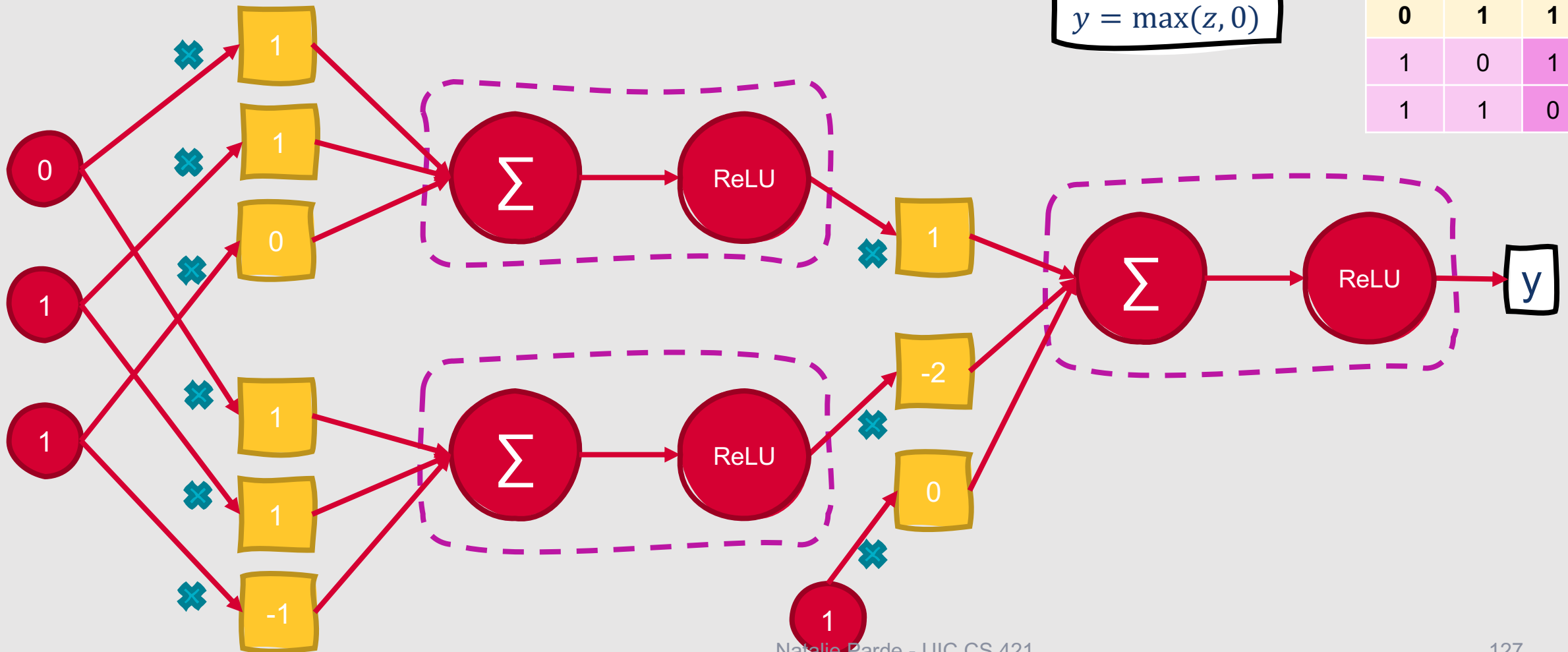
$$y = \max(z, 0)$$



Truth Table Examples: XOR

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

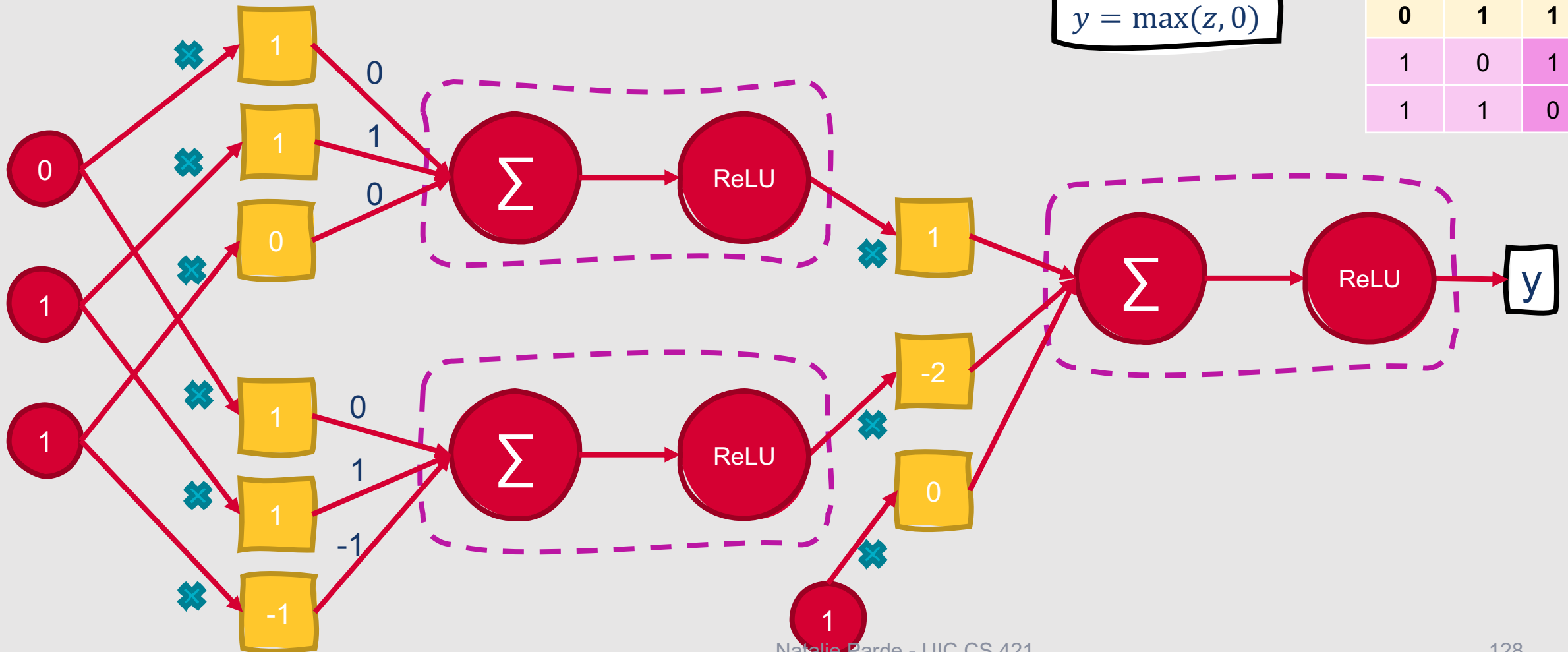
$$y = \max(z, 0)$$



Truth Table Examples: XOR

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

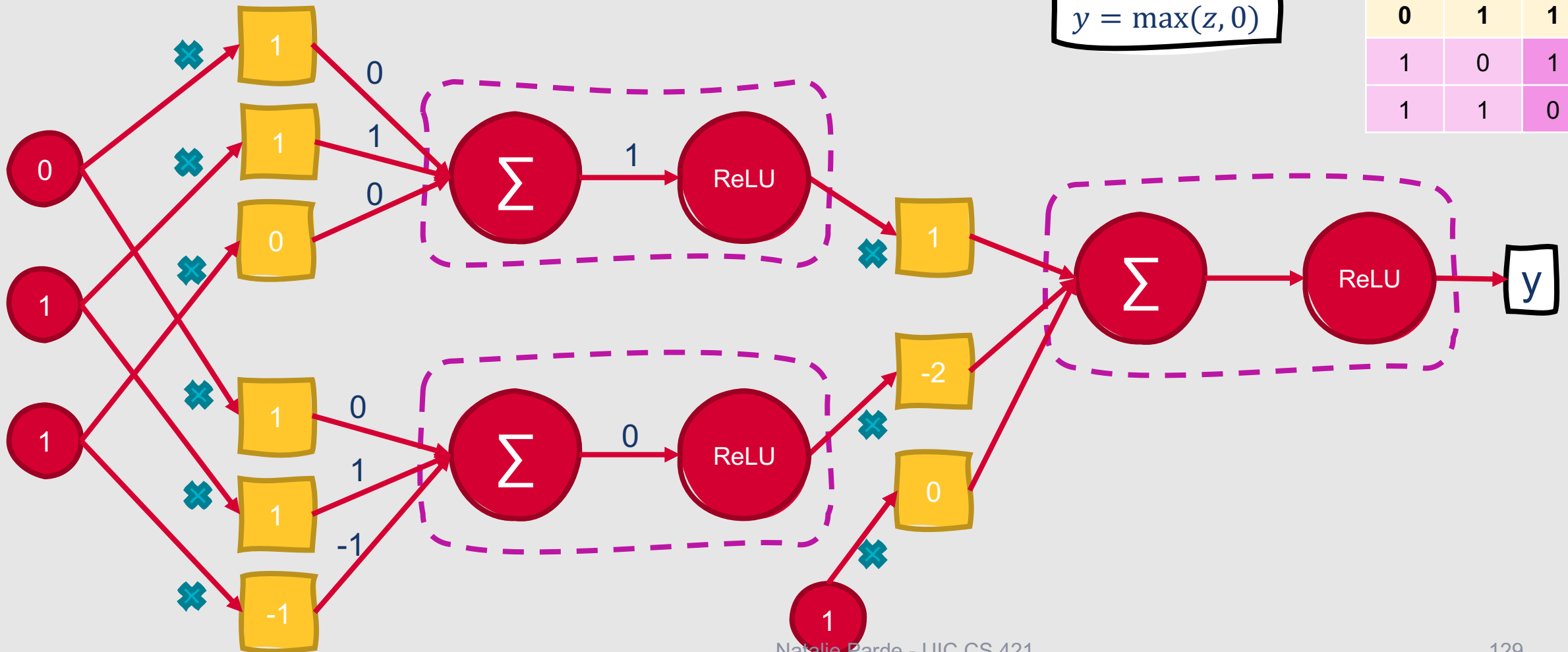
$$y = \max(z, 0)$$



Truth Table Examples: XOR

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

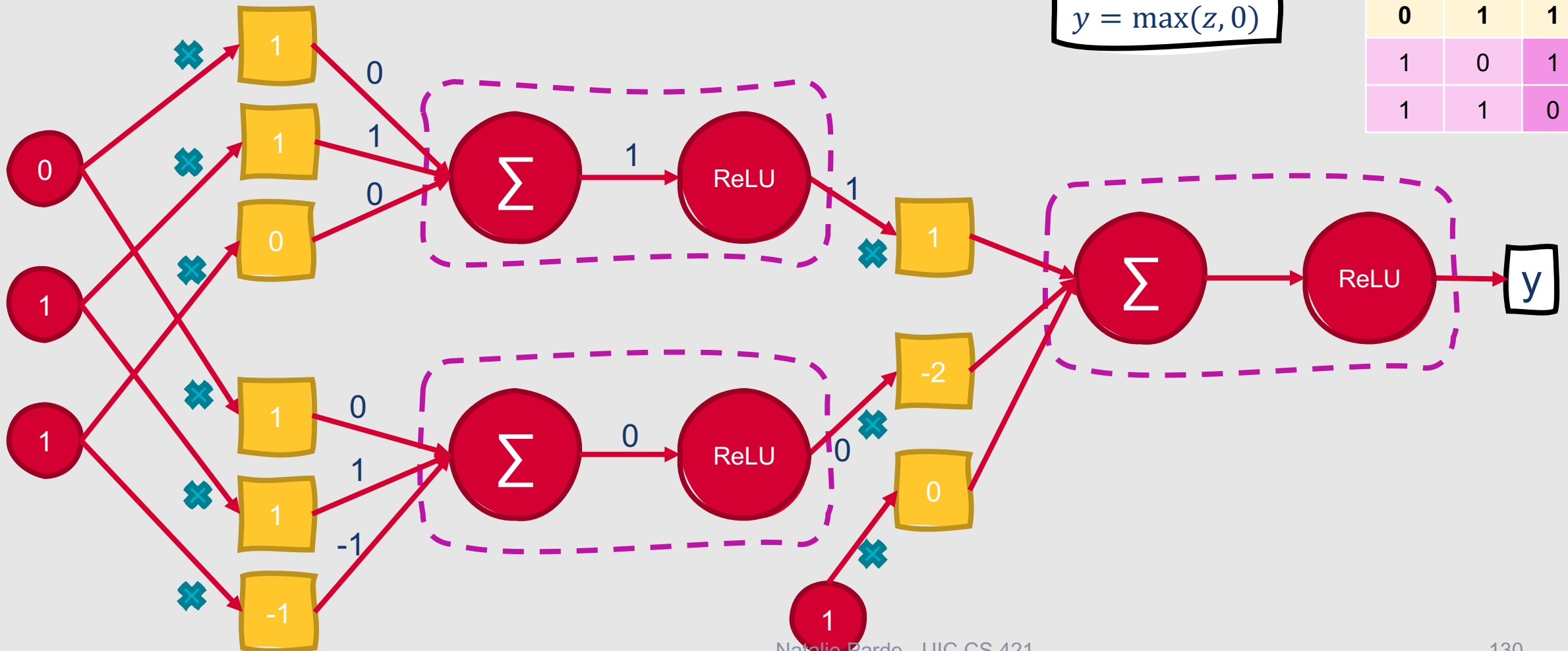
$$y = \max(z, 0)$$



Truth Table Examples: XOR

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

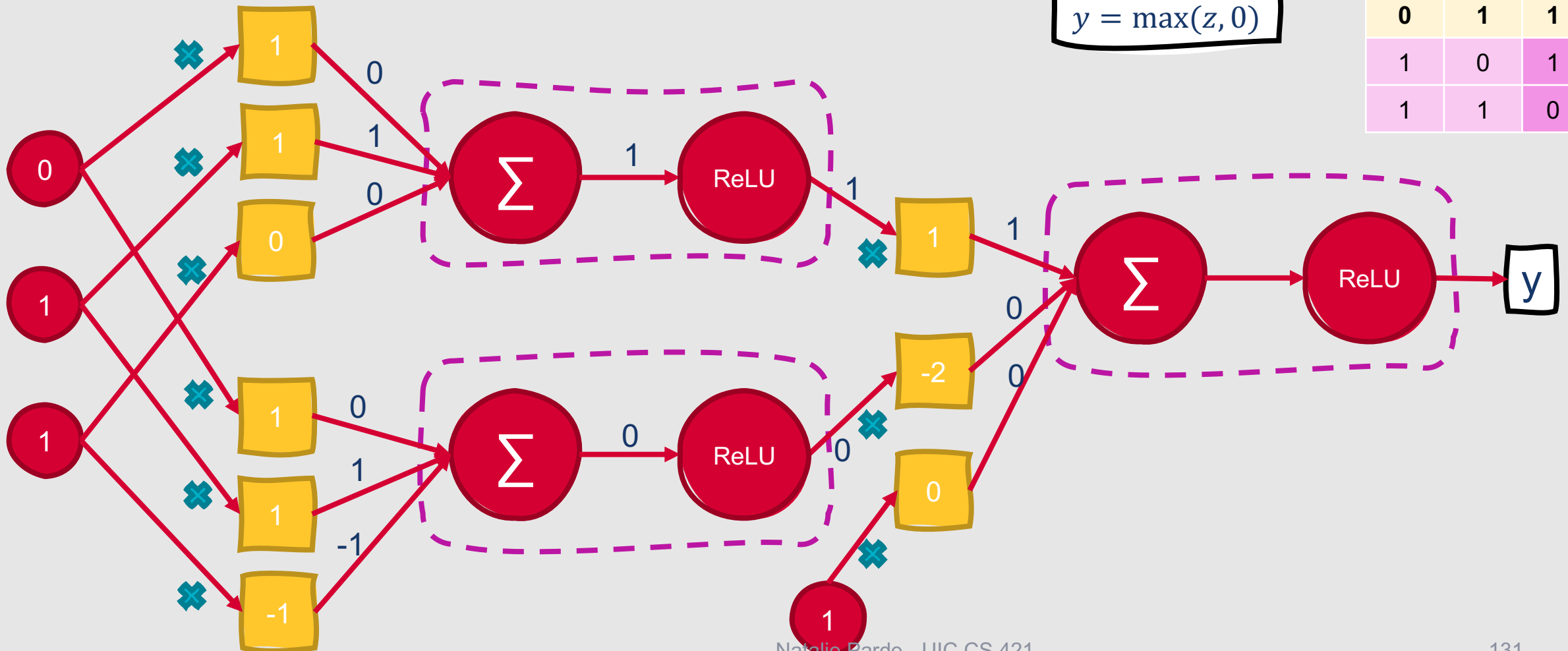
$$y = \max(z, 0)$$



Truth Table Examples: XOR

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

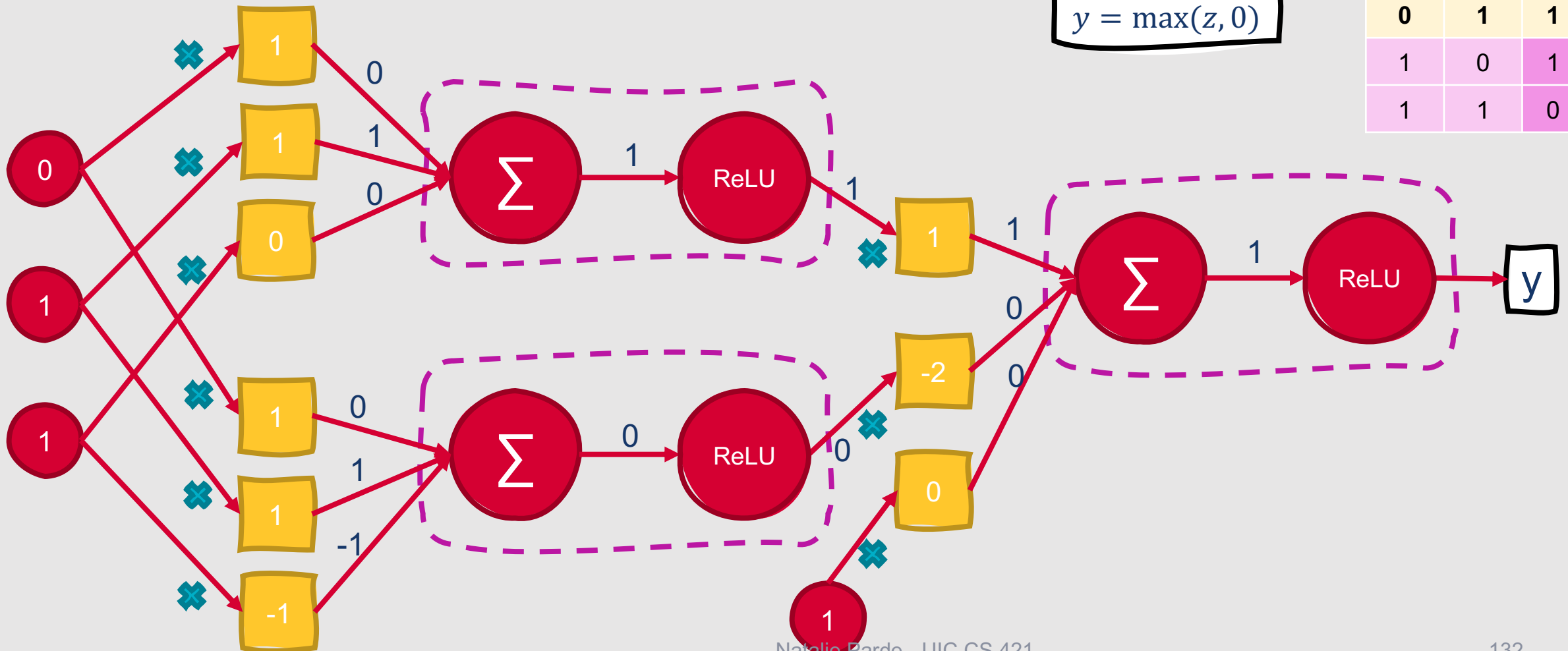
$$y = \max(z, 0)$$



Truth Table Examples: XOR

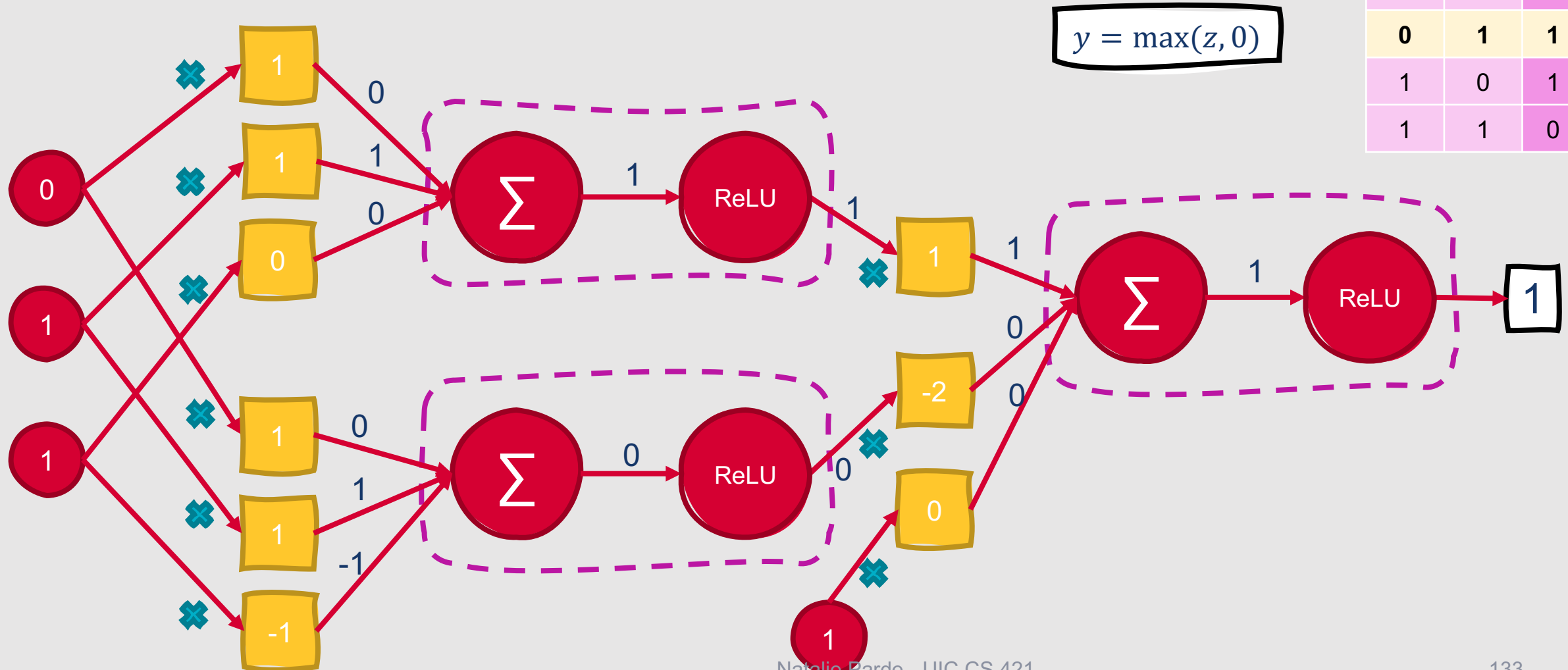
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

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



Truth Table Examples: XOR

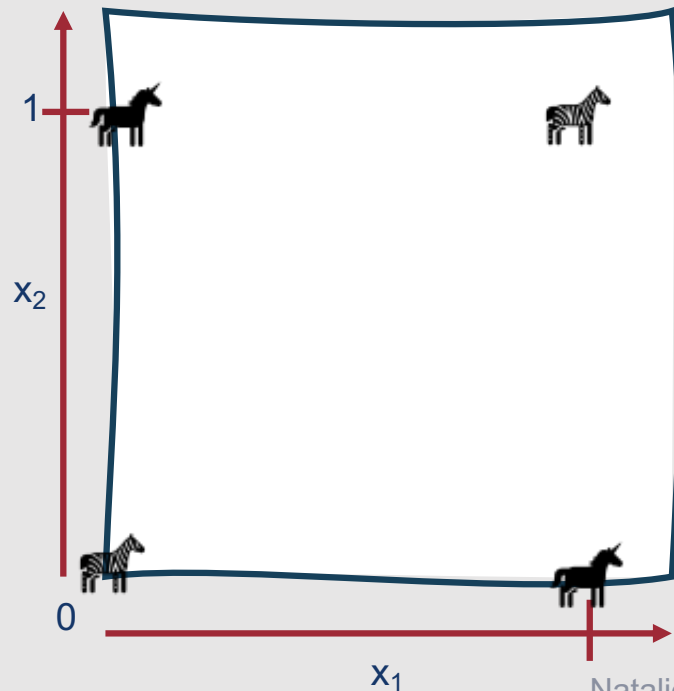
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



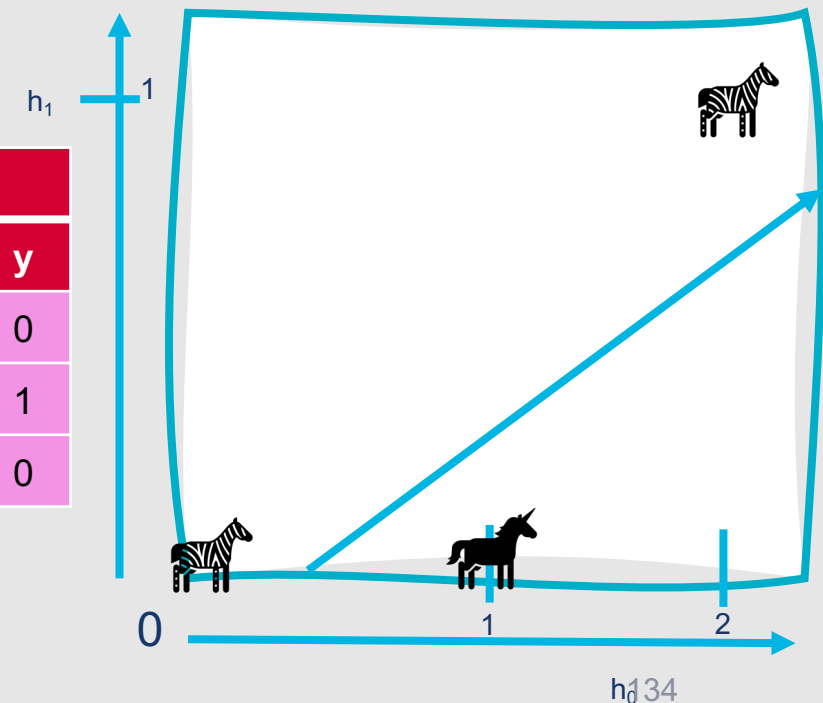
Why does this work?

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

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



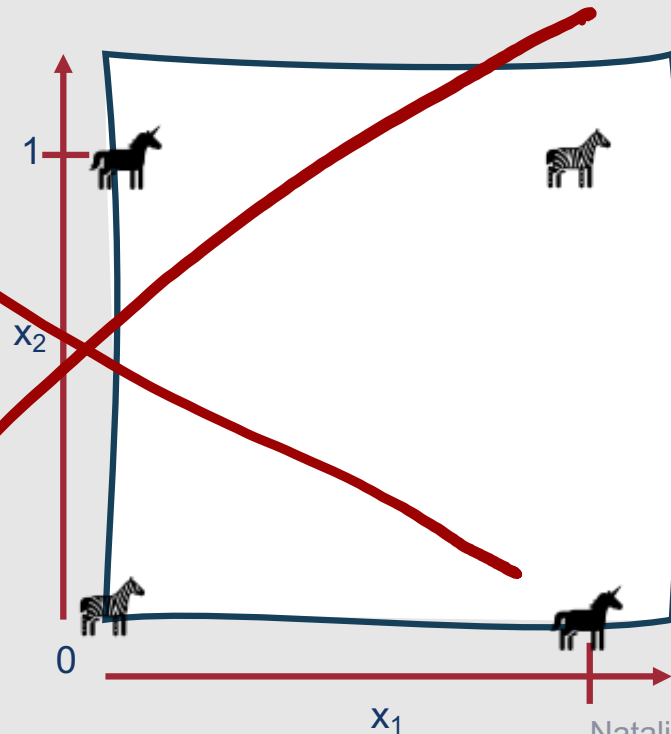
XOR		
h0	h1	y
0	0	0
1	0	1
2	1	0



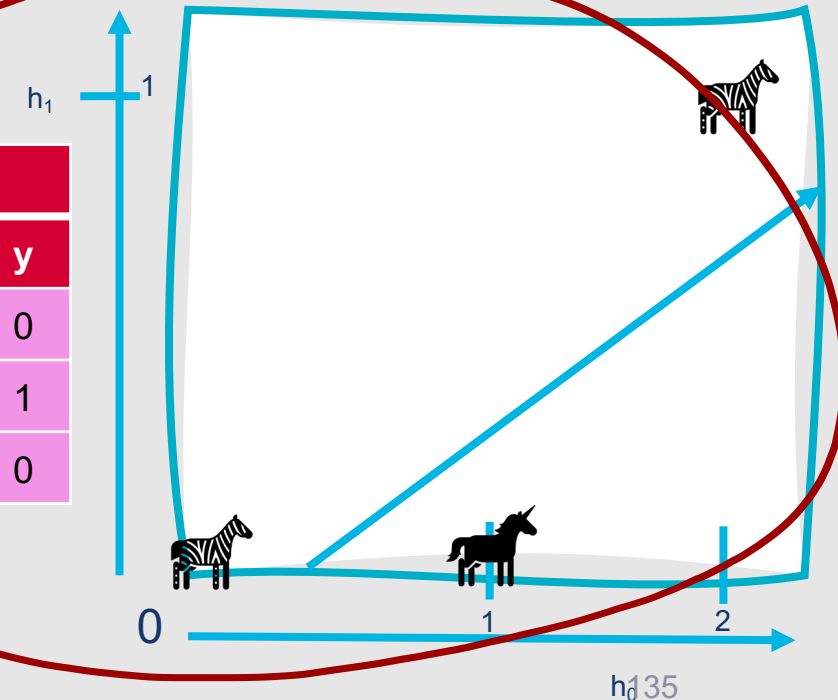
Why does this work?

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

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



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:
 - $z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$
 - $a^{[1]} = g^{[1]}(z^{[1]})$
 - $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
 - $a^{[2]} = g^{[2]}(z^{[2]})$
 - $y' = a^{[2]}$
- We can easily generalize to networks with more layers:
 - For i in $1..n$
 - $z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}$
 - $a^{[i]} = g^{[i]}(z^{[i]})$
 - $y' = a^{[n]}$



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

???



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



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/>

This Week's Topics

Cosine similarity
Word2Vec
Other dense embeddings
Using word embeddings

Thursday

Tuesday

Neural networks
Combining and optimizing
computational units
★ Neural language models

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

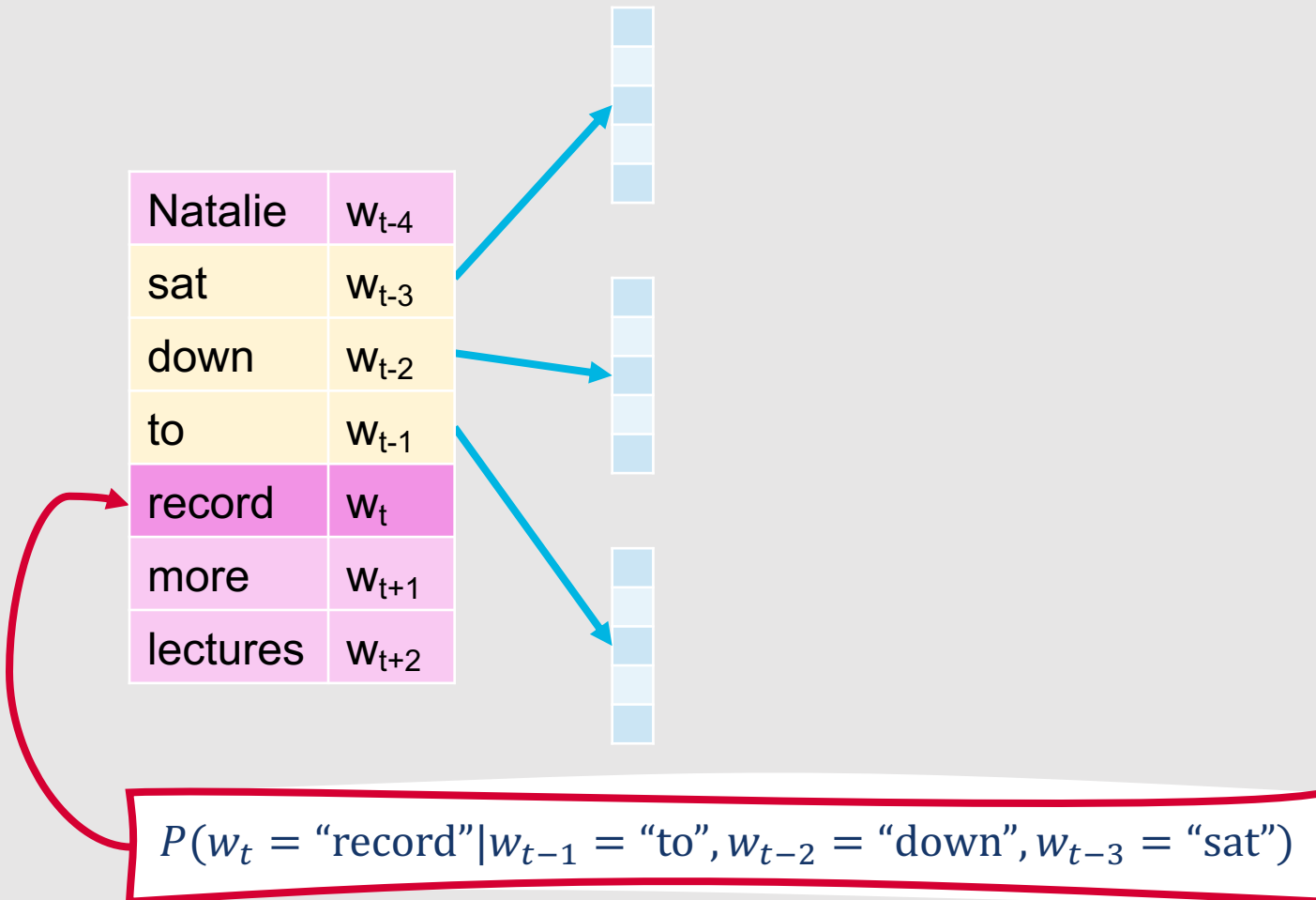
- Input: Representation of some number of previous words
 - $w_{t-1}, w_{t-2}, \text{ etc.}$
- Output: Probability distribution over possible next words
- Goal: Approximate the probability of a word given the entire prior context $P(w_t | w_1^{t-1})$ based on the n previous words
 - $P(w_t | w_1^{t-1})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

Neural Language Model

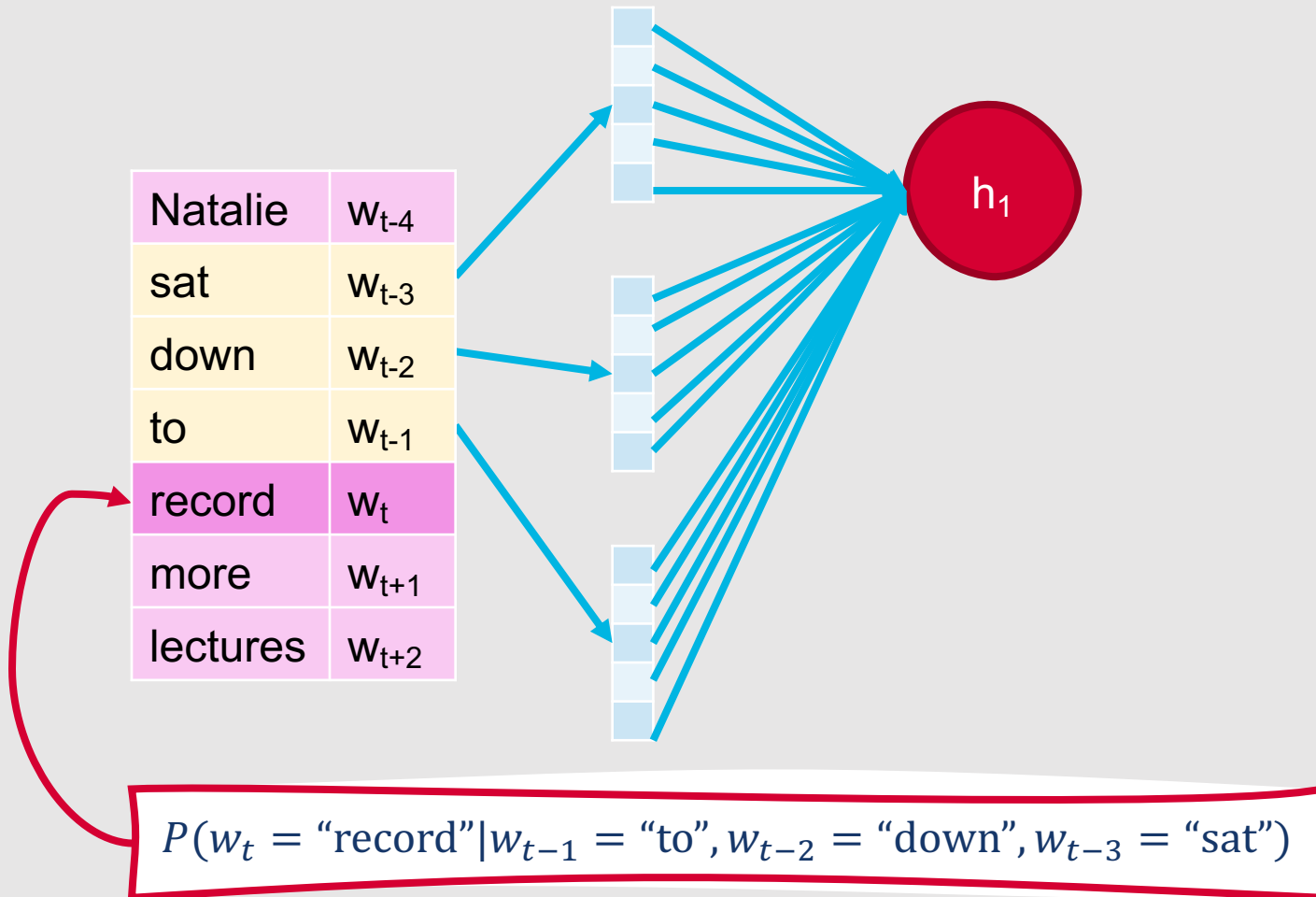
Natalie	w_{t-4}
sat	w_{t-3}
down	w_{t-2}
to	w_{t-1}
record	w_t
more	w_{t+1}
lectures	w_{t+2}


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

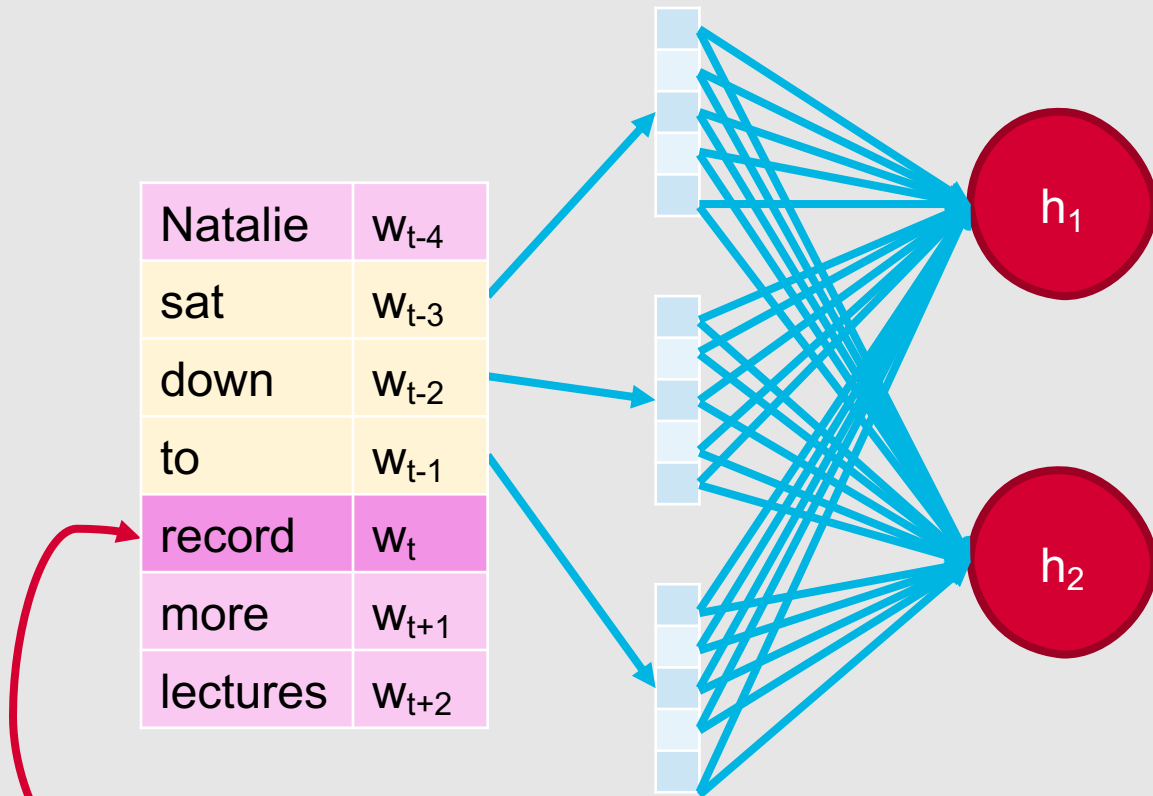
Neural Language Model



Neural Language Model

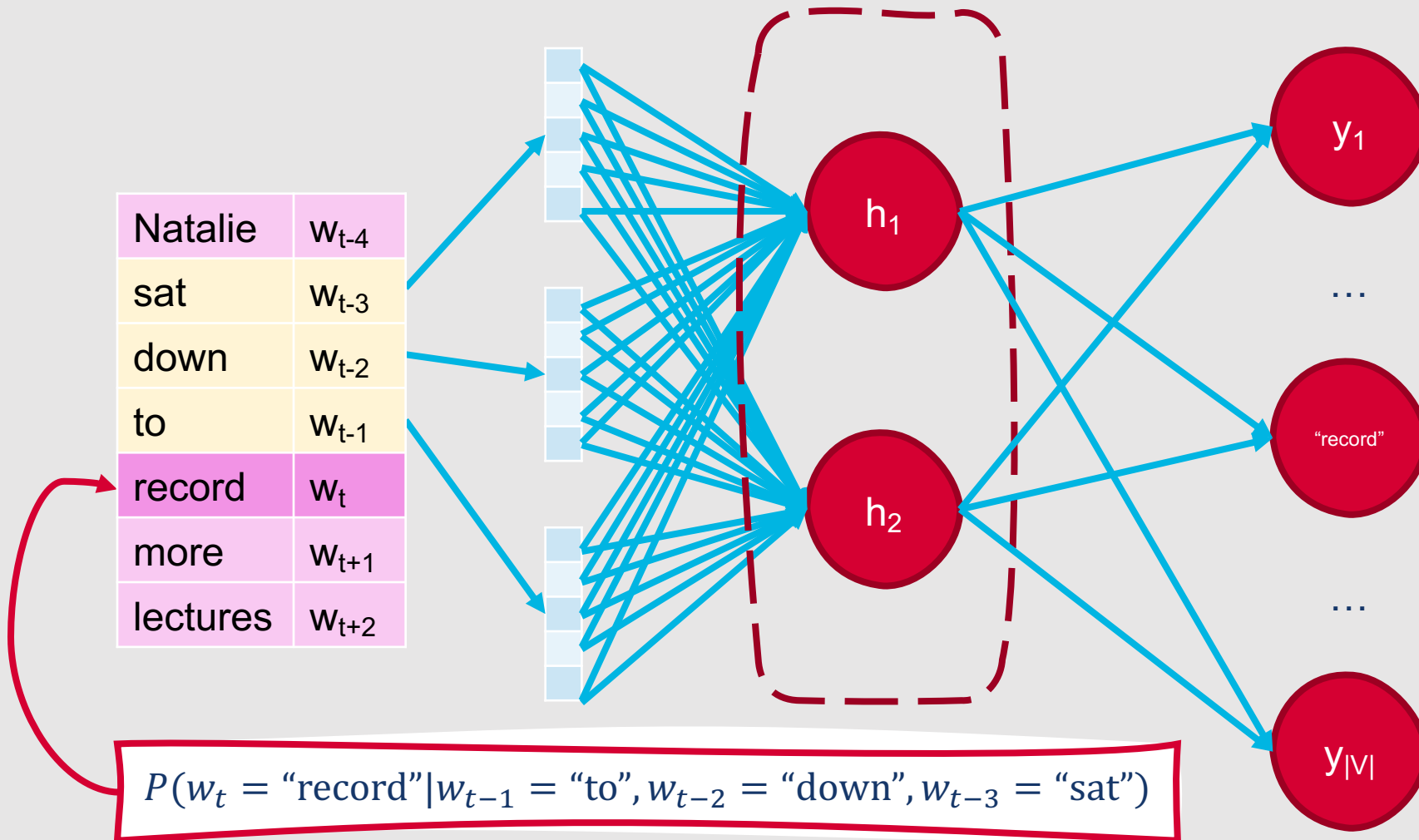


Neural Language Model

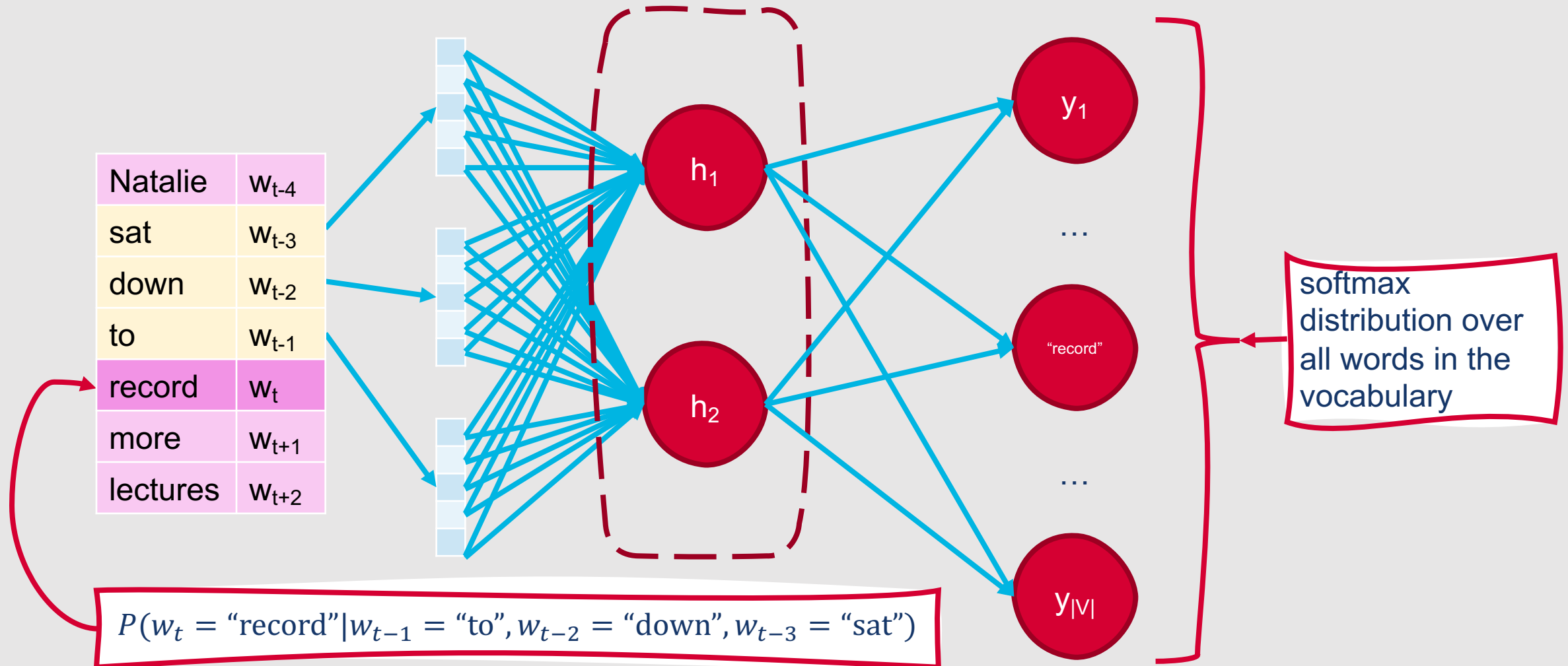


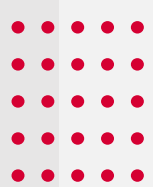
$$P(w_t = \text{"record"} | w_{t-1} = \text{"to"}, w_{t-2} = \text{"down"}, w_{t-3} = \text{"sat"})$$

Neural Language Model



Neural Language Model





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
- 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**